



Programming languages ranking based on energy measurements

Alberto Gordillo¹ · Coral Calero¹ · Ma Ángeles Moraga¹ · Félix García¹ ·
João Paulo Fernandes² · Rui Abreu³ · João Saraiva⁴

Accepted: 28 June 2024

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

Abstract

Software is developed using programming languages whose choice is made based on a wide range of criteria, but it should be noted that the programming language selected can affect the quality of the software product. In this paper, we focus on analysing the differences in energy consumption when running certain algorithms that have been developed using different programming languages. Therefore, we focus on the software quality from the perspective of greenability, in concrete in the aspects related to energy efficiency. For this purpose, this study has conducted an empirical investigation about the most suitable programming languages from an energy efficiency perspective using a hardware-based consumption measurement instrument that obtains real data about energy consumption. The study builds upon a previous study in which energy efficiency of PL were ranked using a software-based approach where the energy consumption is an estimation. As a result, no significant differences are obtained between two approaches, in terms of ranking the PL. However, if it is required to have a more realistic knowledge of consumption, it is necessary to use hardware approaches. Furthermore, the hardware approach provides information about the energy consumption of specific DUT hardware components, such as, HDD, graphics card, and processor, and a ranking for each of component is elaborated. This can provide useful information to make a more informed decision on the choice of a PL, depending on several factors, such as the type of algorithms to be implemented, or the effects on power consumption not only in overall, but also depending on specific DUT hardware components.

Keywords Energy consumption · Programming languages · Ranking · Green software · Hardware measurements

✉ Alberto Gordillo
alberto.gordillo@uclm.es

Extended author information available on the last page of the article

1 Introduction

Most commonly, software applications are developed using high-level, general-purpose programming languages (PLs). Traditionally, stakeholders select programming languages for a software project based on a wide range of criteria. These criteria encompass both technical and non-technical factors, such as abstraction level, project domain (frontend, backend, data science, etc.), expressiveness, documentation, learning curve, library and tool support, and more. As a result, in practice, a great diversity of programming languages is observed in projects (Bissyandé et al., 2013), which in many cases adopt a multilingual approach (Yang et al., 2024). Such as discussed in Section 2, the choice of a suitable programming language (or a set of them) for a software project is, therefore, an important decision that impacts critical aspects such as development productivity and the overall quality of the final product (Bhattacharya & Neamtiu, 2011; Muna, 2022; Ray et al., 2014). In fact, software greenability can be considered as another characteristic of software product quality that can be included in the quality model proposed by ISO/IEC 25010 (IEC, 2011). As proposed in (Calero et al., 2015) software greenability is the degree to which a product lasts over time, optimising the parameters, the amounts of energy and the resources used. Therefore, it is of great importance to assess programming languages in accordance with the amount of energy that the software needs to be run in order to facilitate their selection for each specific context. Yet, there is one criterion that has been largely overlooked in related literature —the carbon footprint of software solutions produced by the chosen programming language(s). According to (Pereira et al., 2017), it is crucial to consider energy efficiency as a key factor when selecting a programming language. While there are other studies emphasizing the importance of considering the energy consumption of programming languages, such as (Lima et al., 2016) and (Pereira et al., 2016), it's worth noting that the number of such studies remains relatively small. The increasing significance of this topic is further emphasized by today's society's high dependence on software, as we rely on various applications for our daily and professional activities. This extensive use of software has led to a significant increase in ICT (Information and Communication Technology) energy consumption. By 2025, it is projected to exceed 20% of the total energy consumption, contributing to up to 5.5% of the world's carbon emissions (Andrae, 2019; Vidal, 2017). While the hardware components of ICT systems are the ones responsible for energy consumption and the resulting emissions, it's the software they run that dictates how this consumption occurs. Consequently, to achieve energy-efficient ICT systems, it's essential to optimize not only their hardware components but also their software components. To capture the energy consumption of software applications, two distinct approaches can be employed: a software-based approach and a hardware-based approach. The former utilizes software tools to estimate energy consumption and analyse resource usage, offering a cost-effective and user-friendly option. In contrast, the latter relies on measurement hardware devices, which yield real-time and highly precise energy consumption measurements, albeit at a higher cost.

Therefore, given the potential importance of making informed decisions when selecting the most suitable programming languages in each context from an energy efficiency perspective, this study has conducted an empirical investigation using a hardware-based consumption measurement instrument. This study builds upon the previous study carried out by (Pereira et al., 2021), in which 27 programming languages based on their energy efficiency were ranked under a software-based approach using RAPL. Our motivation for this research is threefold, as we aim to answer the following research questions:

RQ1) Does the use of hardware-based measurements, when compared with a (software-based) estimation approach, influence ranking programming languages by energy consumption?

RQ2) How do programming languages rank based on the energy consumption for different algorithms?

RQ3) How do programming languages rank with respect to the energy consumption of several DUT hardware components (HDD, graphics card and processor)?

To answer the stated RQs, we selected a set of fourteen programming languages and six algorithms, the ones that run correctly for all the selected languages. The measuring device used was EET (Energy Efficiency Tester), a hardware device that has been used in previous studies with similar goals, and which is able to obtain the software's energy consumption measurements when running on a DUT (Device Under Test).

The main contributions of our work are the following:

1. We corroborate the already existing ranking by adding a new point of empirical evidence.
2. We assess the actual consumption of the PLs with a hardware-based approach versus the obtained previous study results which were based on estimations.
3. We delve deeper into the analysis of energy consumption of PLs, as it is essential to consider not only the overall consumption of each language but also various DUT hardware components of interest, such as the processor, graphics card, and hard drive.

The remainder of this article is structured as follows: In Section 2, we explore the state of the art and the background related to this work. In Section 3, we introduce the FEETINGS framework, which provides the hardware measurement environment used in the study. The results obtained are analysed in Section 4. Section 5 discusses the threats to the validity of the study, and finally, in Section 6, we present the conclusions and proposed lines of future work.

2 Related work

Traditionally, programming language evaluation has focused on productivity and quality dimensions. As reported in (Ray et al., 2014), empirical studies can be classified in: (i) Controlled Experiments, being examples of these studies: comparison of static and dynamic typing in PLs on quality and productivity (Hanenberg, 2010; Kleinschmager et al., 2012); effects of the type of language (procedural, imperative, functional,...) on reliability, quality and/or development effort (Harrison et al., 1996; Pankratius et al., 2012); (ii) Surveys, focusing on aspects such as capturing developers' opinions on why some languages are more popular than others (Meyerovich & Rabkin, 2013), or which PLs are most used by hackers (Koch et al., 2022); (iii) Repository Mining, addressing the PLs effects of PLs on software reliability (Koch et al., 2022), productivity and/or quality (Bhattacharya & Neamtiu, 2011; Kochhar et al., 2016), security challenges, among others.

However, comparative studies on the effects of programming languages on energy consumption are in a more incipient state. In more general terms, the unquestionable influence of software on energy consumption is fostering the research in the field of green software (Kern et al., 2018; Penzenstadler et al., 2014; Pinto & Castor, 2017). In the last few years, several works discuss what is and how to address software

sustainability by identifying the need to assess its energy consumption (Becker et al., 2015; Calero & Piattini, 2015; Fonseca et al., 2019; Naumann et al., 2011; Pinto & Castor, 2017; Venters et al., 2014). However, as stated by (Pang et al., 2015), programmers do not have much experience in software energy consumption. Pinto and Castor (Pinto & Castor, 2017) conducted a survey with software developers to know their perceptions of software energy consumption issues and concluded that they do not completely understand how to write, maintain and evolve energy-efficient software systems. Currently, software developers have to rely on Q&A websites, blog posts or YouTube videos when trying to optimise energy consumption, but these resources are not supported by empirical evidence, and may even be inaccurate (Li & Halfond, 2014; Manotas et al., 2016). Fortunately, some efforts have been made to obtain more reliable results. (Pereira et al., 2020) propose an energy leakage location technique, called SPELL, to identify inefficient energy consumption in the source code of software systems. (Chowdhury et al., 2019) propose the GreenOracle model, which is based on dynamic traces of system calls and CPU utilisation, to estimate software energy consumption. The model was subsequently improved, resulting in the GreenScaler model (Chowdhury & Hindle, 2016). Other addressed lifecycle stages have been design, as for instance about how software architecture influence on energy consumption (Cabot et al., 2019; Guamán & Pérez, 2021); and maintenance (Manotas et al., 2016) (Cruz et al., 2019) (Mancebo et al., 2021a, b).

More focused on LPs energy consumption evaluation, typically studies have tackled the topic by analysing one or a few of them as target. In (Ray et al., 2014), it is examined how different programming languages and compiler optimization flags affect energy consumption. The energy efficiency of C, C++, Java, Python and various compiler flags in the context of Fast Fourier Transform, Linked List Insertion/Deletion, and Quicksort implementations were assessed. Real-time power consumption data from the Intel Power Governor library highlighted that improved performance in sequential code typically reduces energy usage, with the choice of programming language playing a pivotal role. Optimized C++ often outperformed Java and even optimized C in specific scenarios, with -O3 and -O2 compiler optimization flags enhancing energy efficiency. Other studies are focused on a specific PL such as the study in (Koch et al., 2022) about examining the energy efficiency of programs written in Haskell, for which dual-pronged approach is employed to see the impact of strictness and concurrency on energy consumption. RAPL was applied to obtain the estimation of energy consumption. The results revealed that subtle changes can have significant effects on energy efficiency, as for instance selecting the MVar data sharing primitive over TMVar can result in a 60% energy saving. As a result, some green guidelines for Haskell programs development are suggested. Moreover, it was concluded that performance does not consistently correlate with energy consumption in concurrent scenarios as compared with sequential ones. In the research by (Corral-García et al., 2021) the challenges faced by beginners and non-expert programmers in the realm of high-performance computing (HPC) are addressed by introducing a transcompiler to enhance the performance of their HPC routines and tasks written in C or C++. The findings highlight the impact of even small code fragments on execution times and suggest that these techniques can significantly benefit HPC users, particularly for long-running or resource-intensive computations. More specifically, in HPC context, energy consumption of C optimisations is addressed in (Kelefournas & Djemame, 2019) by using an energy estimation model. Other studies also remark the important of an optimal choice of hardware platform and programming language, taking into consideration the intended use and required computing power (Choroszucho et al., 2019; Georgiouet al., 2018). In

particular, C/C++ and Python, were tested to assess their suitability for implementing solutions involving hardware microcontroller-based platforms.

A more complete comparative among PLs effect on energy consumption is provided by (Pereira et al., 2021), who proposed to the best of our knowledge the first classification of this type, studying the energy efficiency of twenty-seven programming languages when applied to ten different programming problems. The relevance and interest on the proposed ranking may be witnessed not only by the significant number of citations made on the article describing it (Pereira et al., 2017) but also by the numerous discussion threads one can find about both the ranking and the paper (Pereira et al., 2017). This ranking was established based on the benchmark solutions obtained from the Computer Language Benchmark Game (CLBG) ('The Computer Language Benchmarks Game', n.d.). In (Pereira et al., 2021), the authors extended their previous work (Pereira et al., 2017) by studying an additional set of benchmark problems. These benchmarks were taken from a chrestomathy program repository called Rosetta Code ('Rosetta Code', n.d.) therefore providing a complementary analysis based on solutions that might be more idiomatic about the considered programming languages and/or representative of real-world code. This complementary analysis confirmed, in essence, the previously established general ranking, therefore providing an additional level of validation to it.

The background study by (Pereira et al., 2021) motivated and inspired this research, which shares the following similarities: (s1) the target PLs of this research are a representative subset of the former study; (s2) the selected algorithms and test cases are the same in both studies; (s3) the test cases were executed on a DUT with similar characteristics. In addition to adding a further point of empirical evidence to the existence ranking with a hardware-based measurement approach, this study aims to add further contributions, mainly by: (c1) facilitating a better awareness of the energy consumption impact of the selected PLs with actual consumption values obtained with a hardware measuring instrument; (c2) to analyse the impact of PLs on the energy consumption of specific hardware components.

3 Methods: applying the FEETINGS framework to the ranking of programming languages

In order to make our hardware-based study reliable and replicable, we have followed FEETINGS (Mancebo et al., 2021a, b), a Framework for Energy Efficiency Testing to Improve Environmental Goals of the Software. In this section we present the instantiation of FEETINGS to our study.

3.1 The FEETINGS framework

FEETINGS (Mancebo et al., 2021a, b) is a framework to promote reliable capture and analysis of software energy consumption data and is composed of:

- The **conceptual component**, which consists of an ontology, called GSMO (Green Software Measurement Ontology), containing the concepts related to the software energy measurement.
- The **methodological component**, which includes GSMP (Green Software Measurement Process), a process with the required steps to measure the software's energy

consumption in a systematic way. GSMP has been built from the conceptualization provided by GSMO. GSMP has been modelled with SPEM v2 (OMG, 2008) (Software & Systems Process Engineering Meta-Model Specification) and implemented as an electronic process guide by using Eclipse Process Framework Composer (available at: <https://alarcos.esi.uclm.es/FEETINGS>). GSMP consists of seven phases which are divided into different activities, as illustrated in Fig. 1 and further described next.

1. Scope Definition, which starts specifying the requirements for the evaluation of energy efficiency and identifying and describing the software to be analysed (software entity).

- Green Software Measurement Process (GSMP) - Phase I: Scope Definition - A1.1 Elicitation of requirements - A1.2 Define the goal - A1.3 Choose a Software Entity - A1.4 Development of Test Cases - Phase II: Measurement Environment Setting - A2.1 Select a measuring instrument - A2.2 Define specifications of the DUT - A2.3 Select a set of the measures provided by MI - A2.4 Close unnecessary software applications and processes - A2.5 Obtain the baseline energy consumption of the DUT - Phase III: Measurement Environment Preparation - A3.1 Close unnecessary software applications and processes - A3.2 Determine number of repetitions of measurements - A3.3 Configure the testbed - Phase IV: Perform the Measurements - Iteration - A4.1 Measure the energy consumed - A4.2 Collect the raw data - Phase V: Test Case Data Analysis - Iteration - A5.1 Prepare and describe the raw data to be analyzed - A5.2 Statistical analysis of test case measurement - Phase VI: Software Entity Data Analysis - A6.1 Calculate the energy consumption by the software entity - A6.2 State conclusions about the software entity's energy consumption data - Phase VII: Reporting the results - A7.1 Carry out the laboratory package - A7.2 Document the case study

Fig. 1 GSMP phases summary

- As an example, in this study, the software entities are the programming languages that we are going to analyse. At the end of this phase, the representative test cases to be run for the analysis of the energy consumption of the software entity are defined.
2. **Measuring Environment Setting.** Firstly, the selection of the measuring instrument to obtain the energy data of the software entities to be analysed takes place, which may be a hardware device or a software tool. In the particular case of this study, the EET (Energy Efficiency Tester) hardware meter has been chosen. Then, the Device Under Test (DUT), i.e. the hardware where the tests cases will be run, is specified. Next, the definition or selection of the set of measures to be used for the analysis is addressed, such as the energy related ones (total energy consumption, consumption by DUT component, etc.), which can be complemented with other measures such as code complexity. The baseline power consumption, i.e. the idle power consumption of the DUT, is also determined during this phase. To ensure accurate results, it is essential to ensure that no software is running in the background and all services and processes that could affect the measurement are stopped.
 3. **Measurement Environment Preparation,** which is initiated as for the baseline power consumption measurement, checking that no software is running in the background and that all services and processes that could affect the measurement are stopped. Once this is completed, the number of times each measurement should be repeated is established. This choice depends on the stated objective and the available (Dirlewanger, 2006; Pinto & Castor, 2017) we recommended to define 20-30 measurements to have a wide enough sample size (central limit theorem) (Kern et al., 2018). As a result, in this context a measurement is considered as a set of energy consumption samples from a single test case run. Finally, the testbed is conFig.d, where the software entities and the services required in the DUT need to be installed and the chosen software entities are prepared, so that they are ready to be evaluated by means of the defined test cases. The procedure also establishes that, once the measurements for one of the software entities are completed, the DUT must be restored, so that it returns to its initial state, and this procedure is repeated for the different software entities.
 4. **Performing the Measurements,** where the energy consumption measurements will be iteratively carried out and collected. Firstly, for each test case of software entity the measurements are carried out and the testbed is cleaned at the end of each iteration to avoid affecting the power consumption of the next test case to be run. Then, the raw energy consumption data taken from the measuring instrument are collected, by including the details of the DUT, the definition of the run test case, the current configuration, the start and end time of the run and the power values.
 5. **Test Case Data Analysis,** which consists of firstly turning the raw data into useful information for the analysis, namely by doing data wrangling as suggested in (Penzensadler et al., 2014). Once the data has been prepared for analysis, the descriptive statistics for each test case are obtained.
 6. **Software Entity Data Analysis,** which starts with the calculation of the energy consumption of the software entity, where the power of the baseline (phase 2) must be subtracted from the average power of the measurements of the software entity for each component (including the DUT). The consumption without the baseline is calculated (Eq. 1) by subtracting the average power of the baseline ($P_{Baseline}$) from the average power of the measurement ($P_{Measurement}$) and multiplying the result by the average duration of the measurement ($T_{Measurement}$):

$$\text{Energy Consumption without baseline} = (P_{\text{Measurement}} - P_{\text{Baseline}}) * T_{\text{Measurement}} \quad (1)$$

The energy consumptions presented in this study does not include baseline energy consumption, which is that consumed by the operating system and the components (network card, sound card, among others) that, even while not being used, consume a minimum amount of energy.

7. Reporting the Results, which focuses on the development of the resulting laboratory package (LP) to promote the replicability of the empirical study conducted (Brooks et al., 1996), (Basili et al., 1986) and (Solari et al., 2018) on documenting the process and results obtained in the study to be sent to the client and other stakeholders. Guidelines such as the proposed by Jedlitschka and Pfahl (Jedlitschka & Pfahl, 2005) can be followed for this purpose.

Technological component, which supports the process and includes EET (Energy Efficiency Tester) and ELLIOT. EET is shown in Fig. 2. This is a hardware-type measuring instrument with a sampling frequency of 100 Hz, i.e. it captures 100 energy data points per second. EET is able to obtain the energy consumption of different components of the DUT, for which it includes internal sensors to capture energy data of the processor, graphic card and hard disk, along with one external sensor for the monitor and to obtain the total consumption of the DUT (without including the monitor consumption). To obtain the consumption data, the sensors are connected to the energy distribution lines that feed these components.

3.2 Application of GSMP to this study

Table 1 summarises the application of GSMP phases and activities to our study, conducted during 2021. For the study, a subset of 14 out of the 27 programming languages analysed in (Pereira et al., 2021) was used. These programming languages were selected taking into account that compiled languages, interpreted languages and languages running on virtual machines had to be chosen. We selected six algorithms to our study, the ones that run correctly for all the selected languages. The compilers used on the study were the same than

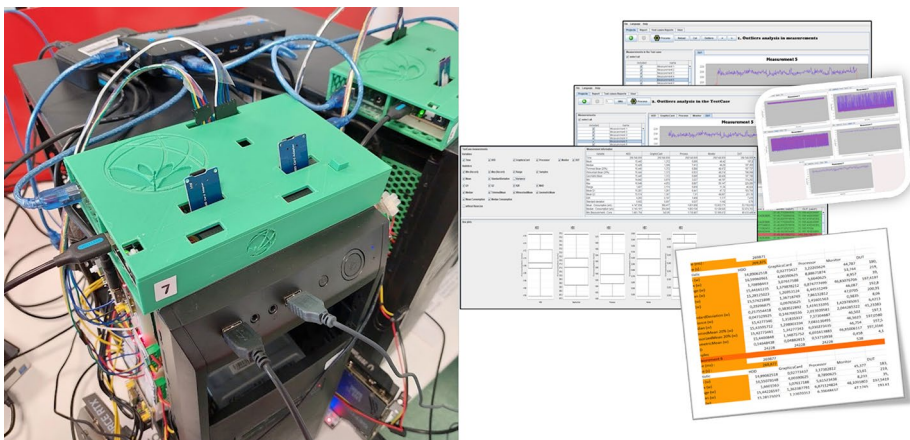


Fig. 2 EET connected to a DUT (Device Under Test) and ELLIOT reports

Table 1 GSMP process applied to this study

Phase	Application
I. Scope Definition	Software Entity Class: Programming Language Software Entities: C, C++, Java, Ada, Pascal, Haskell, JavaScript, Dart, PHP, Erlang, JRuby, Ruby, Perl and Python Test cases Selected algorithms of the Computer Language Benchmarks Game (CLBG): • Binary trees • Fannkuch-redux • Fasta • Mandelbrot • N-body • Spectral-norm Test case run Each algorithm coded in each one of the programming languages Hardware measuring instrument: EET (Energy Efficiency Tester) Device Under Test (DUT): Monitor Motherboard Processor RAM Graphics card Hard Disk Power supply O.S Measures: Execution time DUT Energy Consumption Processor energy consumption Graphics Card energy consumption Hard disk energy consumption

Table 1 (continued)

Phase	Application
III. Measurement Environment Preparation	For each programming language under study: • Check that the DUT is clean • Check that there is not any software running in the background • Install the corresponding compiler
IV. Performing the measurement	For each test case run (PL x algorithm): • Compile the algorithm from the makefile • Repeat: 30 times • Sequentially execute the algorithm 10 times (to facilitate the analysis of time differences) except Perl, Python and Ruby • Obtain the measures results
Phases III and IV are repeated for each one of the programming languages	
V. Test Case Data Analysis	Analyse the energy consumption data for each test case Check that the measurements are correct (outliers, wrong executions and so on) and eliminate the wrong measures if it is necessary
VI. Software Entity Data Analysis	For each programming language: • Calculate the mean of the energy consumption for each algorithm and for each component (HDD, processor and graphic card) • Calculate the mean of the energy consumption for the programming language taking into account the mean of energy consumption of all the algorithms • State conclusions (see next section of this paper)
VII. Reporting the result	This paper

in the previous study to minimize bias. These algorithms are described in Table 2. Thus, the software entity class was defined as the programming languages, whereas the software entities were the 14 programming languages selected and the test cases were six algorithms selected. Finally, the test cases run is defined as the implementation of each algorithm on the different programming languages.

Regarding the DUT used for this study (to see Table 1), it is important to note that the most similar configuration possible was sought, although as a result it was not possible to have a DUT identical to that of the SwE study. Namely, both studies were conducted on Linux, with SwE utilising Linux Ubuntu Server 16.10 and HwM using Ubuntu 16.04.2 LTS. In relation to the hardware, the SwE study involved a Haswell Intel(R) Core(TM) i5-4460 CPU @ 3.20 GHz with 16 GB of RAM. As they are not exactly the same, variations in the results are to be expected. For this reason, we only compare the relative positions of the languages in the rankings of the two studies.

In the next section we present the analyses from the obtained measurement results (Phase VI).

4 Results

This section presents the energy consumption results obtained for each programming language using the hardware measurement device and following the steps defined in GSMP. The aim is to analyse the results in order to produce a ranking of programming languages according to their consumption by using the EET hardware meter (referred as HwM) and compare it with the obtained previous findings (Pereira et al., 2021) in which a software estimator was used (referred as SwE). In addition to a general comparison at the level of the programming languages, further comparisons considering the type of language (compiled, interpreted, and virtual machine) and the paradigm used (object-oriented, scripting, imperative, and functional) are carried out. It is also checked whether there are differences in energy consumption between different programming languages depending on the algorithm they solve (Binary-trees, Fannkuch-redux, Fasta, Mandelbrot, N-body, and Spectral-norm).

Taking into account that the hardware measurement device also provides energy consumption data at the level of some of the computer's hardware component, in particular for the hard disk, graphics card and processor, the differences in consumption between the different components are discussed.

The experimental package can be found at the repository of this study (Gordillo et al., 2024).

4.1 Comparison of SwE and HwM studies

This section discusses the general comparison of the rankings obtained in HwM and SwE studies, analysing the overall and per-algorithm energy results, respectively.

4.1.1 Overall ranking

Firstly, we compare programming languages based on the average time and energy used to solve analysed algorithms. Table 3 shows the global mean of time and energy for each programming language (PL) evaluated with each approach (SwE, HwE).

Table 2 Algorithms implemented in the test cases

Algorithm	Description	Data size
Binary-trees	Allocate and deallocate many binary trees	21
Fannkuch-redux	Indexed-access to tiny integer-sequence. It is used to evaluate the performance of computer systems and implementations of permutation algorithms	12
Fasta	Generate and write random DNA sequences. It works by heuristically searching and comparing short segments of DNA sequences to calculate their similarity	25,000,000
Mandelbrot	It is an iterative method for generating fractals, specifically the Mandelbrot set	16,000
N-body	Is a numerical method used to simulate the motion of N interacting particles in a physical system, such as planets in a solar system, considering the gravitational forces between them	50,000,000
Spectral norm	The Spectral-Norm algorithm calculates the largest singular value of the matrix used to measure its magnitude. This algorithm is used as a performance test to evaluate implementations of numerical algorithms	5,500

Table 3 Mean values of energy and time of the 14 program languages SwE vs. HwM

PL	Time SwE (s)	Time HwM (J)	Energy SwE (J)	Energy HwM (J)	Type	Paradigm
C	2.364	1.201	66.643	206.135	Compiled	Imperative
C++	2.286	1.227	67.162	219.207	Compiled	OO, Imperative
Ada	3.501	1.941	96.143	312.421	Compiled	OO, Imperative
Java	3.992	1.904	113.273	337.046	Virtual Machine	OO
Pascal	7.601	4.486	157.002	671.455	Compiled	Imperative
Haskell	8.236	5.293	206.918	684.366	Compiled	Functional
JavaScript	13.367	8.203	192.353	851.364	Interpreted	OO, Scripting
Dart	13.773	8.765	199.232	956.296	Interpreted	OO, Scripting
PHP	78.338	39.583	2380.527	6462.751	Interpreted	OO, Scripting
Erlang	83.407	41.769	2805.190	6868.359	Virtual Machine	Imperative, Functional
JRuby	113.818	80.307	3527.398	12,492.886	Interpreted	Scripting
Ruby	166.949	105.054	5636.157	17,183.644	Interpreted	Scripting, Functional
Python	143.993	120.475	6286.523	19,067.487	Interpreted	OO, Scripting
Perl	191.956	115.179	6641.842	19,229.640	Interpreted	OO, Imperative, Functional

As can be observed, the time and energy values obtained in SwE and HwM are different, which was expected as one study is based on estimates and the other on actual consumption measurements. Furthermore, although we have maintained as many common variables as possible to replicate the original SwE study, the difference can be also explained by the fact that the computer used to run the test cases is not exactly the same.

Despite the differences in the data obtained, Fig. 3 shows that the ranking of programming languages (from lowest to highest energy consumption) is nearly the same in both studies. The left-hand side of Fig. 3 shows the position obtained by each language using SwE, while the right-hand side shows the ranking based on the HwM results. Haskell has achieved better results, moving up two positions in the ranking and pushing JavaScript and Dart down one position, although they remain in the same order. Table 3 also shows that Dart's results closely match to JavaScript's results in both rankings. Therefore, the difference is mainly due to the disparities of Haskell in execution time and energy consumption.

Next, Ward's hierarchical clustering method (Murtagh & Legendre, 2011) was used to group the languages with similar energy consumption. The RStudio tool was used for the clustering process and after analysing different options, a number of 5 clusters was chosen as the most suitable option. The main purpose of these clusters is to better understand which set of programming languages could be used to develop software with similar energy behaviour. The aim is therefore to provide a label similar to those used in other areas, such as the energy efficiency of a home or household appliances. For this purpose, the left-hand side of Fig. 4 shows the clusters obtained, while the right-hand side presents the same information but in graphical form, identifying the tier list for the overall ranking of PLs, based on the position of the languages in the consumption ranking. As can be seen, each level is labelled with the letter A, B, C, D or E being A the most efficient and E the least efficient, and to each label is assigned a colour, namely dark green, light green, yellow, orange and red, respectively.



Fig. 3 Ranking of the 14 languages SwE vs. HwM

Some conclusions can be drawn from these results. On the one hand, although all compiled languages are ranked with the label A, C and C++ are more efficient than Pascal and Haskell. Regarding interpreted languages, JavaScript and Dart stand out as energy-efficient options, while Perl, Ruby and Python appear to be less energy-efficient. Among virtual machine languages, Java is the best choice, while Erlang is the worst. Therefore, in general, compiled languages have better energy performance.

On the other hand, depending on the type of paradigm, it can be concluded that in the object-oriented paradigm, C++ is the best option, followed by Ada and Java, while Python and Perl are the least recommended. In the scripting paradigm, JavaScript and Dart are highly recommended, while JRuby, Ruby and Python are less recommended. In the Imperative paradigm, C, C++ and Ada are preferred, with a small advantage of C, while Pascal

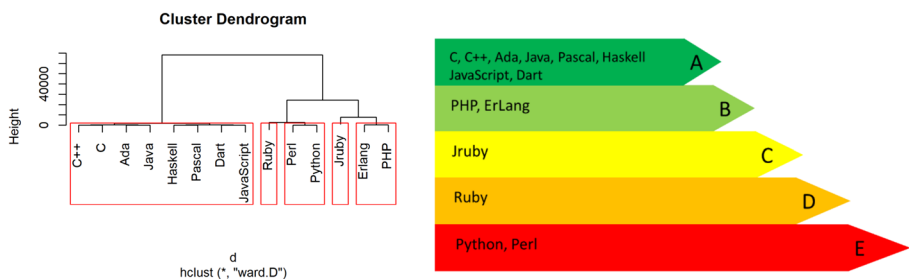


Fig. 4 Clusters in Programming languages for DUT energy consumption (a) and tier list (b)

is not recommended. Finally, in the functional paradigm, Haskell is the lowest energy consuming language, while Perl and Ruby are the least recommended.

4.1.2 Comparison by algorithm

In this section, programming language rankings obtained in SwE and HwE studies are assessed from algorithm focus, i.e., to check whether there are differences between the energy consumption of the selected PLs depending on the type of algorithm being solved.

First, Table 4 shows the results of the current study (HwM), showing the position obtained by each PL in each algorithm as well as the global ranking (considering the energy consumption of all algorithms). The background colours indicate the cluster in which each language has been classified for each algorithm. The colours follow the same conventions as the label colours shown in the previous section.

As it can be observed in Table 4, the energy performance that a language exhibits in general also corresponds to this performance in a large part of the algorithms analysed. However, in some cases, some differences are observed (e.g. C and C++ exchange positions for the Spectral-Norm algorithm, as well as Ada and Java for the Binary-trees and Fasta algorithms; Ruby is ranked tenth for the binary-trees algorithm but reaches the last position for the fankuch-redux algorithm; Haskell ranks third for the spectral-norm algorithm, but eighth for the fasta and n-body algorithms), which gives us an interesting insight into the need, not only to consider one PL or another when making a decision, but also to take into account the type of problem to be solved.

Analysing by language type the results in Table 4 show that the best compiled languages are C and C++. In terms of virtual machine languages, Java is the only one that has achieved good results regardless of the algorithm it solves. The only interpreted PLs that have emerged as a good option in the 6 algorithms studied are JavaScript and Dart. So, in general, these are the only two PL that we can recommend among the interpreted program languages, but they are far from the compiled ones. Therefore, from the previous analysis it can be confirmed that compiled PLs are generally a better option than interpreted PLs.

From the perspective of paradigms, the most optimal languages are those that employ the imperative style: C, C++, and Ada. Additionally, object-oriented languages, such as C++, Ada, and Java, also demonstrate considerable energetic efficiency. The remaining

Table 4 Ranking of the 14 languages

PL	Binary-trees	Fankuch-redux	Fasta	Mandelbrot	N-body	Spectral-norm	Overall	Type	Paradigm
C	1	1	1	1	1	2	1	Compiled	Imperative
C++	2	2	2	2	2	1	2	Compiled	OO, Imperative
Ada	4	3	4	3	3	6	3	Compiled	OO, Imperative
Java	3	4	3	4	5	5	4	Virtual machine	OO
Pascal	7	5	6	8	4	4	5	Compiled	Imperative
Haskell	6	6	8	6	8	3	6	Compiled	Functional
JavaScript	8	7	5	5	6	7	7	Interpreted	OO, Scripting
Dart	9	8	7	7	7	8	8	Interpreted	OO, Scripting
PHP	12	10	9	9	11	10	9	Interpreted	OO, Scripting
Erlang	5	9	10	11	10	9	10	Virtual machine	Imperative, Functional
JRuby	11	13	12	14	9	13	11	Interpreted	Scripting
Ruby	10	14	13	12	12	12	12	Interpreted	Scripting, Functional
Python	13	12	14	10	13	14	13	Interpreted	OO, Scripting
Perl	14	11	11	13	14	11	14	Interpreted	OO, Imperative, Functional

paradigms achieve their optimal positions in the middle of the table, with Haskell leading in functional programming and JavaScript and Dart excelling in scripting. The ranking of PLs by type and paradigm can be consulted at the complementary material included in the repository (Gordillo et al., 2024).

On the other hand, Fig. 5 shows the comparison between studies.

Such as can be observed in Fig. 5, the results of the HwE study largely confirm the ranking obtained in SwE. There is only some fluctuation in the position of some PLs, especially among languages that have obtained very close data and therefore small energy variations modify their position in the ranking. The detailed energy and time data of both studies are shown in Appendix A (Table 5).

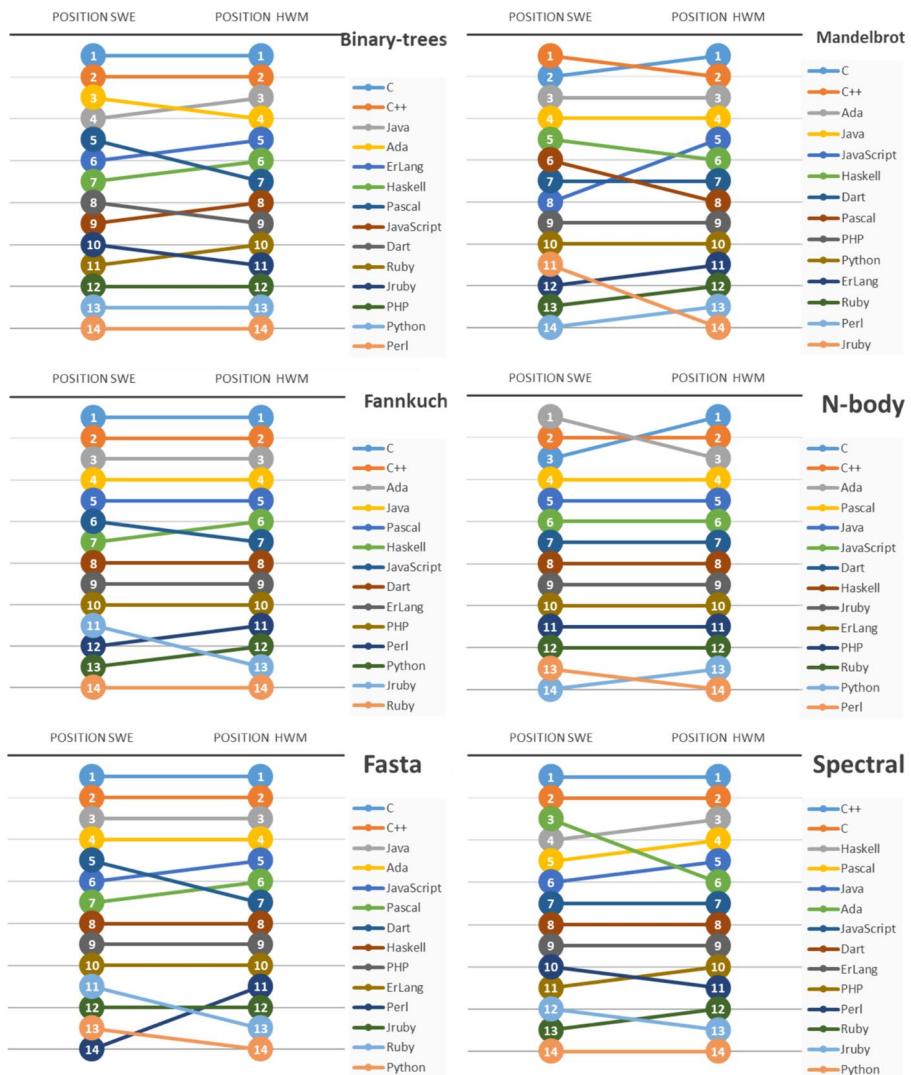


Fig. 5 Comparison of rankings between SwE and HwM data by PL and algorithm

4.2 Components comparison

In this section we move the focus to a more detailed analysis of consumption per DUT hardware component (HDD, graphics card and processor), so that we can determine how much these hardware components are affecting the overall consumption of a PL (Section 4.2.1) and the consumption of a PL applied on a specific algorithm (Section 4.2.2).

4.2.1 Overall ranking

Table 5 shows the results of the energy consumption of each PL for each computer component. It is also shown the total consumption at computer level, i.e. DUT column, which was previously analysed in Section 4.1. To complement this data, Fig. 6 illustrates graphically the portion of the total consumption (DUT) that is due to each component, which confirms as expected, that the highest consumption was due to the processor. The energy consumption of the HDD and graphics card had little influence compared to the processor, which can be explained for the characteristics of the benchmark algorithms, for instance, which do not require much HDD access along with minimal interaction by means of User Interfaces.

Based on the above results, new rankings for each computer component have been elaborated, as shown in Fig. 7. The left side of the figure shows the position of each PL according to its total consumption (DUT) in relation to the position of that PL according to its consumption of graphics card (left side) and processor (right side). As it can be noted the position differences between the Device Under Test (DUT) and the processor are minimal. These differences are notable in the cases of Haskell-JavaScript-Dart, mainly due to Dart's efficient processor use. In contrast, the two lower positions, Python-Perl, show minimal differences.

Table 5 Mean values of energy per component for the 14 program languages

PL	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)	Type	Paradigm
C	206.135	7.632	15.918	72.202	Compiled	Imperative
C++	219.207	3.521	32.129	83.957	Compiled	OO, Imperative
Ada	312.421	26.849	21.953	96.203	Compiled	OO, Imperative
Java	337.046	11.125	17.922	108.070	Virtual machine	OO
Pascal	671.455	21.021	27.208	218.353	Compiled	Imperative
Haskell	684.366	27.320	38.009	277.230	Compiled	Functional
JavaScript	851.364	39.497	43.357	310.059	Interpreted	OO, Scripting
Dart	956.296	44.519	53.239	263.375	Interpreted	OO, Scripting
PHP	6462.751	240.212	284.506	1896.346	Interpreted	OO, Scripting
Erlang	6868.359	226.986	276.987	2066.120	Virtual machine	Imperative, Functional
Jruby	12,492.886	419.669	831.175	4079.473	Interpreted	Scripting
Ruby	17,183.644	628.897	1308.912	6035.509	Interpreted	Scripting, Functional
Python	19,067.487	610.824	846.523	6848.738	Interpreted	OO, Scripting
Perl	19,229.640	632.679	823.432	6278.473	Interpreted	OO, Imperative, Functional

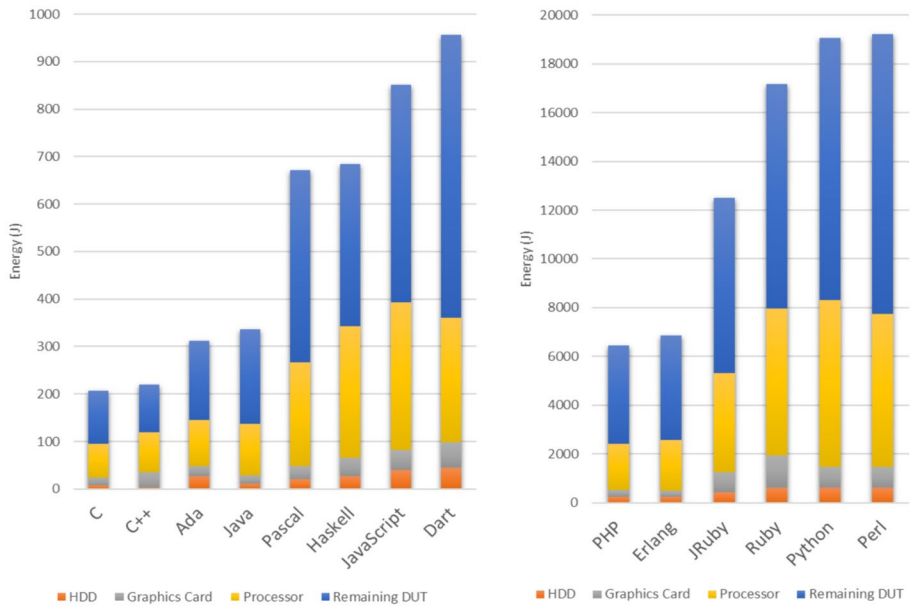


Fig. 6 Percentage relationship between components and DUT

When comparing the position obtained by the PLs taking into account the power consumption of the DUT with the position obtained according to the power consumption of the graphics card, we can observe that the behaviour of the PLs is different. For instance, C++ uses the graphics card more extensively than its close counterparts in the ranking, in contrast to Java. At the bottom of the figure, Ruby shows a higher use of the graphics card compared to its close language counterparts, while Python demonstrates either lower usage or more moderate consumption of the graphics card. This suggests that PLs make a different use of this resource for their computational operations, given that the benchmark programmes do not require, for example, an intensive use of the graphic card to interact with the user through a graphic interface.

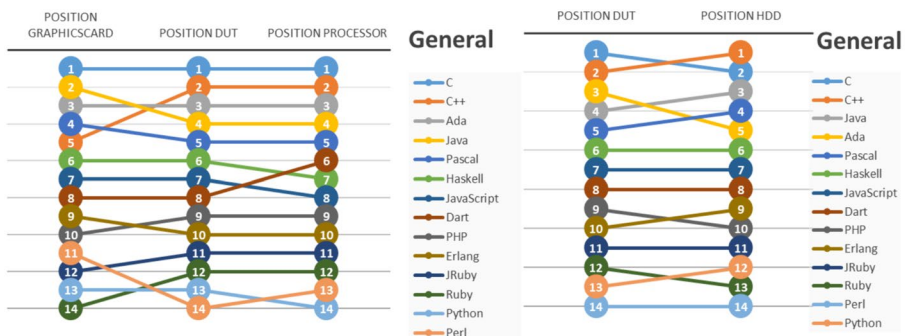


Fig. 7 Ranking of the 14 languages for components

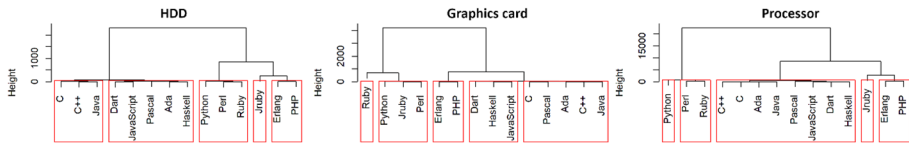


Fig. 8 Clusters in Programming languages for components

On the other hand, the right side of Fig. 7 shows the position obtained by the PLs due to overall energy consumption (DUT) with respect to their ranking due to their effect on HDD consumption. In this case there are notable variations. C uses the HDD more than C++, and Ada exhibits higher HDD usage compared to its close language counterparts in the ranking. In the lower positions of the ranking, Python demonstrates more moderate HDD consumption than its counterparts.

As for the rest of the results of this study the Ward's hierarchical clustering method (Murtagh & Legendre, 2011) has been applied to group the PLs by DUT hardware component (Fig. 8) and corresponding tier lists have been generated (Fig. 9).

As can be seen in Figs. 8 and 9, there is a fairly uniform behaviour in the ranking, with each language moving up or down by a maximum of one position. For example, JRuby gets the label D for HDD and graphics card tier list but gets the letter C for Processor, similarly Python is classified as E considering HDD and processor consumption but is labelled D for the graphics card.

4.2.2 Comparison by component and algorithm

This section compares the power consumption of each DUT hardware component depending on the programming language and the algorithm to be solved (detailed data can be found in Appendix C).

With regards to HDD, as Table 6 shows, in general there are not many fluctuations considering the consumption derived from the algorithm being solved. The most significant case is the Perl language when solving the spectral norm algorithm, as it obtains better results than for the rest of the algorithms. In terms of language type, the best compiled, virtual machine and interpreted languages are C++, Java and JavaScript respectively. From a paradigm perspective, C is the best option for imperative programming, while C++ performs best for the object-oriented paradigm. In the case of the functional paradigm, Haskell achieves the best results, and finally, among scripting languages, JavaScript obtains the best position.

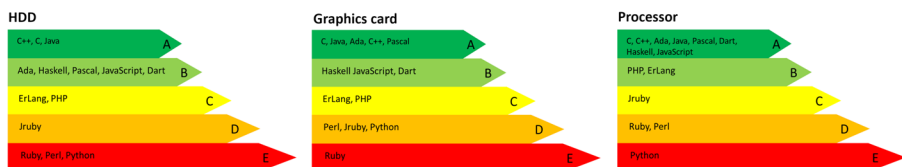


Fig. 9 Tier lists in Programming languages for components

Table 6 Comparison of HDD tiers list by algorithms

PL	Binary-trees	Fankuch-redux	Fasta	Mandelbrot	N-body	Spectral-norm	Overall	type	Paradigm
C++	1	1	1	1	1	2	1	Compiled	OO, Imperative
C	2	3	3	2	2	3	2	Compiled	Imperative
Java	3	2	2	4	4	5	3	Virtual machine	OO
Pascal	8	4	6	8	3	1	4	Compiled	Imperative
Ada	5	6	8	5	8	6	5	Compiled	OO, Imperative
Haskell	6	5	7	7	7	4	6	Compiled	Functional
JavaScript	9	7	4	3	6	7	7	Interpreted	OO, Scripting
Dart	10	8	5	6	5	8	8	Interpreted	OO, Scripting
Erlang	4	9	10	11	10	11	9	Virtual machine	Imperative, Functional
PHP	12	11	9	9	11	9	10	Interpreted	OO, Scripting
JRuby	11	13	12	13	9	13	11	Interpreted	Scripting
Python	13	12	14	10	14	14	12	Interpreted	OO, Scripting
Ruby	7	14	13	12	12	12	13	Interpreted	Scripting, Functional
Perl	14	10	11	14	13	10	14	Interpreted	OO, Imperative, Functional

Next, Table 7 shows the ranking according to the energy consumption of the graphics card. It can again be noted that compiled languages obtain better values than interpreted languages for the majority of the algorithms. However, there are some exceptions. For example, although Ruby is generally the worst language in terms of the graphics card energy consumption, for the binary-trees algorithm this consumption is better than other interpreted languages such as Perl, JRuby or Python. The same applies to the JRuby programming language and the N-body algorithm. In this case it consumes less than PHP, despite that in the overall ranking PHP obtains better position.

Despite these exceptions, there is a tendency for the graphics card to consume less energy when the algorithms are implemented using one of the top eight languages of the Table 7 (C-Dart), regardless of the algorithm. In terms of programming paradigms, C emerges as the best language for imperative programming, and Java for the object-oriented programming. In the functional paradigm, Haskell offers the most favourable results and, finally, for scripting languages, JavaScript is the best choice.

Table 7 Comparison of Graphics card tier lists by algorithms

PL	Binary-trees	Fankuch-redux	Fasta	Mandelbrot	N-body	Spectral-norm	General	Type	Paradigm
C	1	2	2	2	8	4	1	Compiled	Imperative
Java	2	4	3	3	5	5	2	Virtual machine	OO
Ada	3	1	7	5	6	6	3	Compiled	OO, Imperative
Pascal	6	3	6	7	2	1	4	Compiled	Imperative
C++	8	5	1	1	1	2	5	Compiled	OO, Imperative
Haskell	5	6	8	8	7	3	6	Compiled	Functional
JavaScript	7	7	4	4	3	7	7	Interpreted	OO, Scripting
Dart	10	8	5	6	4	8	8	Interpreted	OO, Scripting
Erlang	4	9	9	11	10	9	9	Virtual machine	Imperative, Functional
PHP	11	10	10	9	11	11	10	Interpreted	OO, Scripting
Perl	14	11	11	13	13	10	11	Interpreted	OO, Imperative, Functional
JRuby	13	13	12	14	9	13	12	Interpreted	Scripting
Python	12	12	14	12	12	14	13	Interpreted	OO, Scripting
Ruby	9	14	13	10	14	12	14	Interpreted	Scripting, Functional

Table 8 Comparison of Processor tier lists by algorithms

PL	Binary-trees	Fankuch-redux	Fasta	Mandelbrot	N-body	Spectral-norm	Overall	Type	Paradigm
C	1	1	1	1	1	4	1	Compiled	Imperative
C++	2	3	3	2	4	2	2	Compiled	OO, Imperative
Ada	3	2	4	3	2	6	3	Compiled	OO, Imperative
Java	4	4	2	4	5	5	4	Virtual machine	OO
Pascal	6	5	6	8	3	1	5	Compiled	Imperative
Dart	7	7	7	5	6	8	6	Interpreted	OO, Scripting
Haskell	8	6	8	7	8	3	7	Compiled	Functional
JavaScript	9	8	5	6	7	7	8	Interpreted	OO, Scripting
PHP	12	10	9	9	9	10	9	Interpreted	OO, Scripting
Erlang	5	9	10	10	11	9	10	Virtual machine	Imperative, Functional
IRuby	10	13	12	14	10	12	11	Interpreted	Scripting
Ruby	11	14	13	12	12	13	12	Interpreted	Scripting, Functional
Perl	14	11	11	13	14	11	13	Interpreted	OO, Imperative, Functional
Python	13	12	14	11	13	14	14	Interpreted	OO, Scripting

Finally, Table 8 shows the ranking obtained concerning the processor energy consumption. In this case, there is a uniform energy consumption behaviour, although some languages show significant variations. For example, the language Perl is classified as label B, C, D and E depending on the algorithm it solves. This variation indicates that, for this language, the type of algorithm it solves affects the processor energy consumption (or at least affects it more than in the other cases). On the other hand, the Erlang programming language achieves a good result for the binary tree algorithm, being better than the Pascal language, which is a compiled language and generally consumes less energy. Considering different programming paradigms, C emerges as the language of choice for imperative programming, while C++ highlight in the domain of object-oriented programming. In the case of functional programming, Haskell is the top PL, and finally, regarding scripting languages, Dart is the best.

Table 9 Comparison of average tier list by component

PL	DUT Average	HDD Average	Graphics card Average	Processor Average	Type	Paradigm
C	1	2	1	1	Compiled	Imperative
C++	2	1	5	2	Compiled	OO, Imperative
Ada	3	5	3	3	Compiled	OO, Imperative
Java	4	3	2	4	Virtual machine	OO
Pascal	5	4	4	5	Compiled	Imperative
Haskell	6	6	6	7	Compiled	Functional
JavaScript	7	7	7	8	Interpreted	OO, Scripting
Dart	8	8	8	6	Interpreted	OO, Scripting
PHP	9	10	10	9	Interpreted	OO, Scripting
Erlang	10	9	9	10	Virtual machine	Imperative, Functional
IRuby	11	11	12	11	Interpreted	Scripting
Ruby	12	13	14	12	Interpreted	Scripting, Functional
Python	13	12	13	14	Interpreted	OO, Scripting
Perl	14	14	11	13	Interpreted	OO, Imperative, Functional

4.2.3 Overall discussion

Finally, to summarise the results obtained in the HwM study, Table 9 shows the position of the PLs considering the overall consumption, i.e. the DUT consumption, the consumption of the hard disk, the graphics card and the processor as well as the colour of the cluster in which they have been classified.

As shown in Table 9, based on average energy consumption, the compiled languages C and C++ emerge as the top choices, closely followed by Ada. It can be highlighted that C++ has a higher consumption of the graphics card compared to its close counterparts. Java is the virtual machine language that has always been in the top four positions and has been labelled as A in all cases. JavaScript and Dart are the interpreted languages with the best behaviour and are in the middle of the rankings, achieving an A label for energy consumption of the DUT and processor and a B label for the hard disk and graphics card.

An analysis based on programming paradigms reveals that the most prominent positions are occupied by imperative languages, including C, C++, and Ada. These positions also encompass C++, Ada, and Java, which excel in the object-oriented paradigm. Haskell emerges as the best performing functional language, with JavaScript and Dart following as the top choices in the scripting paradigm.

Table 10 Effect size of the consumption (J) averages of the measurements by language

Language vs Language	HDD	Graphics Card	Processor	DUT
C++ vs C	small	small	medium	small
Erlang vs Dart	small	Small	small	small
Erlang vs Haskell	medium	medium	medium	medium
Erlang vs JavaScript	large	medium	medium	small
Erlang vs Pascal	medium	medium	medium	medium
Erlang vs PHP	small	Small	small	small
Haskell vs Dart	medium	Small	medium	small
Haskell vs Pascal	small	Small	small	small
Java vs C	small	Small	medium	small
Java vs C++	small	medium	small	small
JavaScript vs Ada	medium	Large	large	medium
JavaScript vs Dart	medium	Small	medium	small
JavaScript vs Haskell	small	Small	small	small
JavaScript vs Pascal	small	Small	small	small
JRuby vs Python	small	Small	small	small
Pascal vs Dart	small	Small	medium	small
Perl vs Erlang	medium	medium	medium	medium
Perl vs JRuby	medium	medium	small	medium
Perl vs PHP	small	Small	small	small
Perl vs Python	medium	Large	medium	large
Perl vs Ruby	small	Large	small	small
PHP vs JRuby	medium	Large	medium	medium
PHP vs Ruby	medium	Large	medium	medium
Ruby vs JRuby	small	Small	small	small
Ruby vs Python	small	Small	small	small

4.3 Statistical test of the study data

To confirm whether the differences obtained in this study can be considered statistically significant the non-parametric Mann-Whitney test was applied, given the non-normality of the data distribution according to the results of the Kolmogorov-Smirnov test. The stated null hypotheses is that there are no significant differences in the instant power (average) or consumption values (DUT and components) between the compared groups (by language or by language-algorithm). The details of these statistical tests and the obtained results are included in the provided experimental package (Gordillo et al., 2024). As the resulting p-value was less than 0.05 for all comparisons, the null hypothesis can be rejected. Furthermore, effect sizes were calculated by applying the r formula proposed by Cohen (Cohen, 2013) and adapted to the results of the Mann-Whitney test. The value of r is calculated as follows:

$$r = |z| * \sqrt{n} \quad (2)$$

Depending on the value of r , the effect is either small (if the value is less than 0.3), medium (if the value is between 0.3 and 0.5) or large (if the value is greater than 0.5). In the case of the comparison of LPs by mean consumption, for most of the compared groups the effect size was large, as reported in the experimental package (Gordillo et al., 2024). In fact, of the 91 cross-language comparisons, the effect size was medium in 24 cases (26%) for HDD, Processor and DUT, and in 20 cases (22%) for Graphics Card. Table 10 shows the results obtained for the compared groups that did not have a large effect size for the different hardware components. Finally, it is worth noting that the following pairs of LPs have obtained a small effect size for all hardware components and overall (DUT) (marked

Table 11 Effect size of the power averages of the measurements by language

Language vs Language	HDD	Graphics Card	Processor	DUT
JRuby vs Dart	small	small		
JRuby vs Python	small	small		
Pascal vs JRuby	small		Small	
Pascal vs Ruby	small		Small	
Perl vs Erlang			Médium	
PHP vs Dart	small			
PHP vs JRuby	small		Médium	
PHP vs Pascal	small			
PHP vs Python	small			
PHP vs Ruby	small		Small	
Python vs Dart		small	Small	
Ruby vs Dart	small	small		
Ruby vs JRuby	medium	small	Small	
Ruby vs Python	medium	small		
JRuby vs Dart				small

Table 12 Effect size of the consumption averages of the measurements by language-algorithm

HDD	TestCase	Effect size
	n-body: Perl vs PHP	medium
	binary-trees: Pascal vs Ruby	Small
	fasta: Ruby vs Jruby	Small
Processor	TestCase	Effect size
	fasta: Ruby vs Jruby	Small
	fannkuch-redux: JRuby vs Python	small
DUT	TestCase	Effect size
	binary-trees: Java vs C++	small
	binary-trees: PHP vs Jruby	small

in grey in Table 10): Erlang vs Dart and PHP; Haskell vs Pascal; JavaScript vs Haskell and Pascal; JRuby vs Python; Perl vs PHP; Ruby vs JRuby and Python.

In relation to the comparisons by average power (W), as we can see in Table 11, from 91 comparisons only 15 (16.4%) have low values for some hardware component. In the case of DUT, the only one was JRuby vs Dart.

Finally, the comparisons by average consumption between test cases (PL x algorithm) revealed that only 1% (5 out of 456 comparisons) achieved a small effect size, while 1 achieved a medium effect size (0.2%). Ruby vs JRuby stands out in the fasta algorithm, which obtained a small effect for both HDD and Processor. Table 12 presents these comparisons with the least favourable results. Concerning the comparisons by average power of the test cases, of the 456 results per component or DUT, only binary trees had a small effect on Perl versus Erlang for the processor component.

5 Threats to validity

In this section we outline the potential threats to the validity of our study (Wohlin et al., 2012) and detail the actions we have undertaken to mitigate their impact.

With regard to **construct validity**, to mitigate as much as possible the possible threats to the reliability of measurements we used a hardware instrument (EET) which provides more realistic and accurate results of energy consumption. Furthermore, it provides about 100 energy samples per second, which can be considered as a suitable sampling rate, aspect which was reinforced repeating each case test run 30 times to assure a higher reliability of the measurements. The provided energy consumption results are specific to this hardware instrument, and they can differ If other hardware instruments are applied, although it would not be expected high differences such as confirmed with a previous study in which EET was validated versus a gold standard (Mancebo et al., 2021a, b).

In relation to **Internal Validity**, we dealt with several factors that may affect the results of this study, according to the methodology described in Section 2: several repetitions were performed to mitigate the possible atypical values related with consumption; the DUT was restarted and prepared to be in the same stage before the execution of each test case to

ensure that the conditions were as much as possible the same. For instance, the OS processes running in the background were kept the same and at a minimum necessary to run each test case; and the baseline consumption of the DUT in each test case was subtracted from the test case measurements to avoid as much as possible confounding effects. On the other hand, we tried to replicate (Pereira et al., 2021)'s study under the same conditions as far as possible, selecting the closest version of OS possible, and the exact versions of CBLG algorithms. We had to select from these algorithms the subset which could be executed in all the languages with the same version of compilers used in the original study. Although more modern versions of the compiler could have been used, we opted for the exact versions to mitigate the effects mentioned above. In addition, the DUT used in this study was not possible to be identical to the SwE study, but the most similar configuration was set to mitigate this. Anyway, some proportional variations in the results are to be expected. Overall, and bearing in mind that the objective was to make a ranking, it is not considered to be a serious threat that could affect the position in which the languages have been placed.

Finally, with regard to **external validity**, this study selected a representative set of PLs to fulfil the stated research goal focused on evaluate the existing PLs ranking and going beyond estimation results by providing actual energy measurements. New programming languages can be incorporated in future studies to provide additional evidence. Another limitation is related to the algorithms selected for the study. The CBLG benchmark was used to do a fair comparison with the reference ranking. This benchmark has a varied set of algorithms that may have a different effect on the energy consumed depending on the different PC resources used, for which we consider this is enough representative for our purpose. In any case, it would be desirable to extend this study with other benchmarks to reinforce the external validity obtained and to conduct empirical studies which analyse the impact on real software projects.

6 Conclusions and further work

Nowadays, programmers have a wide variety of programming languages to choose from to meet their objectives. Sometimes the programming language to be used can be a constraint of the project, but many other times you can choose and the choice of one language or another can affect the green aspect. Among all the selection criteria we can distinguish between both technical and non-technical factors, such as abstraction level, project domain, for instance for programming the frontend, the backend, or the project is of data science. Other factors are related with the learning curve, the existing documentation, supporting libraries included in the language, and even more. The choice of a suitable programming language (or a set of them) for a software project is, therefore, an important decision that impacts productivity, quality and the greenability of the project. This implies considering a trade-off (García-Mireles et al., 2018) when choosing a language that must satisfy the traditional criteria (quality and productivity) but at the same time should take *greenability* into account. This study aims to contribute to this by helping developers and managers to make more informed trade-off decisions by addressing how the choice of programming language can affect the energy consumption of the developed software.

In order to do that, an experiment has been conducted in which fourteen programming languages and six algorithms have been analysed using a hardware-based consumption measurement instrument. This study is based on a previous work in which the same programming languages and algorithms (among others) were analysed but using a software-based approach.

As a result of this study, interesting lessons learned have emerged that would also help to focus future work on the issue. Firstly, evidence is contributed to the comparison between consumption measurement approaches (hardware-based, HwE) and consumption estimation approaches (software-based, SwE). Our results shed some light on the suitability of the application of estimators when the main objective of the research is to carry out rankings or comparative studies between one approach or another. In fact, no significant differences are obtained in the HwE and SwE rankings that would suggest that it is the measurement or estimation approach that is really affecting the results. However, if what is required is a more realistic knowledge of consumption, it is necessary to use hardware approaches. Similarly, it is of interest when it is possible to carry out studies that combine hardware approaches, and the information obtained is also complemented by estimation approaches. For example, RAPL allows to obtain valuable information on processor usage (cores, DRAM) that could be used to explain in a more concise way the power consumption of the processor (cores, DRAM).

Secondly, it has been illustrated how analysing the power consumption of specific DUT hardware components can provide useful information to make a more informed decision on the choice of a PL. We have therefore created a new ranking for each of the components (HDD, graphics card, and processor). Furthermore, to complete the study, energy consumption has been analysed considering the type of PL, the paradigm and the algorithm that was solved. As a conclusion, it has been detected for future work the need to expand the variety and type of benchmarks with algorithms that allow for a more in-depth analysis of the impact of Programming Languages (PLs) on specific components of the PC. For example, with algorithms that require more intensive work on the graphics card or HDD.

Finally, the results obtained with the study and the preceding studies represent a first advance on an empirical knowledge base on the influence of PLs on consumption that deserves to be extended by addressing other elements that could also be important in the decision, such as their combination with compilers and their optimizations. According to the results obtained, lower-level compiled languages such as C or C++ also exhibit energy efficiency, but on the other hand this could affect the productivity of programmers who prefer other more high-level alternatives, or simply because the PL chosen includes the necessary libraries to do the job. In these cases, suitable combinations of PL with compiler, e.g., Python with a compiler that optimizes its code, could satisfy the energy requirements, in addition to meeting time to market or other project constraints. This requires focusing on new studies that analyse these language-compiler combination effects.

Appendix A

Table 13 Mean values of energy and time of the 14 program languages in all algorithms. Table 13 shows the time and energy result of all algorithms for each programming language

Binary-trees					Mandelbrot				
Language	Time SwE (s)	Time HwM (s)	Energy SwE (J)	Energy HwM (J)	Language	Time SwE (s)	Time HwM (s)	Energy SwE (J)	Energy HwM (J)
C	1.125	0.622	39.800	148.258	C	1.142	0.479	35.390	112.623
C++	1.129	0.630	41.230	155.727	C++	0.864	0.480	32.440	125.188
Java	3.306	1.440	111.840	346.193	Ada	3.192	1.033	97.590	247.562
Ada	2.822	1.983	95.020	385.866	Java	3.657	1.226	114.230	300.556
Erlang	7.327	3.647	266.140	774.015	JavaScript	8.278	1.999	233.040	494.818
Haskell	11.582	7.046	270.150	816.672	Haskell	6.028	4.193	169.600	569.658
Pascal	16.079	10.287	214.640	856.780	Dart	10.046	2.740	214.740	571.253
JavaScript	21.349	11.455	312.140	1246.836	Pascal	7.190	4.981	212.670	766.604
Dart	17.197	12.054	290.270	1262.578	PHP	73.741	26.315	3374.290	5152.519
Ruby	26.634	11.415	855.120	2412.344	Python	16.333	74.095	7790.160	14118.908
Iruby	19.276	12.871	720.530	2682.286	Erlang	194.726	81.386	9172.720	15407.026
PHP	42.316	17.612	1397.510	3402.135	Ruby	245.248	106.537	11050.980	20358.194
Python	45.003	20.242	1793.460	3979.624	Perl	389.587	153.444	16965.060	29988.293
Perl	96.097	67.624	3542.200	11474.379	Iruby	217.118	159.168	8401.950	33834.189
Fannkuch-redux					N-body				
Language	Time SwE (s)	Time HwM (s)	Energy SwE (J)	Energy HwM (J)	Language	Time SwE (s)	Time HwM (s)	Energy SwE (J)	Energy HwM (J)
C	6.076	2.557	215.920	527.574	C	4.190	2.240	59.450	176.833
C++	6.123	2.492	219.890	548.432	C++	3.770	2.339	57.300	199.007
Ada	7.351	2.963	264.980	628.843	Ada	4.098	2.623	51.790	216.298
Java	8.241	3.451	311.380	735.303	Pascal	5.702	3.095	64.870	220.815

Table 13 (continued)

Famkuch-redux						N-body					
Language	Time SwE (s)	Time HwM (s)	Energy SwE (J)	Energy HwM (J)	Language	Time SwE (s)	Time HwM (s)	Energy SwE (J)	Energy HwM (J)	Language	Time SwE (s)
Pascal	9.807	4.104	343.550	844.977	Java	5.839	3.389	65.150	272.382	Java	5.839
Haskell	14.666	10.476	433.680	1593.806	JavaScript	6.763	5.078	78.740	343.813	JavaScript	6.763
JavaScript	33.663	24.803	413.900	1805.279	Dart	6.827	4.137	84.330	361.120	Dart	6.827
Dart	38.678	27.233	487.290	2485.435	Haskell	10.037	5.610	127.820	407.153	Haskell	10.037
ErLang	101.839	39.444	4148.380	7654.466	Jruby	98.407	68.549	1889.850	8188.760	Jruby	98.407
PHP	125.975	46.713	5731.880	8985.842	ErLang	150.698	100.138	1998.660	9416.256	ErLang	150.698
Perl	249.418	105.385	11133.490	19947.253	PHP	179.360	118.795	2468.300	11879.277	PHP	179.360
Python	31.558	117.657	12784.090	21469.747	Ruby	281.470	210.540	3870.670	20646.820	Ruby	281.470
Jruby	219.148	145.133	7819.030	25314.842	Python	559.214	364.146	7879.920	35443.585	Python	559.214
Ruby	315.583	224.567	14064.980	42901.450	Perl	335.391	322.761	4663.480	35896.837	Perl	335.391
Fasta						Spectral-norm					
Language	Time SwE (s)	Time HwM (s)	Energy SwE (J)	Energy HwM (J)	Language	Time SwE (s)	Time HwM (s)	Energy SwE (J)	Energy HwM (J)	Language	Time SwE (s)
C	0.973	1.149	27.640	114.083	C++	0.665	0.160	17.230	23.288	C++	0.665
C++	1.164	1.264	34.880	142.337	C	0.677	0.160	21.660	27.459	C	0.677
Java	1.249	1.490	35.860	143.848	Haskell	1.373	0.217	34.740	30.833	Haskell	1.373
Ada	2.765	2.512	40.450	167.676	Pascal	1.349	0.105	37.650	33.187	Pascal	1.349
JavaScript	5.098	3.776	64.840	233.328	Java	1.659	0.426	41.180	79.640	Java	1.659
Pascal	5.478	4.343	68.630	295.862	Ada	0.775	0.530	27.030	90.204	Ada	0.775
Dart	4.787	4.210	63.610	333.188	JavaScript	5.048	2.108	51.460	134.893	JavaScript	5.048
Haskell	5.728	4.217	205.520	664.002	Dart	5.102	2.217	55.150	185.507	Dart	5.102
PHP	29.508	19.970	430.730	1908.741	ErLang	17.998	6.687	767.430	1311.304	ErLang	17.998
ErLang	27.852	19.310	477.810	1947.949	PHP	19.128	8.093	880.450	1640.479	PHP	19.128

Table 13 (continued)

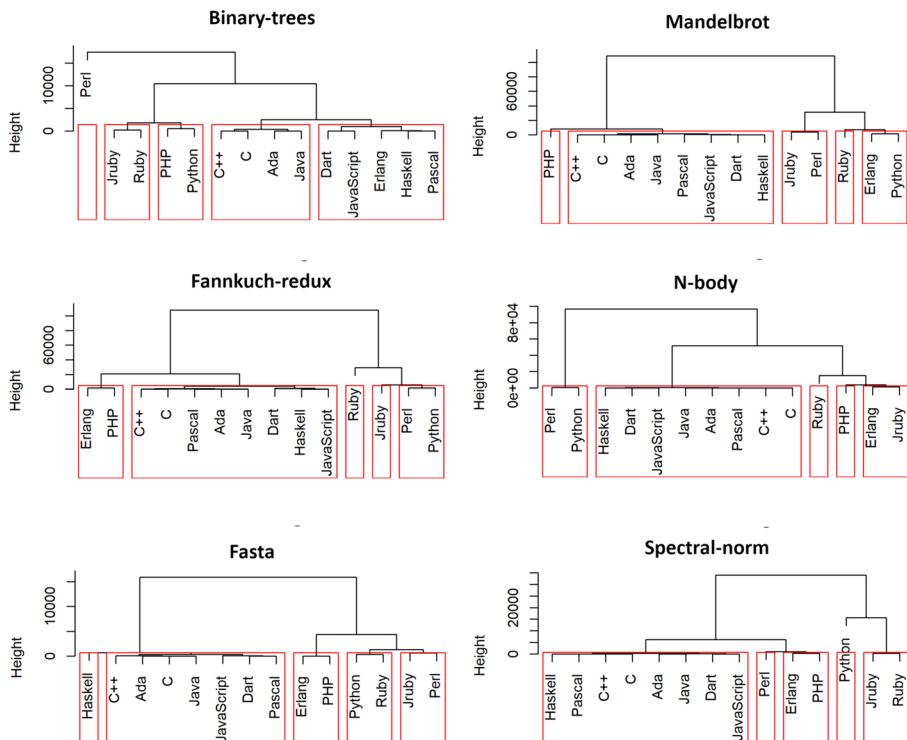
Language	Spectral-norm					
	Time SwE (s)	Time HwM (s)	Energy SwE (J)	Energy HwM (J)	Language	Time SwE (s)
Perl	61.463	31.868	2684.330	3101.250	Perl	19.782
Jruby	49.509	37.142	912.930	3722.026	Ruby	71.540
Ruby	61.216	42.440	852.300	4144.757	Jruby	79.449
Python	74.111	49.800	1061.410	4508.876	Python	137.738
						96.909
						862.490
						2386.203
						3122.890
						6694.626
						1420.100
						6978.705
						18617.474

Table 14 Standard deviation of energy of the 14 program languages in all algorithms Table 14

TestCase	HDD	GraphicsCard	Processor	DUT
Ada	4,117044145	2,605081221	20,19223942	70,81378648
C	2,147145825	3,72768583	22,00685816	68,53727902
C++	0,792544467	41,00613461	25,62311821	73,9916664
Dart	0,491809366	0,412163592	12,91856013	49,47540082
Java	1,286228893	2,558218725	22,8374847	71,92801296
JSRuby	3,460725814	38,9769536	17,85346656	50,95541611
Perl	7,946296538	4,913497282	21,18491748	53,87749007
PHP	3,324390341	1,217144498	17,27686032	51,62263353
Haskell	0,195780487	0,488298042	11,64893547	31,34134239
JavaScript	0,158469654	0,321404,796	24,81865213	72,54326701
Pascal	0,154784969	0,267011014	26,53736993	97,86342773
Erlang	0,470862174	0,84737853	16,52191026	52,49918075
Python	3,277377695	3,912137333	17,97783696	54,45943983
Ruby	6,695049608	52,44450035	21,28774456	55,84387203

Appendix B

Figure 10 shows the clusters of the programming languages for each algorithm, using Ward's method

**Fig. 10** Clusters of Programming languages for each algorithm considering energy consumption

Appendix C

Figures 11, 12 and 13 show the clusters identified based on the energy consumed by the hard disk, graphics card and processor respectively. Figure 11 HDD clusters by algorithm Fig. 12 Graphics Card clusters by algorithm.

Table 15 Mean values of energy for components of the 14 program languages in all algorithms. Table 15 provides an overview of the energy consumption for individual components associated with each programming language for the different algorithms

Binary-trees					Mandelbrot				
Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)	Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)
C	148.258	4.125	8.074	48.570	C	112.623	2.467	6.923	37.691
C++	155.727	0.953	68.058	56.766	C++	125.188	1.590	2.698	46.670
Java	346.193	6.244	15.675	112.407	Ada	247.562	12.238	13.243	77.143
Ada	385.866	27.967	21.612	104.508	Java	300.556	9.645	10.115	94.216
Erlang	774.015	18.092	23.404	234.080	JavaScript	494.818	9.613	10.692	173.911
Haskell	816.672	34.543	46.510	342.369	Haskell	569.658	21.458	31.693	237.101
Pascal	856.780	47.877	64.886	304.394	Dart	571.253	14.085	17.843	154.426
JavaScript	1246.836	54.199	67.048	450.176	Pascal	766.604	22.775	30.861	280.146
Dart	1262.578	54.924	77.690	313.275	PHP	5152.519	148.487	210.281	1407.087
Ruby	2412.344	47.686	70.693	916.061	Python	14118.908	383.032	684.402	5030.773
Jruby	2682.286	75.421	259.506	901.878	Erlang	15407.026	452.967	645.177	4174.968
PHP	3402.135	79.335	107.860	1027.403	Ruby	20358.194	488.060	641.863	7457.606
Python	3979.624	83.788	160.127	1487.715	Perl	29988.293	1610.192	1937.509	9069.707
Perl	11474.379	296.506	430.762	3689.755	Jruby	33834.189	752.594	2386.915	10024.029
Fannkuch-redux					N-body				
Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)	Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)
C	527.574	18.384	23.491	183.973	C	176.833	10.531	39.864	66.903
C++	548.432	6.274	67.450	206.066	C++	199.007	6.692	12.637	84.871

Table 15 (continued)

Fannkuch-redux					N-body				
Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)	Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)
Ada	628.843	64.437	18.953	194.878	Ada	216.298	35.186	30.154	67.676
Java	735.303	16.340	27.677	231.424	Pascal	220.815	14.161	18.221	75.151
Pascal	844.977	18.727	25.211	308.241	Java	272.382	21.170	29.139	89.793
Haskell	1593.806	53.185	80.588	664.447	JavaScript	343.813	23.963	25.553	125.354
JavaScript	1805.279	117.390	133.065	694.132	Dart	361.120	23.107	26.205	103.013
Dart	2485.435	123.380	161.412	681.823	Haskell	407.153	28.545	38.158	184.138
ErLang	7654.466	196.074	270.915	2434.608	Julby	8188.760	380.722	511.881	2561.242
PHP	8985.842	510.895	321.433	3035.411	ErLang	9416.256	592.961	710.176	2722.526
Perl	19947.253	492.537	549.476	6488.091	PHP	11879.277	685.765	1030.015	1173.309
Python	21469.747	591.421	679.809	8210.868	Ruby	20,646.820	1013.570	7441.619	6710.430
Julby	25314.842	877.390	972.797	9972.429	Python	35443.585	1727.844	2275.679	11403.111
Ruby	42901.450	3000.688	2941.086	14098.116	Perl	35896.837	1667.450	2293.536	11635.961
Fasta					Spectral-norm				
Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)	Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)
C	114.083	11.608	9.984	39.841	C++	23.288	0.514	0.919	9.945
C++	142.337	4.795	6.581	52.883	C	27.459	0.695	2.623	10.780
Java	143.848	9.594	10.201	45.079	Haskell	30.833	1.162	1.484	10.662
Ada	167.676	25.519	31.516	64.978	Pascal	33.187	0.507	0.590	8.739
JavaScript	233.328	19.377	19.418	92.216	Java	79.640	2.321	5.922	26.395
Pascal	295.862	21.420	27.144	103.129	Ada	90.204	6.220	7.267	27.878
Dart	333.188	20.956	23.452	103.225	JavaScript	134.893	10.042	10.476	49.273
Haskell	664.002	22.933	32.071	269.141	Dart	185.507	12.602	12.536	52.056
PHP	1908.741	97.259	115.732	561.224	ErLang	1311.304	40.257	37.836	412.173
ErLang	1947.949	99.766	112.481	593.920	PHP	1640.479	37.909	62.018	588.193

Table 15 (continued)

Fasta		Spectral-norm							
Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)	Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)
Perl	3101.250	139.663	203.502	968.586	Perl	2386.203	38.544	51.885	853.184
Jruby	3722.026	161.588	213.672	1243.601	Ruby	6694.626	163.427	297.201	2316.315
Ruby	4144.757	182.110	236.087	1413.088	Jruby	6978.705	284.093	413.951	1898.206
Python	4508.876	238.760	279.586	1887.015	Python	18,617.474	634.374	714.266	5882.539

Table 16 Standard deviation of the measurements for each component of the 14 programming languages for all. Algorithms. Table 16 shows the standard deviations between measurements, sorted by algorithm and language

Binary-trees						Mandelbrot					
Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)		Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)	
C	10,832	0,421	3,340	0,781		C	3,604	0,171	1,993	0,491	
C++	9,964	0,059	0,532	1,725		C++	3,381	0,058	0,074	0,847	
Java	4,117	0,289	4,001	3,092		Ada	5,198	0,814	1,747	1,477	
Ada	3,066	0,839	1,045	0,677		Java	2,426	0,727	1,177	0,610	
ErLang	2,436	0,084	0,354	1,210		JavaScript	2,568	0,031	0,112	0,446	
Haskell	2,980	0,056	0,628	0,788		Haskell	1,619	0,021	0,091	0,491	
Pascal	24,568	0,038	0,566	9,317		Dart	1,312	0,459	0,958	0,298	
JavaScript	2,526	0,015	0,618	0,800		Pascal	13,972	0,019	0,150	5,538	
Dart	2,810	0,018	0,697	0,301		PHP	0,793	1,178	1,056	5,459	
Ruby	7,167	0,026	0,330	2,751		Python	1,573	0,030	1,883	1,381	
Iruby	1,007	0,648	31,554	1,423		ErLang	1,211	0,183	0,401	8,220	
PHP	1,514	0,106	1,660	0,816		Ruby	0,861	0,042	0,408	1,697	
Python	1,475	0,040	1,032	2,817		Perl	1,908	4,160	5,654	2,117	
Perl	1,410	0,033	0,921	2,731		Iruby	1,798	0,044	28,012	2,063	
Fannkuch-redux						N-body					
Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)		Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)	
C	1,812	0,853	0,459	1,269		C	0,810	0,331	3,798	1,055	
C++	1,962	0,666	40,878	1,208		C++	3,341	0,033	0,075	0,280	
Ada	2,214	8,068	3,721	3,888		Ada	1,616	0,640	2,048	0,539	
Java	4,177	0,646	0,372	2,684		Pascal	5,967	0,022	0,271	1,924	
Pascal	25,211	0,063	0,270	9,633		Java	1,598	1,408	2,120	2,915	
Haskell	6,411	0,049	0,214	2,446		JavaScript	1,402	0,022	0,139	0,356	
JavaScript	0,967	0,013	0,063	0,149		Dart	2,220	0,020	0,738	1,171	

Table 16 (continued)

Fannkuch-redux						N-body					
Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)		Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)	
Dart	2,030	0,035	0,251	0,246		Haskell	1,996	0,019	0,326	0,771	
ErLang	3,395	0,046	0,127	0,334		Ruby	1,593	0,144	11,687	0,515	
PHP	0,870	1,626	0,879	3,348		ErLang	1,643	0,039	0,922	2,211	
Perl	1,289	0,034	0,601	2,830		PHP	2,323	0,418	1,589	0,267	
Python	1,311	0,471	0,888	2,270		Ruby	5,612	0,011	3,164	1,629	
Julia	1,503	0,407	5,557	0,793		Python	3,556	0,015	0,655	1,243	
Julia	1,390	4,197	2,998	0,997		Perl	3,737	0,013	1,697	1,427	
Fasta						Spectral-norm					
Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)		Language	DUT (J)	HDD (J)	Graphics Card (J)	Processor (J)	
C	5,812	3,485	3,986	1,688		C++	3,679	0,054	0,127	0,921	
C++	6,453	0,208	0,170	2,142		C	3,338	0,082	0,644	0,827	
Java	2,072	0,623	0,715	2,927		Haskell	7,362	0,061	0,175	1,466	
Ada	1,956	2,748	1,303	0,831		Pascal	3,938	0,067	0,160	1,534	
JavaScript	1,167	0,047	0,116	0,353		Java	3,707	0,104	2,342	0,706	
Pascal	14,839	0,056	0,125	5,876		Ada	6,208	1,216	2,567	1,694	
Dart	1,301	0,020	0,231	0,066		JavaScript	0,849	0,023	0,123	0,199	
Haskell	3,105	0,059	0,094	0,985		Dart	1,815	0,020	0,211	0,145	
PHP	0,945	0,190	0,987	0,172		ErLang	0,794	0,017	0,149	3,416	
ErLang	4,679	0,028	1,181	0,628		PHP	1,847	0,282	1,473	1,326	
Perl	2,036	0,024	1,203	0,395		Perl	7,015	0,043	0,868	2,530	
Julia	3,803	0,053	0,102	1,389		Ruby	1,559	0,081	0,390	0,187	
Julia	3,180	0,015	0,179	0,402		Julia	2,555	0,038	0,174	1,189	
Python	1,762	0,023	1,065	0,587		Python	2,829	1,207	0,949	1,174	

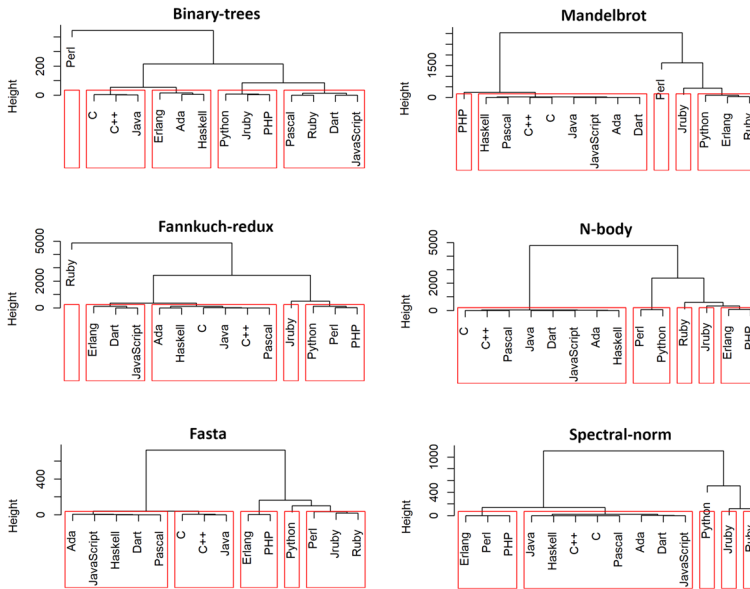


Fig. 11 HDD clusters by algorithm

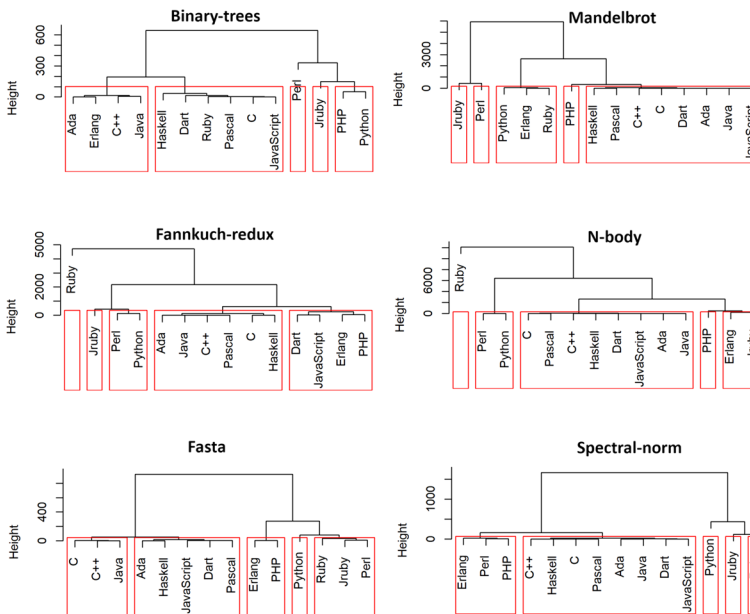


Fig. 12 Graphics Card clusters by algorithm

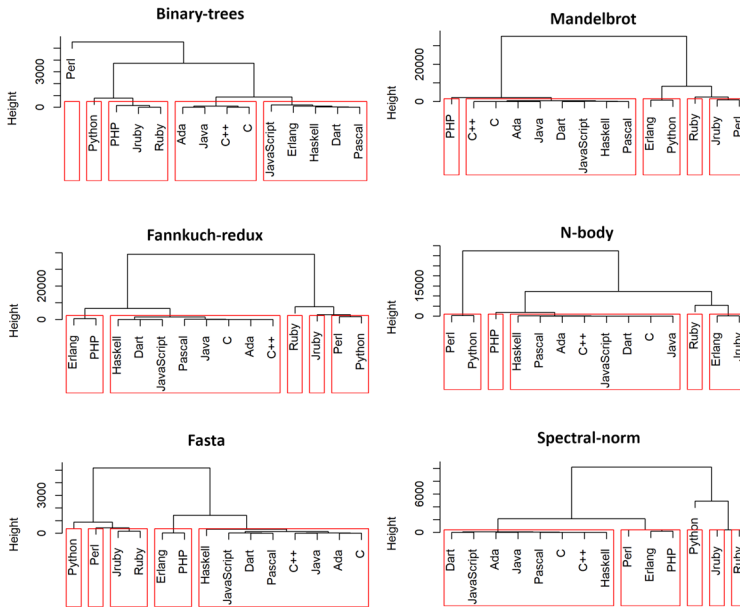


Fig. 13 Processor clusters by algorithm

Acknowledgements The work has received support from the following projects: OASSIS (PID2021-122554OB-C31/ AEI/10.13039/ 501100011033/FEDER, UE); EMMA (Project SBPLY/21/180501/000115, funded by CECD (JCCM) and FEDER funds); SEEAT (PDC2022-133249-C31 funded by MCIN/AEI/ <https://doi.org/10.13039/501100011033> and European Union NextGenerationEU/PRTR); PLAGEMIS (TED2021-129245B-C22 funded by MCIN/AEI/ <https://doi.org/10.13039/501100011033> and European Union NextGenerationEU/PRTR). Financial support for the execution of applied research projects, within the framework of the UCLM Own Research Plan, co-financed at 85% by the European Regional Development Fund (FEDER) UNION (2022-GRIN-34110).

Authors contribution The planning of the comparison was carried out by Alberto Gordillo, Coral Calero, M^a Moraga, Félix García, João Paulo Fernandes, Rui Abreu, and João Saraiva. The preparation of the replication was conducted by Alberto Gordillo, Coral Calero, M^a Moraga, and Félix García. The experiment execution was performed by Alberto Gordillo. The manuscript was written by Alberto Gordillo, Coral Calero, M^a Moraga, and Félix García. Finally, the comparison of the studies and the review of both the experiment and the manuscript were conducted by Alberto Gordillo, Coral Calero, M^a Moraga, Félix García, João Paulo Fernandes, Rui Abreu, and João Saraiva.

Data availability No datasets were generated or analysed during the current study.

Declarations

Conflict of interest The authors declare no competing interests.

References

- Andrae, A. S. (2019). Prediction Studies of Electricity Use of Global Computing in 2030. *International Journal of Science and Engineering Investigations (IJSEI)*, 8, 27–33.
- Basili, V. R., Selby, R. W., & Hutchens, D. H. (1986). Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, 12(7), 733–743.

- Becker, C., Chitchyan, R., Duboc, L., Easterbrook, S., Penzenstadler, B., Seyff, N., & Venters, C. C. (2015). *Sustainability design and software: The karlskrona manifesto*. 2, 467–476. IEEE.
- Bhattacharya, P., & Neamtiu, I. (2011). *Assessing programming language impact on development and maintenance: A study on C and C++*. 171–180.
- Bissyandé, T. F., Thung, F., Lo, D., Jiang, L., & Réveillere, L. (2013). Popularity, interoperability, and impact of programming languages in 100,000 open source projects. *2013 IEEE 37th Annual Computer Software and Applications Conference*, 303–312. IEEE.
- Brooks, A., Daly, J. W., Miller, J., Roper, M., & Wood, M. I. (1996). *Replication of experimental results in software engineering*.
- Cabot, J., Capilla, R., Carrillo, C., Muccini, H., & Penzenstadler, B. (2019). Measuring systems and architectures: A sustainability perspective. *IEEE Software*, 36(3), 98–100.
- Calero, C., & Piattini, M. (2015). *Green in software engineering* (Vol. 3). Springer.
- Calero, C., Moraga, M. Á., Bertoa, M. F., & Duboc, L. (2015). Green software and software quality. *Green in Software Engineering*, 231–260.
- Choroszucho, A., Golonko, P., Bednarek, J., Sumorek, M., & Żukowski, J. (2019). *Comparison of high-level programming languages efficiency in embedded systems*. 11176, 1800–1808. SPIE.
- Chowdhury, S. A., & Hindle, A. (2016). *Greenoracle: Estimating software energy consumption with energy measurement corpora*. 49–60.
- Chowdhury, S., Borle, S., Romansky, S., & Hindle, A. (2019). Greenscaler: Training software energy models with automatic test generation. *Empirical Software Engineering*, 24, 1649–1692.
- Cohen, J. (2013). *Statistical power analysis for the behavioral sciences*. Routledge.
- Corral-García, J., Lemus-Prieto, F., & Pérez-Toledano, M. -Á. (2021). Efficient code development for improving execution performance in high-performance computing centers. *The Journal of Supercomputing*, 77(4), 3261–3288.
- Cruz, L., Abreu, R., Grundy, J., Li, L., & Xia, X. (2019). *Do energy-oriented changes hinder maintainability?* 29–40. IEEE.
- Dirlwanger, W. (2006). *Measurement and rating of computer systems performance and of software efficiency: An introduction to the ISO/IEC 14756 method and a guide to its application*. Kassel University Press Kassel.
- Fonseca, A., Kazman, R., & Lago, P. (2019). A manifesto for energy-aware software. *IEEE Software*, 36(6), 79–82.
- García-Mireles, G. A., Moraga, M. Á., García, F., Calero, C., & Piattini, M. (2018). Interactions between environmental sustainability goals and software product quality: A mapping study. *Information and Software Technology*, 95, 108–129.
- Georgiou, S., Kechagia, M., Louridas, P., & Spinellis, D. (2018). *What are your programming language's energy-delay implications?* 303–313.
- Gordillo, A., Calero, C., Moraga, M. Á., García, F., Fernandes, J. P., Abreu, R., & Saraiva, J. (2024). *Repository of programming languages ranking based on energy measurements*. Retrieved from Repository of Programming Languages Ranking based on Energy Measurements website: <https://github.com/GrupoAlarcos/Programming-Languages-Ranking-based-on-Energy--Measurements>
- Guamán, D., & Pérez, J. (2021). *Supporting Sustainability and Technical Debt-Driven Design Decisions in Software Architectures*.
- Hanenberg, S. (2010). *An experiment about static and dynamic type systems: Doubts about the positive impact of static type systems on development time*. 22–35.
- Harrison, R., Samaraweera, L., Dobie, M. R., & Lewis, P. H. (1996). Comparing programming paradigms: An evaluation of functional and object-oriented programs. *Software Engineering Journal*, 11(4), 247–254.
- IEC, I. (2011). ISO/IEC 25010: System and Software engineering-System and software Quality Requirements and Evaluation (SQuaRE)-System and software quality models. *Switzerland: ISO*.
- Jedlitschka, A., & Pfahl, D. (2005). Reporting guidelines for controlled experiments in software engineering. *2005 International Symposium on Empirical Software Engineering, ISESE 2005*, 10 pp.-. <https://doi.org/10.1109/ISESE.2005.1541818>
- Kelefouras, V., & Djemame, K. (2019). A methodology correlating code optimizations with data memory accesses, execution time and energy consumption. *The Journal of Supercomputing*, 75(10), 6710–6745.
- Kern, E., Hilty, L. M., Guldner, A., Maksimov, Y. V., Filler, A., Gröger, J., & Naumann, S. (2018). Sustainable software products—Towards assessment criteria for resource and energy efficiency. *Future Generation Computer Systems*, 86, 199–210.
- Kleinschmager, S., Robbes, R., Stefik, A., Hanenberg, S., & Tanter, E. (2012). *Do static type systems improve the maintainability of software systems? An empirical study*. 153–162. IEEE.

- Koch, C., Müller, K., & Sultanow, E. (2022). Which programming languages do hackers use? A survey at the German Chaos Computer Club. *arXiv Preprint arXiv:2203.12466*
- Kochhar, P. S., Wijedasa, D., & Lo, D. (2016). *A large scale study of multiple programming languages and code quality. 1*, 563–573. IEEE.
- Li, D., & Halfond, W. G. (2014). *An investigation into energy-saving programming practices for android smartphone app development*. 46–53.
- Lima, L. G., Soares-Neto, F., Lieuthier, P., Filho, F. C., Melfe, G., & Fernandes, J. P. (2016). Haskell in Green Land: Analyzing the Energy Behavior of a Purely Functional Language. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), 1*, 517–528.
- Mancebo, J., Calero, C., & García, F. (2021a). Does maintainability relate to the energy consumption of software? *A Case Study. Software Quality Journal*, 29(1), 101–127.
- Mancebo, J., Calero, C., García, F., Moraga, M. A., & de Guzman, I.G.-R. (2021b). FEETINGS: Framework for Energy Efficiency Testing to Improve Environmental Goal of the Software. *Sustainable Computing: Informatics and Systems*, 30, 100558. <https://doi.org/10.1016/j.suscom.2021.100558>
- Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., . . . Clause, J. (2016). *An empirical study of practitioners' perspectives on green software engineering*. 237–248.
- Meyerovich, L. A., & Rabkin, A. S. (2013). *Empirical analysis of programming language adoption*. 1–18.
- Muna, A. (2022). Assessing programming language impact on software development productivity based on mining oss repositories. *ACM SIGSOFT Software Engineering Notes*, 44(1), 36–38.
- Murtagh, F., & Legendre, P. (2011). Ward's hierarchical clustering method: Clustering criterion and agglomerative algorithm. *arXiv Preprint arXiv:1111.6285*
- Naumann, S., Dick, M., Kern, E., & Johann, T. (2011). The GREENSOFT Model: A reference model for green and sustainable software and its engineering. *Sustainable Computing: Informatics and Systems*, 1(4), 294–304.
- OMG. (2008). Software process engineering metamodel 2.0. Retrieved from Software process engineering metamodel 2.0 website: <https://www.omg.org/spec/SPEM/2.0/About-SPEM>
- Pang, C., Hindle, A., Adams, B., & Hassan, A. E. (2015). What do programmers know about software energy consumption? *IEEE Software*, 33(3), 83–89.
- Pankratius, V., Schmidt, F., & Garretón, G. (2012). *Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java*. 123–133. IEEE.
- Penzenstadler, B., Raturi, A., Richardson, D., Calero, C., Femmer, H., & Franch, X. (2014). *Systematic mapping study on software engineering for sustainability (SE4S)*. 1–14.
- Pereira, R., Couto, M., Saraiva, J., Cunha, J., & Fernandes, J. P. (2016). The Influence of the Java Collection Framework on Overall Energy Consumption. *Proceedings of the 5th International Workshop on Green and Sustainable Software*, 15–21. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/2896967.2896968>
- Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). *Energy efficiency across programming languages: How do energy, time, and memory relate?* (pp. 256–267). Association for Computing Machinery.
- Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J. P., & Saraiva, J. (2020). SPELLing out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software*, 161, 110463.
- Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2021). Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205, 102609.
- Pinto, G., & Castor, F. (2017). Energy efficiency: A new concern for application software developers. *Communications of the ACM*, 60(12), 68–75.
- Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2014). *A large scale study of programming languages and code quality in github*. 155–165.
- Rosetta code. (n.d.). Retrieved 21 December 2022 from https://rosetta-code.org/wiki/Rosetta_Code
- Solari, M., Vegas, S., & Juristo, N. (2018). Content and structure of laboratory packages for software engineering experiments. *Information and Software Technology*, 97, 64–79. <https://doi.org/10.1016/j.infsof.2017.12.016>
- The computer language benchmarks game. (n.d.). Retrieved 21 December 2022, from <https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html>
- Venters, C. C., Jay, C., Lau, L., Griffiths, M. K., Holmes, V., Ward, R. R., . . . Xu, J. (2014). *Software sustainability: The modern tower of babel*. 1216, 7–12. CEUR.
- Vidal, J. (2017). Tsunami of data could consume one fifth of global electricity by 2025'. *Climate Home News*, 11.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Yang, H., Nong, Y., Wang, S., & Cai, H. (2024). Multi-Language Software Development: Issues, Challenges, and Solutions. *IEEE Transactions on Software Engineering*.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.



Alberto Gordillo is currently a PhD student in Advanced Information Technologies. He graduated in Computer Engineering in 2020 and obtained a master's degree also in Computer Engineering in 2022. He is a member of the Alarcos Research Group, University of Castilla-La Mancha (UCLM), Ciudad Real, Spain. He works on process automation and software sustainability by performing energy consumption measurements among other research. He holds the ITIL4 professional certification (Information Technology Infrastructure Library). Contact him at alberto.gordillo@uclm.es.



Coral Calero is currently a Full Professor with the University of Castilla-La Mancha (UCLM), Ciudad Real, Spain. She is a member of the Alarcos Research Group, being responsible of the "Green and Sustainable software" line research. She received the M.Sc. degree in 1996 from the University of Seville and the Ph.D. degree in Computer science in 2001 from the UCLM. She holds the professional certifications PMP (Project Management Professional), Scrum Master, and Scrum Manager. She is member of the Spanish Committee on Research Ethics. Contact her at Coral.Calero@uclm.es.



Mª Ángeles Moraga is currently an Associate Professor and a Member of the Alarcos Research Group, University of Castilla-La Mancha (UCLM), Ciudad Real, Spain. Mª Angeles Moraga received the M.Sc. degree in 2003, and the Ph.D. degrees in computer science in 2006, from the UCLM. She works on software quality and measures, and software sustainability. She holds the following professional certifications: PMP (Project Management Professional), Scrum Master I - PSM I, and Scrum Manager. Contact her at MariaAngeles.Moraga@uclm.es.



Félix García is Full Professor at the University of Castilla-La Mancha (UCLM) and a member of the Alarcos Research Group, Ciudad Real, Spain. His research interests include software sustainability, business process management, software processes, software measurement, and agile methods. He holds the following professional certifications: PMP (Project Management Professional), CISA (Certified Information Systems Auditor), Scrum Master I - PSM I, and Scrum Manager. Contact him at Felix.Garcia@uclm.es.



João Paulo Fernandes is a Visiting Associate Professor at New York University Abu Dhabi. Prof. Fernandes is a computer scientist passionate about addressing sustainability challenges within software development and usage and creating a greener, more eco-friendly future. His research interests focus on optimizing the energy efficiency of software systems by providing information and tools for developers to build energy-efficient systems and for end users to adopt energy-efficient usage patterns. He has founded/coordinated and/or is involved in projects and initiatives such as GreenHub, Green Sw Lab, GreenHaskell, GreenStamp, and Sustrainable. His scientific achievements have been described in over 75 publications in high-quality international conferences and journals such as IEEE TSE, ACM TOCE, EMSE, JSS, ICSME, SANER, EASE, and SLE, and include award-winning publications, e.g., the "Data Showcase Special

Mention Award" at MSR'19. He has served on over 50 international conferences and journals program committees and was awarded the Outstanding Reviewer Award at SBES'19.



Prof. Rui Abreu holds a Ph.D. in Computer Science - Software Engineering from the Delft University of Technology, The Netherlands, and a M.Sc. in Computer and Systems Engineering from the University of Minho, Portugal. His research revolves around software quality, with emphasis in automating the testing and debugging phases of the software development life-cycle as well as self-adaptation. Dr. Abreu has extensive expertise in both static and dynamic analysis algorithms for improving software quality. He is the recipient of 8 Best Paper Awards, including a Distinguished Paper Award at ESEC/FSE 2019, and his work has attracted considerable attention. Before joining FEUP as a Full Professor, he was an Associate Professor at IST, ULisbon and a member of the Model-Based Reasoning group at PARC's System and Sciences Laboratory and an Assistant Professor at the University of Porto.



João Saraiva is an Associate Professor at the department of Informatics, University of Minho, Portugal, and a senior research member of HASLab/INESC TEC. He obtained a Ph.D. degree in Computer Science from Utrecht University in 1999. His main research contributions have been in the field of programming language design and implementation, green software, and functional programming.

Authors and Affiliations

Alberto Gordillo¹ · Coral Calero¹ · M^a Ángeles Moraga¹ · Félix García¹ ·
João Paulo Fernandes² · Rui Abreu³ · João Saraiva⁴

Coral Calero
coral.calero@uclm.es

M^a Ángeles Moraga
mariaangeles.moraga@uclm.es

Félix García
felix.garcia@uclm.es

João Paulo Fernandes
jpaulo@fe.up.pt

Rui Abreu
rma@fe.up.pt

João Saraiva
saraiva@di.uminho.pt

¹ Institute of Technology and Information Systems, University of Castilla-La Mancha, Ciudad Real, Spain

² New York University, Abu Dhabi, UAE

³ DEI-FEUP, University of Porto, Porto, Portugal

⁴ University of Minho, Minho, Portugal