Energy Efficiency across Programming Languages

How Do Energy, Time, and Memory Relate?

Rui Pereira
HASLab/INESC TEC
Universidade do Minho, Portugal
ruipereira@di.uminho.pt

Marco Couto
HASLab/INESC TEC
Universidade do Minho, Portugal
marco.l.couto@inesctec.pt

Francisco Ribeiro, Rui Rua
HASLab/INESC TEC
Universidade do Minho, Portugal
fribeiro@di.uminho.pt
rrua@di.uminho.pt

Jácome Cunha
NOVA LINCS, DI, FCT
Univ. Nova de Lisboa, Portugal
jacome@fct.unl.pt

João Paulo Fernandes
Release/LISP, CISUC
Universidade de Coimbra, Portugal
jpf@dei.uc.pt

João Saraiva
HASLab/INESC TEC
Universidade do Minho, Portugal
saraiva@di.uminho.pt

Abstract

This paper presents a study of the runtime, memory usage and energy consumption of twenty seven well-known software languages. We monitor the performance of such languages using ten different programming problems, expressed in each of the languages. Our results show interesting findings, such as, slower/faster languages consuming less/more energy, and how memory usage influences energy consumption. We show how to use our results to provide software engineers support to decide which language to use when energy efficiency is a concern.

CCS Concepts • **Software and its engineering → Software performance**; **General programming languages**;

Keywords Energy Efficiency, Programming Languages, Language Benchmarking, Green Software

ACM Reference Format:
Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy Efficiency across Programming Languages: How Do Energy, Time, and Memory Relate?. In *Proceedings of 2017 ACM SIGPLAN International Conference on Software Language Engineering (SLE'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3136014.3136031

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 execute 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. In fact, in the last century *performance* in software languages was in almost all cases synonymous of *fast execution time* (embedded systems were probably the single exception).

In this century, 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, it is usual to see mobile phone users (which are powerful computers) avoiding 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 [28]. 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 [15, 27] and android language [25], the energy impact of different programming practices both in mobile [18, 22, 31] and desktop applications [26, 32], the energy efficiency of applications within the same scope [2, 17], or even on how to predict energy consumption in several software systems [4, 14], 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. 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 hardly 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

SLE'17, October 23–24, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5525-4/17/10…$15.00
https://doi.org/10.1145/3136014.3136031

this subject [1, 21, 27, 29, 35, 38]. A similar question arises when comparing software languages: *is a faster language, a greener 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.

In this paper we analyze the performance of twenty seven software languages. We consider ten different programming problems that are expressed in each of the languages, following exactly the same algorithm, as defined in the *Computer Language Benchmark Game* (CLBG) [12]. We compile/execute such programs using the state-of-the-art compilers, virtual machines, interpreters, and libraries for each of the 27 languages. Afterwards, we analyze the performance of the different implementation considering three variables: execution time, memory consumption and energy consumption. Moreover, we analyze 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 variable considered. Our results show interesting findings, such as, slower/-faster software languages consuming less/more energy, and how memory usage influences energy consumption. Moreover, we discuss how to use such results to provide software engineers support to decide which language to use when energy efficiency is a concern.

This work builds on previous work [6] which presents a framework to allow the monitoring of the energy consumption of executable software programs. In that work, the C-based framework was used to define a preliminary ranking of ten languages (where only energy was considered). We reuse the energy monitoring framework (briefly described in Section 2.2) to analyze the energy efficiency of 27 languages and (almost) 270 programs. We have also extended it in order to monitor memory consumption, as well.

This paper is organized as follows: Section 2 exposes the detailed steps of our methodology to measure and compare energy efficiency in software languages, followed by a presentation of the results. Section 3 contains the analysis and discussion on the obtained results, where we first analyze whether execution time performance implies energy efficiency, then we examine the relation between peak memory usage and memory energy consumption, and finally we present a discussion on how energy, time and memory relate in the 27 software languages. In Section 4 we discuss the threats to the validity of our study. Section 5 presents the related work, and finally, in Section 6 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.

With this in mind, we begin by trying to answer the following research question:

- **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.

The following subsections will detail the methodology used to answer this question, and the results we obtained.

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 [12]. (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 [6, 21, 25].

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 compiler for that language is proprietary. 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

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

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;

(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 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 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) tool [10], which is capable of providing accurate energy estimates at a very fine-grained level, as it has already been proven [13, 30]. Also, the current version of RAPL allows it to be invoked from any program written in *C* and `Java` (through jRAPL [23]).

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 [1]) is described in Listing 1.

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

[1]The measuring framework and the complete set of results are publicly available at https://sites.google.com/view/energy-efficiency-languages

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 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.

Each benchmark solution was executed and measured 10 times, in order to obtain 10 energy consumption and execution time samples. We did so to reduce the impact of cold starts and cache effects, and to be able to analyze the measurements' consistency and avoid outliers. We followed the same approach when gathering results for memory usage.

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%).[2]

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 [2]. 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: 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 the remainder shown in the online appendix for this paper [1]. 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 same example benchmark, `Rust`, while the most energy efficient, would drop 9 positions if ordered by peak memory usage.

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`.

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 1-3 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. 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 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.

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

The second set, Figures 4-6 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 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.

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 peak memory usage which allowed us to analyze how these two relate with energy consumption. The analysis and discussion of our results is shown in the next section.

3 Analysis and Discussion

In this section 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.

- **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 memory usage 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 a given scenarios.

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 [38], and the opposite was also observed [21, 29, 35].

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 [1] and [27], 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 1 to 3 (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

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

binary-trees	Energy	Time	Ratio	Mb
(c) C	39.80	1125	0.035	131
(c) C++	41.23	1129	0.037	132
(c) Rust ⇓$_2$	49.07	1263	0.039	180
(c) Fortran ⇑$_1$	69.82	2112	0.033	133
(c) Ada ⇓$_1$	95.02	2822	0.034	197
(c) Ocaml ↓$_1$ ⇑$_2$	100.74	3525	0.029	148
(v) Java ↑$_1$ ⇓$_{16}$	111.84	3306	0.034	1120
(v) Lisp ↓$_3$ ⇓$_3$	149.55	10570	0.014	373
(v) Racket ↓$_4$ ⇓$_6$	155.81	11261	0.014	467
(i) Hack ↑$_2$ ⇓$_9$	156.71	4497	0.035	502
(v) C# ↓$_1$ ⇓$_1$	189.74	10797	0.018	427
(v) F# ↓$_3$ ⇓$_1$	207.13	15637	0.013	432
(c) Pascal ↓$_3$ ⇑$_5$	214.64	16079	0.013	256
(c) Chapel ↑$_5$ ⇑$_4$	237.29	7265	0.033	335
(v) Erlang ↑$_5$ ⇑$_1$	266.14	7327	0.036	433
(c) Haskell ↑$_2$ ⇓$_2$	270.15	11582	0.023	494
(i) Dart ↓$_1$ ⇑$_1$	290.27	17197	0.017	475
(i) JavaScript ↓$_2$ ⇓$_4$	312.14	21349	0.015	916
(i) TypeScript ↓$_2$ ⇓$_2$	315.10	21686	0.015	915
(c) Go ↑$_3$ ⇑$_{13}$	636.71	16292	0.039	228
(i) Jruby ↑$_2$ ⇓$_3$	720.53	19276	0.037	1671
(i) Ruby ⇑$_5$	855.12	26634	0.032	482
(i) PHP ⇑$_3$	1,397.51	42316	0.033	786
(i) Python ⇑$_{15}$	1,793.46	45003	0.040	275
(i) Lua ↓$_1$	2,452.04	209217	0.012	1961
(i) Perl ↑$_1$	3,542.20	96097	0.037	2148
(c) Swift	n.e.			

fannkuch-redux	Energy	Time	Ratio	Mb
(c) C ⇓$_2$	215.92	6076	0.036	2
(c) C++ ⇑$_1$	219.89	6123	0.036	1
(c) Rust ⇓$_{11}$	238.30	6628	0.036	16
(c) Swift ⇓$_5$	243.81	6712	0.036	7
(c) Ada ⇓$_2$	264.98	7351	0.036	4
(c) Ocaml ↓$_1$	277.27	7895	0.035	3
(c) Chapel ↑$_1$ ⇓$_{18}$	285.39	7853	0.036	53
(v) Lisp ↓$_3$ ⇓$_{15}$	309.02	9154	0.034	43
(v) Java ↑$_1$ ⇓$_{13}$	311.38	8241	0.038	35
(c) Fortran ⇓$_1$	316.50	8665	0.037	12
(c) Go ↑$_2$ ⇑$_7$	318.51	8487	0.038	2
(c) Pascal ⇑$_{10}$	343.55	9807	0.035	2
(v) F# ↓$_1$ ⇓$_7$	395.03	10950	0.036	34
(v) C# ↑$_1$ ⇓$_5$	399.33	10840	0.037	29
(i) JavaScript ↓$_1$ ⇓$_2$	413.90	33663	0.012	26
(c) Haskell ↑$_1$ ⇑$_8$	433.68	14666	0.030	7
(i) Dart ⇓$_7$	487.29	38678	0.013	46
(v) Racket ⇑$_3$	1,941.53	43680	0.044	18
(v) Erlang ⇑$_3$	4,148.38	101839	0.041	18
(i) Hack ⇓$_6$	5,286.77	115490	0.046	119
(i) PHP	5,731.88	125975	0.046	34
(i) TypeScript ↓$_4$ ⇑$_4$	6,898.48	516541	0.013	26
(i) Jruby ↑$_1$ ⇓$_4$	7,819.03	219148	0.036	669
(i) Lua ↓$_3$ ⇑$_{19}$	8,277.87	635023	0.013	2
(i) Perl ↑$_2$ ⇑$_{12}$	11,133.49	249418	0.045	12
(i) Python ↑$_2$ ⇑$_{14}$	12,784.09	279544	0.046	12
(i) Ruby ↑$_2$ ⇑$_{17}$	14,064.98	315583	0.045	8

fasta	Energy	Time	Ratio	Mb
(c) Rust ⇓$_9$	26.15	931	0.028	16
(c) Fortran ↓$_6$	27.62	1661	0.017	1
(c) C ↑$_1$ ⇓$_1$	27.64	973	0.028	3
(c) C++ ↑$_1$ ⇓$_2$	34.88	1164	0.030	4
(v) Java ↑$_1$ ⇓$_{12}$	35.86	1249	0.029	41
(c) Swift ⇓$_9$	37.06	1405	0.026	31
(c) Go ↓$_2$	40.45	1838	0.022	4
(c) Ada ↓$_2$ ⇑$_3$	40.45	2765	0.015	3
(c) Ocaml ↓$_2$ ⇓$_{15}$	40.78	3171	0.013	201
(c) Chapel ↑$_5$ ⇓$_{10}$	40.88	1379	0.030	53
(v) C# ↑$_4$ ⇓$_5$	45.35	1549	0.029	35
(i) Dart ⇓$_6$	63.61	4787	0.013	49
(i) JavaScript ⇓$_1$	64.84	5098	0.013	30
(c) Pascal ↓$_1$ ⇑$_{13}$	68.63	5478	0.013	0
(i) TypeScript ↓$_2$ ⇓$_{10}$	82.72	6909	0.012	271
(v) F# ↑$_2$ ⇑$_3$	93.11	5360	0.017	27
(v) Racket ↓$_1$ ⇑$_5$	120.90	8255	0.015	21
(c) Haskell ↑$_2$ ⇓$_8$	205.52	5728	0.036	446
(v) Lisp ⇓$_2$	231.49	15763	0.015	75
(i) Hack ⇓$_3$	237.70	17203	0.014	120
(i) Lua ⇑$_{18}$	347.37	24617	0.014	3
(i) PHP ↓$_1$ ⇑$_{13}$	430.73	29508	0.015	14
(v) Erlang ↑$_1$ ⇑$_{12}$	477.81	27852	0.017	18
(i) Ruby ↓$_1$ ⇑$_2$	852.30	61216	0.014	104
(i) JRuby ↑$_1$ ⇓$_2$	912.93	49509	0.018	705
(i) Python ↓$_1$ ⇑$_{18}$	1,061.41	74111	0.014	9
(i) Perl ↑$_1$ ⇑$_8$	2,684.33	61463	0.044	53

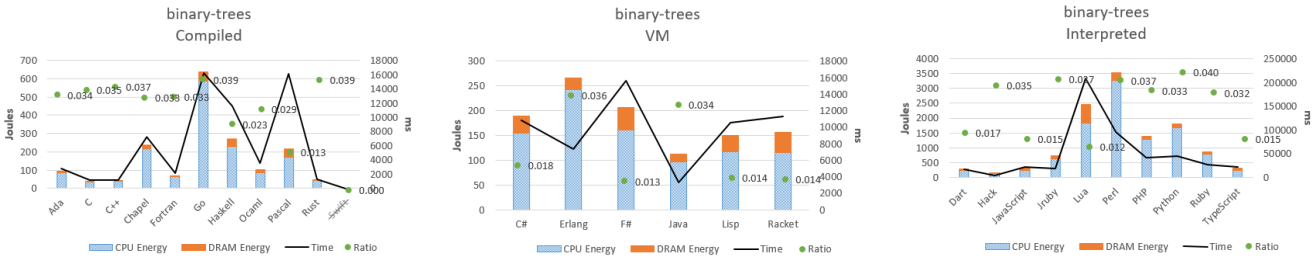

Figure 1. Energy and time graphical data for binary-trees

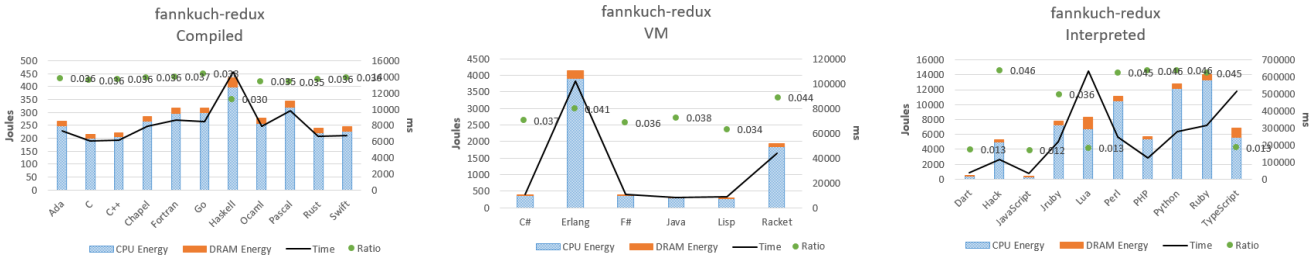

Figure 2. Energy and time graphical data for fannkuch-redux

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.

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%)

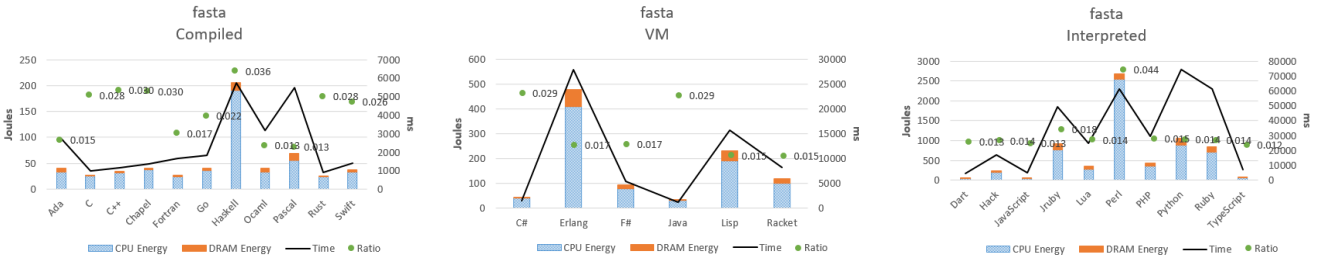

Figure 3. Energy and time graphical data for fasta

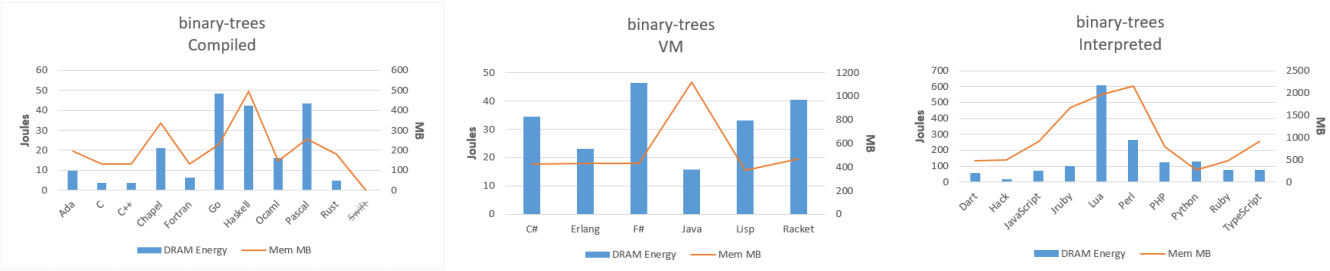

Figure 4. Energy and memory graphical data for binary-trees

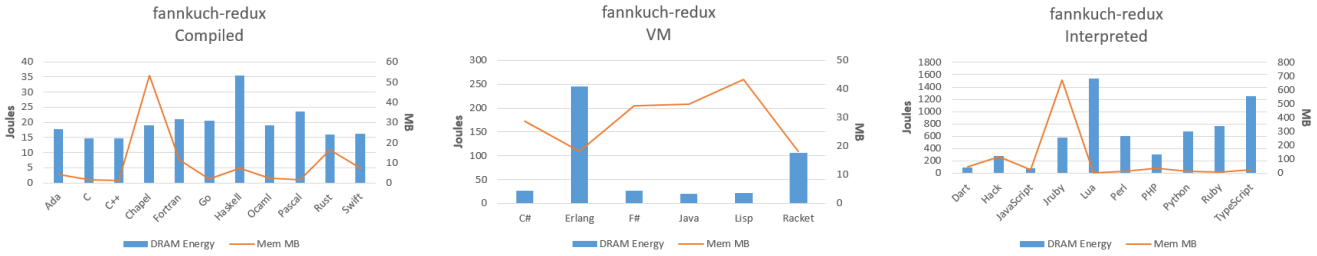

Figure 5. Energy and memory graphical data for fannkuch-redux

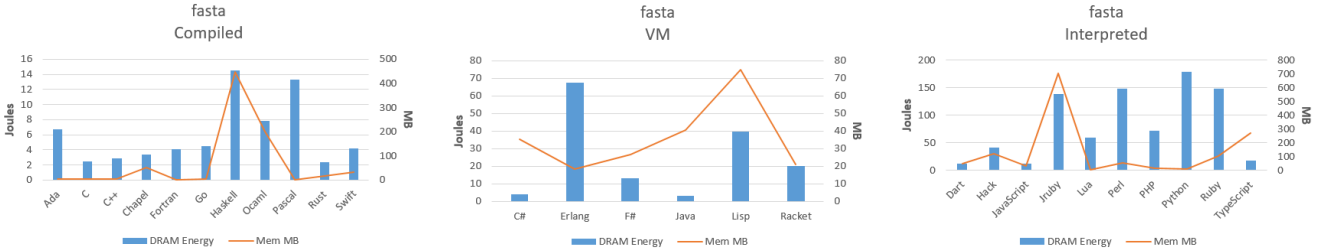

Figure 6. Energy and memory graphical data for fasta

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

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

Total					
	Energy		Time		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

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.

Moreover, 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: continuous memory usage and peak memory usage. With the data we have collected, we will try to answer the latter scenario.

The top 5 languages, also presented in Table 4, which needed the least amount of memory space (on average) to execute the solutions were: `Pascal` (66Mb), `Go` (69Mb), `C` (77Mb), `Fortran` (82Mb), and `C++` (88Mb); these are all compiled languages. The bottom 5 languages were: `JRuby` (1309Mb), `Dart` (570Mb), `Erlang` (475Mb), `Lua` (444Mb), and `Perl` (437Mb); of these, only `Erlang` is not an interpreted language.

On average, the compiled languages needed 125Mb, the virtual machine languages needed 285Mb, and the interpreted needed 426Mb. If sorted by their programming paradigm, the imperative languages needed 116Mb, the object-oriented 249Mb, the functional 251Mb, and finally the scripting needed 421Mb.

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

Looking at the visual data from Figures 4-6, 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. To verify this, we first tested both the DRAM energy consumption and peak memory usage for normality using the Shapiro-Wilk [33] test. As the data is not normally distributed, we calculated the Spearman [39] rank-order correlation coefficient. The result was a Spearman ρ value equal to 0.2091, meaning it is between no linear 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.

As future work, we wish to measure the continuous memory usage, or in other words the total amount of memory used over time, to understand if this is what leads to higher

DRAM energy consumption. We expect there to be a stronger relationship between these two as factors such as garbage collection, cache usage, register location, and the data management efficiency of each language (read/write) to have a strong impact on the energy consumption.

3.3 Energy vs. Time vs. Memory

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 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 5 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 [8, 9]. 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.

We used this technique, and in particular the software available at [37], to calculate different rankings for the analyzed software languages. In Table 5 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, that is, a set containing the languages that are equivalent to each other for the underlying objectives. In other words, each line is a single rank or position. A single software language in a position signifies that the language was 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 5 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 5 to avoid cluttering the paper with to many tables with numbers).

4 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 subsection some threats to the validity of our study,

Table 5. Pareto optimal sets for different combination of objectives.

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

divided into four categories [3], namely: conclusion validity, internal validity, construct validity, and external validity.

Conclusion Validity From our experiment it is clear that different programming paradigms and even languages within the same paradigm have a completely different impact on energy consumption, time, and memory. We also see interesting cases where the most energy efficient is not the fastest, and believe these results are useful for programmers. For a better comparison, we not only measured CPU energy consumption but also DRAM energy consumption. This allowed us to further understand the relationship between DRAM energy consumption and peak memory usage, while also understanding the behavior languages have in relation the energy usage derived from the CPU and DRAM. Additionally, the way we grouped the languages is how we felt is the most natural to compare languages (by programming paradigm, and how the language is executed). Thus, this was the chosen way to present the data in the paper. Nevertheless, all the data is available and any future comparison groups such as ".NET languages" or "JVM languages" can be very easily analyzed.

Internal Validity This category concerns itself with what factors may interfere with the results of our study. 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 memory 10 times, removed the furthest 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.

Construct Validity We analyzed 27 different programming languages, each with roughly 10 solutions to the proposed problems, totaling out to almost 270 different cases. These solutions were developed by 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 problem. Albeit certain paradigms or languages could have an advantage for certain problems, and others may be implemented in a not so traditional sense. Nevertheless, there is no basis to suspect that these projects are best or worst than any other kind we could have used.

External Validity We concern ourselves with the generalization of the results. The obtained solutions were the best performing ones at the time we set up the study. As the CLBG is an ongoing "competition", we expect that more advanced and more efficient solutions will substitute the ones we obtained as time goes on, and even the languages' compilers might evolve. Thus this, along with measurements in different systems, might produce slightly different resulting values if replicated. Nevertheless, unless there is a huge leap within the language, the comparisons might not greatly differ. 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. Thus we believe these results can be further generalized, and other researchers can replicate our methodology for future work.

5 Related Work

The work presented in this paper extends the work presented by [6], where the energy consumption monitoring approach for different programming languages was introduced. The main focus of [6] was the methodology and the comparison of the CPU-based energy efficiency in 10 of the 28 languages. We made a wider and more in-depth analysis, since in addition to including all languages, we also included the DRAM-based energy consumption and peak memory usage values, and presented a discussion on how energy, time and energy relate in software, and on different languages divided by type and paradigm.

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 [20], to analyze dynamic scripting languages [36] and compiler optimizations [34], or even to benchmark a JIT compiler for PHP [16]. At the best of our knowledge, CLGB was only used once for energy consumption analysis. In [21], the authors used the provided `Haskell` implementations, among other benchmarks, to analyze the energy efficiency of `Hakell` programs from strictness and concurrency perspectives, while also analyzing the energy influence of small implementation changes. The authors of [25] also used CLBG to compare `JavaScript`, `Java`, and `C++` in an Android setting.

While several works have shown indications that a more time efficient approach does not always lead to the most energy efficient solution [1, 21, 25, 27, 29, 35], 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 [5, 19], or just monitoring how energy consumption evolves over time [11]. Other studies aimed at a more extensive energy consumption analysis, by comparing the energy efficiency of similar programs in specific usage scenarios [4, 17], or by providing conclusions on the energy impact of different implementation decisions [7]. Several other works have shown that several factors, such as different design patterns [22, 31], coding practices [21, 26, 29, 32], and data structures [15, 23, 24, 27], actually have a significant influence in the software's energy efficiency.

6 Conclusions

In this paper, we first present an analysis and comparison of the energy efficiency of 27 well-known software languages from the popular software repository *The Computer Language Benchmarks Game*. We are able to show which were the most energy efficient software languages, execution types, and paradigms across 10 different benchmark problems.

Through also measuring the execution time and peak memory usage, we were able to relate both to energy 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.

Finally, as often times developers have limited resources and may be concerned with more than one efficiency characteristic we calculated which were the best/worst languages according to a combination of the previous three characteristics: Energy & Time, Energy & Peak Memory, Time & Peak Memory, and Energy & Time & Peak Memory.

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 project POCI-01-0145-FEDER-016718 and UID/EEA/50014/2013. The first author is also sponsored by FCT grant SFRH/BD/112733/2015.

References

[1] Sarah Abdulsalam, Ziliang Zong, Qijun Gu, and Meikang Qiu. 2015. Using the Greenup, Powerup, and Speedup metrics to evaluate software energy efficiency. In *Proc. of the 6th Int. Green and Sustainable Computing Conf.* IEEE, 1–8.

[2] Shaiful Alam Chowdhury and Abram Hindle. 2016. GreenOracle: estimating software energy consumption with energy measurement corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR, 2016*. 49–60.

[3] Thomas D Cook and Donald T Campbell. 1979. *Quasi-experimentation: design & analysis issues for field settings*. Houghton Mifflin.

[4] Marco Couto, Paulo Borba, Jácome Cunha, João P. Fernandes, Rui Pereira, and João Saraiva. 2017. Products go Green: Worst-Case Energy Consumption in Software Product Lines. (2017).

[5] Marco Couto, Tiago Carção, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2014. Detecting Anomalous Energy Consumption in Android Applications. In *Programming Languages: 18th Brazilian*

Symposium, SBLP 2014, Maceio, Brazil, October 2-3, 2014. Proceedings, Fernando Magno Quintão Pereira (Ed.). 77–91.

[6] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. 2017. Towards a Green Ranking for Programming Languages. In *Programming Languages: 21st Brazilian Symposium, SBLP 2017, Fortaleza, Brazil, September, 2017.*

[7] Luis Cruz and Rui Abreu. 2017. Performance-based Guidelines for Energy Efficient Mobile Applications. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft '17)*. IEEE Press, 46–57.

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

[9] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *Trans. Evol. Comp* 6, 2 (2002), 182–197.

[10] Martin Dimitrov, Carl Strickland, Seung-Woo Kim, Karthik Kumar, and Kshitij Doshi. 2015. Intel® Power Governor. https://software.intel.com/en-us/articles/intel-power-governor. (2015). Accessed: 2015-10-12.

[11] F. Ding, F. Xia, W. Zhang, X. Zhao, and C. Ma. 2011. Monitoring Energy Consumption of Smartphones. In *Proc. of the 2011 Int. Conf. on Internet of Things and 4th Int. Conf. on Cyber, Physical and Social Computing*. 610–613.

[12] Isaac Gouy. *The Computer Language Benchmarks Game*. http://benchmarksgame.alioth.debian.org/

[13] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Performance Evaluation Review* 40, 3 (2012), 13–17.

[14] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. 2013. Estimating Mobile Application Energy Consumption Using Program Analysis. In *Proc. of the 2013 Int. Conf. on Software Engineering (ICSE '13)*. IEEE Press, 92–101.

[15] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy profiles of java collections classes. In *Proc. of the 38th Int. Conf. on Software Engineering*. ACM, 225–236.

[16] Andrei Homescu and Alex Şuhan. 2011. HappyJIT: A Tracing JIT Compiler for PHP. *SIGPLAN Not.* 47, 2 (Oct. 2011), 25–36.

[17] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann. 2015. EcoDroid: An Approach for Energy-based Ranking of Android Apps. In *Proc. of 4th Int. Workshop on Green and Sustainable Software (GREENS '15)*. IEEE Press, 8–14.

[18] Ding Li and William G. J. Halfond. 2014. 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)*.

[19] Ding Li, Shuai Hao, William GJ Halfond, and Ramesh Govindan. 2013. Calculating source line level energy information for android applications. In *Proc. of the 2013 Int. Symposium on Software Testing and Analysis*. ACM, 78–89.

[20] Wing Hang Li, David R. White, and Jeremy Singer. 2013. JVM-hosted Languages: They Talk the Talk, but Do They Walk the Walk?. In *Proc. of the 2013 Int. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '13)*. ACM, 101–112.

[21] Luís Gabriel Lima, Gilberto Melfe, Francisco Soares-Neto, Paulo Lieuthier, João Paulo Fernandes, and Fernando Castor. 2016. Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. In *Proc. of the 23rd IEEE Int. Conf. on Software Analysis, Evolution, and Reengineering (SANER'2016)*. IEEE, 517–528.

[22] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining energy-greedy API usage patterns in Android apps: an empirical study. In *Proc. of the 11th Working Conf. on Mining Software Repositories*. ACM, 2–11.

[23] Kenan Liu, Gustavo Pinto, and Yu David Liu. 2015. Data-oriented characterization of application-level energy optimization. In *Fundamental Approaches to Software Engineering*. Springer, 316–331.

[24] Irene Manotas, Lori Pollock, and James Clause. 2014. SEEDS: A Software Engineer's Energy-Optimization Decision Support Framework. In *Proc. of the 36th Int. Conf. on Software Engineering*. ACM, 503–514.

[25] Wellington Oliveira, Renato Oliveira, and Fernando Castor. 2017. A study on the energy consumption of Android app development approaches. In *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 42–52.

[26] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva. 2017. Helping Programmers Improve the Energy Efficiency of Source Code. In *Proc. of the 39th Int. Conf. on Soft. Eng. Companion*. ACM.

[27] Rui Pereira, Marco Couto, João Saraiva, Jácome Cunha, and João Paulo Fernandes. 2016. The Influence of the Java Collection Framework on Overall Energy Consumption. In *Proc. of the 5th Int. Workshop on Green and Sustainable Software (GREENS '16)*. ACM, 15–21.

[28] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining questions about software energy consumption. In *Proc. of the 11th Working Conf. on Mining Software Repositories*. ACM, 22–31.

[29] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Understanding energy behaviors of thread management constructs. In *Proc. of the 2014 ACM Int. Conf. on Object Oriented Programming Systems Languages & Applications*. ACM, 345–360.

[30] Efraim Rotem, Alon Naveh, Avinash Ananthakrishnan, Eliezer Weissmann, and Doron Rajwan. 2012. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* 32, 2 (2012), 20–27.

[31] Cagri Sahin, Furkan Cayci, Irene Lizeth Manotas Gutierrez, James Clause, Fouad Kiamilev, Lori Pollock, and Kristina Winbladh. 2012. Initial explorations on design pattern energy usage. In *Green and Sustainable Software (GREENS), 2012 1st Int. Workshop on*. IEEE, 55–61.

[32] Cagri Sahin, Lori Pollock, and James Clause. 2014. How do code refactorings affect energy usage?. In *Proc. of 8th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement*. ACM, 36.

[33] SS Shaphiro and MB Wilk. 1965. An analysis of variance test for normality. *Biometrika* 52, 3 (1965), 591–611.

[34] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Optimization Coaching: Optimizers Learn to Communicate with Programmers. In *Proc. of ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, 163–178.

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

[36] Kevin Williams, Jason McCandless, and David Gregg. 2010. Dynamic Interpretation for Dynamic Scripting Languages. In *Proc. of the 8th Annual IEEE/ACM Int. Symposium on Code Generation and Optimization (CGO '10)*. ACM, 278–287.

[37] Matthew Woodruff and Jon Herman. 2013. pareto.py: a ϵ – *nondomination* sorting routine. https://github.com/matthewjwoodruff/pareto.py. (2013).

[38] Tomofumi Yuki and Sanjay Rajopadhye. 2014. Folklore confirmed: Compiling for speed= compiling for energy. In *Languages and Compilers for Parallel Computing*. Springer, 169–184.

[39] Daniel Zwillinger and Stephen Kokoska. 1999. *CRC standard probability and statistics tables and formulae*. Crc Press.