Ranking Programming Languages by Energy Efficiency

Rui Pereira[a,b], Marco Couto[c,b], Francisco Ribeiro[c,b], Rui Rua[c,b], Jácome Cunha[c,b], João Paulo Fernandes[d], João Saraiva[c,b]

[a] *C4 — Centro de Competências em Cloud Computing (C4-UBI), Universidade da Beira Interior, Rua Marquês d'Ávila e Bolama, 6201-001, Covilhã, Portugal*
[b] *HASLab/INESC Tec, Portugal*
[c] *Universidade do Minho, Portugal*
[d] *Departamento de Engenharia Informática, Faculdade de Engenharia da Universidade do Porto & CISUC, Portugal*

Abstract

This paper compares a large set of programming languages regarding their efficiency, including from an energetic point-of-view. Indeed, we seek to establish and analyze different rankings for programming languages based on their energy efficiency. The goal of being able to rank programming languages based on their energy efficiency is both recent, and certainly deserves further studies. We have taken rigorous and strict solutions to 10 well defined programming problems, expressed in (up to) 27 programming languages, from the well known Computer Language Benchmark Game repository. This repository aims to compare programming languages based on a strict set of implementation rules and configurations for each benchmarking problem. We have also built a framework to automatically, and systematically, run, measure and compare the energy, time, and memory efficiency of such solutions. Ultimately, it is based on such comparisons that we propose a series of efficiency rankings, based on single and multiple criteria.

Our results show interesting findings, such as how slower/faster languages can consume less/more energy, and how memory usage influences energy consumption. We also present a simple way to use our results to provide software engi-

Email addresses: `rui.a.pereira@inesctec.pt` (Rui Pereira), `marco.l.couto@inesctec.pt` (Marco Couto), `francisco.j.ribeiro@inesctec.pt` (Francisco Ribeiro), `rui.a.rua@inesctec.pt` (Rui Rua), `jacome@di.uminho.pt` (Jácome Cunha), `jpf@dei.uc.pt` (João Paulo Fernandes), `jas@di.uminho.pt` (João Saraiva)

neers and practitioners support in deciding which language to use when energy efficiency is a concern.
In addition, we further validate our results and rankings against implementations from a chrestomathy program repository, Rosetta Code., by reproducing our methodology and benchmarking system. This allows us to understand how the results and conclusions from our rigorously and well defined benchmarked programs compare to those based on more representative and real-world implementations. Indeed our results show that the rankings do not change apart from one programming language.
Keywords: Energy Efficiency, Programming Languages, Language Benchmarking, Green Software

1. Introduction

Software language engineering provides powerful techniques and tools to design, implement and evolve software languages. Such techniques aim at improving programmers productivity - by incorporating advanced features in the language design, like for instance powerful modular and type systems - and at efficiently executing such software - by developing, for example, aggressive compiler optimizations. Indeed, most techniques were developed with the main goal of helping software developers in producing faster programs.

More recently, this reality is quickly changing and software energy consumption is becoming a key concern for computer manufacturers, software language engineers, programmers, and even regular computer users. Nowadays, even mobile phone (which are powerful computers) users tend to avoid using CPU intensive applications just to save battery/energy. While the concern on the computers' energy efficiency started by the hardware manufacturers, it quickly became a concern for software developers too [1]. In fact, this is a recent and intensive area of research where several techniques to analyze and optimize the energy consumption of software systems are being developed. Such techniques already provide knowledge on the energy efficiency of data structures [2, 3, 4], the energy impact of different programming practices both in mobile [5, 6, 7] and desktop applications [8, 9], the energy efficiency of applications within the same scope [10, 11], or even on how to predict energy consumption in several software systems [12, 13], among several other works.

An interesting question that frequently arises in the software energy efficiency area is whether *a faster program is also an energy efficient program*, or not. In other words, does a faster program consume less energy due to performing the task quicker?. If the answer is yes, then optimizing a program for speed also means optimizing it for energy, and this is exactly what the compiler construction community has been doing since the very beginning of software languages. However, energy consumption does not depends only on execution time, as shown in the equation $E_{nergy} = T_{ime} \times P_{ower}$. In fact, there are several research works showing different results regarding this subject [14, 15, 16, 17, 2, 18].

A similar question arises when comparing software languages: *is a faster language, a more energy efficient one?* Comparing software languages, however, is an extremely complex task, since the performance of a language is influenced by the quality of its compiler, virtual machine, garbage collector, available libraries, etc. Indeed, a software program may become faster by improving its source code, but also by "just" optimizing its libraries and/or its compiler.

Such questions arise by both researchers and programmers as there is still a large difficulty of how to analyze, interpret, and optimize the energy consumption in software. In fact, studies [19, 1] have shown that programmers are very concerned with the energy consumption of their software, and many times seek help. There are many misconceptions within the programming community as to what causes high energy consumption and in what ways they can be solved [20]. Recent works [21, 22] argue that there are two main roadblocks in regards to energy efficiency software development: the *lack of tools* and *lack of knowledge*. The research work presented in this paper aims at helping with the lack of knowledge for energy efficient software development by tackling one of the initial steps in software development and providing further useful information:

what programming language should be chosen?

In previous works [23, 24], we have made coherent and consistent efforts to assess and compare the performance of (a total of) 27 of the most widely used software languages. We considered (a total of) ten different programming problems that are expressed in each of the languages, following the exact same algorithm, as defined in the *Computer Language Benchmark Game* (CLBG) [25]. We compiled/executed such programs using the state-of-the-art compilers, virtual machines, interpreters, and libraries for each language. Afterwards, we analyzed the performance of the different implementations considering runtime performance, i.e., execution time and peak memory consumption, and energy consumption. Moreover, we analyzed those results according to the languages' execution type (compiled, virtual machine and interpreted), and programming paradigm (imperative, functional, object oriented, scripting) used. For each of the execution types and programming paradigms, we compiled a software language ranking according to each objective individually considered (e.g., time or energy consumption). We have also proposed global rankings for all the possible combinations of objectives (e.g., time and energy consumption). While the study was performed on a server based system, there is clear evidence that such results can be mapped to embedded systems [26].

Additionally, our previous work was openly welcomed by the community of researchers and industrial practitioners with excitement and interest in our findings, sparking countless amounts of discussions. This result is expected, as our previously mentioned studies have shown that the programming community is very much concerned with the energy consumption of their software and seek ways to improve it, and insights tackling the lack of knowledge in the field is very appreciated. In fact, this can be clearly seen by practitioners' responses and discussions through online news pages[1], social-media[2], external teaching

[1]TheNewStack: `https://bit.ly/2yXESG3`

[2]Twitter threads: `https://bit.ly/3fYzFyG` — `https://bit.ly/3dRdZCN`

material[3], and Reddit[4] discussions. Such information has helped many rethink (and even publicly change) their programming language of choice as energy efficiency is a concern of theirs. For language developers, such information is also important in order to compare language performance (energy and time) against competitors. In fact, the competition in the computing world, from both software and hardware developers, drives and motivates further evolution. Language developers become excited to see that their language is competing to be a very efficient one, for example as shown through the Rust language newsletter[5].

This paper extends our previous work in two ways.

First, we have considered an alternative dimension within our earlier work. Indeed, one of the objectives we considered was peak memory usage, which did not prove to be correlated with memory energy consumption. Now, we are presenting total memory usage, or the accumulative amount of memory used through the application's lifecycle, as another possibility for analyzing memory behavior. Finally, we also present statistical correlation tests between energy, runtime, and memory.

Second, we present a second large study in order to provide a validation of our previous energy ranking that uses a more idiomatic and day-to-day code example base. Indeed, we consider a chrestomathy repository, *Rosetta Code* [27], of alternative solutions to programming problems that is maintained with the main goal of assisting programmers in understanding syntactic or semantic aspects of programming languages outside their domain of expertise. Thus, the solutions that are gathered have a clarity and pedagogical concern, which is essentially different when compared to CLBG, whose solutions are strictly performance-oriented. To validate, we considered 9 tasks from *Rosetta Code*, and their solutions in (up to) the 27 programming languages that we have previously considered. With this, we are also able to study the energy efficiency of program solutions from

[3]Programming course: `https://bit.ly/2Z2XcZ7`
[4]Reddit: `https://bit.ly/2LpSOeP` — `https://bit.ly/3dOBexh`
[5]Rust newsletter: `this-week-in-rust.org/blog/2017/09/19/this-week-in-rust-200/`

a performance-oriented source (CLBG) and an educational source (Rosetta), allowing us to analyze how performance vs. comprehensibility affects energy consumption.

With the proposition of a secondary ranking serving as a validation, we are interested in finding efficiency trends that confirm or contradict our earlier findings with respect to the efficiency of programming languages, and the representativeness of our benchmarks. This is aligned with our perspective that the insights provided by one ranking, if considered in isolation, are more subject to imprecise systematization, and might indeed benefit from complementary perspectives provided by different rankings. We believe that this is actually an idea that generalizes to traditional rankings, e.g., when considering the prestigious of worldwide Universities, and the multiple rankings that attempt to analyze it. Comparable rankings between the study and validation can solidify the results we have presented, while at the same time allowing us to understand how normal and more representative day-to-day programming styles and tendencies (Rosetta Code) compare to those focused on pure performance (CLBG).

As we have previously mentioned, the work presented in this paper extends previous work [23, 24]. While this work has been previously introduced, we go back to fully describing the methodology and results as to provide readers the full-picture and complete comprehensive overview of our context. This provides a fully self contained look at our presented research work. Thus, in this paper we present and answer the following research questions:

- **RQ1**: *Can we compare the energy efficiency of software languages?* This will allow us to have results in which we can in fact compare the energy efficiency of popular programming languages. In having these results, we can also explore the relations between energy consumption, execution time, and memory usage.

- **RQ2**: *Is the faster language always the most energy efficient?* Properly understanding this will not only address if energy efficiency is purely a performance problem, but also allow developers to have a greater understanding of how energy and time relates in a language, and between languages.

- **RQ3**: *How does memory usage relate to energy consumption?* Insight on how peak memory (highest amount of used memory in a given instance) and total memory (the accumulative amount of memory used) affects energy consumption will allow developers to better understand how to manage memory if their concern is energy consumption.

- **RQ4**: *Can we automatically decide what is the best programming language considering energy, time, and memory usage?* Often times developers are concerned with more than one (possibly limited) resource. For example, both energy and time, time and memory space, energy and memory space or all three. Analyzing these trade-offs will allow developers to know which programming languages are best in specific scenarios.

- **RQ5**: *How do the results of our energy consumption analysis of programming languages gathered from rigorous performance benchmarking solutions compare to results of average day-to-day solutions?* As the results and ranking we gathered are based off a competitive benchmarking structure for the performance of programming languages, the solutions may be very specific to the problem at hand and stray away from a typical and representative idiomatic style of programming for an everyday user. It is important to understand how such results compare to the day-to-day programming styles, which follow more flexible solutions and are written by more everyday users. This will allow us to conclude if our results are representative and generalizable to a degree, and understand in what cases and why differences occur.

The remainder of this paper is organized as follows: Section 2 details previous work which was used as the basis of this paper, which includes the steps of our rigorous and strict methodology to measure and compare the energy efficiency in software languages; this section also includes the description of our data set from CLBG and the study's results. Section 3 presents a discussion and ranking on the energy efficiency of each programming language based on the results. We describe in Section 4, how we structured a new validating study based on our previous methodology in order to produce a secondary ranking using the Rosetta Code chrestomathy repository of representative programs in order to validate our prior one and understand if our results are generalizable. In Section 5 we discuss the threats that may affect the validity of the insights we are drawing. Section 6 presents the related work, and finally, in Section 7 we present the conclusions of our work.

2. Measuring Energy in Software Languages

The initial motivation and primary focus of this work is to understand the energy efficiency across various programming languages. This might seem like a simple task, but it is not as trivial as it sounds. To properly compare the energy efficiency between programming languages, we must obtain various comparable implementations with a good representation of different problems/solutions.

The following subsections will detail the methodology used to answer this question, and the results we obtained. A large part of this section extends previous work [23, 24]. We feel this detailed description of previous work is necessary to provide a complete and comprenhensive description for the reader, allowing a better context of the additional contributions presented in this section, and our study presented in Section 4. This also allows us to present all the results from our studies and research questions in a fully contained manner, where we may focus completely on the topic at hand: ranking programming languages by energy efficiency.

2.1. The Computer Language Benchmarks Game

In order to obtain a comparable, representative and extensive set of programs written in many of the most popular and most widely used programming languages we have explored The Computer Language Benchmarks Game [25]. (CLBG).

The CLBG initiative includes a framework for running, testing and comparing implemented coherent solutions for a set of well-known, diverse programming problems. The overall motivation is to be able to compare solutions, within and between, different programming languages. While the perspectives for comparing solutions have originally essentially analyzed runtime performance, the fact is that CLBG has recently also been used in order to study the energy efficiency of software [17, 23, 4].

In its current stage, the CLBG has gathered solutions for 13 benchmark problems, such that solutions to each such problem must respect a given algorithm and specific implementation guidelines. Solutions to each problem are expressed in, at most, 28 different programming languages.

The complete list of benchmark problems in the CLBG covers different computing problems, as described in Table 1. Additionally, the complete list of programming languages in the CLBG is shown in Table 2, sorted by their paradigms.

2.2. Design and Execution

Our case study to analyze the energy efficiency of software languages is based on the CLBG.

From the 28 languages considered in the CLBG, we excluded *Smalltalk* since the specific compiler used by the CLBG is proprietary and was not available at the time of the study. Also, for comparability, we have discarded benchmark problems whose language coverage is below the threshold of 80%. By language coverage we mean, for each benchmark problem, the percentage of programming languages (out of 27) in which solutions for it are available. This criteria excluded `chameneos-redux`, `meteor-contest` and `thread-ring` from our study.

We then gathered the most efficient (i.e. fastest) version of the source code in each of the remaining 10 benchmark problems, for all the 27 considered programming languages.

The CLBG documentation also provides information about the specific compiler/runner version used for each language, as well as the compilation/execution

Table 1: CLBG corpus of programs.

Benchmark	Description	Input
`n-body`	Double precision N-body simulation	50M
`fannkuch-redux`	Indexed access to tiny integer sequence	12
`spectral-norm`	Eigenvalue using the power method	5,500
`mandelbrot`	Generate Mandelbrot set portable bitmap file	16,000
`pidigits`	Streaming arbitrary precision arithmetic	10,000
`regex-redux`	Match DNA 8mers and substitute magic patterns	fasta output
`fasta`	Generate and write random DNA sequences	25M
`k-nucleotide`	Hashtable update and k-nucleotide strings	fasta output
`reverse-complement`	Read DNA sequences, write their reverse-complement	fasta output
`binary-trees`	Allocate, traverse and deallocate many binary trees	21
`chameneos-redux`	Symmetrical thread rendezvous requests	6M
`meteor-contest`	Search for solutions to shape packing puzzle	2,098
`thread-ring`	Switch from thread to thread passing one token	50M

options considered (for example, optimization flags at compile/run time). We strictly followed those instructions and installed the correct compiler versions, and also ensured that each solution was compiled/executed with the same options used in the CLBG. Once we had the correct compiler and benchmark solutions for each language, we tested each one individually to make sure that we could execute it with no errors and that the output was the expected one.

The next step was to gather the information about energy consumption, execution time and peak memory usage for each of the compilable and executable solutions in each language. It is to be noted that the CLBG already contains measured information on both the execution time and peak memory usage. We measured both not only to check the consistency of our results

Table 2: Languages sorted by paradigm

Paradigm	Languages
Functional	Erlang, F#, Haskell, Lisp, Ocaml, Perl, Racket, Ruby, Rust;
Imperative	Ada, C, C++, F#, Fortran, Go, Ocaml, Pascal, Rust;
Object-Oriented	Ada, C++, C#, Chapel, Dart , F#, Java, JavaScript, Ocaml, Perl, PHP, Python, Racket, Rust, Smalltalk, Swift, TypeScript;
Scripting	Dart, Hack, JavaScript, JRuby, Lua, Perl, PHP, Python, Ruby, TypeScript;

against the CLBG, but also since different hardware specifications would bring about different results. For measuring the energy consumption, we used Intel's Running Average Power Limit (RAPL)[6] tool [28], which is capable of providing accurate energy estimates at a very fine-grained level, as it has already been proven [29, 30]. Also, the current version of RAPL allows it to be invoked from any program written in *C* and `Java` (through jRAPL [31]).

In order to properly compare the languages, we needed to collect the energy consumed by a single execution of a specific solution. In order to do this, we used the `system` function call in `C`, which executes the string values which are given as arguments; in our case, the command necessary to run a benchmark solution (for example, the `binary-trees` solution written in `Python` is executed by writing the command `/usr/bin/python binarytrees.py 21`).

The energy consumption of a solution will then be the energy consumed by the `system` call, which we measured using RAPL function calls. The overall process (i.e., the workflow of our energy measuring framework [7]) is described in Listing 1.

```
...
for (i = 0 ; i < N ; i++){
  time_before = getTime(...);
  //performs initial energy measurement
```

[6]This software requires an Intel processor (Sandybridge or higher) and atleast a Linux 3.13 kernel.

[7]The measuring framework is publicly available at `https://sites.google.com/view/energy-efficiency-languages`

```
  rapl_before(...);

  //executes the program
  system(command);

  //computes the difference between
  //this measurement and the initial one
  rapl_after(...);
  time_elapsed = getTime(...) - time_before;
  ...
}
...
```

Listing 1: Overall process of the energy measuring framework.

In order to ensure that the overhead from our measuring framework, using the `system` function, is negligible or non-existing when compared to actually measuring with RAPL inside a program's source code, we design a simple experiment. It consisted of measuring the energy consumption inside of both a C and Java language solution, using RAPL and jRAPL respectively, and comparing the results to the measurements from our C language energy measuring framework. We found the resulting differences to be insignificant, and therefore negligible, thus we conclude that we could use this framework without having to worry about imprecisions in the energy measurements.

Also, we chose to measure the energy consumption and the execution time of a solution together, since the overhead will be the same for every measurement, and so this should not affect the obtained values.

The peak memory usage of a solution was gathered using the `time` tool, available in Unix-based systems. This tool runs a given program, and summarizes the system resources used by that program, which includes the peak of memory usage. To measure the total memory, we used the Python `memory_profiler` [8]. library to obtain the values, and afterwards we calculated, for each language, the average of all solutions.

Each benchmark solution was executed and measured 10 times, in order to obtain 10 energy consumption and execution time samples. As commonly done, we did so to reduce the impact of cold starts for VMs (startups for VMs have a tendency to consume more resources during a few initial starts) and cache

[8]Python memory profiler page: `https://pypi.org/project/memory_profiler/`

effects, and to be able to analyze the measurements' consistency and avoid outliers. Additionally, between each measurement, we let the system rest for 2 minutes to allow a cool-down, as to not overheat and in turn affect energy measurements (which are susceptible to this). We followed the same approach when gathering results for memory usage. As the benchmarking problems are not memory heavy, there is no current concern in regards to memory swapping in large problems.

In order to compare different programming languages considering multiple characteristics (energy, time, and peak memory), and in order to help answer **RQ4**, we used a multi-objective optimization algorithm known as the Pareto optimization [32, 33]. Using the software available at [34], we calculated various Pareto fronts to produce rankings of a combination of the analyzed characteristics. To do so, the algorithm can be repeatedly applied to discover the most optimal language in various ranking levels. In other words, after identifying the best language(s), and re-applying the algorithm while removing the identified languages from consideration, we would then produce the second best language(s). Repeating this process, we can identify the third, fourth, and n-th best language(s), thus producing an easy to read Pareto ranking.

For some benchmark problems, we could not obtain any results for certain programming languages. In some cases, there was no source code available for the benchmark problem (i.e., no implementation was provided in a concrete language which reflects a language coverage below 100%).[9]

In other cases, the code was indeed provided but either the code itself was already buggy or failing to compile or execute, as documented in CLBG, or, in spite of our best efforts, we could not execute it, e.g., due to missing libraries [9]. From now on, for each benchmark problem, we will refer as its execution coverage to the percentage of (best) solutions for it that we were actually able to successfully execute.

All studies were conducted on a desktop with the following specifications:

[9]In these cases, we will include an `n.a.` indication when presenting their results.

Linux Ubuntu Server 16.10 operating system, kernel version 4.8.0-22-generic, with 16GB of RAM, a Haswell Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz.

2.3. Results

The results from our study are partially shown in this section, with all the remaining benchmark results (including means, standard deviation, and box-plot data) shown in the online appendix for this paper [7]. Table 3, and the left most tables under *Results - A. Data Tables* in the appendix, contains the measured data from different benchmark solutions. We only show the results for `binary-trees`, `fannkuch-redux`, and `fasta` within the paper, which are the first 3 ordered alphabetically. Each row in a table represents one of the 27 programming languages which were measured.

The 4 rightmost columns, from left to right, represent the average values for the *Energy* consumed (Joules), *Time* of execution (milliseconds), *Ratio* between Energy and Time, and the amount of peak memory usage in *Mb*. The *Energy* value is the sum of CPU and DRAM energy consumption. Additionally, the *Ratio* can also be seen as the average Power, expressed in Kilowatts (kW). The rows are ordered according to the programming language's energy consumption, from lowest to highest. Finally, the right most tables under *Results - A. Data Tables* contain the standard deviation and average values for our measured CPU, DRAM, and Time, allowing us to understand the variance.

The first column states the name of the programming languages, preceded by either a (c), (i), or (v) classifying them as either a compiled, interpreted, or virtual-machine language, respectively. In some cases, the programming language name will be followed with a $\uparrow_x/\downarrow_y$ and/or $\Uparrow_x/\Downarrow_y$ symbol. The first set of arrows indicates that the language would go up by x positions ($\uparrow_x$) or down by y positions ($\downarrow_y$) if ordered by *execution time*. For example in Table 3, for the `fasta` benchmark, `Fortran` is the second most energy efficient language, but falls off 6 positions down if ordered by execution time. The second set of arrows states that the language would go up by x positions ($\Uparrow_x$) or down by y positions ($\Downarrow_y$) if ordered according to their *peak memory usage*. Looking at the

Table 3: Results for binary-trees, fannkuch-redux, and fasta

binary-trees	Energy (J)	Time (ms)	Ratio (J/ms)	Mb
(c) C	39.80	1125	0.035	131
(c) C++	41.23	1129	0.037	132
(c) Rust $\Downarrow_2$	49.07	1263	0.039	180
(c) Fortran $\Uparrow_1$	69.82	2112	0.033	133
(c) Ada $\Downarrow_1$	95.02	2822	0.034	197
(c) Ocaml $\downarrow_1$ $\Uparrow_2$	100.74	3525	0.029	148
(v) Java $\uparrow_1$ $\Downarrow_{16}$	111.84	3306	0.034	1120
(v) Lisp $\downarrow_3$ $\Downarrow_3$	149.55	10570	0.014	373
(v) Racket $\downarrow_4$ $\Downarrow_6$	155.81	11261	0.014	467
(i) Hack $\uparrow_2$ $\Downarrow_9$	156.71	4497	0.035	502
(v) C# $\downarrow_1$ $\Downarrow_1$	189.74	10797	0.018	427
(v) F# $\downarrow_3$ $\Downarrow_1$	207.13	15637	0.013	432
(c) Pascal $\downarrow_3$ $\Uparrow_5$	214.64	16079	0.013	256
(c) Chapel $\uparrow_5$ $\Uparrow_4$	237.29	7265	0.033	335
(v) Erlang $\uparrow_5$ $\Uparrow_1$	266.14	7327	0.036	433
(c) Haskell $\uparrow_2$ $\Downarrow_2$	270.15	11582	0.023	494
(i) Dart $\downarrow_1$ $\Uparrow_1$	290.27	17197	0.017	475
(i) JavaScript $\downarrow_2$ $\Downarrow_4$	312.14	21349	0.015	916
(i) TypeScript $\downarrow_2$ $\Downarrow_2$	315.10	21686	0.015	915
(c) Go $\uparrow_3$ $\Uparrow_{13}$	636.71	16292	0.039	228
(i) Jruby $\uparrow_2$ $\Downarrow_3$	720.53	19276	0.037	1671
(i) Ruby $\Uparrow_5$	855.12	26634	0.032	482
(i) PHP $\Uparrow_3$	1,397.51	42316	0.033	786
(i) Python $\Uparrow_{15}$	1,793.46	45003	0.040	275
(i) Lua $\downarrow_1$	2,452.04	209217	0.012	1961
(i) Perl $\uparrow_1$	3,542.20	96097	0.037	2148
(c) Swift	n.e.			

fannkuch-redux	Energy (J)	Time (ms)	Ratio (J/ms)	Mb
(c) C $\Downarrow_2$	215.92	6076	0.036	2
(c) C++ $\Uparrow_1$	219.89	6123	0.036	1
(c) Rust $\Downarrow_{11}$	238.30	6628	0.036	16
(c) Swift $\Downarrow_5$	243.81	6712	0.036	7
(c) Ada $\Downarrow_2$	264.98	7351	0.036	4
(c) Ocaml $\downarrow_1$	277.27	7895	0.035	3
(c) Chapel $\uparrow_1$ $\Downarrow_{18}$	285.39	7853	0.036	53
(v) Lisp $\downarrow_3$ $\Downarrow_{15}$	309.02	9154	0.034	43
(v) Java $\uparrow_1$ $\Downarrow_{13}$	311.38	8241	0.038	35
(c) Fortran $\Downarrow_1$	316.50	8665	0.037	12
(c) Go $\uparrow_2$ $\Uparrow_7$	318.51	8487	0.038	2
(c) Pascal $\Uparrow_{10}$	343.55	9807	0.035	2
(v) F# $\downarrow_1$ $\Downarrow_7$	395.03	10950	0.036	34
(v) C# $\uparrow_1$ $\Downarrow_5$	399.33	10840	0.037	29
(i) JavaScript $\downarrow_1$ $\Downarrow_2$	413.90	33663	0.012	26
(c) Haskell $\uparrow_1$ $\Uparrow_8$	433.68	14666	0.030	7
(i) Dart $\Downarrow_7$	487.29	38678	0.013	46
(v) Racket $\Uparrow_3$	1,941.53	43680	0.044	18
(v) Erlang $\Uparrow_3$	4,148.38	101839	0.041	18
(i) Hack $\Downarrow_6$	5,286.77	115490	0.046	119
(i) PHP	5,731.88	125975	0.046	34
(i) TypeScript $\downarrow_4$ $\Uparrow_4$	6,898.48	516541	0.013	26
(i) Jruby $\uparrow_1$ $\Downarrow_4$	7,819.03	219148	0.036	669
(i) Lua $\downarrow_3$ $\Uparrow_{19}$	8,277.87	635023	0.013	2
(i) Perl $\uparrow_2$ $\Uparrow_{12}$	11,133.49	249418	0.045	12
(i) Python $\uparrow_2$ $\Uparrow_{14}$	12,784.09	279544	0.046	12
(i) Ruby $\uparrow_2$ $\Uparrow_{17}$	14,064.98	315583	0.045	8

fasta	Energy (J)	Time (ms)	Ratio (J/ms)	Mb
(c) Rust $\Downarrow_9$	26.15	931	0.028	16
(c) Fortran $\downarrow_6$	27.62	1661	0.017	1
(c) C $\uparrow_1$ $\Downarrow_1$	27.64	973	0.028	3
(c) C++ $\uparrow_1$ $\Downarrow_2$	34.88	1164	0.030	4
(v) Java $\uparrow_1$ $\Downarrow_{12}$	35.86	1249	0.029	41
(c) Swift $\Downarrow_9$	37.06	1405	0.026	31
(c) Go $\downarrow_2$	40.45	1838	0.022	4
(c) Ada $\downarrow_2$ $\Uparrow_3$	40.45	2765	0.015	3
(c) Ocaml $\downarrow_2$ $\Downarrow_{15}$	40.78	3171	0.013	201
(c) Chapel $\uparrow_5$ $\Downarrow_{10}$	40.88	1379	0.030	53
(v) C# $\uparrow_4$ $\Downarrow_5$	45.35	1549	0.029	35
(i) Dart $\Downarrow_6$	63.61	4787	0.013	49
(i) JavaScript $\Downarrow_1$	64.84	5098	0.013	30
(c) Pascal $\downarrow_1$ $\Uparrow_{13}$	68.63	5478	0.013	0
(i) TypeScript $\downarrow_2$ $\Downarrow_{10}$	82.72	6909	0.012	271
(v) F# $\uparrow_2$ $\Uparrow_3$	93.11	5360	0.017	27
(v) Racket $\downarrow_1$ $\Uparrow_5$	120.90	8255	0.015	21
(c) Haskell $\uparrow_2$ $\Downarrow_8$	205.52	5728	0.036	446
(v) Lisp $\Downarrow_2$	231.49	15763	0.015	75
(i) Hack $\Downarrow_3$	237.70	17203	0.014	120
(i) Lua $\Uparrow_{18}$	347.37	24617	0.014	3
(i) PHP $\downarrow_1$ $\Uparrow_{13}$	430.73	29508	0.015	14
(v) Erlang $\uparrow_1$ $\Uparrow_{12}$	477.81	27852	0.017	18
(i) Ruby $\downarrow_1$ $\Uparrow_2$	852.30	61216	0.014	104
(i) JRuby $\uparrow_1$ $\Downarrow_2$	912.93	49509	0.018	705
(i) Python $\downarrow_1$ $\Uparrow_{18}$	1,061.41	74111	0.014	9
(i) Perl $\uparrow_1$ $\Uparrow_8$	2,684.33	61463	0.044	53

same example benchmark, `Rust`, while the most energy efficient, would drop 9 positions if ordered by peak memory usage.

Table 4: Normalized global results for Energy, Time, and Memory

Total					
	Energy (J)		Time (ms)		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Table 4 shows the global results (on average) for *Energy*, *Time*, and *Mb* normalized to the most efficient language in that category. Since the `pidigits` benchmark solutions only contained less than half of the languages covered, we did not consider this one for the global results. The base values are as follows: *Energy* for `C` is 57.86J, *Time* for `C` is 2019.26ms, and *Mb* for `Pascal` is 65.96Mb. For instance, `Lisp`, on average, consumes 2.27x more energy (131.34J) than `C`, while taking 2.44x more time to execute (4926.99ms), and 1.92x more memory (126.64Mb) needed when compared to `Pascal`.

In order to better assess and interpret the differences among the considered languages in terms of elapsed time, memory and energy usage considering statistical evidences, a hierarchical clustering method was used. The graphical result of such a method is presented in the form of dendrograms, a format typically used to observe hierarchical relationship between objects, where the objects are joined together in a hierarchical manner from the closest grouping to the furthest. In these kinds of diagrams, it is crucial to look at the heights in which any two objects are joined together, as they reflect the distance between the clusters. In order to generate such clusters, we selected the 4 benchmarks (fannkuch-redux, fasta, n-body, spectral-norm) where all 27 languages were represented, as to maintain the 100% code coverage requirement needed to run such an analysis. Next, we used the Python package plotly[10] to generate the distinct clusters of languages for Energy consumption (CPU + DRAM), Time, and Memory.

Figures 1, 3, 5 show the generated dendogram for energy, time and memory for all considered languages. The X-axis in each Figure detail the different programming languages under analysis. The Y-axis represent the total amount of energy consumed in Joules (J), the total execution time in milliseconds (ms), and the peak memory in Megabytes (Mb) for Figures 1, 3, and 5 respectively. Looking at these diagrams, it is clearly noticeable that there are languages significantly distant from the group at the center of the graph (the most efficient languages in the given metric category). Given the distance between these groups of languages, and in order to obtain a more visually informative dendrogram, the same method of hierarchical clustering was applied, but now only considering the innermost languages. The results of this new procedure are shown in Figures 2, 4, 6. Here again the X-axis represent the different programming languages while the Y-axis represent the total amount of energy (Joules), time spent (ms), and peak memory (Mb) for the three respective Figures.

To better visualize and interpret the data, we also generated two different sets of graphical data for each of the benchmarks. The first set, Figures 7-9 and the left most figures under *Results - C. Energy and Time Graphs* in the appendix, contains the results of each language for a benchmark, consisting of three joint parts: a bar chart, a line chart, and a scatter plot. The bars represent the energy consumed by the languages, with the CPU energy consumption on the bottom half in blue dotted bars and DRAM energy consumption on the top half in orange solid bars, and the left y-axis representing the average Joules.

[10]Plotly package: `https://github.com/plotly/plotly.py`

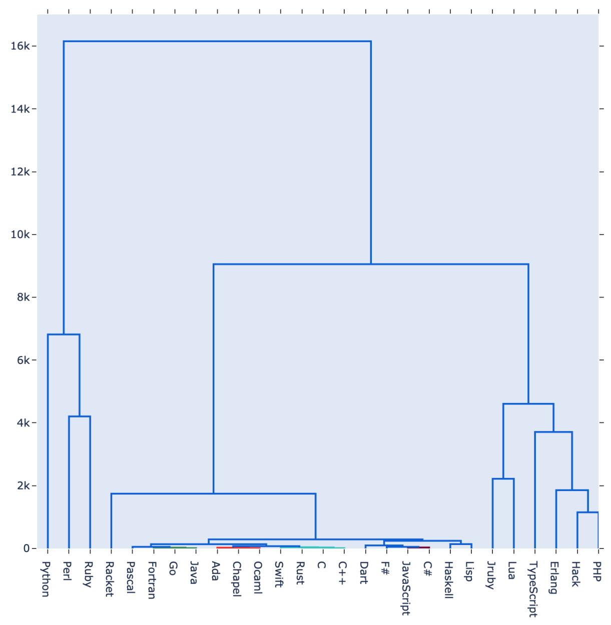


Figure 1: Dendogram representing CPU+DRAM energy consumption

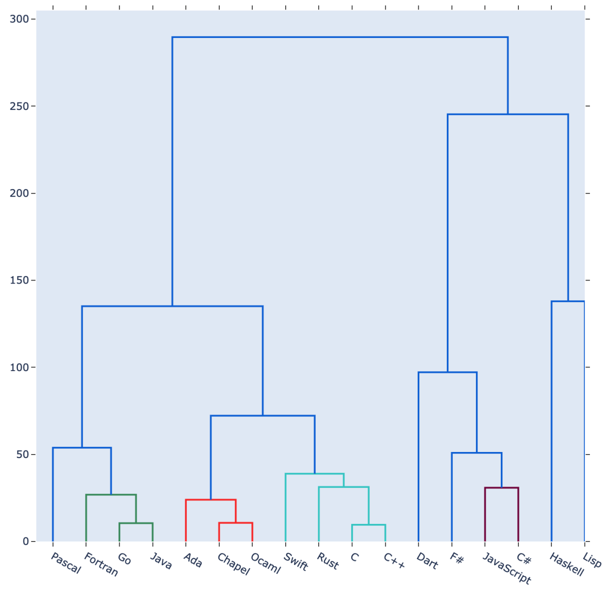


Figure 2: Dendogram representing CPU+DRAM energy consumption of the most efficient languages

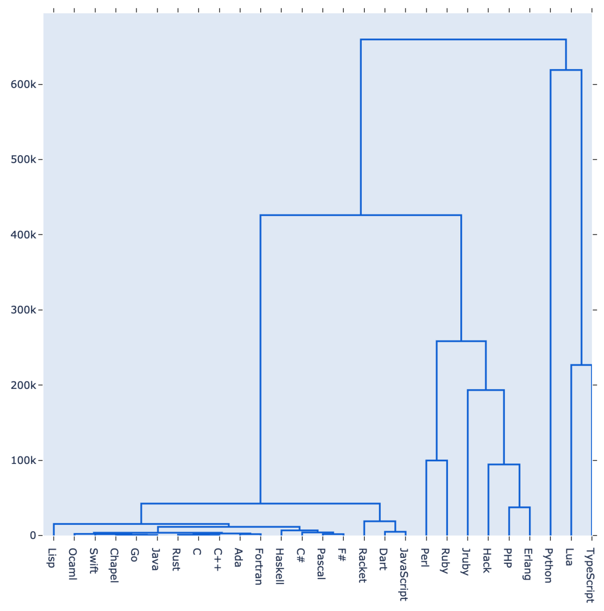


Figure 3: Dendogram representing runtime of all languages

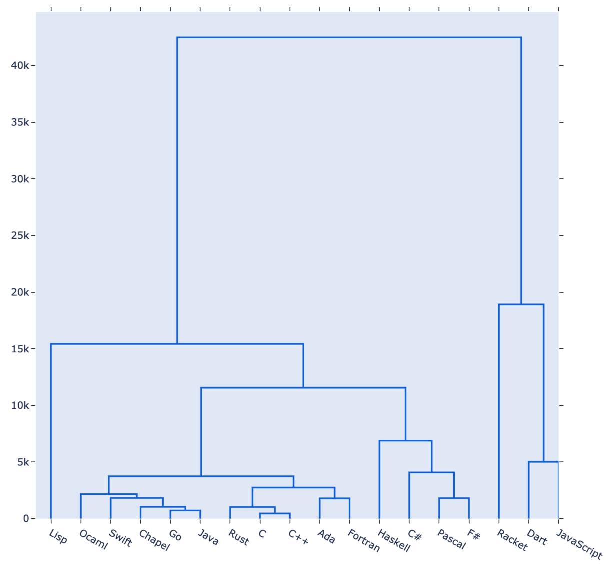


Figure 4: Dendogram representing runtime of the fastest languages

The execution time is represented by the line chart, with the right y-axis representing average time in milliseconds. The joining of these two charts allow

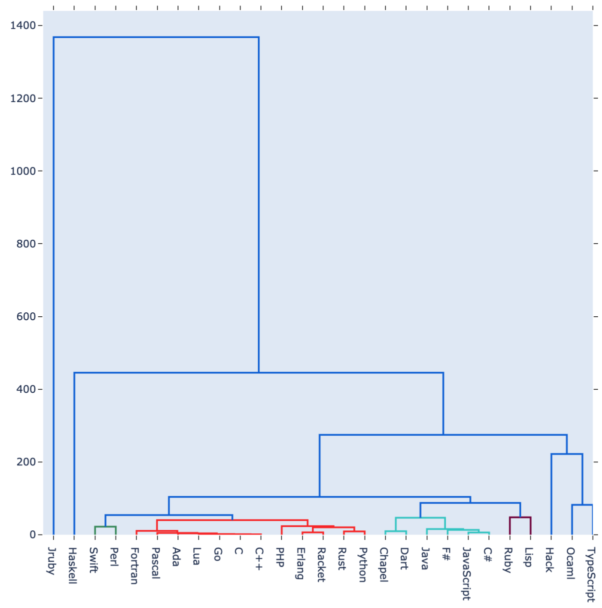


Figure 5: Dendogram representing Memory usage

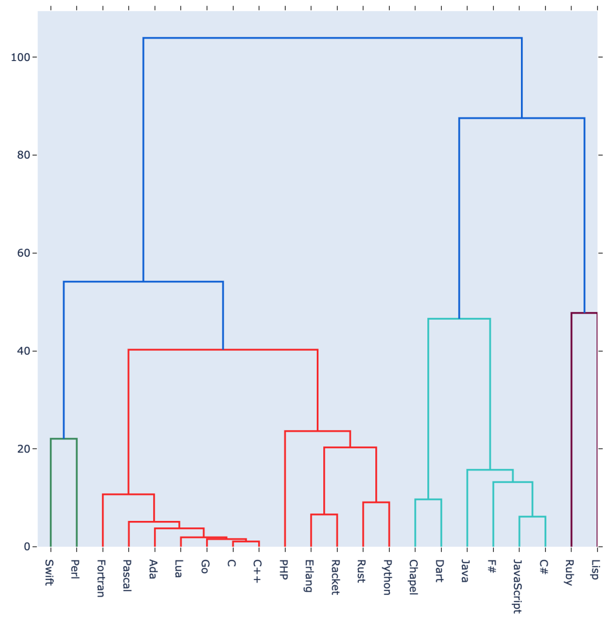


Figure 6: Dendogram representing Memory usage of the most memory efficient languages

us to better understand the relationship between energy and time. Finally, a scatter plot on top of both represents the ratio between energy consumed and execution time.

The ratio plot allows us to understand if the relationship between energy and time is consistent across languages. A variation in these values indicates that energy consumed is not directly proportional to time, but dependent on the language and/or benchmark solution.

The second set, Figures 10-12 and the right most figures under *Results - C. Energy and Time Graphs* in the appendix, consists of two parts: a bar chart, and a line chart. The blue bars represent the DRAM's energy consumption for each of the languages, with the left y-axis representing the average Joules. The orange line chart represents the peak memory usage for each language, with the right y-axis representing the average peak Mb. The joining of these two allows us to look at the relation between DRAM energy consumption and the peak memory usage for each language in each benchmark.

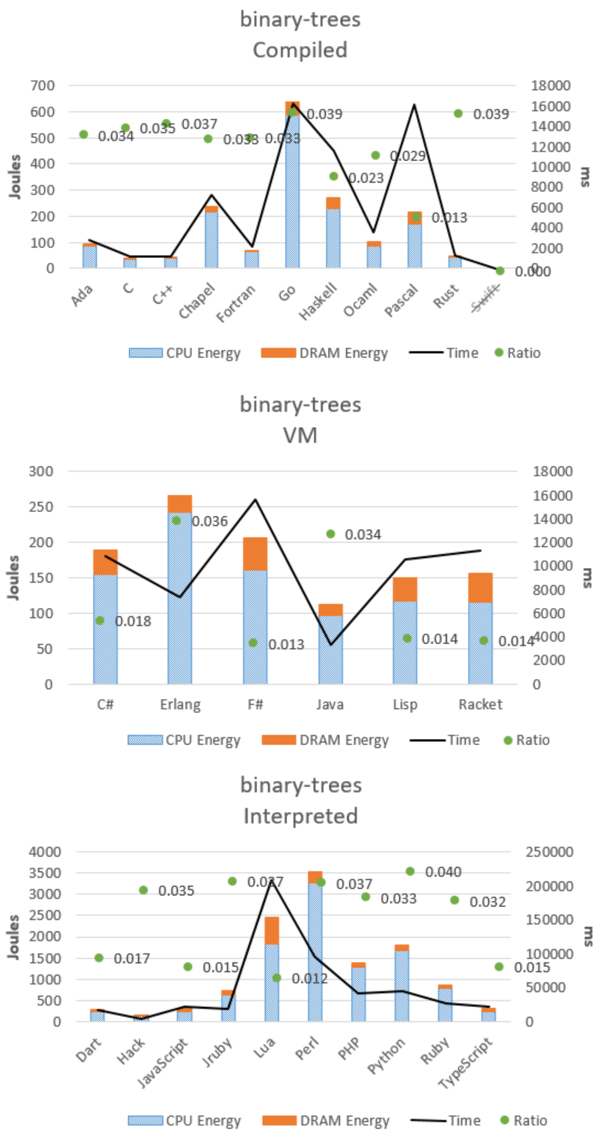
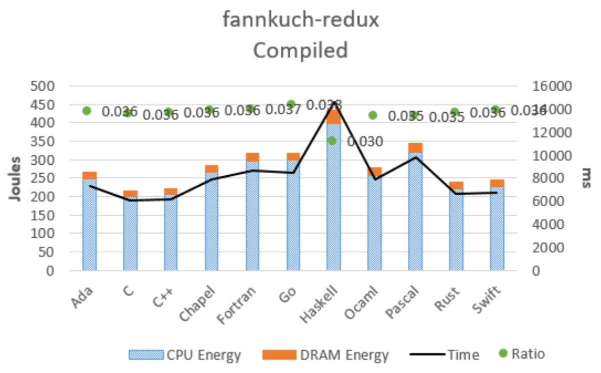
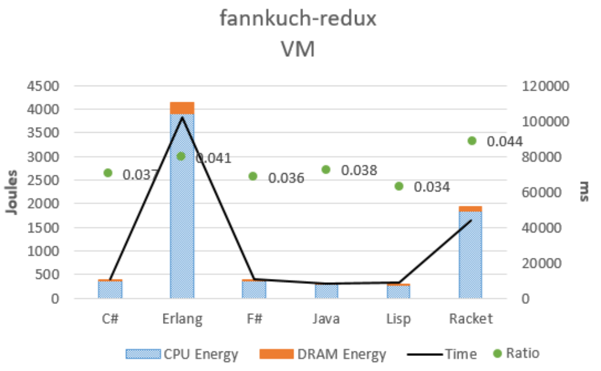


Figure 7: Energy and time graphical data for binary-trees





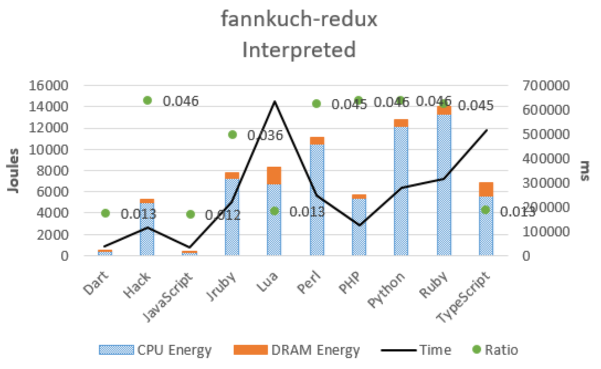


Figure 8: Energy and time graphical data for fannkuch-redux

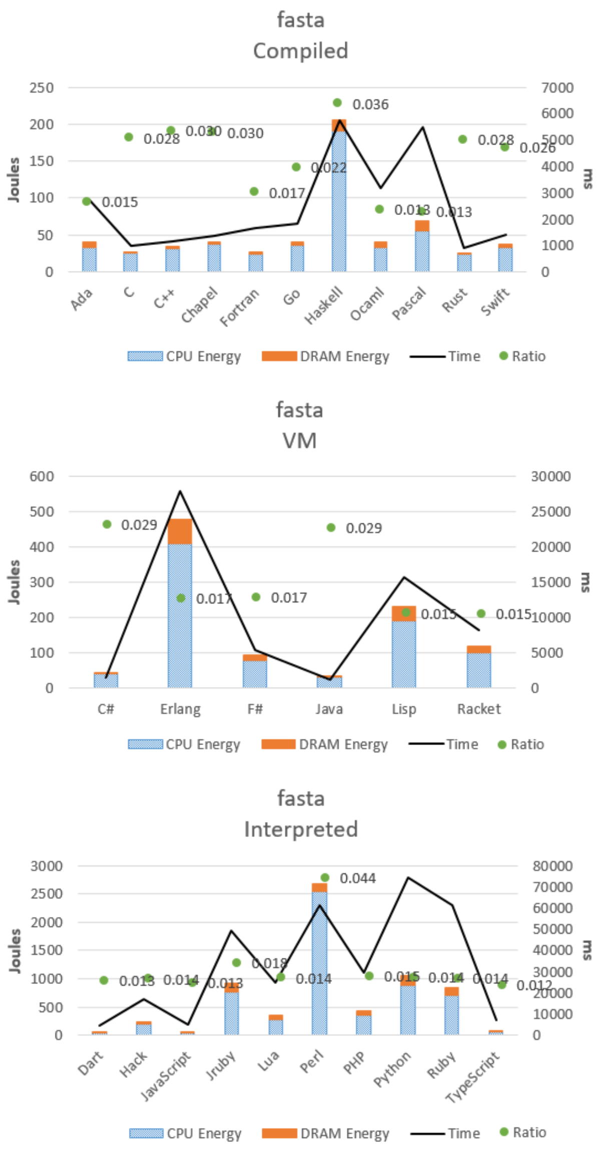


Figure 9: Energy and time graphical data for fasta

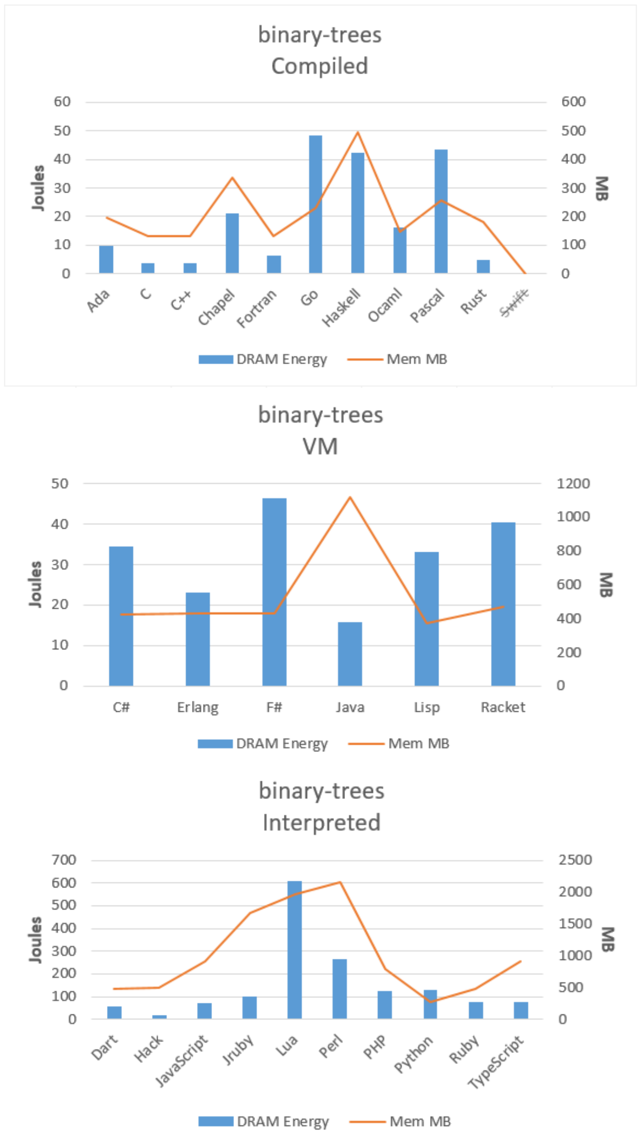


Figure 10: Energy and memory graphical data for binary-trees

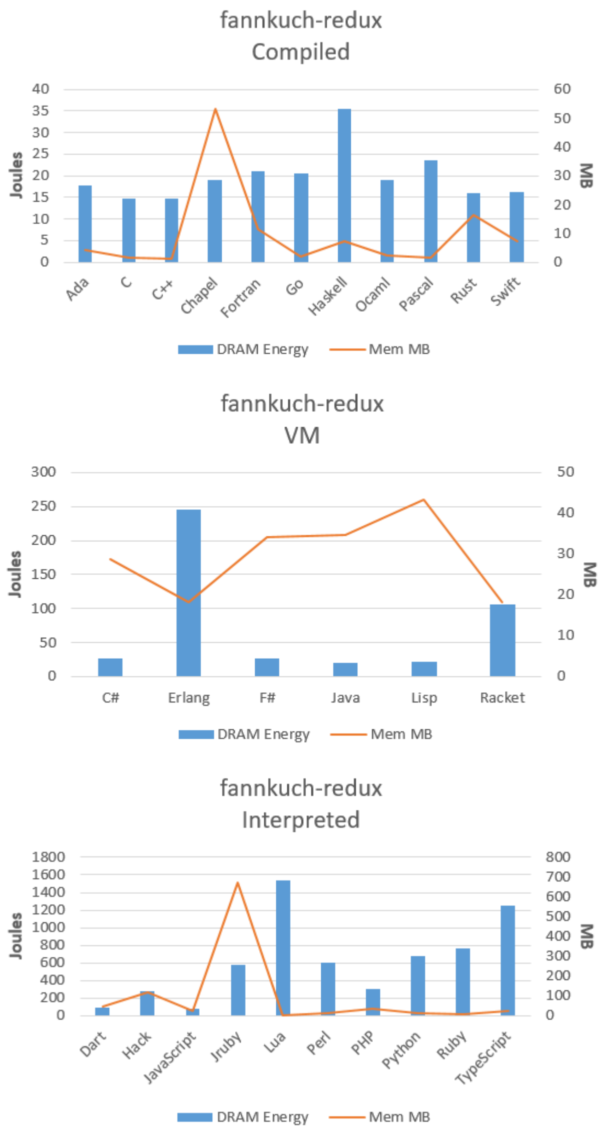


Figure 11: Energy and memory graphical data for fannkuch-redux

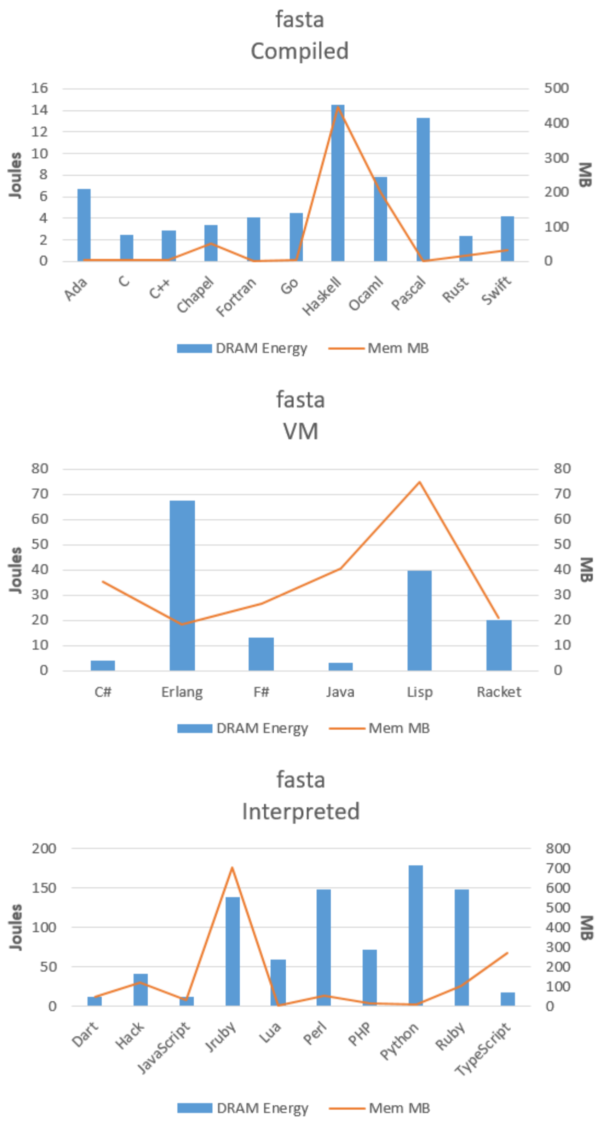


Figure 12: Energy and memory graphical data for fasta

In order to evaluate the relationship between energy consumption, runtime and memory usage with statistical support, we first applied both the methods of Shapiro [35] and Anderson–Darling [36], and verified that our data was non-parametric. As such, we used the Spearman correlation coefficient. The Spearman's correlation coefficient is a statistical measure of the strength of a monotonic relationship between paired data. In this way, we evaluated the level of correlation between the Energy-Memory, Energy-Time and Memory-Time pairs, in order to be able to evaluate how these metrics are related between the different languages and whether high/low values of one of these metrics have significant consequences in one of the pairs. Since the Spearman method returns a ρ value between -1 and 1, indicating a negative and positive correlation, we also applied the of Rea and Parker [37] method to obtain a nominal classification of the correlation value module. This value can be interpreted as a measure of effect size between the paired data. The classification assigned to each range of values according to Rea and Parker is as follows:

- $0.00 < 0.10$ - Negligible
- $0.10 < 0.20$ - Weak
- $0.20 < 0.40$ - Moderate
- $0.40 < 0.60$ - Relatively strong
- $0.60 < 0.80$ - Strong
- $0.80 <= 1.00$ - Very strong

Table 5 illustrates the values obtained for the complete set of problems of the 27 languages considered. For each pair of metrics of each language, the respective Spearman ρ values and respective classification are presented.

Finally, Table 6 summarizes the results of the DRAM energy consumption (Joules), peak memory usage (Mb), and total memory usage (Mb).

Table 5: Correlation values for all languages

Language	**Energy & Time**		**Energy & Memory**		**Memory & Time**	
	Spearman ρ	**Correlation**	**Spearman ρ**	**Correlation**	**Spearman ρ**	**Correlation**
Ada	0.950000	Very strong	0.150000	Weak	0.000000	None
C	0.966667	Very strong	-0.583333	Relatively strong	-0.616667	Strong
C++	0.966667	Very strong	-0.033333	Negligible	-0.166667	Weak
Chapel	0.916667	Very strong	0.250000	Moderate	0.183333	Weak
Fortran	0.952381	Very strong	0.190476	Weak	0.190476	Weak
Go	0.976190	Very strong	0.285714	Moderate	0.166667	Weak
Haskell	0.857143	Very strong	0.214286	Moderate	0.071429	Negligible
Ocaml	0.933333	Very strong	0.516667	Relatively strong	0.433333	Relatively strong
Pascal	0.928571	Very strong	0.000000	None	0.000000	None
Rust	0.783333	Strong	-0.309626	Moderate	-0.393309	Moderate
Dart	0.916667	Very strong	0.266667	Moderate	0.266667	Moderate
Hack	0.904762	Very strong	-0.476190	Relatively strong	-0.476190	Relatively strong
JavaScript	0.983333	Very strong	-0.066667	Negligible	-0.066667	Negligible
Jruby	0.966667	Very strong	-0.300000	Moderate	-0.366667	Moderate
Lua	0.928571	Very strong	-0.333333	Moderate	-0.333333	Moderate
Perl	0.983333	Very strong	-0.333333	Moderate	-0.333333	Moderate
PHP	0.900000	Very strong	-0.216667	Moderate	-0.333333	Moderate
Python	0.933333	Very strong	-0.866667	Very strong	-0.866667	Very strong
Ruby	0.933333	Very strong	-0.816667	Very strong	-0.916667	Very strong
TypeScript	1.000000	Very strong	-0.257143	Moderate	-0.257143	Moderate
C#	0.966667	Very strong	0.527201	Relatively strong	0.443519	Relatively strong
Erlang	0.904762	Very strong	-0.095238	Negligible	-0.095238	Negligible
F#	0.866667	Very strong	0.716667	Strong	0.716667	Strong
Java	0.833333	Very strong	0.083333	Negligible	-0.233333	Moderate
Lisp	0.857143	Very strong	0.214286	Moderate	0.214286	Moderate
Racket	0.933333	Very strong	-0.083333	Negligible	-0.083333	Negligible
Swift	1.000000	Very Strong	0.142857	Weak	0.142857	Weak

Table 6: Results for DRAM Energy Consumption and Total Memory

	DRAM (J)	Peak MB	Total MB
(c) C	5.28	77	626
(c) Rust	5.70	102	1087
(c) C++	8.54	88	2274
(c) Ada	10.00	97	3020
(c) Pascal	15.24	66	3046
(v) Erlang	205.36	475	5457
(c) Go	15.49	69	5797
(v) Lisp	23.84	127	7544
(c) Haskell	22.40	162	8126
(c) Chapel	12.37	264	10513
(c) Fortran	24.16	82	10715
(v) Java	12.89	397	13935
(v) C#	18.62	188	14351
(c) Swift	25.72	179	23102
(v) F#	35.28	280	30218
(i) Dart	36.24	570	33891
(c) OCaml	19.62	186	36839
(v) Racket	63.29	232	38921
(i) TypeScript	272.30	309	52967
(i) JavaScript	42.70	303	88831
(i) Python	358.75	185	116265
(i) PHP	155.13	169	188136
(i) Hack	133.88	221	194589
(i) Ruby	353.00	262	203864
(i) Perl	326.82	437	255738
(i) Lua	487.50	444	690087
(i) JRuby	383.85	1309	890144

3. Analysis and Discussion

By turning to the CLBG, we were able to use a large set of software programming languages which solve various different programming problems with similar solutions. This allowed us to obtain a comparable, representative, and extensive set of programs, written in several of the most popular languages, along with the compilation/execution options, and compiler versions. With these joined together with our energy measurement framework, which uses the accurate Intel RAPL tool, we were able to measure, analyze, and compare the energy consumption, and in turn the energy efficiency, of software languages, thus answering **RQ1** as shown with our results. Additionally, we were also able to measure the execution time and the peak and total memory usage, allowing us to analyze how these two relate with energy consumption.

In the following subsections, we will present an analysis and discussion on the results of our study. While our main focus is on understanding the energy efficiency in languages, we will also try to understand how energy, time, and memory relate. Additionally, in this section we will try to answer the following three research questions, each with their own designated subsection.

3.1. Is Faster, Greener?

A very common misconception when analyzing energy consumption in software is that it will behave in the same way execution time does. In other words, reducing the execution time of a program would bring about the same amount of energy reduction. In fact, the `Energy` equation, `Energy (J) = Power (W) x Time(s)`, indicates that reducing time implies a reduction in the energy consumed. However, the `Power` variable of the equation, which cannot be assumed as a constant, also has an impact on the energy. Therefore, conclusions regarding this issue diverge sometimes, where some works do support that energy and time are directly related [14], and the opposite was also observed [17, 15, 16].

The data presented in the aforementioned tables and figures lets us draw an interesting set of observations regarding the efficiency of software languages when considering both energy consumption and execution time. Much like [18] and [2], we observed different behaviors for energy consumption and execution time in different languages and tests.

By observing the data in Table 4, we can see that the `C` language is, overall, the fastest and most energy efficient. Nevertheless, in some specific benchmarks there are more efficient solutions (for example, in the `fasta` benchmark it is the third most energy efficient and second fastest).

Execution time behaves differently when compared to energy efficiency. The results for the 3 benchmarks presented in Table 3 (and the remainder shown in the appendix) show several scenarios where a certain language energy consumption rank differs from the execution time rank (as the arrows in the first column indicate). In the `fasta` benchmark, for example, the `Fortran` language is second most energy efficient, while dropping 6 positions when it comes to execution time. Moreover, by observing the *Ratio* values in Figures 7 to 9 (and the remainder in the appendix under *Results - C. Energy and Time Graphs*), we clearly see a substantial variation between languages. This means that the average power is not constant, which further strengthens the previous point. With this variation, we can have languages with very similar energy consumptions and completely different execution times, as is the case of languages `Pascal` and `Chapel` in the `binary trees` benchmark, which energy consumption differ roughly by 10% in favor of `Pascal`, while `Chapel` takes about 55% less time to execute.

Compiled languages tend to be, as expected, the fastest and most energy efficient ones. On average, compiled languages consumed 120J to execute the solutions, while for virtual machine and interpreted languages this value was 576J and 2365J, respectively. This tendency can also be observed for execution time, since compiled languages took 5103ms, virtual machine languages took 20623ms, and interpreted languages took 87614ms (on average). Grouped by the different paradigms, the imperative languages consumed and took on average 125J and 5585ms, the object-oriented consumed 879J and spent 32965ms, the functional consumed 1367J and spent 42740ms and the scripting languages consumed 2320J and spent 88322ms.

Moreover, the top 5 languages that need less energy and time to execute the solutions are: `C` (57J, 2019ms), `Rust` (59J, 2103ms), `C++` (77J, 3155ms), `Ada` (98J, 3740ms), and `Java` (114J, 3821ms); of these, only `Java` is not compiled. As expected, the bottom 5 languages are all interpreted: `Perl` (4604J), `Python` (4390J), `Ruby` (4045J), `JRuby` (2693J), and `Lua` (2660Js) for energy; `Lua` (167416ms), `Python` (145178ms), `Perl` (132856ms), `Ruby` (119832ms), and `TypeScript` (93292ms) for time.

Looking at the dendogram clustering results in Figure 1 and Figure 3 (for energy and execution time respectively), we see that the most efficient and least efficient group of languages tend to, more or less, follow the same ranking to the results we present in Table 4. The most energy efficient clusters, more clearly visible in Figure 2, contain the top energy efficient languages languages present in Table 4, while the same repeats for the least energy efficient clusters.

The most notable differences can be attributed to the fact that these results are based off only 4 of the benchmarks, in order to represent all 27 languages with 100% coverage (as necessary for the hierachichal clustering analysis), while the results in Table 4 take into consideration the remainder. This also explains why the Pascal programming language is shown to be the one of the least energy efficient languages of the innermost cluster, as it was out performed by most of the languages in the 4 considered benchmarks. Nevertheless, statistically speaking, our presented ranking can be considered sound. It is also important to notice that dendograms do not give information regarding how many clusters should exist unless when the ultrametric tree inequality holds (which is not the case).

The CPU-based energy consumption always represents the majority of the energy consumed. On average, for the compiled languages, this value represents 88.94% of the energy consumed, being the remaining portion assigned to DRAM. This value is very similar for virtual machine (88.94%) and interpreted languages (87.98%). While, as explained in the last point, the overall average consumption for these 3 language types is very different, the ratio between CPU and DRAM based energy consumption seems to generally maintain the same proportion.

This might indicate that optimizing a program to reduce the CPU-based energy consumption will also decrease the DRAM-based energy consumption. However, it is interesting to notice that this value varies more for interpreted languages (min of 81.57%, max of 92.90%) when compared to compiled (min of 85.27%, max of 91.75%) or virtual machine languages (min of 86.10%, max of 92.43%).

With these results, we can try to answer the question raised in **RQ2**: *Is the faster language always the most energy efficient?* By looking solely at the overall results, shown in Table 4, we can see that the top 5 most energy efficient languages keep their rank when they are sorted by execution time and with very small differences in both energy and time values. This does not come as a surprise, since in 9 out of 10 benchmark problems, the fastest and most energy efficient programming language was one of the top 3. Additionally, it is common knowledge that these top 3 language (`C`,`C++`, and `Rust`) are known to be heavily optimized and efficient for execution performance, as our data also shows. Thus, as time influences energy, we had hypothesized that these languages would also produce efficient energy consumptions as they have a large advantage in one of the variables influencing energy, even if they consumed more power on average.

Nevertheless, if we look at the remaining languages in Table 4, we can see that only 4 languages maintain the same energy and time rank (`OCaml`, `Haskel`, `Racket`, and `Python`), while the remainder are completely shuffled. Additionally, looking at individual benchmarks we see many cases where there is a different order for energy and time.

By analyzing the data in Table 5, we can observe that there is always a level of correlation between consumption and execution time. We have statistical support to conclude that the decrease in the execution time of a program of any language will translate with large probability in the reduction of the energy consumption. In the cases of the Swift and TypeScript languages, the correlation values are even equal to 1, which indicates a positive and perfect monotic relationship, leading to the conclusion that these languages will consume less energy whenever the execution time is reduced. However, this direct correlation is not surprising and even expected as we already know that $E_{nergy} = T_{ime} \times P_{ower}$.

While there is indeed this correlation, the weight and size between a change in time and energy is not always proportional as we have previously seen.

Furthermore, the tables in *Results - A. Data Tables* in the appendix also allows us to understand that this question does not have a concrete and ultimate answer. Although the most energy efficient language in each benchmark is almost always the fastest one, the fact is that there is no language which is consistently better than the others. This allows us to conclude that the situation on which a language is going to be used is a core aspect to determine if that language is the most energy efficient option. For example, in the `regex-redux` benchmark, which manipulates strings using regular expressions, interpreted languages seem to be an energy efficient choice (`TypeScript`, `JavaScript` and `PHP`, all interpreted, are in the top 5), although they tend to be not very energy efficient in other scenarios. Thus, the answer for **RQ2** is: No, a faster language is **not always** the most energy efficient.

3.2. Memory Impact on Energy

How does memory usage affect the memory's energy consumption? There are two main possible scenarios which may influence this energy consumption: peak memory usage and total memory usage.

Peak Memory. To answer this question, we calculated for each language the average peak value, considering the solutions each language had. The top 5 languages, also presented in Table 4, with the lowest value were: `Pascal` (66 Mb), `Go` (69 Mb), `C` (77 Mb), `Fortran` (82 Mb), and `C++` (88 Mb); these are all compiled languages. The bottom 5 languages were: `JRuby` (1,309 Mb), `Dart` (570 Mb), `Erlang` (475 Mb), `Lua` (444 Mb), and `Perl` (437 Mb); of these, only `Erlang` is not an interpreted language.

When comparing to the dendograms of the Figures 5, 6, we see several relevant changes relatively to the data presented in Table 4. Although Pascal is in one of the innermost clusters, it is relatively far from the most memory efficient clusters. However, the most notable changes are probably the presence of Haskell and Hack in the outermost clusters. As previously mentioned, the differences of the dendogram results against those presented in Table 4 come down to the fact that in this case, only 4 of the benchmarks were able to be considered. Despite the changes mentioned, the results of the dendograms are inline with what we have shown in Table 4.

On average, the peak memory usage of compiled languages was 125 Mb, or the virtual machine languages was 285 Mb, and for the interpreted was 426 Mb. If sorted by their programming paradigm, the imperative languages had a peak of 116 Mb, the object-oriented 249Mb, the functional 251Mb, and finally the scripting had 421 Mb.

Additionally, the top 5 languages with the least amount of DRAM energy used (on average) were: `C` (5 J), `Rust` (6 J), `C++` (8 J), `Ada` (10 J), and `Java` (11 J); of these, only `Java` is not a compiled language. The bottom 5 languages were: `Lua` (430 J), `JRuby` (383 J), `Python` (356 J), `Perl` (327 J), and `Ruby` (295 J); all are interpreted languages. On average, the compiled languages consumed 14J, the virtual machine languages consumed 52 J, and the interpreted languages consumed 236 J.

Looking at the visual data from Figures 10-12, and the right most figures under *Results - C. Energy and Time Graphs* in the appendix, one can quickly see that there does not seem to be a consistent correlation between the DRAM energy consumption and the peak memory usage. By looking at the results presented in Table 5 (Energy and Memory), we can see that most of the languages obtained non-significant values of Spearman's ρ, meaning it is close to no monotonic relationship. The only exceptions are Python and Ruby, with classification of "Very Strong" but showing values of negative monotonic correlation, pointing to the possibility of high energy consumption being associated with reduced memory usage for these languages.

To further verify this correlation on a global point of view and specifically only on DRAM consumption, we tested both the DRAM energy consumption and peak memory usage for normality using the Shapiro-Wilk [35] test. As the data is not normally distributed, we calculated the Spearman [38] rank-order correlation coefficient. The result was a Spearman ρ value equal to 0.2091, meaning it is between no monotonic relationship ($\rho = 0$) and a weak uphill positive relationship ($\rho = 0.3$).

While we did expect the possibility of little correlation between the DRAM's energy consumption and peak memory usage, we were surprised that the relationship is almost non-existent. Thus, answering the first part of **RQ3**, this indicates that the DRAM's energy consumption has very little to do with how much memory is saved at a given point, but possibly more of how it is used.

Total Memory. Since there was no apparent relation between DRAM's energy consumption and peak memory usage, let us turn our attentions towards the other way of analyzing memory behavior, which is total memory usage.

The average values presented in the Table 6, and most importantly the order in which the languages appear, gives as a clear first impression that the DRAM's energy consumption relates differently with peak memory usage and total memory usage. In the previous section, we saw that the top 5 languages with lowest peak memory usage were `Pascal`, `Go`, `C`, `Fortran`, and `C++`. For total memory usage, the top 5 less consuming languages are `C` (626 Mb), `Rust` (1,087 Mb), `C++` (2,274 Mb), `Ada` (3,020 Mb), and `Pascal` (3,046 Mb). In fact, almost every other language switches places from one ranking to another.

In order to test if there is a correlation between DRAM energy consumption and total memory usage, we repeated the statistical test performed for peak memory usage. Once again, the Shapiro-Wilk test revealed the values were not normally distributed, thus we calculated the Spearman correlation coefficient, which resulted in a ρ value of 0.744, indicating a strong positive relationship. Answering the second part of **RQ3**, we now know that there is a strong uphill relationship between total memory usage and DRAM energy consumption. The more memory is used over a program's lifecyle, the more DRAM energy consumption is spent.

There seems to be in fact a clear relation between the DRAM energy and total memory used, where a lower memory usage value leads to less energy consumed. Since the opposite was observed for peak memory usage (i.e., almost no relation with DRAM energy), these results may indicate that, it might be more energy efficient to store high amounts of memory at once and releasing it right afterwards than total memory usage throughout the execution. Nevertheless, this should be further explored in order to properly understand if this is possible.

3.3. Energy vs. Time vs. Memory

Table 7: Pareto optimal sets for different combination of objectives.

Time & Memory	**Energy & Time**
C • Pascal • Go	C
Rust • C++ • Fortran	Rust
Ada	C++
Java • Chapel • Lisp • Ocaml	Ada
Haskell • C#	Java
Swift • PHP	Pascal • Chapel
F# • Racket • Hack • Python	Lisp • Ocaml • Go
JavaScript • Ruby	Fortran • Haskell • C#
Dart • TypeScript • Erlang	Swift
JRuby • Perl	Dart • F#
Lua	JavaScript
	Racket
	TypeScript • Hack
	PHP
	Erlang
	Lua • JRuby
	Ruby

Energy & Memory	**Energy & Time & Memory**
C • Pascal	C • Pascal • Go
Rust • C++ • Fortran • Go	Rust • C++ • Fortran
Ada	Ada
Java • Chapel • Lisp	Java • Chapel • Lisp • Ocaml
OCaml • Swift • Haskell	Swift • Haskell • C#
C# • PHP	Dart • F# • Racket • Hack • PHP
Dart • F# • Racket • Hack • Python	JavaScript • Ruby • Python
JavaScript • Ruby	TypeScript • Erlang
TypeScript	Lua • JRuby • Perl
Erlang • Lua • Perl	
JRuby	

There are many situations where a software engineer has to choose a particular software language to implement his algorithm according to functional or non functional requirements. For instance, if he is developing software for wearables, it is important to choose a language and apply energy-aware techniques to help save battery. Another example is the implementation of tasks that run in background. In this case, execution time may not be a main concern, and they may take longer than the ones related to the user interaction.

With the fourth research question **RQ4**, we try to understand if it is possible to automatically decide what is the best programming language when considering energy consumption, execution time, and peak memory usage[11] needed by their programs, globally and individually. In other words, if there is a "best" programming languages for all three characteristics, or if not, which are the best in each given scenario.

To this end, we present in Table 7 a comparison of three language characteristics: energy consumption, execution time, and peak memory usage. In order to compare the languages using more than one characteristic at a time we use a multi-objective optimization algorithm to sort these languages, known as Pareto optimization [32, 33]. It is necessary to use such an algorithm because in some cases it may happen that no solution simultaneously optimizes all objectives. For our example, energy, time, and memory are the optimization objectives. In these cases, a dominant solution does not exist, but each solution is a set, in our case, of software languages. Here, the solution is called the Pareto optimal.

In Table 7 we present four multi-objective rankings: time & memory, energy & time, energy & memory, and energy & time, & memory. For each ranking, each line represents a Pareto optimal set, or in other words the Pareto front, that is, a set containing the languages that are equivalent to each other for the underlying objectives. Simply put, each line is a single rank or position.

A single software language in a position signifies that the language was

[11]We only considered peak memory and not total memory, as we wish to focus on optimizing the limited resources a programmer may have: battery, time, and memory space.

clearly the best for the analyzed characteristics. Multiple languages in a line imply that a tie occured, as they are essentially similar; yet ultimately, the languages lean slightly towards one of the objectives over the other as a slight trade-off.

The most common performance characteristics of software languages used to evaluate and choose them are execution time and memory usage. If we consider these two characteristics in our evaluation, `C`, `Pascal`, and `Go` are equivalent. However, if we consider energy and time, `C` is the best solution since it is dominant in both single objectives. If we prefer energy and memory, `C` and `Pascal` constitute the Pareto optimal set. Finally, analyzing all three characteristics, this scenario is very similar as for time and memory.

It is interesting to see that, when considering energy and time, the sets are usually reduced to one element. This means, that it is possible to actually decide which is the best language. This happens possibly because there is a mathematical relation between energy and time and thus they are usually tight together, thus being common that a language is dominant in both objectives at the same time. However, there are cases where this is not true. For instance, for `Pascal` and `Chapel` it is not possible to decide which one is the best as `Pascal` is better in energy and memory use, but worse in execution time. In these situations the developer needs to intervene and decide which is the most important aspect to be able to decide for one language.

It is also interesting to note that, when considering memory use, languages such as `Pascal` tend to go up in the ranking. Although this is natural, it is a difficult analysis to perform without information such as the one we present.

Given the information presented in Table 7 we can try to answer **RQ4: Can we automatically decide what is the best software language considering energy, time, and memory usage?** If the developer is only concerned with execution time and energy consumption, then yes, it is almost always possible to choose the best language. Unfortunately, if memory is also a concern, it is no longer possible to automatically decide for a single language. In all the other rankings most positions are composed by a set of Pareto optimal languages, that is, languages which are equivalent given the underlying characteristics. In these cases, the developer will need to make a decision and take into consideration which are the most important characteristics in each particular scenario, while also considering any fuctional/non-functional requirements necessary for the development of the application. Still, the information we provide in this paper is quite important to help group languages by equivalence when considering the different objectives. For the best of our knowledge, this is the first time such work is presented. Note that we provide the information of each individual characteristic in Table 4 so the developer can actually understand each particular set (we do not show such information in Table 7 to avoid cluttering the paper with to many tables with numbers).

4. Validation on a Chrestomathy Program Repository

The computer language benchmark game was created with the main goal of comparing the execution time of different software languages. Thus, in CLBG, software developers submit solutions that use all advanced mechanisms of the language with the single purpose of implementing a very fast solution (provided that solutions follow a predefined and strict algorithm).

The fastest solution to a problem, however, may not represent the usual programming practices followed by the programmers within the respective languages. For example, the algorithms required by CLBG do not consider lazy evaluation since this evaluation mechanism is only supported by a limited number of languages (which in turn can execute non-lazy code). As a consequence, languages like Haskell and OCaml cannot use lazy evaluation to save work, thus (potentially) providing faster solutions.

In this section, we present a new empirical study to validate our previous ranking that considered only perfomance-oriented programs, which were included in CLBG. Thus, we consider the programming chrestomathy repository *Rosetta Code* [27]. This repository[12] was created to gather solutions to the

[12]`http://www.rosettacode.org/wiki/Rosetta_Code`

same (programming) task in as many different languages as possible. It has a large choice of programming problems across many languages: considering almost 900 tasks throughout of 700 languages! These submissions are completely public and allow a high level of freedom for the solutions.

In a clear distinction when compared to CLBG, the purpose of *Rosetta Code* is to demonstrate similarities and differences among languages, and by doing so to support a programmer with a background in one approach to a problem in learning another. Indeed, if a programmer is trained or has instruction in one programming language or programming approach, by reading comparable solutions to a problem in a different language or using a different programming approach can aid him in understanding such new language or approach.

When compared to CLGB, *Rosetta Code* also does not force any particular algorithm, rule or implementation style for a solution. Actually, the repository makes available multiple solutions to the same problem within the same programming language. Such solutions may use, e.g., different constructions provided by the language: in C++ there are implementations based on *Templates*, and also others using standard C-like solutions. This also happens in object-oriented languages, where in sorting algorithms some solutions use static-arrays and others use collections. This leniency in implementation rules allows for a much better day-to-day programming representation, with more varied algorithmic approaches and solutions to such common problems.

Additionally, while CLBG provides unit tests and their expected output for each of the tasks, *Rosetta Code* does not, often times even only containing programming snippets which are not executable.

In the next section, we describe in detail the study that we have designed and conducted in order to compare and validate the energy efficiency of programming languages, using programs from *Rosetta Code* as our code base.

4.1. Design and Execution

The commendable effort put into the creation and maintenance of *Rosetta Code* has resulted in the compilation of programs written in circa 700 different programming languages, to solve nearly 900 programming tasks.[13]

For comparability, we have restricted our study to the same 27 languages that were represented in the CLBG repository.

In order to decide which tasks to consider in our analysis, we started by sorting all the available tasks by their (decreasing) number of languages for which *Rosetta Code* provides at least one solution. We found 51 tasks with at least 20 implementations in different languages, having preliminarily excluded the remaining ones. We decided not to consider tasks with less than 20 languages as this would hinder the representativeness of the task among languages.

Since we need to be able to compare implementations (regarding, e.g., the energy they use) we then analyzed each of the 51 tasks by hand to choose the ones that implement some kind of algorithm. We ended up choosing tasks such as the computation of the Fibonacci number or the merge sort algorithm and discarded generic tasks such the ones showing how to implement loops or *"hello world"* like programs. From this manual inspection, we marked 7 tasks as interesting to further analyze and 20 other as possibly interesting. The 7 tasks included in the first category implement algorithms that are time and thus (potentially) energy consuming. The 20 tasks in the second category, although implementing some kind of well-known algorithm, tend to be too fast to get interesting energy and time readings. Nevertheless, to have a more representative set of tasks we explored two of these programs, too. The remaining tasks were excluded either because they did not implemented something comparable among languages or because the computations were too trivial. The final set of nine tasks is shown in Table 8.

Although the programming tasks we ended up selecting all had solutions in (at least) 20 different programming languages, still we were not able to consider all such solutions. In some cases, solutions required deprecated libraries, or libraries of which we are unaware of despite our best effort. In other cases, the

[13]Even if, of course, there does not necessarily exist a solution for every of the 700 languages in each of the 900 tasks.

Table 8: Rosetta Code chosen set of programs.

Benchmark	**Description**	**Input**
`MergeSort`	To sort a collection of integers using merge sort	10k random integers
`QuickSort`	To sort a collection of integers using quick sort	10k random integers
`Hailstone`	Generate the hailstone sequence for specific numbers	*Rosetta
`Fibonacci`	Compute Fibonacci number	fib(47)
`Ackermann`	Compute the Ackermann Function	*Rosetta
`N-Queens Problem`	Solve the n-queens puzzle	12-queens
`100-doors`	Solve the 100 doors problem	*Rosetta
`Remove duplicates`	Remove duplicated elements in a sequence	2^{17} random elements
`Sieve of Eratosthenes`	Compute algorithm that finds the prime numbers up to a given integer	10k

implementations were incomplete, did not compile, or had incorrect solutions. The final set of languages to be evaluated for each programming task that we considered is shown in Table 9, along with the running totals for each language and for each task.

When presented with a choice of different implementations for a given language, we chose the algorithm or implementation most similar to all the other remaining implementations for that given task. Additionally, as we also wanted to be as less intrusive as possible, we tried to avoid as much as possible changing any original code. Thus some implementations were discarded because they required a complete rewrite of the code.

For every solution utilizable in each programming language, we then needed to define unit tests, normalize the I/O, and make the implementations executable e.g., by adding a main function.

The units tests needed to be sufficiently complex to significantly exercise the corresponding implementations, but not too much so that they would not terminate, or cause run-time overflows. As shown by our ranking in Section 3,

Table 9: Rosetta Code chosen set of languages for each task.

	MergeSort	QuickSort	Hailstone	Fibonacci	Ackermann	N-Queens	100-doors	Remove-duplicates	Sieve-of-Eratosthenes	Total
(c) Ada			✓	✓	✓		✓		✓	5
(c) C	✓	✓	✓	✓	✓	✓	✓	✓		8
(c) C++			✓	✓	✓		✓	✓	✓	6
(c) Chapel				✓	✓				✓	3
(i) Dart		✓	✓	✓		✓	✓		✓	6
(v) Erlang	✓	✓	✓	✓	✓		✓	✓	✓	8
(c) Fortran			✓	✓		✓	✓			4
(c) Go	✓	✓	✓	✓	✓			✓	✓	7
(c) Haskell	✓	✓	✓	✓	✓	✓			✓	7
(v) Java	✓	✓	✓	✓		✓	✓	✓	✓	8
(i) JavaScript	✓	✓	✓	✓		✓	✓	✓		7
(v) Lisp	✓	✓		✓					✓	4
(i) Lua	✓	✓	✓	✓	✓		✓	✓	✓	8
(c) OCaml	✓	✓	✓	✓	✓		✓		✓	7
(c) Pascal	✓	✓	✓	✓		✓	✓		✓	7
(i) Perl	✓	✓	✓	✓	✓	✓	✓	✓	✓	9
(i) PHP	✓	✓		✓	✓		✓	✓	✓	7
(i) Python	✓	✓	✓	✓		✓	✓	✓		7
(v) Racket	✓	✓	✓	✓	✓			✓	✓	7
(i) Ruby	✓	✓	✓	✓		✓	✓	✓	✓	8
(c) Rust	✓	✓	✓	✓	✓	✓		✓	✓	8
Total	16	17	18	21	13	11	15	13	17	142

we need to be very careful when selecting the inputs as some languages can finish 79.58x slower and eventually requiring limited computing resources. As some solutions had hard-coded executions, and others read from files, we needed to normalize this aspect of execution for all programming languages. The inputs are shown in the top-right column in Table 8. In some cases, the input is stated as *Rosetta, meaning that the input is based on the specific task defined on the *Rosetta Code* site.

We have also confirmed that all solutions for the same task produced the correct output, and those which did not were discarded (this was, for example, the case of the C implementation for the Sieve of Eratosthenes).

The result of our process of selecting and normalizing all implementations is a curated repository. This curated repository, together with scripts to execute each program, are publicly available for others to use[14].

As our focus is currently on the energy efficiency of languages only, we do not measure the memory consumption (peak nor total). Since our measurement infrastructure collects energy measurements at the same time as execution time, both are presented alongside each other in the results. In order to execute this study, we used the same compilers, energy measurement benchmark, and same desktop machine as detailed in Section 2.2.

In the next section, we present and analyze the results we obtained when executing our study.

4.2. Analysis and Discussion

This section presents the results of energy consumption and runtime execution for each of the nine *Rosetta Code* tasks that we selected.

For each task, we include a table ordering the languages by the energy consumption from lowest (more energy efficient) to highest (less energy efficient). We recall that both the energy, presented in Joules, and the execution time, presented in milliseconds, for each task, is the average of ten measurements. Tables 10, 11 and 12 contain such results.

In the remainder of this section, we analyze, one by one and in detail, the results we believe have the most profound impact when compared to our earlier ranking based on the CLBG, and try to understand why such differences occured.

Looking at the results for the sorting algorithms (merge and quicksort, presented in Table 10) we can see that Java is not performing as well as before. In fact, while most imperative implementations use the same array as the data

[14] `https://github.com/greensoftwarelab/RosettaExamples`

structure to store the original and the sorted list of integer numbers (which is obtained by changing elements among positions), the Java implementations in this repository use a more OO-based approach: they use (List) collections, and build new structures which are dynamically populated with sorted elements using *add* methods. This overhead does influence the performance of Java. We can also see surprising differences between different sorting algorithm implementations: both Pascal and PHP solutions are very efficient performing quicksort, which is not replicated by the merge sort implementations. For these two languages, the merge sort implementations use additional temporary arrays for merging.

For the (exponential) Fibonacci problem, whose results are presented in Table 10, and although we were careful defining test cases so that all implementations would execute in a timely manner, there is one language - Python - that could not terminate (within a 24 hour timeout!) for the defined input. While there are small differences between this specific ranking and the overall CLBG one, the four most efficient solutions - Ada, Rust, C, C++ - are the same and do conclude the task very quickly and efficiently.

The results of the four tasks shown in Table 11 also generally follow the CLBG-based ranking. The most energy inefficient languages in our earlier ranking - Ruby, Python, Perl - also appear in the bottom of the individual rankings. This also occurs in the other individual rankings in Tables 10 and 12. C wins in three of these four tasks, and ranks third in the Remove-duplicates task. The Remove-duplicates task, however, does not require the sorting of the resulting elements. Thus, most languages do not sort the result, while the implementations in C, C++, Erlang and Java produce a sorted result. Obviously, this extra work influences both energy and time consumption. In this task, Java is once again penalized by the usage of multiple collections (List and Set!).

For the Sieve of Eratosthenes, the results presented in Table 12 are also aligned with the results obtained with CLBG. A remarkable outlier, however, is observed for the Chapel implementation: although it is very well ranked based on CLBG, it is the most inefficient language for this task! In spite of our best effort in trying to understand this corner case, we believe it deserves a more detailed study of its own, that we leave for future reference, and ideally with the involvement of an expert of Chapel. Naturally, we have confirmed that the algorithm implemented in Chapel is the correct one, and so, the result it produces is also correct.

When globally considering all tasks, we can see that the C programming language is yet again generally the most energy efficient language and also the fastest. As shown in the CLBG-based ranking, the compiled languages are also the best performing ones, whilst the interpreted ones are handicapted by their execution mechanisms. In fact, for most of the tasks, the languages follow the CLBG ranking.

The differences, which are most often seen through a language being placed further down in the ranking, are also very much explainable. These cases are due to specific implementations/languages which are penalized by the usage of poor implemented solutions (used in *Rosetta Code*) and also by chosen algorithm (for example the overhead of sorting the final results) and extra or inefficient data structure (for example the overuse of auxiliary structures).

Table 10: Results for: MergeSort, QuickSort, Hailstone and Fibonacci

MergeSort

	Energy (J)	Time (ms)
(c) C	0.03	6
(c) Rust	0.03	7
(c) Go	0.04	6
(c) OCaml	0.08	9
(v) Lisp	0.26	16
(c) Haskell	0.29	18
(c) Pascal	0.52	46
(i) Ruby	0.63	53
(i) Lua	0.68	54
(i) JavaScript	0.72	61
(i) Perl	0.72	60
(i) Python	1.14	86
(v) Java	1.43	83
(i) PHP	3.03	254
(v) Racket	5.86	392

QuickSort

	Energy (J)	Time (ms)
(c) Pascal	0.02	3
(c) C	0.02	4
(c) Rust	0.03	6
(c) Go	0.05	9
(c) OCaml	0.09	9
(i) PHP	0.23	20
(v) Lisp	0.25	18
(i) Lua	0.26	23
(c) Haskell	0.29	20
(i) Perl	0.32	28
(i) Ruby	0.61	45
(i) Python	0.73	61
(i) JavaScript	0.78	60
(v) Java	1.49	87
(v) Erlang	1.50	101
(i) Dart	1.70	114
(v) Racket	2.24	169

Hailstone

	Energy (J)	Time (ms)
(c) C	0.27	29
(c) Pascal	0.34	16
(c) Ada	0.46	22
(c) Fortran	0.65	34
(c) Go	0.68	31
(c) OCaml	0.68	32
(c) Rust	0.91	39
(c) C++	1.78	78
(i) JavaScript	2.76	119
(v) Java	4.12	160
(v) Racket	6.28	284
(i) Dart	7.08	293
(c) Haskell	7.99	309
(v) Erlang	16.50	696
(i) Ruby	18.35	776
(i) Lua	22.47	987
(i) Python	46.42	1896
(i) Perl	67.27	2771

Fibonacci

	Energy (J)	Time (ms)
(c) Ada	32.12	2477
(c) Rust	61.94	4704
(c) C	70.91	5241
(c) C++	82.78	6136
(c) Chapel	84.34	6126
(c) OCaml	105.22	8555
(v) Java	108.36	8325
(c) Go	150.51	11172
(c) Pascal	152.08	11822
(c) Fortran	185.45	14742
(i) Dart	251.70	18531
(v) Racket	275.15	21627
(i) JavaScript	311.22	22352
(v) Lisp	497.41	40442
(v) Erlang	799.22	62462
(c) Haskell	2068.83	143187
(i) PHP	2360.63	171315
(i) Lua	3816.43	290291
(i) Ruby	9657.99	663070
(i) Perl	14380.33	1010606
(i) Python	∞	∞

Table 11: Results for: Ackermann, N-queens, 100-doors and Remove-duplicates

Ackermann		
	Energy (J)	Time (ms)
(c) C	0.00	1
(c) Chapel	0.01	2
(c) Ada	17.79	1349
(c) OCaml	26.37	1922
(c) Rust	41.69	1726
(c) C++	58.18	1345
(c) Haskell	178.49	9036
(v) Racket	237.24	15421
(c) Go	275.78	21421
(v) Erlang	290.60	21110
(i) PHP	3215.77	207815
(i) Lua	5316.18	180432
(i) Perl	14475.82	1008595

N-queens		
	Energy (J)	Time (ms)
(c) C	0.05	4
(c) Fortran	0.10	10
(c) Pascal	0.56	53
(v) Java	2.15	167
(i) Dart	2.90	224
(c) Haskell	2.90	204
(c) Rust	4.53	384
(i) JavaScript	8.22	574
(i) Ruby	19.18	1412
(i) Perl	41.23	2881
(i) Python	111.37	7401

100-doors		
	Energy (J)	Time (ms)
(c) C	0.01	1
(c) C++	0.09	10
(c) Fortran	0.35	34
(c) OCaml	0.85	69
(v) Java	1.14	67
(i) Dart	1.20	83
(i) JavaScript	1.59	107
(i) PHP	6.95	422
(c) Pascal	7.39	530
(c) Ada	13.88	1193
(i) Perl	17.54	1217
(i) Ruby	33.02	2513
(v) Erlang	35.42	2267
(i) Python	63.58	4445
(i) Lua	108.23	9188

Remove-duplicates		
	Energy (J)	Time (ms)
(c) Rust	0.01	1
(c) C++	0.12	5
(c) C	0.14	10
(c) Go	0.32	13
(i) Lua	0.51	21
(i) Perl	1.31	53
(i) JavaScript	1.73	73
(v) Erlang	2.36	96
(v) Java	2.96	214
(i) PHP	2.99	121
(i) Python	4.93	206
(i) Ruby	6.13	259
(v) Racket	7.54	318

Table 12: Results for: Sieve of Eratosthenes

Sieve of Eratosthenes		
	Energy (J)	Time (ms)
(c) Pascal	0.02	3
(c) C++	0.03	3
(c) Rust	0.03	4
(c) OCaml	0.05	7
(c) Ada	0.06	8
(c) Haskell	0.10	13
(c) Go	0.11	10
(v) Lisp	0.15	11
(i) PHP	0.30	22
(i) Ruby	0.51	42
(i) Perl	0.64	49
(i) Lua	0.69	37
(v) Java	1.64	89
(v) Racket	1.97	148
(i) Dart	1.98	133
(v) Erlang	2.36	162
(c) Chapel	2280.27	174549

Having produced individual energy-sorted rankings for each of the 9 tasks we considered, we now wish to produce an overall language ranking so that we can compare the ranks of languages in a performance-tailored program corpus (the

CLBG) to one more oriented to program comprehension (the *Rosetta Code*). To produce such overall ranking we use the Schulze method [39] to agregate the results of the individual rankings in Tables 10, 11 and 12 into a combined one. We needed to use a different method to produce this ranking, compared to the CLBG one, because the range of values is very large and the number of implementation differ much more between tasks. Table 13 shows the *Rosetta Code* overall ranking that we obtained.

Table 13: Rosetta Code global ranking based on Energy

Rosetta Code Global Ranking	
Position	Language
1	C
2	Pascal
3	Ada
4	Rust
5	C++, Fortran
6	Chapel
7	OCaml, Go
8	Lisp
9	Haskell, JavaScript
10	Java
11	PHP
12	Lua, Ruby
13	Perl
14	Dart, Racket, Erlang
15	Python

This *Rosetta Code* based ranking is similar to the our earlier CLBG ranking. The top six languages in CLBG continue to be in the top five this new ranking, with the exception of Java. As we discussed before the *Rosetta Code* implementations in Java rely on the widely used Java Collection Framework, which require more work when compared to imperative-based solutions that use static arrays. We can also see that the Chapel language also dropped in our *Rosetta Code* based ranking.

These results also show that interpreted languages like PHP, Lua, Ruby, Perl, Python continue at the bottom being the least energy efficient software languages.

4.3. Conclusions

As expected, the fact that one specific solution uses a different, more efficient approach to solve a task did influence the results of the *Rosetta Code* study and the ranking of the different languages. This occurs in two situations: i) the requirements for a task on *Rosetta Code* are not completely defined; thus, there are solutions that perform work that is not specified (for example, sorting the list after removing duplicates); and ii) some solutions that use additional temporary data structure which also force additional computational work to be performed.

In this new study to validate our previous work, we use the most natural and understandable solutions available in *Rosetta Code*, and we did not change the program's repository: as discussed in previous sections only strictly necessary editions (such as adding test cases or main functions) were performed on programs. If we were forcing the different implementations for a task to perform exactly the same algorithm, we were essentially re-doing the CLBG-based study. This could easily be done, for example in the sorting tasks, by adding a solution in C that sort dynamically linked lists, instead of sorting the original array, but diverges from our intentions here.

In regards to **RQ5: How do the results of our energy consumption analysis of programming languages gathered from rigorous performance benchmarking solutions compare to results of average day-to-day solutions?**, we have seen that the results of our validation study (*Rosetta Code*) against our original study (CLBG) are very much comparable. As expected, the results show many similarities, even though one repository source is to help learn and comprehend programs in various languages and the other is tailored to analyze the performance of languages and the other. Carefully analyzing the cases where the rankings would differ, we were able to quickly localize and explain such occurrences down to improper or inefficient use of the programming language.

If wanting to understand how programming languages compare in terms of energy efficiency, and are written by general everyday programmers (through very simple and direct solutions), Table 13 shows such results. On the other hand, Table 4 details the results of the potential each language has on reducing their energy consumption if programmed by more experienced programmers or those taking into consideration basic algorithmic optimizations. With this validation study, we have shown that our results based off the rigorous benchmarking of highly optimized programs from CLBG (presented in Section 3) are representative and detail a good overall look at the energy efficiency of programming languages written by either expert or non-expert programmers.

5. Threats to Validity

The goal of our study was to both measure and understand the energetic behavior of several programming languages, allowing us to bring about a greater insight on how certain languages compare to each other mainly in terms of energy consumption, but also performance and memory. We present in this section some possible threats to the validity of our study and in what ways we have tried to minimize their effects, divided into four categories [40, 41], namely: conclusion validity, internal validity, construct validity, and external validity.

Conclusion Validity. This first category describes threats which may influence our capacity to draw correct conclusions [41].

Fishing is a possible threat as one may be searching for particular results, thus making the analysis not independent [41]. In the case of our study, we are not evaluating a programming language(PL) that we may have proposed and hence have no particular interest in the outcome. Thus, we are not searching for a particular result, and as such, this threat does not apply to our study.

A common threat is the *reliability of measures.* In our case, when measuring the energy consumption of the various different programming languages, other factors alongside the different implementations and actual languages themselves may contribute to variations, i.e. specific versions of an interpreter or virtual machine. To avoid this, we executed every language and benchmark solution equally. In each, we measured the energy consumption (CPU and DRAM), execution time, and peak and total memory 10 times, removed the lowest and highest 20% outliers, and calculated the median, mean, standard deviation, min, and max values. This allowed us to minimize the particular states of the tested machine, including uncontrollable system processes and software. However, the measured results are quite consistent, and thus reliable. In addition, the used energy measurement tool has also been proven to be very accurate.

Another common threat is the *reliability of treatment implementation.* The implementations used to evaluate the PLs were produced by external developers. We simply reused the settings from CLBG which were also applied to *Rosetta* tasks. Thus, these implementations are independent from this study and are the best available as the CLBG is a running contest of the performance of.

Regarding *random heterogeneity of subjects*, we used all the available languages in the CLBG, that is, 27 different PLs. Although there are hundreds of languages, this set includes many popular languages and also more academic ones, thus covering a vast set of PLs. In fact, several communities from news pages, to social-media, to Reddit have found our work broad enough to be interesting.

Internal Validity. This category concerns itself with what factors may interfere with the results of our study, that is, that may influence the relationship between the treatment and the outcome [41].

Instrumentation is one of the possible causes of internal validity [41]. This refers to the artifacts used during the experiment. In our case, we used scripts to collect the energy, time and memory used during the execution of the programs. However, these are simple scripts used to call RAPL for measurement during the execution of programs. They were previously validated and tested [23, 24] and are also publicly available in the paper's online appendix.

Construct Validity. This category concerns the generalization of the results to the concept or theory behind the experiment [41].

Inadequate preoperational explication of constructs is a possible issue related to the constructs not being well defined prior to being measured [41]. In our case we evaluated the energy, time and memory used by programs, and thus the measurements were obvious, making this issue minor or nonexistent in our study.

Another possible issue is the *mono-operation bias* concerned with the under-representation of a construct. We have used about 10 programs to evaluate each PL. These programs were proposed by others to evaluate the performance of PLs and thus were designed to stress the languages within the context of a contest. Thus, they seem to represent an interesting way of evaluating the PLs.

Regarding the *mono-method bias*, we have indeed used just a single tool to measure energy and time (RAPL), and another tool for memory (the Unix-based `time` tool). However, both known to be very precise for measuring energy, time, and memory, thus their results are reliable.

The *interaction of different treatments* is also a possible issue. However, we have used different and independent programs to evaluate the languages. Between each measuring execution (as common practice in measuring energy consumption), there was a two minute idle time rest to allow the system to cool-down, as to reduce over heating (which may affect energy measurements), and to allow the system to treat garbage collecting.

External Validity. This type of threat is concerned with the generalization of the results to an industrial setting [41].

A common threat is termed *interaction of selection and treatment* meaning the population chosen is not representative, in our case the PLs [41]. In the first study we analyzed 27 different programming languages. These PLs include popular languages among industry such as C/C++, Java, C#, JavaScript, Ruby, PHP or Python[15]. Thus, our study applies also to an industrial setting, at least regarding the PLs used.

Another external threat is the *interaction of setting and treatment*, that is, the experimental setting might not represent the industrial setting. Each PL

[15]See `http://pypl.github.io/PYPL.html` for a list of popular languages based on Google searches.

was evaluated with roughly 10 solutions to the proposed problems, totaling out to almost 270 different cases. The implementation solutions we measured were developed by external experts in each of the programming languages, with the main goal of "winning" by producing the best solution for performance time. While the different languages contain different implementations, they were written under the same rules, all produced the same exact output, and were implemented to be the fastest and most efficient as possible. Having these different yet efficient solutions for the same scenarios allows us to compare the different programming languages in a quite just manner as they were all placed against the same problems. Moreover, the compilers and computers used are recent and thus in line with nowadays industry. For the *Rosetta* the solutions are not so curated. In any case, the authors have reviewed and used solutions that were correct thus solving the underlying problem.

While our benchmarking system is server based, studies have shown that there is no statistical difference between server platforms and embedded systems in regards to energy based readings [26], thus the results can be generalized directly to embedded systems. In regards to the generalization to mobile systems, this can not be completely assured as results differ slightly between server/embedded based systems and mobile, and sometimes even between independent studies in mobile, if analyzing on a small scale. Overall however, the results seem to maintain their tendencies [17, 4]

In general, in this category of threats it is paramount to report the characteristics of the experiment in order to understand its applicability to other contexts [41].The actual approach and methodology we used also favors easy replications. This can be attributed to the CLBG containing most of the important information needed to run the experiments, these being: the source code, compiler version, and compilation/execution options. Moreover, all the material used and produced is publicly available at `https://sites.google.com/view/energy-efficiency-languages`. Thus we believe these results can be further generalized, and other researchers and industry can replicate our methodology for future work.

6. Related Work

The work presented in this paper extends previous work in [23] and [24]. In this extended version, an analysis on total memory usage was performed to better understand the relationship between continuous memory usage and DRAM energy consumption. Additionally, we replicated our study on a different repository, the *Rosetta Code* chrestomathy repository. This not only allowed us to validate our previous programming language energy ranking using the CLBG, but also to understand how different are the results of programs on a repository for performance based benchmarking and a repository for learning and comprehensibility.

The CLBG benchmark solutions have already been used for validation purpose by several research works. Among other examples, CLGB was used to study dynamic behavior of non-Java JVM languages [42], to analyze dynamic scripting languages [43] and compiler optimizations [44], or even to benchmark a JIT compiler for PHP [45]. At the best of our knowledge, CLGB was only used once for energy consumption analysis. In [17], the authors used the provided `Haskell` implementations, among other benchmarks, to analyze the energy efficiency of `Haskell` programs from strictness and concurrency perspectives, while also analyzing the energy influence of small implementation changes.

A similar study using the *Rosetta Code* repository was performed [26], where the authors looked at the energy-delay implications on 14 programming languages, on three different computing platforms (embedded, laptop, and server. They too produced very similar results to ours across the three platforms, and found that there is no statistical differences between server platforms and embedded systems. Thus, our server based benchmarking system can easily be generalized for embedded devices. They further explored [46] the energy-delay implications within inter-process communication systems and observed how energy consumption and run-time performance can very significantly across different programming language implementations.

While several works have shown indications that a more time efficient approach does not always lead to the most energy efficient solution [17, 15, 16, 18, 2, 4], these results were not the intended focus nor main contribution, but more of a side observation per se. We focused on trying to understand and directly answer this question of how energy efficiency and time relate.

Nevertheless, the energy efficiency in software problem has been growing in interest in the past few years. In fact, studies have emerged with different goals and in different areas, with the common vision of understanding how development aspects affect the energy consumption in diversified software systems. For instance, for mobile applications, there are works focused on analyzing the energy efficiency of code blocks [47, 48, 49, 50], or just monitoring how energy consumption evolves over time [51]. Other studies aimed at a more extensive energy consumption analysis, by comparing the energy efficiency of similar programs in specific usage scenarios [13, 11], or by providing conclusions on the energy impact of different implementation decisions [52]. Several other works have shown that several factors, such as different design patterns [6, 7], Android keyboards [53] and energy footprints [54], coding practices [15, 55, 17, 56, 8, 9, 49], and data structures [2, 31, 3, 19, 57, 58], actually have a significant influence in the software's energy efficiency.

In the context of Android, a particular line of work relates to ours. The authors of [59, 4] also used CLBG and the *Rosetta Code* repository to compare `JavaScript`, `Java`, and `C/C++` in a setting specifically targeted for Android systems. Using programs from these sources they concluded that most of the times `JavaScript` consumes less energy, but there is no overall winner. Moreover, they also advocate that faster programs are not always the ones that consume less energy.

7. Conclusions

In this paper, we present an extended work of a series of systematic comparisons and rankings over the energy efficiency of 27 well-known software languages. These comparisons take as their original code base programs from a popular programming language benchmarking competition, *The Computer Language Benchmarks Game* (CLBG), where experts in different languages compete to produce the most performance efficient solution.

We were able to show which were the most energy efficient software languages, execution types, and paradigms across 10 different benchmark problems. We were also able to relate execution time and memory consumption to energy consumption to understand not only how memory usage affects energy consumption, but also how time and energy relate. This allowed us to understand if a faster language is always the most energy efficient. As we saw, this is not always the case. Additionally, we have created rankings, group clusterings, and correlation tables of programming languages based on their energy consumption, execution time, and memory usage. In addition, we further analyzed the correlation between energy consumption and memory usage.

As often times developers have limited resources and may be concerned with more than one objective, or efficiency characteristic, we established rankings of the best/worst languages according to a combination of different objectives: limited battery, limited time, and limited memory capacity.

In order to properly assess our original findings, we revisited this study and presented a new empirical study based on a chrestomathy repository, *Rosetta Code* [27] in order to validate our original study. This allows us to also understand if the analyzed performance-oriented solutions, of which we based our results on, are representative of day-to-day programming and by non-expert programmers.

This new validation considered 9 tasks from *Rosetta Code*, and their solutions in the programming languages that we have previously considered. These results showed many similarities when compared to the strict and rigorous performance-oriented benchmarks used to produce our programming language rankings. Additionally, albeit very few (and highly explainable) differences, we have concluded the originally presented rankings and results are representative of both expert and non-expert programmers.

Our work helps contribute another stepping stone in bringing more information to developers to allow them to become more energy-aware when programming.

Acknowledgments

We would like to thank Luís Cruz (University of Porto) for the help that he provided. This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within projects: UID/EEA/50014/2013 (POCI-01-0145-FEDER-016718) and PTDC/CCI-INF/29583/2017 (POCI-01-0145-FEDER-029583). The second, third, and fourth author are also sponsored by FCT grant SFRH/BD/132485/2017, SFRH/BD/144938/2019, and SFRH/BD/146624/2019 respectively. This work is also supported by operation Centro-01-0145-FEDER-000019 — C4 — Centro de Competências em Cloud Computing, cofinanced by the European Regional Development Fund (ERDF) through the Programa Operacional Regional do Centro (Centro 2020), in the scope of the Sistema de Apoio à Investigação Científica e Tecnológica - Programas Integrados de IC&DT, and the first author was financed by post-doc grant reference C4_SMDS_L1-1_D. Additionally, this paper acknowledges the support of the Erasmus+ Key Action 2 (Strategic partnership for higher education) project No. 2020-1-PT01-KA203-078646: "SusTrainable - Promoting Sustainability as a Fundamental Driver in Software Development Training and Education". The information and views set out in this paper are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

References

[1] G. Pinto, F. Castor, Y. D. Liu, Mining questions about software energy consumption, in: Proceedings of the 11th Working Conference on Mining Software Repositories, ACM, 2014, pp. 22–31.

[2] R. Pereira, M. Couto, J. Saraiva, J. Cunha, J. P. Fernandes, The Influence of the Java Collection Framework on Overall Energy Consumption, in: Proceedings of the 5th International Workshop on Green and Sustainable Software, GREENS, ACM, 2016, pp. 15–21.

[3] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, A. Hindle, Energy profiles of java collections classes, in: Proceedings of the 38th International Conference on Software Engineering, ICSE, ACM, 2016, pp. 225–236.

[4] W. Oliveira, R. Oliveira, F. Castor, A study on the energy consumption of android app development approaches, in: Proceedings of the 14th International Conference on Mining Software Repositories, MSR, IEEE Press, 2017, pp. 42–52.

[5] D. Li, W. G. J. Halfond, An investigation into energy-saving programming practices for android smartphone app development, in: Proceedings of the 3rd International Workshop on Green and Sustainable Software, GREENS, 2014.

[6] C. Sahin, F. Cayci, I. L. M. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, in: Proceedings of 4th International Workshop on Green and Sustainable Software, GREENS, IEEE, 2012, pp. 55–61.

[7] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, D. Poshyvanyk, Mining energy-greedy api usage patterns in android apps: an empirical study, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR, 2014, pp. 2–11.

[8] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, in: Proceedings of 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, 2014, p. 36.

[9] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, J. Saraiva, Helping programmers improve the energy efficiency of source code, in: Proceedings of the 39th International Conference on Software Eng. Companion, ICSE-C, ACM, 2017.

[10] S. A. Chowdhury, A. Hindle, Greenoracle: estimating software energy consumption with energy measurement corpora, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR, 2016, pp. 49–60.

[11] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, P. Ammann, Ecodroid: An approach for energy-based ranking of android apps, in: Proceedings of 4th International Workshop on Green and Sustainable Software, GREENS '15, IEEE Press, 2015, pp. 8–14.

[12] S. Hao, D. Li, W. G. J. Halfond, R. Govindan, Estimating mobile application energy consumption using program analysis, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, 2013, pp. 92–101.

[13] M. Couto, P. Borba, J. Cunha, J. P. Fernandes, R. Pereira, J. Saraiva, 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, 2017, pp. 84–93.

[14] T. Yuki, S. Rajopadhye, Folklore confirmed: Compiling for speed= compiling for energy, in: Languages and Compilers for Parallel Computing, Springer, 2014, pp. 169–184.

[15] G. Pinto, F. Castor, Y. D. Liu, Understanding energy behaviors of thread management constructs, in: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA, 2014, pp. 345–360.

[16] A. E. Trefethen, J. Thiyagalingam, Energy-aware software: Challenges, opportunities and strategies, Journal of Computational Science 4 (6) (2013) 444 – 449.

[17] L. G. Lima, G. Melfe, F. Soares-Neto, P. Lieuthier, J. P. Fernandes, F. Castor, Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language, in: Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER, IEEE, 2016, pp. 517–528.

[18] S. Abdulsalam, Z. Zong, Q. Gu, M. Qiu, Using the greenup, powerup, and speedup metrics to evaluate software energy efficiency, in: Proceedings of the 6th International Green and Sustainable Computing Conference, IGCC, IEEE, 2015, pp. 1–8.

[19] I. Manotas, L. Pollock, J. Clause, Seeds: A software engineer's energy-optimization decision support framework, in: Proceedings of the 36th International Conference on Software Engineering, ACM, 2014, pp. 503–514.

[20] C. Pang, A. Hindle, B. Adams, A. E. Hassan, What do programmers know about software energy consumption?, IEEE Software 33 (3) (2015) 83–89.

[21] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, J. Clause, An empirical study of practitioners' perspectives on green software engineering, in: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), IEEE, 2016, pp. 237–248.

[22] G. Pinto, F. Castor, Energy efficiency: a new concern for application software developers, Communications of the ACM 60 (12) (2017) 68–75.

[23] M. Couto, R. Pereira, F. Ribeiro, R. Rua, J. Saraiva, Towards a green ranking for programming languages, in: Proceedings of the 21st Brazilian Symposium on Programming Languages, SBLP, 2017, pp. 7:1–7:8, (best paper award).

[24] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, J. 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, SLE, ACM, 2017, pp. 256–267.

[25] I. Gouy, The Computer Language Benchmarks Game.
URL `http://benchmarksgame.alioth.debian.org/`

[26] S. Georgiou, M. Kechagia, P. Louridas, D. Spinellis, What are your programming language's energy-delay implications?, in: Proceedings of the 15th International Conference on Mining Software Repositories, MSR, ACM, 2018, pp. 303–313.

[27] M. Mol, Rosetta Code.
URL `http://rosettacode.org/`

[28] M. Dimitrov, C. Strickland, S.-W. Kim, K. Kumar, K. Doshi, Intel® power governor, `https://software.intel.com/en-us/articles/intel-power-governor`, accessed: 2015-10-12 (2015).

[29] M. Hähnel, B. Döbel, M. Völp, H. Härtig, Measuring energy consumption for short code paths using RAPL, SIGMETRICS Performance Evaluation Review 40 (3) (2012) 13–17.

[30] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, D. Rajwan, Power-management architecture of the intel microarchitecture code-named sandy bridge, IEEE Micro 32 (2) (2012) 20–27.

[31] K. Liu, G. Pinto, Y. D. Liu, Data-oriented characterization of application-level energy optimization, in: Fundamental Approaches to Software Engineering, Springer, 2015, pp. 316–331.

[32] K. Deb, M. Mohan, S. Mishra, Evaluating the ε-domination based multiobjective evolutionary algorithm for a quick computation of pareto-optimal solutions., Evolutionary Computation Journal 13 (4) (2005) 501–525.

[33] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, Trans. Evol. Comp 6 (2) (2002) 182–197.

[34] M. Woodruff, J. Herman, pareto.py: a $\varepsilon - nondomination$ sorting routine, https://github.com/matthewjwoodruff/pareto.py (2013).

[35] S. S. Shapiro, M. B. Wilk, An analysis of variance test for normality (complete samples)†, Biometrika 52 (3-4) (1965) 591–611. arXiv:https://academic.oup.com/biomet/article-pdf/52/3-4/591/962907/52-3-4-591.pdf, doi:10.1093/biomet/52.3-4.591.
URL https://doi.org/10.1093/biomet/52.3-4.591

[36] T. W. Anderson, D. A. Darling, Asymptotic theory of certain goodness of fit criteria based on stochastic processes, Ann. Math. Statist. 23 (2) (1952) 193–212. doi:10.1214/aoms/1177729437.
URL https://doi.org/10.1214/aoms/1177729437

[37] L. Rea, Ñames, R. A. Parker, R. Allen, Designing and conducting survey research, Jossey-Bass Publishers, 2016.

[38] C. Spearman, The proof and measurement of association between two things, The American Journal of Psychology 15 (1) (1904) 72–101.
URL http://www.jstor.org/stable/1412159

[39] M. Schulze, A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method, Social Choice and Welfare 36 (2) (2011) 267–303.

[40] T. D. Cook, D. T. Campbell, Quasi-experimentation: design & analysis issues for field settings, Houghton Mifflin, 1979.

[41] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Computer Science, Springer, 2012.

[42] W. H. Li, D. R. White, J. Singer, Jvm-hosted languages: They talk the talk, but do they walk the walk?, in: Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, PPPJ, ACM, 2013, pp. 101–112.

[43] K. Williams, J. McCandless, D. Gregg, Dynamic interpretation for dynamic scripting languages, in: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO, ACM, 2010, pp. 278–287.

[44] V. St-Amour, S. Tobin-Hochstadt, M. Felleisen, Optimization coaching: Optimizers learn to communicate with programmers, in: Proceedings of ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA, ACM, 2012, pp. 163–178.

[45] A. Homescu, A. Şuhan, Happyjit: A tracing jit compiler for php, SIGPLAN Not. 47 (2) (2011) 25–36.

[46] S. Georgiou, D. Spinellis, Energy-delay investigation of remote inter-process communication technologies, Journal of Systems and Software 162 (2020) 110506.

[47] M. Couto, T. Carção, J. Cunha, J. P. Fernandes, J. Saraiva, Detecting anomalous energy consumption in android applications, in: F. M. Quintão Pereira (Ed.), Programming Languages: Proceedings of 18th Brazilian Symposium on Programming Languages, SBLP, 2014, pp. 77–91.

[48] D. Li, S. Hao, W. G. Halfond, R. Govindan, Calculating source line level energy information for android applications, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ACM, 2013, pp. 78–89.

[49] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, J. Saraiva, Spelling out energy leaks: Aiding developers locate energy inefficient code, Journal of Systems and Software 161 (2020) 110463.

[50] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, J. Saraiva, Helping programmers improve the energy efficiency of source code, in: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2017, pp. 238–240.

[51] F. Ding, F. Xia, W. Zhang, X. Zhao, C. Ma, Monitoring energy consumption of smartphones, in: Proceedings of the 2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing, 2011, pp. 610–613.

[52] L. Cruz, R. Abreu, Performance-based guidelines for energy efficient mobile applications, in: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, MOBILESoft, IEEE Press, 2017, pp. 46–57.

[53] R. Rua, T. Fraga, M. Couto, J. Saraiva, Greenspecting android virtual keyboards, in: Proceedings of the 7th International Conference on Mobile Software Engineering and Systems, MOBILESoft, 2020.

[54] L. Cruz, R. Abreu, On the energy footprint of mobile testing frameworks, IEEE Transactions on Software Engineering.

[55] M. Couto, J. Saraiva, J. P. Fernandes, Energy refactorings for android in the large and in the wild, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2020, pp. 217–228.

[56] L. G. Lima, F. Soares-Neto, P. Lieuthier, F. Castor, G. Melfe, J. P. Fernandes, On haskell and energy efficiency, Journal of Systems and Software 149 (2019) 554–580.

[57] W. de Oliveira Júnior, R. O. dos Santos, F. J. C. de Lima Filho, B. F. de Araújo Neto, G. H. L. Pinto, Recommending energy-efficient java collections, in: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, 2019, pp. 160–170.

[58] L. Cruz, R. Abreu, Catalog of energy patterns for mobile applications, Empirical Software Engineering 24 (4) (2019) 2209–2235.

[59] W. Oliveira, W. Torres, F. Castor, B. H. Ximenes, Native or web? a preliminary study on the energy consumption of android development models, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Vol. 1, 2016, pp. 589–593. `doi:10.1109/SANER.2016.93`.