Towards a Green Ranking for Programming Languages [*]

Marco Couto, Rui Pereira
HASLab/INESC TEC
Universidade do Minho
Portugal
marco.l.couto@inesctec.pt,
ruipereira@di.uminho.pt

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

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

ABSTRACT

While in the past the primary goal to optimize software was the run time optimization, nowadays there is a growing awareness of the need to reduce energy consumption. Additionally, a growing number of developers wish to become more energy-aware when programming and feel a lack of tools and the knowledge to do so.

In this paper we define a ranking of energy efficiency in programming languages. We consider a set of computing problems implemented in ten well-known programming languages, and monitored the energy consumed when executing each language. Our preliminary results show that although the fastest languages tend to be the lowest consuming ones, there are other interesting cases where slower languages are more energy efficient than faster ones.

CCS CONCEPTS

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

KEYWORDS

Energy Efficiency, Programming Languages, Language Benchmarking, Green Software

ACM Reference format:
Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. 2017. Towards a Green Ranking for Programming Languages . In *Proceedings of SBLP 2017, Fortaleza, CE, Brazil, September 21–22, 2017,* 8 pages.
https://doi.org/10.1145/3125374.3125382

1 INTRODUCTION

The performance of computers was a main concern since their creation in the previous century. Today, this is no exception and performance is still the main goal for both computer manufacturers and software developers. In this century, however, the focus on computer performance is changing: program's execution time is no longer the only concern in terms of computer performance. The quick adoption of non-wired computer devices in recent years is making energy consumption one of the main bottlenecks when building computers and developing their software.

While computer manufacturers began developing energy efficient hardware since the very beginning of the world wide adoption of mobile computing devices, only recently has the software engineering community started to become concerned about the energy efficiency of software systems [23]. Today, this topic has become an intensive area of research, where techniques to monitor energy consumption by programs have been defined, such as the use of consumption models in mobile devices [34], the use of estimations as provided by the Intel RAPL [6, 8, 18], the Qualcomm TrepN frameworks [1], or even the use of external energy measurement devices [2, 19]. Using such monitoring frameworks, several techniques have been proposed to reason about energy consumption in programming languages, for example, analyzing the energy efficiency of Java and Haskell data structures [16, 22, 25], analyzing how different programming coding practices influence energy consumption [28], studying the impact of testing techniques in software energy consumption [14], etc.

An interesting question that frequently arises in the area of software energy efficiency is whether *a faster program is also an energy efficient program*, or not. Indeed, there are research papers pointing in completely different directions, for example [1, 16, 22, 24, 31, 33].

This question also arises when comparing the performance of different programming languages. However, comparing programming languages performance is an extremely complex task. Programs written in different languages which implement exactly the same computing problem may use different algorithms. Moreover, the reused language libraries, the quality of the compiler, and its (aggressive) optimizations all greatly influence the performance of the resulting programs. Thus, a programming language may become *faster*, not by changing its programs, but by "just" improving its libraries and/or its compiler (or virtual machine). Nevertheless there is some work aiming at comparing performance of programming languages, such as the *"Computer Language Benchmarks Game"* (CLBG) project [2] which compares computer language performance, in terms of execution time and memory consumption. It includes a repository of programs written in different languages which implement solutions to a set of predefined computing problems. This

[*]This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project POCI-01-0145-FEDER-016718. The second author is also sponsored by FCT grant SFRH/BD/112733/2015

SBLP 2017, September 21–22, 2017, Fortaleza, CE, Brazil
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5389-2/17/09…$15.00
https://doi.org/10.1145/3125374.3125382

[1]https://developer.qualcomm.com/software/trepn-power-profiler
[2]More information about the CLBG can be found here: http://benchmarksgame.alioth.debian.org

language benchmark project was developed to provide a runtime ranking of programming languages.

In this paper we reuse the computing problems and solutions proposed in CLBG to define a ranking of energy efficiency in programming languages. We consider the computing problems of CLBG implemented in ten well-known programming languages (*C*, *C#*, *Fortran*, *Go*, *Java*, *JRuby*, *Lua*, *OCaml*, *Perl*, and *Racket*) in order to answer the following questions: *Is the fastest language, the most energy efficient one? Are there languages which run slower, while consuming less energy than others?*

We developed a framework using an energy measurement tool provided by Intel (RAPL), in order to monitor the the consumption when executing a program. We have used this monitoring framework to measure the energy of all executable programs included in CLBG. Our preliminary results show that the *C* language is both the fastest and greenest language, while also showing interesting cases where slower languages are greener than others: for example, when considering all the benchmarks, *Lua* is slower by, roughly 12%, than *Perl*, while consuming 53% less energy!

The remaining of this paper is structured as follows: Section 2 presents the computer language benchmark game. Section 3 describes the methodology we use to monitor the energy consumption of the ten languages we consider in our study. We present the energy consumption of each program and we discuss the obtained results. Section 4 details the threats to the validity of our study. In Section 5 we describe related work, and in Section 6 we present the conclusions of our work.

2 COMPUTER LANGUAGE BENCHMARK GAME

In order to compare the performance, both in terms of execution time and energy consumption, of different programming languages, we need solutions/programs for the same problems expressed in each of those languages. Developing programs written in different programming languages (and programming paradigms) is both a complex and time consuming task. Thus, in our study we reused the repository of programs available in the "Computer Language Benchmarks Game[3]" (CLBG) project. The project compares programming language performance, in terms of runtime and memory consumption, by implementing a solution to a set of computing problems in different programming languages.

The CLBG includes solutions for thirteen different problems, written in twenty eight different programming languages. These benchmarks have been used in several research works, to study the dynamic behavior of non-Java JVM languages [15], to analyze dynamic scripting languages [32] and compiler optimizations [30], to benchmark a JIT compiler for PHP [11], and in the design of the R language [20]. In fact, the repository of programs included in CLBG are also particularly suitable to analyze the energy efficiency of programming languages due to the following characteristics:

- The solutions do represent the state-of-the-art on how to solve the thirteen problems in each of the programming language. Indeed, the programs in the repository were developed by experts in each of the languages considered.
- The source code for the different program solutions is available as open source, and consequently the solutions (and their authors) are known. Thus, it is possible to see that each program follows common programming practices used by the respective programming language community.
- The solutions use the same underlying algorithm. Therefore they may ignore programming language features that would improve performance using a different algorithm.
- The inputs provided to the programs are known, and all programs produce the same output for similar inputs (in other words the programs are correct).
- Finally, the compiler version and options to compile/execute the program's source code in each language is also specified in the CLBG.

[3]The word *game* in this benchmark means that programmers from different languages contribute programs that compete (but try to remain comparable) with each others.

The considered programs in the CLBG cover different computing problems, as described in Table 1.

Table 1: CLBG corpus of programs used in our study.

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

Unfortunately, in the current state of the CLBG, not all implementations for these thirteen computing problems are available. In our study we only consider the first ten computing problems, as their implementations are the most commonly available.

3 ENERGY EFFICIENCY IN PROGRAMMING LANGUAGES

In this section, we present our case study for comparing the energy efficiency of different computer languages.

We start by describing the followed methodology and the tools used to compare the energy efficiency of 10 different programming languages (Section 3.1). We chose to analyze the 10 following languages: *C*, *C#*, *Fortran*, *Go*, *Java*, *JRuby*, *Lua*, *OCaml*, *Perl*, and *Racket*. We used the program solutions available in the CLBG, implemented in these 10 languages, as the objects for our case study. Finally, we present the obtained results for all 10 languages (Section 3.2).

3.1 Design and Execution

The CLBG proposes 13 benchmark problems to be solved in various different languages. Of these 13 we chose the first 10 shown in Table 1, as explained in the previous section. We gathered the source code of the most efficient (i.e., fastest) benchmark solutions for the 10 benchmark problems, for each of our considered 10 programming languages. As the CLBG lists the performance of each proposed solution, we already knew which ones were the most efficient.

These benchmark solutions were compiled and executed according to the information provided by the submission's documentation page. These documentation pages also provided information on the compiler versions, and compilation/executions options.

To obtain energy measurements, we used Intel's Running Average Power Limit (RAPL) tool [6], which has already been proven to provide very accurate energy measurements [8, 26]. Each benchmark was measured 10 times, both energy consumption and execution time. By following this approach we aim to reduce the impact of cold starts and cache effects, while also analyzing the consistency of the measured results to check consistency and avoid outliers.

Since the RAPL tool is currently only usable within the C and Java (through jRAPL [18]) languages, we needed to develop a measuring framework to allow the energy measurement of every language. Our goal was to measure the energy consumed by the whole program, and not by independent code blocks (such as methods), in order to properly compare the languages. Therefore, we developed a measuring framework written in the *C* language which measures the execution of an external program, using the `system` function. The overall process of this framework is described in Listing 1.

```
...
int main() {
  ...
  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.

As we can see, both the execution time and energy are measured for every execution of `command`. This variable holds a string describing how to execute a particular benchmark, in a particular language. For example, for executing the *binary trees* benchmark in the *C* language, the value of the `command` variable will be `"./binarytrees.o 21"`, where *binarytrees.o* is the name of the executable file resulting of prior compilation, and *21* is the argument provided to the *binary trees* benchmark. For interpreted languages (such as *Perl*) or languages which require a virtual machine to be executed (such as *Java*), the path to the interpreter/virtual machine is also provided. To execute the *fasta* benchmark in *Perl*, the variable `command` would be `"/usr/bin/perl fasta.perl 25000000"`.

Before advancing, we needed to certify that there would be minimal or no overhead from our measuring framework using the `system` function. To do so, we measured the energy consumption of both a C and Java language program, using RAPL and jRAPL respectively, and compared the results to the measurements from our C language energy measuring framework. The differences were insignificant, consistent, and negligible. Therefore, we may use our C language energy measurement framework without worry of overheads or imprecise measurements.

In a few cases, we were not able to generate the results for all of the 10 benchmarks solutions. This was due to one of three reasons: (1) there was no available source code for the missing benchmark, (2) the implementation contained errors, or (3) there were missing libraries which could not be found. Nevertheless, by choosing these 10 languages, we assure that every benchmark problem is implemented in at least 70% of our chosen languages, and that all programming paradigms are included.

We ran these studies 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. After measuring each benchmark solution 10 times, we extracted and treated the CPU energetic consumption values and carefully analyzed the results to check for consistency and accurate measurements as described in Section 4.

3.2 Results and Discussion

This section presents the obtained results in our study, followed by an analysis and discussion of these results. Each table, from Table 2 to Table 6, represents the results from a particular benchmark, where each row represents one of the 10 considered languages. The values in each line, from left to right, include the average values for the *Energy* consumed (Joules), *Time* of execution (milliseconds), and *Ratio* (ratio between Energy and Time). This ratio can also be seen as the average *Power* consumed in Kilowatts (kW). In these tables, the programming languages are ordered in regards to their energy consumption, from lowest to highest.

Table 7 shows the global results (summation) for *Energy* and *Time* normalized to the C language. So for example, C# consumes 2.21x more energy than C, while taking 2.44x more to finish. In this table, the left side orders the programming language in regards to their energy consumption while the right orders the language in regards to their execution time.

In some cases, across all tables from Table 2 to Table 7, the programming language name will have an $\uparrow_x$ or $\downarrow_x$ symbol. This states that the language would go up by x positions ($\uparrow_x$) or down by x positions ($\downarrow_x$) if ordered by the execution time. For example in Table 3, Fortran is the most energy efficient language, but would fall off 3 positions down if ordered by the execution time.

To further understand these values, we also generated graphs for each of the benchmarks (Figures 1 - 8). Each graph contains the results of each language relative for a benchmark, consisting of a combination of three parts: a bar chart, a line chart and a scatter plot.

Table 2: Results for binary-trees and fannkuch-redux

binary-trees			
	Energy	Time	Ratio
C	36.06	1124.67	0.032
Fortran	63.56	2112.17	0.030
Ocaml$\downarrow_1$	84.63	3525.47	0.024
Java$\uparrow_1$	96.09	3305.65	0.029
Racket$\downarrow_1$	115.45	11260.66	0.010
C#$\uparrow_1$	155.19	10797.15	0.014
Go	588.14	16291.66	0.036
Jruby	617.96	19276.14	0.032
Lua$\downarrow_1$	1841.62	209217.00	0.009
Perl$\uparrow_1$	3276.56	96097.28	0.034

fannkuch-redux			
	Energy	Time	Ratio
C	201.11	6076.28	0.033
Ocaml	258.20	7895.43	0.033
Java	291.46	8240.84	0.035
Fortran$\downarrow_1$	295.56	8665.26	0.034
Go$\uparrow_1$	298.01	8487.00	0.035
C#	373.13	10839.74	0.034
Racket	1836.00	43680.23	0.042
Lua$\downarrow_2$	6732.44	634877.63	0.011
Jruby$\uparrow_1$	7242.76	219148.25	0.033
Perl$\uparrow_1$	10526.19	249357.50	0.042

Table 3: Results for fasta and k-nucleotide

fasta			
	Energy	Time	Ratio
Fortran$\downarrow_3$	23.54	1661.43	0.014
C$\uparrow_1$	25.22	973.27	0.026
Java$\uparrow_1$	32.73	1248.89	0.026
Ocaml$\downarrow_2$	32.97	3170.54	0.010
Go	35.93	1838.39	0.020
C#$\uparrow_3$	41.47	1549.05	0.027
Racket	100.64	8254.71	0.012
Lua	287.82	24616.91	0.012
Jruby	775.08	49508.94	0.016
Perl	2535.80	61462.63	0.041

k-nucleotide			
	Energy	Time	Ratio
C	81.20	2958.04	0.027
Java	127.32	4116.17	0.031
C#	189.55	7138.50	0.027
Go	247.79	8004.35	0.031
Ocaml	294.12	13847.16	0.021
Fortran$\downarrow_1$	393.77	41656.29	0.009
Racket$\downarrow_1$	511.60	44238.51	0.012
Lua$\downarrow_1$	981.73	88074.58	0.011
Perl$\uparrow_3$	1227.12	35615.50	0.034
Jruby	2269.92	88523.14	0.026

Table 4: Results for pidigits and regex-redux

pidigits			
	Energy	Time	Ratio
C	6.19	546.23	0.011
Racket	8.43	731.89	0.012
Go	11.27	772.52	0.015
C#	11.44	943.64	0.012
Perl	16.11	1306.91	0.012
Jruby	121.26	8198.40	0.015

regex-redux			
	Energy	Time	Ratio
C	22.60	804.82	0.028
Ocaml$\downarrow_2$	133.62	12977.91	0.010
Java$\uparrow_1$	177.79	5693.63	0.031
Perl$\uparrow_1$	218.66	7163.59	0.031
Racket$\downarrow_2$	289.46	26152.36	0.011
Jruby$\uparrow_1$	313.13	13477.41	0.023
C#$\uparrow_1$	485.58	14722.95	0.033

Table 5: Results for mandelbrot and n-body

mandelbrot			
	Energy	Time	Ratio
C	32.60	1141.53	0.029
Java$\downarrow_1$	105.34	3657.16	0.029
Go$\uparrow_1$	107.53	3451.28	0.031
C#	140.49	3960.59	0.035
Fortran$\downarrow_1$	146.60	8633.04	0.017
Ocaml$\uparrow_1$	194.56	6863.35	0.028
Racket	418.33	44516.96	0.009
Lua	3392.11	100441.83	0.034
Jruby	6972.23	217158.63	0.032
Perl	16042.30	390013.25	0.041

n-body			
	Energy	Time	Ratio
Fortran	41.69	3580.94	0.012
C	49.24	4190.34	0.012
Java	51.03	5839.28	0.009
Ocaml	51.60	5856.66	0.009
Go	52.98	5899.14	0.009
C#	53.38	6117.06	0.009
Racket	227.73	22259.99	0.010
Jruby	1432.98	98407.31	0.015
Lua	3462.55	177251.13	0.020
Perl	3853.19	335390.75	0.011

Table 6: Results for reverse-complement and spectral-norm

reverse-complement			
	Energy	Time	Ratio
C	5.64	227.90	0.030
Ocaml	7.58	286.78	0.022
Go	7.91	366.17	0.024
Fortran$\downarrow_2$	9.99	937.56	0.029
C#$\uparrow_1$	10.68	581.44	0.024
Java$\uparrow_1$	14.59	628.79	0.022
Perl	17.50	1187.04	0.041
Racket	25.17	2112.06	0.034
Jruby	114.38	4533.48	0.013
Lua	119.38	9305.06	0.011

spectral-norm			
	Energy	Time	Ratio
Fortran	19.37	667.03	0.011
C	20.02	676.51	0.025
Go	32.16	1331.62	0.022
C#	32.36	1374.42	0.018
Ocaml$\downarrow_1$	36.57	1683.41	0.026
Java$\uparrow_1$	37.16	1658.64	0.023
Racket	79.31	2341.39	0.012
Perl	814.68	19781.86	0.015
Lua$\downarrow_1$	1019.49	95405.64	0.013
Jruby$\uparrow_1$	1031.88	79449.25	0.025

The bars represent the energy consumed by the languages, with the left y-axis representing the average Joules. The execution time is represented by the line chart, with the right y-axis representing the average time in milliseconds. By combining the two charts we can easily understand the relationship between energy and time.

Table 7: Normalized global results for Energy and Time

Total			
	Energy	Time	
C	1.00	1.00	C
Java	1.68	1.65	Java
Ocaml$\downarrow_1$	2.13	2.44	C#$\downarrow_2$
Fortran$\downarrow_2$	2.20	2.48	Ocaml$\uparrow_1$
C#$\uparrow_2$	2.21	2.63	Go$\downarrow_1$
Go$\uparrow_1$	3.04	3.91	Fortran$\uparrow_2$
Racket	7.35	10.29	Racket
Lua$\downarrow_2$	39.54	44.68	Jruby$\downarrow_1$
Jruby$\uparrow_1$	45.35	68.45	Perl$\downarrow_1$
Perl$\uparrow_1$	84.89	77.10	Lua$\uparrow_2$

Finally, the scatter plot on top of both represents the ratio between energy consumed and execution time. This plot allows us to observe if the relation between energy and time is consistent between languages. A variation in these values indicates that the energy consumed is not directly proportional to the execution time, but also dependent on the language and/or benchmark solution.

From the data presented in the aforementioned tables and plots, we can indeed draw a set of relevant and interesting observations, and answer our previous questions. For example, a common perception to the energy consumption problem is that it is a performance issue, whereby reducing execution time would bring about energy efficiency. While some works do support that claim [33], the opposite has also been observed [16, 24, 31], where an increase in execution time brings about a decrease in the energy consumed. We have observed both cases, as do [1, 22].

- The results clearly show that the *C* language is both the fastest and greenest language. However, in certain cases there are other more energy efficient solutions.
- From our study, we can see several examples where language `A` consumes more energy to run a benchmark than language `B`, while taking less time to do so. Such an example can be seen in Figure 1, for instance, where the `Java` language has higher energy consumption then the `OCaml` language, yet a slightly lower execution time. The same can be observed in the `Lua` and `Perl` languages, for example, in Figure 1, 2, and 8. In these cases `Lua` consumes less energy while having a higher execution time.
- Energy consumption is **not always** directly proportional to execution time. In almost every graph/table, we can observe that execution time does not behave in the same way the energy consumption does in proportion to one another. For instance, in Table 2 we can see that the `C#` language consumes roughly 4 times more energy to execute the `binary-trees` benchmark then the `C` language. However, in terms of execution time, it takes the `C#` language around 9 times more to finish then the `C` language.
- By observing the ratio values (or in other words the average Power in kW), the previous discussion is further strengthened. While performance optimization would indeed (in most cases) reduce the energy consumed, the fact of the matter is we still have the `Power` variable in the `Energy` equation (`Joules = Watts x Second`). For this to be purely a performance issue, we would have to assume

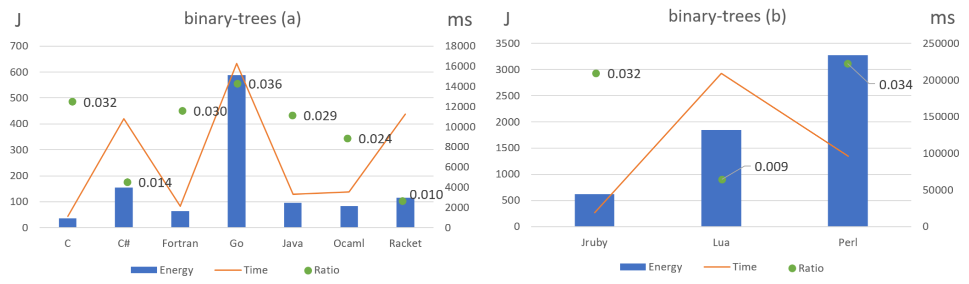

Figure 1: Graphical data for binary-trees

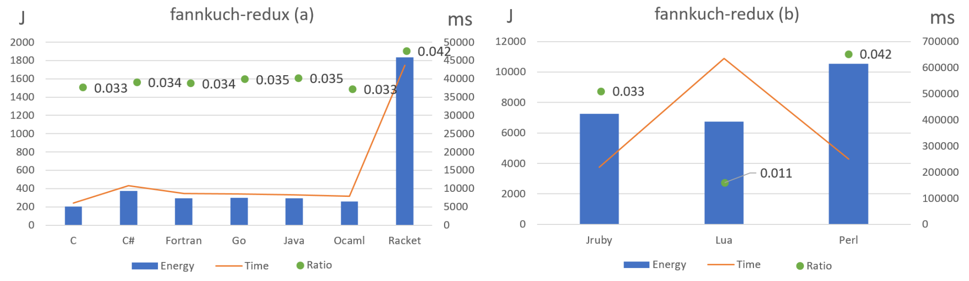

Figure 2: Graphical data for fannkuch-redux

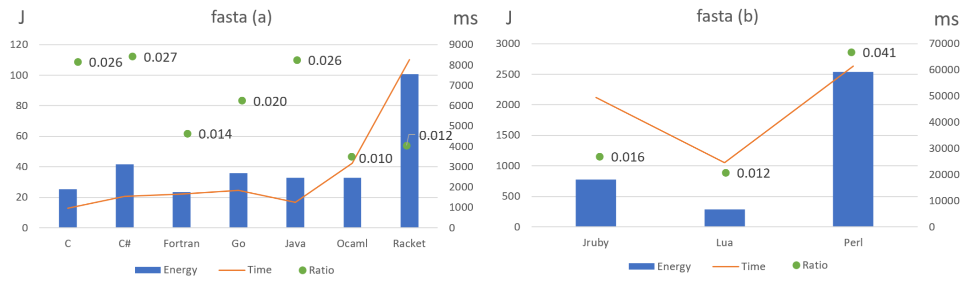

Figure 3: Graphical data for fasta

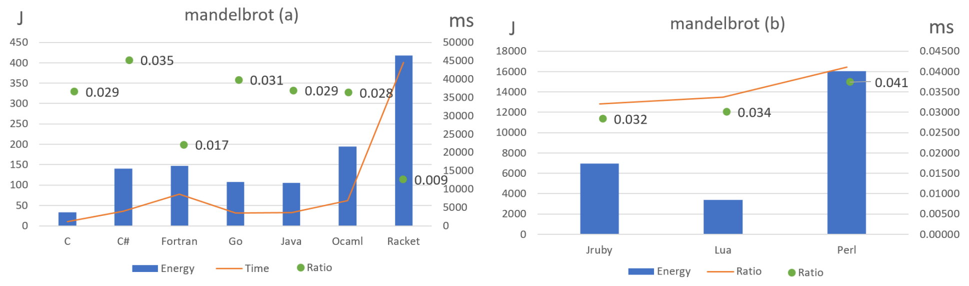

Figure 4: Graphical data for mandelbrot

that the Power is a constant, which we clearly see that is not the case, and may even vary quite heavily in between different languages. For example in Figure 8, JRuby and Lua have almost identical execution times, while Lua has less than half the ratio (average kW) compared to JRuby, and in turn half the energy consumed also.

- We can easily see that the three interpreted languages Lua, JRuby, and Perl have a tendency to not only be slower,

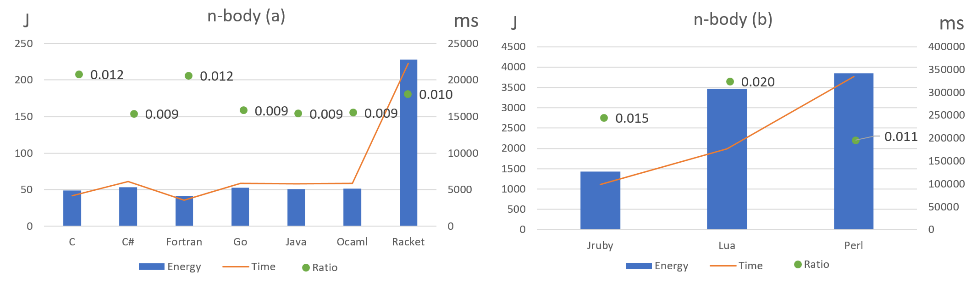

Figure 5: Graphical data for n-body

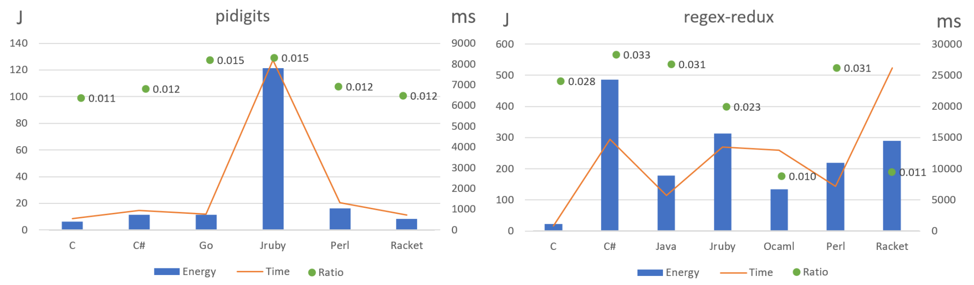

Figure 6: Graphical data for pidigits and regex-redux

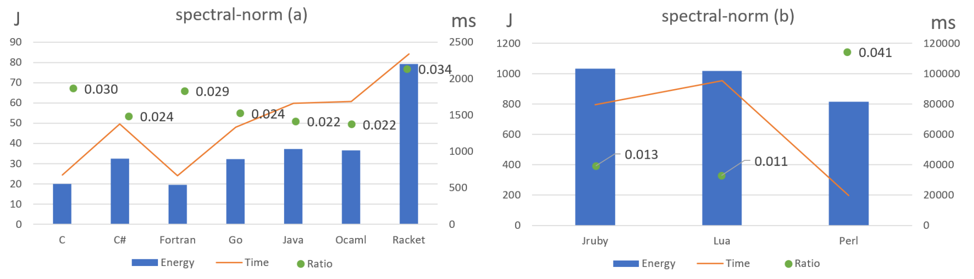

Figure 7: Graphical data for spectral-norm

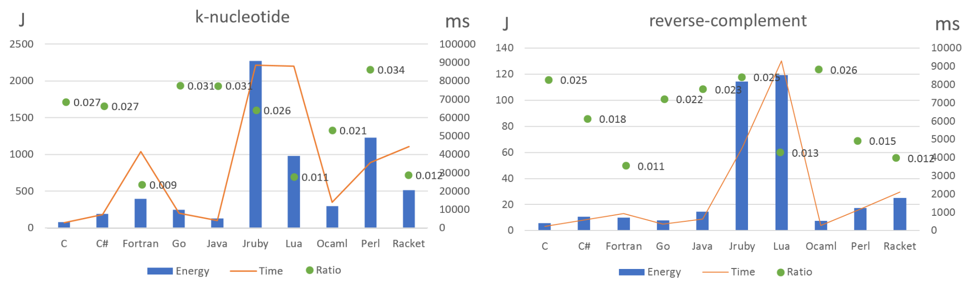

Figure 8: Graphical data for k-nucleotide and reverse-complement

but much more energy consuming (such that, we even had to show the visual data in different scales). Even so, out of the three, `Lua` tends to be the slowest one while also being the lowest consuming one. When compared overall to `Perl` (Table 7, we can see `Lua` is slower by roughly 12% than `Perl`, while consuming 53% less energy!

- Looking at Table 7 we have a better view of the overall results. When comparing all the languages together, we clearly see that `C` and `Java` are the best two, in both energy consumption and execution time. We also observe languages such as `OCaml`, `Fortran`, and `Lua` as being much more energy efficient than they are performance efficient, with the opposite observed for `C#`, `Go`, `Jruby`, and `Perl`.

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 in terms of energy consumption. We present in this subsection some threats to the validity of our study, divided into four categories [4], 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. We also see interesting cases where the most energy efficient is not the fastest, and believe these results are useful for programmers. Nevertheless, the observed energy consumption is only attributed to CPU usage, and while we have shown that energy and performance are sometimes related in non-predictable ways, the impact of other hardware components (such as memory usage and its energy impact) deserve further analysis.

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. To avoid this, we executed every language and benchmark solution equally. In each, we measured the energy consumption and execution time 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 results measured are quite consistent, and thus reliable. In addition, the used measurement tool has also been proven to be very accurate.

Construct Validity. We analyzed 10 different programming languages, each with roughly 10 solutions to the proposed problems, totaling out to almost 100 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, all produced the same exact output and each are 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, while others may be implemented in a not so traditional sense. Nevertheless, where is no basis to suspect that these projects are best or worst than any other kind we could of have used.

External Validity. We concern ourselves with the generalization of the results. The obtained solutions were the best 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

For years, the main reference when it comes to classifying a program's performance and efficiency has been its execution time. Indeed, there is a long lasting series of engineering techniques provided to software developers which aim at helping them build correct programs, while also indicating how to reduce the execution time [30, 32]. With the improvements on hardware devices, programs became more complex and robust, and another perfomance aspect became relevant: the memory usage. Soon, software developers were supplied with tools and techniques for analyzing the memory usage of their programs [3].

With the emergence of different languages, platforms and programming paradigms, the interest in analyzing performance aspects in software has indeed increased, and execution time is no longer the only relevant performance aspect. Software related energy consumption has been gaining an increasing interest in recent years as well, especially among researchers. In fact, this interest is being extended to software developers [23], who have been questioning how to optimize energy consumption through software improvements.

Several studies have been aimed at understanding how the energy consumption of software systems can be influenced by development aspects. In fact, it is already known that several factors, such as the use of different design patterns/coding practices [16, 17, 27, 28], different data structures [10, 16, 18, 22], or even code obfuscation [29] can significatively influence the energy consumed by software. This awareness of energy consumption is notorious within the software testing area, where some works aim at reducing the overall consumption in the testing phase, by reducing the number of tests while maintaining the code coverage [12, 14].

Other energy related research works offer a more thorough energy consumption analysis of programs. While some have been focused on offering techniques able to identify, within a program, code blocks responsible for excessive energy consumption [5, 13, 21], others try to predict the energy consumption of a program by statically analyzing it [7, 9]. The result of the latter studies is an estimate of the energy consumed by the program in a specific scenario.

Regardless of the interest in the energy consumption area, there is no study, to the best of our knowledge, which compares the energy consumption behavior of different programming languages. With this study, we aim at creating the basis for exploring the subject in more detail, by providing evidence that the energy consumption behaves differently depending not only on the program but also on the programming language, while also having different behaviors in comparison to its execution time analysis.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we described a study on the energy efficiency of ten different programming languages, considering a corpus of ten computing problems included in the CLBG project. We monitored the energy consumed and execution time of almost hundred computer programs.

Our results show that compiled languages are, as we expected, both more energy efficient and faster than interpreted ones, while also showing that `C` is the fastest and greenest language in our study. However, the results also show that energy consumption is not always directly proportional to execution time. In fact, there are greener programming languages, while being slower than others.

In this study we only monitored and related the CPU energy consumption to execution time. As future work we plan to study the impact of memory consumption on the energy consumption, namely in terms of the energy consumed by the computer's RAM and hard drive. We also plan to extend our study with more programming languages, namely, languages using different execution models (for example, lazy evaluated languages).

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] Tarsila Bessa, Pedro Quintão, Michael Frank, and Fernando Magno Quintão Pereira. 2016. JetsonLeap: A Framework to Measure Energy-Aware Code Optimizations in Embedded and Heterogeneous Systems. In *Proc. of the 20th Brazilian Symposium on Programming Languages*, Fernando Castor and Yu David Liu (Eds.). Springer Int. Publishing, 16–30.

[3] Wei-Ngan Chin, Huu Hai Nguyen, Shengchao Qin, and Martin Rinard. 2005. Memory Usage Verification for OO Programs. In *Static Analysis: 12th Int. Symposium, SAS 2005, London, UK, September 7-9, 2005. Proceedings*, Chris Hankin and Igor Siveroni (Eds.). Springer Berlin Heidelberg, 70–86.

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

[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.). Springer Int. Publishing, 77–91.

[6] 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.

[7] Neville Grech, Kyriakos Georgiou, James Pallister, Steve Kerrison, Jeremy Morse, and Kerstin Eder. 2015. Static Analysis of Energy Consumption for LLVM IR Programs. In *Proc. of the 18th Int. Workshop on Software and Compilers for Embedded Systems (SCOPES '15).* ACM, 12–21.

[8] 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.

[9] 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.

[10] 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.

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

[12] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. 2016. Energy-aware Test-suite Minimization for Android Apps. In *Proc. of the 25th Int. Symposium on Software Testing and Analysis (ISSTA 2016).* 425–436.

[13] 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.

[14] Ding Li, Yuchen Jin, Cagri Sahin, James Clause, and William GJ Halfond. 2014. Integrated energy-directed test suite optimization. In *Proc. of the 2014 Int. Symposium on Software Testing and Analysis.* ACM, 339–350.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Dustin McIntire, Thanos Stathopoulos, Sasank Reddy, Thomas Schmidt, and William J. Kaiser. 2012. Energy-Efficient Sensing with the Low Power, Energy Aware Processing (LEAP) Architecture. *ACM Trans. Embed. Comput. Syst.* 11, 2, Article 27 (July 2012), 36 pages.

[20] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. Evaluating the Design of the R Language: Objects and Functions for Data Analysis. In *Proc. of the 26th European Conf. on Object-Oriented Programming (ECOOP'12).* Springer-Verlag, 104–131.

[21] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Helping Programmers Improve the Energy Efficiency of Source Code. In *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C '17).* 238–240.

[22] 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.

[23] 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.

[24] 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.

[25] G. Pinto, K. Liu, F. Castor, and Y. D. Liu. 2016. A Comprehensive Study on the Energy Efficiency of Java's Thread-Safe Collections. (Oct 2016), 20–31. https://doi.org/10.1109/ICSME.2016.34

[26] 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.

[27] 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 First Int. Workshop on.* IEEE, 55–61.

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

[29] Cagri Sahin, Philip Tornquist, Ryan McKenna, Zachary Pearson, and James Clause. 2014. How Does Code Obfuscation Impact Energy Usage?. In *Software Maintenance (ICSM), 2013 29th IEEE Int. Conf. on.* IEEE.

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

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

[32] 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.

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

[34] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proc. of the 8th Int. Conf. on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2010, part of ESWeek '10 Sixth Embedded Systems Week, Scottsdale, AZ, USA, October 24-28, 2010.*