Trading Runtime for Energy Efficiency

Leveraging Power Caps to Save Energy across Programming Languages

Simão Cunha
University of Minho
Braga, Portugal
a93262@alunos.uminho.pt

Luís Silva
University of Minho
Braga, Portugal
pg50564@alunos.uminho.pt

João Saraiva
University of Minho
Braga, Portugal
saraiva@di.uminho.pt

João Paulo Fernandes
New York University Abu Dhabi
Abu Dhabi, United Arab Emirates
jpf9731@nyu.edu

Abstract

Energy efficiency of software is crucial in minimizing environmental impact and reducing operational costs of ICT systems. Energy efficiency is therefore a key area of contemporary software language engineering research. A recurrent discussion that excites our community is whether runtime performance is always a proxy for energy efficiency. While a generalized intuition seems to suggest this is the case, this intuition does not align with the fact that energy is the accumulation of power over time; hence, time is only one of the factors in this accumulation. We focus on the other factor, power, and the impact that capping it has on the energy efficiency of running software.

We conduct an extensive investigation comparing regular and power-capped executions of 9 benchmark programs obtained from *The Computer Language Benchmarks Game*, across 20 distinct programming languages. Our results show that employing power caps can be used to trade running time, which is degraded, for energy efficiency, which is improved, in *all* the programming languages and in *all* benchmarks that were considered. We observe overall energy savings of almost 14% across the 20 programming languages, with notable savings of 27% in Haskell. This saving, however, comes at the cost of an overall increase of the program's execution time of 91% in average.

We are also able to draw similar observations using language specific benchmarks for programming languages of different paradigms and with different execution models. This is achieved analyzing a wide range of benchmark programs from the *nofib Benchmark Suite of Haskell Programs*, *DaCapo Benchmark Suite for Java*, and the *Python Performance Benchmark Suite*. We observe energy savings of approximately 8% to 21% across the test suites, with execution time increases ranging from 21% to 46%. Notably, the DaCapo suite exhibits the most significant values, with 20.84% energy savings and a 45.58% increase in execution time.

Our results have the potential to drive significant energy savings in the context of computational tasks for which runtime is not critical, including Batch Processing Systems, Background Data Processing and Automated Backups.

CCS Concepts: • **Software and its engineering → Software performance**; **General programming languages**; • **Hardware → Power estimation and optimization**.

Keywords: Green Software, Power Cap, Energy Efficiency, Programming Languages, Language Benchmarking

ACM Reference Format:
Simão Cunha, Luís Silva, João Saraiva, and João Paulo Fernandes. 2024. Trading Runtime for Energy Efficiency: Leveraging Power Caps to Save Energy across Programming Languages. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3687997.3695638

1 Introduction

The software language design and engineering communities have been working hard to enhance both programmer performance and the efficiency of their programs. Today, software languages are vast and complex ecosystems focused on boosting programmer productivity. Software languages support portable code and feature advanced static and dynamic type systems, numerous reusable libraries, robust testing frameworks and powerful IDEs guided by large language models such as Copilot[1], GPT-4[2], ChatGPT[3] or Codex[4].

SLE '24, October 20–21, 2024, Pasadena, CA, USA

ACM ISBN 979-8-4007-1180-0/24/10
https://doi.org/10.1145/3687997.3695638

[1] https://copilot.microsoft.com/
[2] https://www.openai.com/research/gpt-4
[3] https://www.openai.com/chatgpt
[4] https://www.openai.com/index/openai-codex

Prior to these advances, the effort required to develop software was higher and, if we go back several decades, performance in programming languages was intimately associated with fast execution time due to limited hardware resources.

These days, however, although the development phase is less costly, the cost of running software has become much more relevant [47] and energy efficiency is one of its key components and concerns. In our age of non-wired computing devices, cloud computing, and large data centers, it was estimated that the Information and Communications Technology (ICT) sector alone accounted for more than 7% of global energy usage in 2020 [1]. With growing demand for digital services, this figure is expected to increase, which is especially concerning given that global energy consumption has been rising for seven decades, with most of it still coming from non-renewable sources [41].

Although there is significant research on estimating the costs of software development, maintenance, and evolution [3, 4, 11, 40], there is comparatively little research on the costs of running software and how such costs can be reduced.

Software systems are implemented using programming languages and, as shown in [37], the energy costs of running a program can be drastically influenced by the choice of programming language(s) used to develop it. While the goal of [37] was to establish an energy ranking for 27 software languages, it also provides some evidence that slower/faster software languages can consume less/more energy [37, 38].

Results in this line fuel a long-standing debate that is pervasive in our community: is it the case that runtime efficiency is always a proxy for energy efficiency? While general intuition in the software development community suggests that this is the case, we know for a fact that energy consumption does not depend only on (execution) time but also on the accumulation of power that is dissipated over that time: $energy = time \times power$, or, more rigorously, $energy = \int_0^T power(t)\,dt$. And we have incorporated this knowledge in our everyday life: we know that a driver operating a car can reduce its fuel consumption by limiting the power he/she employs, at the cost of a longer journey.

In this paper, we pursue the hypothesis that controlling the execution environment by capping power can similarly be used to reduce energy consumption.[5] Our goal is to be able of answering research questions that naturally arise from this hypothesis, namely:

RQ1 *Can we reduce the energy consumption of a program by defining a limit on the power used by the CPU? How is the program's runtime affected by defining such a power cap?*

RQ2 *Are all software languages affected by a power cap in the same way? Or are there languages that benefit from it more than others?*

RQ3 *Under a power cap, which are the most energy efficient software languages? Does power capping cause programming languages to be ranked differently regarding their runtime and energy efficiency?*

We start by analyzing the impact of limiting the power of the CPU considering as study objects 9 benchmark problems obtained from The Computer Language Benchmarks Game (CLBG) [13]. We compile/execute solutions to such problems using state-of-the-art compilers, virtual machines, interpreters, and libraries for 20 of the languages available in the latest version of CLBG. We then execute each of the 180 programs with and without imposing a power limit on the CPU, ending up with 360 executions that are pairwise comparable. Afterwards, we analyze the performance of the different implementations considering execution time, and energy consumption.

Our results reveal notable findings, such as *all* programs in *all* languages reduce their energy consumption in such a way that these reductions are *all* statistically significant. The observed reductions in energy consumption range from a minimum of 6% and a maximum of 27%.

There is, however, a(n expected) price to pay: *all* programs increase execution times, also in a statistically significant way. Runtime degradations are even more significant, ranging from a minimum of 35% and a maximum of 128%.

In order to cross-validate these results obtained using subjects from the language agnostic benchmark CLBG, we then seek to confirm them relying on language specific benchmark programs. We consider three programming languages widely used in academia and industry, namely Haskell, Java and Python, which also represent different programming paradigms and execution models. We use 27 benchmarks from the *nofib Benchmark Suite of Haskell Programs*, 20 applications from the *DaCapo Benchmark Suite for Java*, and 75 programs from the *Python Performance Benchmark Suite*.

Our results confirm our empirical observation that by defining a power cap we do reduce energy consumption, and in *all* benchmarks. On the contrary, the execution time of the all the considered benchmarks also increases significantly.

Finally, we seek to understand how power capping may or not affect the energy ranking of programming languages presented in [8, 37, 38]. With this goal in mind, we replicate the ranking methodology employed in previous studies to establish and compare two rankings of the same programming languages: one obtained by running programs without power capping and the other establishing a limit on the amount of power that running a program can utilize.

Our results suggest that power capping implies minimal changes on how programming languages compare among themselves regarding runtime and energy efficiency. While this contribution re-confirms the serialization of languages regarding these metrics, that is to say that ranking programming languages without limiting power dissipation leads

[5] We note that the results of [37] were observed without controlling power in any way.

in essence to the same result as ranking programming languages while capping power.

The remainder of this paper is organised as follows. Section 2 discusses related work. In section 3 we present the methodology we follow to conduct our study. It briefly presents the benchmarks, the use of the Intel's RAPL and RAPLCap energy framework, the calibration of the powercap and how we executed and measured the benchmarks. In Section 4 we present and discuss the results of the language ranking and the three benchmarks. Section 5 presents our conclusions.

2 Related Work

The software language engineering community has done a considerable amount of research on developing techniques and tools to improve programmers productivity. The goal of most of this research is to reduce the costs of software development, although such costs have been mitigated by the always increasing complexity of software systems. Similarly, powerful program and compiler optimizations have been introduced with the main goal of improving execution time.

Only recently the costs of executing software has become a key concern for industry and for the green software research community. Indeed, this community already provided knowledge on the energy efficiency of language data structures [16, 39] and mobile software languages [33], the energy impact of different programming practices both in mobile [28, 30, 44] and desktop applications [45], the energy efficiency of applications within the same scope [5, 18, 43], or even on how to statically predict energy consumption in several software systems [7, 15], among several other works.

In this paper we study the costs of executing software implemented in different programming languages. We build on previous work where 27 of the most known and used programming languages are ranked according to their energy performance [8, 37]. This study considered ten problems implemented in 27 languages of the CLBG. One key result of the work on the energy ranking of languages, is a general framework to compile, execute and measure the energy consumption of large set of programs. This framework relies on a program that uses a system call to execute and measure the energy of the programs under analysis without the need of adding intrusive (energy measurement) code. This framework is available to the research community [6] which makes it very suitable for our work and we have reused it as described in Section 3.1.

The CLBG programs were developed with the single concern for runtime efficiency. In [38] the ranking was validated with programs from Rosetta Code: a programming chrestomathy repository. This repository was also used in [12] which presents a study of the energy consumption of 14 programming languages when executing the same 25 Rosetta programming tasks. Moreover, these programs were executed in three different computer platforms: a server, a laptop, and an embedded system. Another work on the energy efficiency of software languages is presented in [23], which analyses the energy consumption of the CLBG programs implemented in Java, JavaScript, Python, PHP, Ruby, C, C++ and C#. This work studies the impact of the compiler's optimization flags and the use of different hardware in the energy consumption of those programs.

All these studies show that programming languages do have a strong impact on the energy performance of software systems. These studies, however, did not study the impact on the energy and runtime performance of languages when executed under a power limit. In fact, to the best of our knowledge there is no work on the impact of power caps on the performance of software languages. On the contrary, the high performance computing community has done extensive work on using power caps both on CPUs [6, 24, 26, 50] and GPUs [27, 31] in order to reduce the energy consumption of data centers. The definition of power caps is supported by Intel's Running Average Power Limit (RAPL) [10]: the energy measurement system used in the energy ranking of languages. Thus, in our study we use RAPL's PowerCap mechanism, named *RAPLCap*[7], and we extend the energy ranking framework to execute programs under given power caps. This is detailed in Section 3.3.

Software languages are interpreted/compiled so programs are executed. Thus, there is relevant work on analyzing the impact on languages' energy consumption of general compiler optimizations [22, 46, 48] and virtual machines [35]. There is also work on studying compiler energy efficient techniques such as energy driven optimizations [20], energy-efficient register caching [19, 49], and on the improvement of GPU energy efficiency [32]. All these works show important results on improving the energy performance of software languages. In our work, however, we are not studying the impact of such techniques on language performance.

3 Methodology

In this section, we review the methodology that we employed in our experiments. In Section 3.1, we describe in detail the CLBG and the benchmark problems it includes and which ones we used in our experiments. In Section 3.2, we describe the language specific benchmarks that we have additionally considered in order to further establish our findings. In Section 3.3 we describe our procedure to obtain energy measurements and to cap power to a specific value. In Section 3.4 we describe our experimental setup. Finally, in Section 3.5 we provide the details on the statistical analysis that we conducted and in Section 3.6 we report on the experiment equipment that we have used.

[6] https://sites.google.com/view/energy-efficiency-languages

[7] https://github.com/powercap/raplcap

3.1 Language Agnostic Benchmarks

In order to obtain a comparable, representative and extensive set of programs written in many of the most popular and most widely used programming languages we have explored The Computer Language Benchmarks Game (CLBG) [13].

The CLBG initiative includes a framework for running, testing and comparing implemented coherent solutions for a set of well-known, diverse programming problems. The overall motivation is to be able to compare solutions, within and between, different programming languages. While the perspectives for comparing solutions have originally essentially analyzed runtime performance, the fact is that CLBG has also been used in order to study the energy efficiency of software [8, 29, 33].

In its 23.03 version, the CLBG has gathered solutions for 10 benchmark problems, such that solutions to each such problem must respect a given algorithm and specific implementation guidelines. The *pidigits* problem was excluded from our experiments due to installation compatibility issues; *pidigits* was also excluded from the study in [38].

The complete list of 9 benchmark problems from CLBG that we have considered can be found in Table 1. These problems cover a substantial range of computational problems.

Table 1. CLBG corpus of programs.

Benchmark	Description	Input
`n-body`	Double precision N-body simulation	50M
`fannkuch-redux`	Indexed access to tiny integer sequence	12
`spectral-norm`	Eigenvalue using the power method	5,500
`mandelbrot`	Generate Mandelbrot set portable bitmap file	16,000
`regex-redux`	Match DNA 8mers and substitute magic patterns	fasta output
`fasta`	Generate and write random DNA sequences	25M
`k-nucleotide`	Hashtable update and k-nucleotide strings	fasta output
`reverse-complement`	Read DNA sequences, write their reverse-complement	fasta output
`binary-trees`	Allocate, traverse and deallocate many binary trees	21

Solutions to each benchmark problem are expressed in, at most, 26 different programming languages.

In order to streamline our experiment, we decided not to consider Fortran and Smalltalk due to licensing restrictions that prevented their installation.

We also excluded Chapel, Lisp, Pascal, and F# because we either encountered execution errors and/or we were not able to find all implementations for some CLBG benchmark.

The 20 programming languages, and respective versions, that we considered in our study are described in Table 2:

Table 2. Programming Languages and Respective Versions

Language	Version
Ada	GNATMAKE 11.3.0
C	12.2.0-3ubuntu1
C++	Ubuntu 12.2.0-3ubuntu1
C#	.NET SDK 7.0.200
Dart	Dart SDK version: 2.18.6 (stable)
Erlang	Erlang/OTP 25 [erts-13.1.4]
Go	go version go1.20 linux/amd64
Haskell	The Glorious Glasgow Haskell Compilation System, version 9.4.4
Java	openjdk 20 2023-03-21
JavaScript	nodejs build 19.0.1+10-21
Julia	julia version build 19.0.1+10-21
Lua	Lua 5.4.4 Copyright (C)
OCaml	OCaml native-code version 5.0.0
Perl	This is perl 5, version 36
PHP	PHP 8.2.1 (cli)
Python	Python 3.11.1
Racket	Racket v8.7 [cs]
Ruby	ruby 3.2.0
Rust	1.67.0
Swift	Swift version 5.8-dev

3.2 Language Specific Benchmarks

In addition to studying the impact of power capping using benchmarks that were collected within CLBG, we have also considered language-specific benchmarks.

We aimed at studying languages from different paradigms and execution models, and therefore chose to target Haskell, a functional compiled programming language, Python, an interpreted multi-paradigm language, and Java, an object-oriented programming language running on a virtual machine. For these languages, we consider well-established and widely used benchmarks. We have used the same version of these programming languages as described in Table 2.

For Haskell, we considered the *The nofib Benchmark Suite of Haskell Programs* [36].[8]. The suite includes 98 programs, from toy benchmarks to real applications. From this suite, we considered 27 out of 29 programs in the *real* category, which corresponding to real-sized applications. We were not able to use the `ben-raytrace` and `smallpt` programs due to execution errors occurring.

For the Java language, we considered *The DaCapo Benchmark suite* [2], version 23.11-chopin.[9] The suite considers 22 open-source, real-world applications with non-trivial memory loads. Out of those 22 applications, we were able to study 20; we excluded `cassandra` and `h2o` since they only work in older versions of the JDK than the one used in the study.

[8] https://gitlab.haskell.org/ghc/nofib. Version available on January,2024 (last commit SHA 7ffecc8115865fea9995a951091e6ff23cf8ca3a)
[9] https://www.dacapobench.org/

For the Python language, we considered *The Python Performance Benchmark Suite*, version 1.11.0.[10] The suite includes 87 programs, ranging from microbenchmarks focused on Python interpreter start-up time to high-level applicative benchmarks. From this suite, we considered all the programs that are compatible with the Python version that we used; these account for a total of 75 programs, out of which 4 represent real-sized benchmarks.

3.3 Energy Measurements and Power Capping

For measuring the energy consumption, we used Intel's Running Average Power Limit (RAPL) [10]. RAPL has been shown to provide accurate energy estimates for Intel architectures at a very fine-grained level with a negligible overhead [14, 21, 42]. RAPL reports energy estimations of different power domains, physical hardware elements for which it is possible to manage power. Examples of such hardware elements include the processors (PP0), RAM (DRAM), uncore components (L3 cache, memory controllers), and package (the cores plus uncore and on-chip interfaces for memory and IO). Intel's RAPL framework has been used in several studies with overlapping interests and goals as ours [22, 29, 34, 37].

RAPL also supports the possibility of defining a power limit on the CPU while executing a program [17], which consists of a key feature for our study. In fact, *RAPLCap*[11] provides a C interface for managing Intel RAPL power caps. RAPL can be configured via RAPLCap with different values to control the power cap of a specific CPU. Thus, when executing a program it is possible to define the (maximum) power used by the (Intel) CPU.

RAPLCap operates by constantly monitoring and managing the processor's power usage in real-time. It limits energy consumption by dynamically adjusting the voltage and clock frequency of hardware components, ensuring that the average power stays within the limit, preventing excessive power draw and potential wastage.

Different Intel CPUs may have the lowest energy consumption by using different power cap values. In order to know the concrete value that produces the lowest energy consumption in a specific CPU, we considered a calibration phase where we executed with different RAPLCap values a series of benchmark programs [25, 26]. In this phase, we tested the range of values: 2, 10, 15, and 25, all in Watts. We then executed every CLBG benchmark for every programming language with each of these power caps values. We found that the value 15 was the one producing the most energy efficient executions, on average.

In our study, and the remainder of the paper, when we refer to power capping we then mean that the power was capped to 15 Watts. Additionally, we retain only the minimal processes in the operative system of the equipment described in Section 3.6. The measurements in the study include the CPU's idle energy values.

[10] https://github.com/python/pyperformance
[11] https://github.com/powercap/raplcap

3.4 Experimental Setup

3.4.1 Language Agnostic Benchmarks

To examine the effect of power capping on CLBG problems, we collected all previously mentioned solutions for each programming language from the CLBG website and installed the necessary libraries and compiler versions. For each language, every CLBG problem solution was executed 20 times, both with and without power capping. We maintained a consistent temperature during the execution of all programs and benchmarks. After collecting all the measurements, we removed the three highest and three lowest energy-consuming executions to ensure consistent samples for each language and each program across both power cap values. We had an issue in one of the Go executions with spectral-norm which gave us 19 executions for that solution. In this case, specifically, we removed the two highest and three lowest energy-consuming executions to have the same number of executions in all of the problems.

3.4.2 Language Specific Benchmarks

To analyze the impact of power capping on language-specific benchmarks, we collected the repositories of nofib and DaCapo and installed the PyPerformance benchmark via PyPI on the machine described in Section 3.6.

Each benchmark/program was then executed 20 times without power capping and 20 times with power capping. We have also ensured consistent temperature across the execution of all programs and benchmarks. After gathering all the measurements, the 3 highest and the 3 lowest energy-consuming executions were removed to ensure consistent samples for each language and each program across both power cap values.

3.5 Statistical Analysis

We employed statistical analysis to assess whether significant differences are observed between groups of energy and time measurements, ones collected with power capping and the others collected without power capping. This was conducted for each programming language and for each CLBG programs.

We first assessed the normality of the data in each group using the Shapiro-Wilk test. This was crucial for selecting the appropriate statistical test for each paired sample. The next step involved applying statistical tests based on the normality of the samples:

- Paired Samples t Test was applied to samples following a normal distribution.

- Wilcoxon signed-rank test was applied to samples not following a normal distribution.

3.6 Experimental Equipment

All our experiments were executed on a computer with an installation of Linux Ubuntu Server 22.04.3 LTS operating system. The system's specifications are the following: x86_64 Intel(R) Core(TM) i5-4460 3.20 GHz with 16 GB RAM, 4 cores and 4 threads.

4 Results and Analysis

In this section, we present and discuss the results obtained using benchmark problems from both the Language Agnostic Benchmarks (Section 4.1) and the Language Specific Benchmarks (Section 4.2). We consider energy consumption and runtime results, both with and without power caps. The energy usage we report is the package estimation provided by RAPL. Based on these results, we develop a Ranking of Programming Languages, initially without considering the impact of power caps and subsequently with the power cap taken into account (Section 4.3).

4.1 Language Agnostic Benchmarks

We start by providing the results we obtained taking benchmarks from the CLBG. In Table 3, we show, for each of the 9 CLBG benchmarks and each of the 20 programming languages we considered, the changes in runtime and energy efficiency that were observed when capping power, compared to the same executions without power capping. Changes are given in percentage, where an improvement/reduction is shown as a negative percentage. Changes observed in energy consumption are displayed in green and changes regarding runtime are displayed in blue.

As an example, the first entry of the table shows that running fannkuch-redux in Ada with power capping allowed saving slightly over 13% of energy consumption while running time was degraded more than 150%.

Regarding the information synthesized in Table 3, a notable realization can be made: *all* green values are negative and *all* blue values are positive. In practice, this means that capping power implied energy savings and running time degradation in *all* the 180 different combinations of a programming language and a benchmark.

The energy savings that were observed range from a minimum of 6% and a maximum of 27%; the runtime degradation ranged from a minimum of 35% and a maximum of 128%.

Our findings are reaffirmed by the statistical analysis we have conducted.

The Shapiro-Wilk test on all energy groups (totaling 360 groups from 20 languages $\times$ 9 programs $\times$ 2 power cap values) revealed that: 119 groups under power capping followed a normal distribution while 61 did not; 130 groups without power capping followed a normal distribution while 50 did not. Regarding execution time, 116 groups under power capping followed a normal distribution while 64 did not; 96 groups without power capping 0 a normal distribution while 84 did not. To further analyze the data distribution of each group not following a normal distribution, we generated Q-Q plots, which are available in our GitHub repository. Although the data from these groups does not follow a normal distribution, we observe that the points do not significantly deviate from the theoretical quantiles, suggesting no errors in the applied methodology. In fact, we believe that if we collect more samples we will observe normal distributions in all the populations because the effects of power capping on energy consumption are likely to follow the central limit theorem if observed multiple times.

We then applied statistical tests as described in Section 3.5. Whenever we rejected the null hypothesis, this would mean that there is sufficient evidence of differences in energy consumption or execution time when comparing results obtained with power capping with results obtained in its absence.

The Wilcoxon Signed-Rank test and the Paired Samples t test revealed significant differences in energy consumption and execution time across *all* groups of Table 3 compared pairwise, ones with power capping compared to the respective groups without capping, with *all* p-values being below the significance level of 0.05. Thus, these results provide clear evidence that the presence of power capping significantly impacts both energy consumption and execution time of programming languages.

We proceeded by looking at the runtime and energy consumption differences for each programming language. In Figures 1a and 1b, we present the grouped percentage differences in energy consumption and execution time for all programming languages, respectively.

Regarding energy efficiency and the information provided in Figure 1a, Haskell observed the greatest reduction, of 27.3% while Julia exhibits the smallest reduction, of 6.33%. On average, power capping allowed savings of 13.75% across all programming languages.

Regarding execution time and the information provided in Figure 1b, we observe that the Ruby language exhibits the largest increase in execution time, of 127.9% whereas Julia shows the smallest increase, of 35.34%. On average, power capping incurred in a runtime increase of 91% across all programming languages.

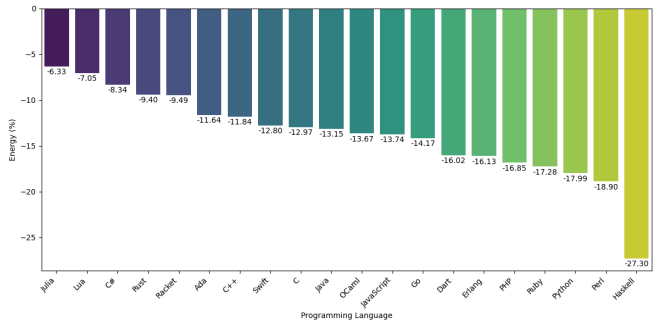
We further inspected the effects of power capping using benchmarks from the CLBG. This was done by grouping programming languages within 3 categories: i) the ones that are compiled, ii) the ones that are interpreted, and iii) the ones running on a Virtual Machine. Our goal with this analysis is to understand whether power capping affects different kinds of programming languages similarly.

In Figures 2, 3 and 4, we present the results for compiled languages, interpreted languages, and programming

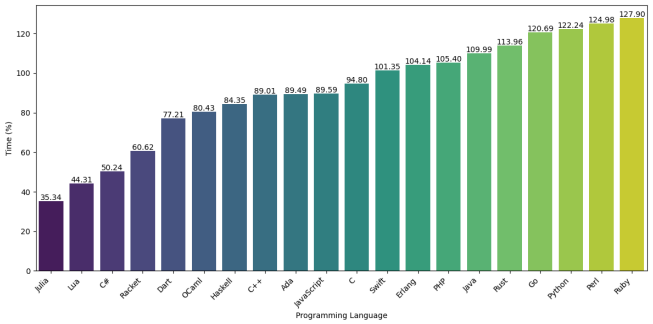
Table 3. Percentual differences in energy consumption (shown in green at the top of each cell) and execution time (shown in blue at the bottom of each cell) for CLBG problems across various programming languages

	fannkuch-redux	n-body	spectral-norm	mandel-brot	regex-redux	fasta	k-nucleotide	reverse-complement	binary-trees
Ada	-13.21 +150.59	-3.43 +21.37	-18.88 +164.46	-11.25 +110.76	-13.07 +110.58	-3.53 +20.35	-12.85 +59.15	-7.53 +40.14	-13.37 +122.72
C	-12.18 +130.04	-3.19 +17.90	-13.31 +151.46	-10.83 +125.93	-10.46 +92.37	-2.21 +13.48	-17.81 +114.15	-13.75 +42.77	-16.36 +131.67
C++	-12.22 +126.92	-3.57 +21.46	-8.50 +97.28	-11.49 +125.73	-8.02 +66.45	-7.93 +45.02	-18.27 +88.05	-8.78 +28.63	-16.48 +132.41
C#	-12.09 +123.29	-2.37 +19.79	-17.19 +95.90	-12.83 +121.72	-8.48 +69.57	-11.43 +118.45	-15.72 +127.19	-12.31 +64.06	-4.84 +26.43
Dart	-14.62 +153.76	-4.34 +20.59	-11.34 +95.37	-13.78 +128.85	-8.76 +50.90	-4.47 +27.52	-24.12 +64.20	-10.16 +42.62	-17.23 +78.08
Erlang	-18.24 +148.13	-5.38 +35.21	-15.75 +164.82	-22.50 +189.50	-10.83 +71.44	-8.55 +40.92	-19.11 +122.00	-11.78 +99.14	-16.99 +111.49
Go	-14.05 +144.80	-4.50 +19.86	-8.86 +96.81	-11.37 +124.91	-7.65 +52.17	-14.18 +117.29	-16.04 +149.59	-6.96 +30.11	-17.17 +160.8
Haskell	-14.16 +157.79	-3.83 +20.51	-7.72 +94.16	-9.41 +99.63	-10.10 +87.22	-11.03 +105.00	-38.19 +71.45	-7.70 +33.58	-14.34 +93.07
Java	-14.05 +157.15	-2.71 +20.37	-8.06 +94.06	-9.32 +124.24	-14.24 +126.24	-13.20 +93.22	-14.00 +135.22	-6.48 +39.14	-27.13 +95.15
JavaScript	-15.37 +152.81	-3.07 +24.61	-8.28 +96.55	-10.41 +114.52	-6.23 +44.48	-16.62 +99.67	-17.74 +84.54	-4.50 +29.66	-17.04 +121.02
Julia	-3.08 +28.49	-3.48 +25.86	-3.10 +15.51	-3.93 +19.36	-5.01 +28.34	-7.12 +20.45	-4.84 +35.14	-12.59 +36.67	-12.25 +59.52
Lua	-1.94 +29.82	-4.80 +32.56	-4.69 +32.53	-16.07 +144.98	-3.59 +27.35	-4.76 +31.02	-5.46 +32.28	-5.69 +31.23	-22.75 +115.56
OCaml	-14.12 +157.00	-3.99 +20.93	-0.86 +13.66	-11.26 +127.49	-9.68 +64.77	-4.69 +25.32	-20.57 +73.15	-12.15 +102.83	-15.64 +102.17
Perl	-22.23 +172.13	-4.37 +38.07	-16.86 +179.84	-22.02 +170.01	-14.08 +124.77	-5.45 +36.16	-21.65 +127.13	-5.52 +33.77	-21.14 +139.45
PHP	-21.26 +169.35	-5.12 +35.51	-19.46 +172.46	-19.57 +175.21	-9.20 +65.12	-6.36 +32.49	-22.11 +119.34	-9.41 +52.18	-21.94 +116.86
Python	-19.82 +180.70	-6.57 +37.54	-20.04 +182.90	-23.44 +175.40	-10.53 +69.61	-4.79 +35.19	-23.84 +133.07	-8.49 +60.52	-19.42 +133.45
Racket	-15.23 +149.17	-3.90 +26.52	-8.77 +93.39	-8.55 +101.56	-10.01 +70.03	-4.63 +25.15	-5.13 +23.32	-4.23 +27.35	-17.73 +59.28
Ruby	-20.24 +180.15	-4.26 +35.22	-16.87 +171.36	-20.55 +167.16	-9.96 +75.75	-5.03 +36.84	-19.60 +149.94	-5.08 +75.07	-15.02 +123.29
Rust	-9.27 +140.18	-4.45 +25.49	-6.41 +107.99	-10.09 +142.36	-9.37 +99.68	-9.00 +73.46	-11.19 +105.46	-11.38 +69.51	-14.73 +149.96
Swift	-12.16 +154.03	-1.17 +24.62	-7.01 +101.94	-11.02 +153.23	-8.85 +76.90	-5.54 +118.87	-20.61 +107.25	-6.57 +38.28	-13.33 +119.75

languages running on a virtual machine, respectively. Each figure includes a dual-axis chart with columns in one axis describing energy results and lines in the other describing runtime results. Green columns/lines represent results with power capping: green columns are on the left, and green lines are at the top. Blue columns/lines represent results without power capping: blue columns are on the right, and blue lines are at the bottom. These results are given in joules, for energy, and in seconds, for time, to provide a comprehensive analysis of the impact of power capping also in terms of its absolute impact.



(a) Impact on Energy



(b) Impact on Runtime

Figure 1. Comparison of Power Capping impact on Energy and Runtime for different Programming Languages

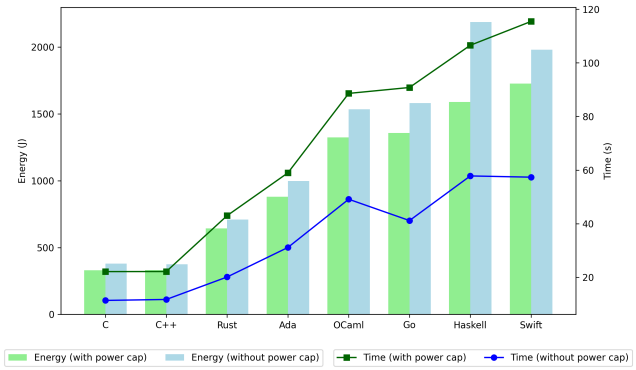


Figure 2. Power Capping Compiled Programming Languages

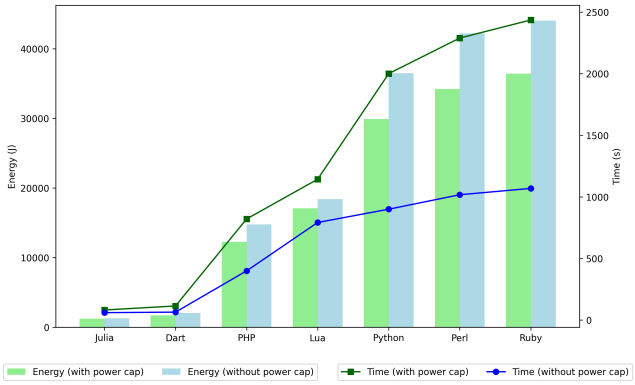


Figure 3. Power Capping Interpreted Programming Languages

Among compiled programming languages, the information in Figure 2 confirms that C and C++ are highly optimized languages, both for energy consumption and runtime; this was consistently observed in previous studies [37, 38]. Cross-checking this information with the one we had seen in Figure 1a, we provide evidence that power capping can be employed to achieve energy savings even within highly energy-efficient programming languages: energy savings of circa 12% were observed for C and C++.

Analyzing the performance of interpreted programming languages taking into account the information presented in Figure 3, we observe that Julia and Dart are notably more efficient than the other interpreted languages, namely PHP, Lua, and even more than Python, Perl and Ruby. This increased efficiency is observed in factors of ten, which is also aligned with the previous studies [37, 38]. We note, however, that previous studies have not considered Julia whose efficiency our works contributes to shed light upon.

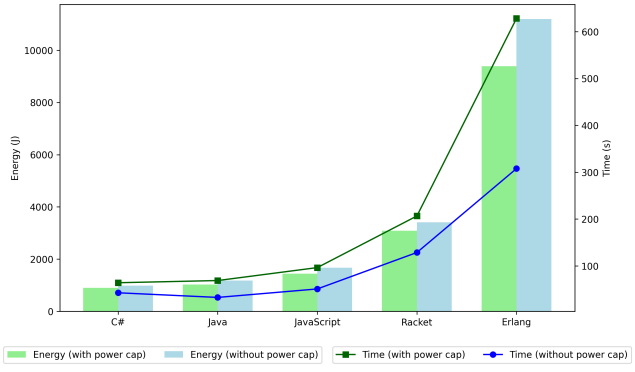


Figure 4. Power Capping Programming Languages running on Virtual Machines

C#, Java and Javascript have demonstrated comparable efficiency, in terms of both runtime and energy. They have also shown to be significantly more efficient than Racket and much more efficient than Erlang; focusing on energy efficiency, the latter is tenfold less efficient.

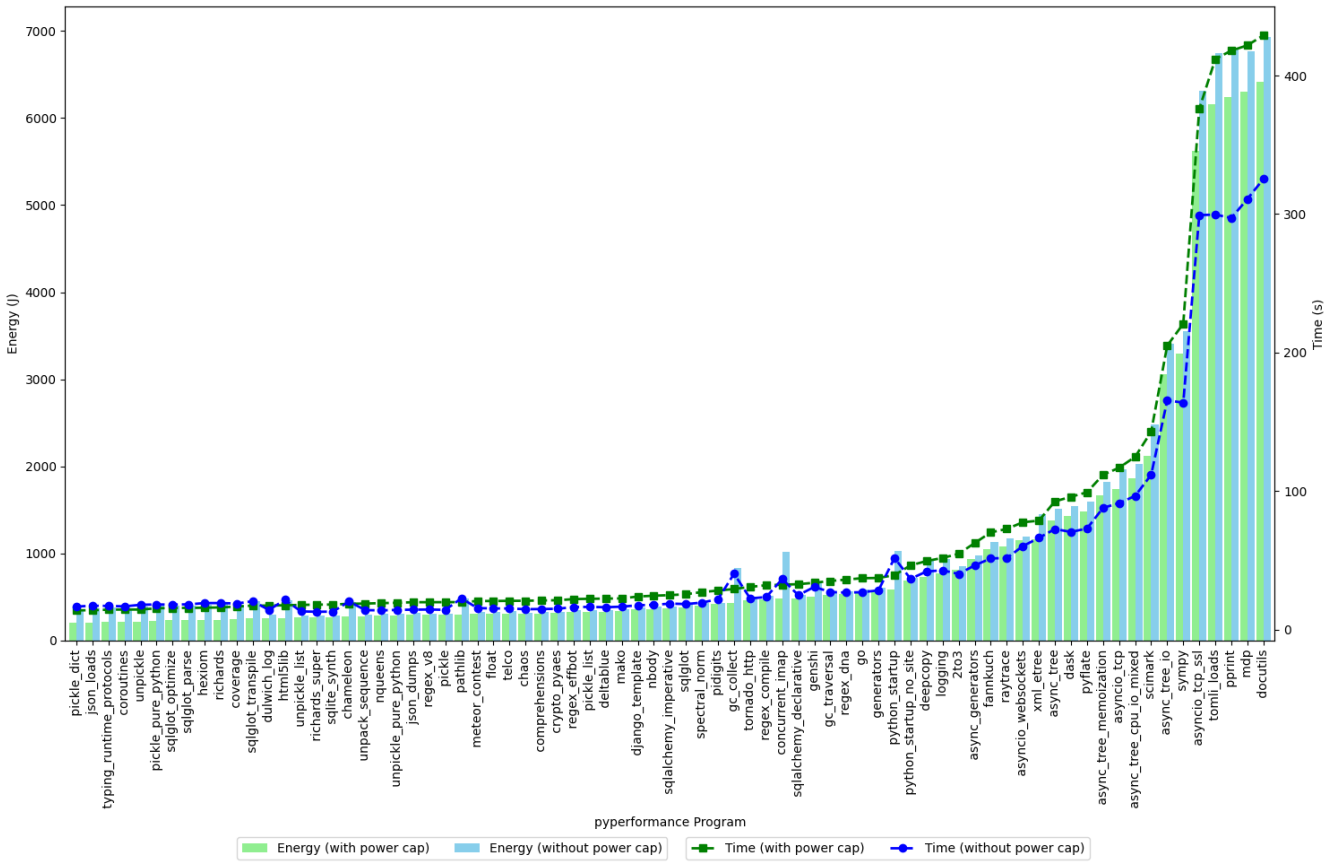


Figure 5. Energy Consumption and Time for pyperformance Programs

4.2 Language Specific Benchmarks

In the previous section, we analyzed the results we obtained from experimenting with benchmark problems taken from the CLBG.

In this section, we consider benchmark problems taken from the language specific benchmarks we described earlier, namely PyPerformance for Python, DaCapo for Java , and nofib for Haskell. Our goal is to assess whether the impact of power capping we observed before is consistently observed for different benchmarks as well.

In Figures 5, 6, and7, we show the runtime and energy consumption of all individual benchmark programs included in PyPerformance, DaCapo, and nofib, respectively. We show those results when programs execute without a powercap and with a powercap.

In Figure 8, we can observe the average values of energy consumption and execution time for each benchmark. We notice that PyPerformance is the highest consumer of both energy and time among the three benchmarks, while nofib is the lowest consumer in both metrics. According to Figure

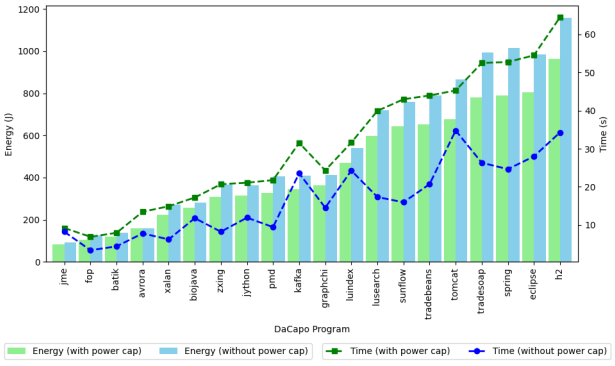


Figure 6. Energy Consumption and Time for DaCapo Programs

9[12], under power capping, DaCapo experiences the most significant changes, with a reduction of 20.84% in energy consumption and an increase of 45.58% in execution time.

[12]Note that darker red indicates worse percentage values, while darker green indicates better percentage values in the metric under consideration.

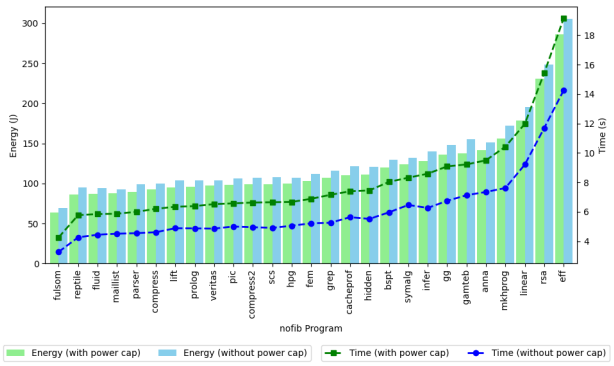


Figure 7. Energy Consumption and Time for nofib Programs

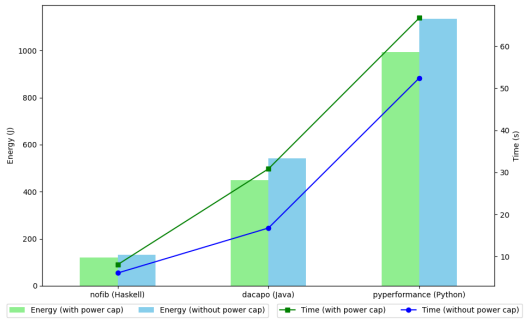


Figure 8. Power Capping Impact per Benchmark

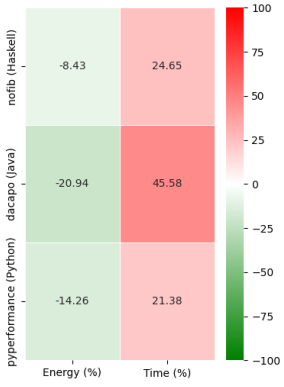


Figure 9. Average Differences per Benchmark

The results of the Language Specific Benchmarks show that the energy savings of Java and Python were similar to the ones observed using Language Agnostic Benchmarks. Specifically, Java achieved energy savings of 13% when considering benchmarks from DaCapo and 17% when using benchmarks from the CLBG. Python, in comparison, showed energy savings of 18% when experimenting with PyPerformance benchmarks and 12% using benchmarks from the CLBG. Additionally, Java demonstrated consistent runtime performance degradation, with an increase in runtime of 110% when considering benchmarks from DaCapo and 84% when considering CLBG benchmarks.

4.3 Power Capping on Ranking of Programming Languages

In this section, we base ourselves on the execution results of the CLBG programs, as described in Section 4.1, to establish a ranking of programming languages regarding their runtime and energy efficiency.

Table 4 compares energy and time values for all languages without power capping, sorted by normalized energy values. We note that here we are closely mimicking the approach followed in [37, 38] to rank programming languages. We did this to establish a baseline ranking that can be compared with a similar ranking, but one obtained in the context of power capping. Establishing a baseline ranking is needed so that both rankings are obtained in the same hardware setup. Notably, C++ and C exhibit the lowest energy consumption values, while on the other end of the spectrum, Perl and Ruby are the highest energy consumers without power capping.

Table 4. Ranking Programming Languages by Energy Consumption - Without Power Cap

Language	Energy Normalized	Time Normalized
C++	1.00	1.00
C	1.01	0.97
Rust	1.89	1.72
C#	2.63	3.64
Ada	2.66	2.66
Java	3.15	2.80
Julia	3.48	5.17
OCaml	4.09	4.19
Go	4.21	3.51
JavaScript	4.45	4.34
Swift	5.27	4.90
Dart	5.43	5.53
Haskell	5.82	4.94
Racket	9.08	10.98
Erlang	29.80	26.28
PHP	39.30	34.13
Lua	48.98	67.67
Python	97.12	76.89
Perl	112.40	86.92
Ruby	117.24	91.30

In Table 5 we compare energy and time values to establish a ranking for the same languages and programs but now with power capping enabled.

Comparing the new ranking with the former we observe minor changes. These changes are denoted by ↑k or ↓k, where k indicates the number of positions changed from the ranking without power capping. As an example, C and C++ have

Table 5. Ranking Programming Languages by Energy consumption - With Power Cap

Language	Energy	Time	Energy Difference (J)	Energy Difference %	Time Difference (ms)	Time Difference %
C ↑1	1.00	1.00	-49.28	-12.97	10759.14	94.80
C++ ↓1	1.00	1.00	-44.47	-11.84	10422.93	89.01
Rust	1.94	1.94	-66.71	-9.40	22896.14	113.96
Ada ↑1	2.66	2.67	-116.11	-11.64	27842.71	89.49
C# ↓1	2.74	2.90	-82.36	-8.34	21437.07	50.24
Java	3.11	3.11	-155.69	-13.15	36028.50	109.99
Julia	3.70	3.71	-82.75	-6.33	21413.36	35.34
OCaml	4.01	4.01	-209.78	-13.67	39498.64	80.43
Go	4.11	4.11	-224.19	-14.17	49663.29	120.68
JavaScript	4.36	4.36	-229.37	-13.74	45557.00	89.59
Haskell ↑2	4.81	4.82	-597.11	-27.30	48761.57	84.35
Dart	5.18	5.19	-326.84	-16.02	49972.64	77.21
Swift ↓2	5.22	5.22	-253.48	-12.80	58124.86	101.35
Racket	9.34	9.34	-323.80	-9.49	77956.14	60.62
Erlang	28.38	28.42	-1805.44	-16.13	320482.79	104.14
PHP	37.10	37.13	-2487.35	-16.85	421207.21	105.40
Lua	51.69	51.73	-1297.66	-7.05	351166.21	44.31
Python	90.44	90.50	-6562.49	-17.99	1100536.14	122.24
Perl	103.51	103.57	-7979.47	-18.90	1272059.64	124.98
Ruby	110.13	110.20	-7608.35	-17.28	1367355.07	127.90

swapped places between the rankings in Table 4 and Table 5, with C moving up one position and C++ moving down one.

We see that only 6 programming languages change position, among which 4 by one position and two by two positions. This seems to confirm that previous rankings of programming languages based on their energy efficiency are marginally affected by power capping.

Additionally, we provide the absolute and percentage differences in energy and time for all programming languages. Notably, the bottom two languages in the ranking of Table 5, Ruby and Perl, consume 110.13 and 103.51 times more energy, respectively, and take 110.20 and 103.57 times longer to execute compared to the top two languages, C and C++. Note that the absolute energy and time metrics refer to the total energy or time consumed by a particular programming language across all the CLBG programs.

5 Conclusions and Future Work

In this paper, we performed a comprehensive study comparing regular and power-capped executions of 9 benchmark programs from *The Computer Language Benchmarks Game* across 20 different programming languages. Furthermore, we identified similar trends using language-specific benchmarks for programming languages of various paradigms and execution models. This was accomplished by examining a broad spectrum of benchmark programs from the *nofib Benchmark Suite of Haskell Programs*, the *DaCapo Benchmark Suite for Java*, and the *Python Performance Benchmark Suite*.

Our findings reveal that implementing power caps can lead to a trade-off where increased running time is exchanged for enhanced energy efficiency across *all* programming languages and benchmarks examined in this study.

Given the research questions we defined at the beginning of our study, we can now address them:

RQ1 According to Figures 1a and 1b, both language-agnostic benchmarks and language-specific benchmarks show improved energy efficiency across all respective benchmarks. However, this improvement comes at the cost of increased runtime in all cases.

RQ2 According to Figures 1a and 1b, all language-agnostic benchmarks showed varying reductions in energy consumption and different increases in runtime. For example, Haskell demonstrated the largest reduction in energy consumption at 27.30% under power capping, whereas Julia showed a reduction of 6.33%. Regarding runtime, Ruby experienced a slowdown of 127.90%, whereas Julia's was 35.34%. Observing the language-specific benchmarks in Figure 9, the differences in energy and runtime are not uniform either. DaCapo, for instance, reduced its energy consumption by 20.94%, whereas nofib only reduced by 8.43%. In terms of execution time, DaCapo increased by 45.58%, while PyPerformance saw a smaller increase of 21.38%.

RQ3 According to the language ranking from Figure 5, we observe that the programming languages show minimal differences in ranking when under power capping compared to their rankings without it. C, C++, and Rust remain the greenest programming languages (and faster) even under power capping. Conversely, Ruby, Perl, and Python continue to be the top energy consumers (and slower).

5.1 Prospects for Further Study

Our study has focused primarily on prominent languages highlighted in the CLBG. However, there is potential for further investigation into additional languages such as Chapel and F#, which were not included in our current analysis. In future work, we plan to overcome this limitation.

Exploring various power capping values is crucial for finely adjusting the balance between energy efficiency and performance. This exploration, which deserves dedicated research in the future, can uncover optimal configurations where energy consumption is minimized without significant performance degradation.

Future research could also involve conducting a multicriteria analysis to systematically optimize the selection of programming languages. This approach would aim to achieve energy consumption reduction goals across diverse computational tasks and environments. Furthermore, delving into additional variables within our dataset could provide insights into how power capping influences various performance metrics beyond energy efficiency alone.

Finally, using RAPL we are only able to obtain energy readings for some of the components of a system. The energy savings we observe should, therefore, be interpreted as relative to those components. It would be highly valuable to expand this perspective with studies that determine the overall system improvements obtained by capping power.

Data Availability Statement

The data that support the findings of this study are openly available in Energy-Languages-PowerCap at https://doi.org/10.6084/m9.figshare.27087901.v1 [9].

Acknowledgements

This work is financed by national funds through Portuguese funding agency, FCT - *Fundação para a Ciência e a Tecnologia* within project `UIDB/50014/2020`, *DOI 10.54499-/UIDB/50014/2020*, and by COST Action 19135: "CERCIRAS - Connecting Education and Research Communities for an Innovative Resource Aware Society".

References

[1] Anders Andrae. 2020. New perspectives on internet electricity use in 2030. *Engineering and Applied Science Letters* 3 (Jun 2020), 19–31.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM, 169–190.

[3] Barry Boehm. 1981. *Software Engineering Economics*. Prentice Hall.

[4] Gerardo Canfora and Aniello Cimitile. 2000. Software Maintenance. *Journal of Software Maintenance: Research and Practice* 12, 6 (2000), 365–395.

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

[6] Stefano Conoci, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. 2018. Adaptive Performance Optimization under Power Constraint in Multi-thread Applications with Diverse Scalability. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE '18)*. ACM, 16–27.

[7] Marco Couto, Paulo Borba, Jácome Cunha, João Paulo Fernandes, Rui Pereira, and João Saraiva. 2017. Products go Green: Worst-Case Energy Consumption in Software Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A (SPLC '17)*. ACM, 84–93.

[8] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. 2017. Towards a Green Ranking for Programming Languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages (SBLP '17)*. ACM.

[9] Simão Cunha, Luís Silva, João Paulo Fernandes, and João Saraiva. 2024. Energy-Languages-PowerCap. (9 2024). https://doi.org/10.6084/m9.figshare.27087901.v1

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

[11] Steven Fraser, Dennis Mancl, Bill Opdyke, Judith Bishop, Pradeep Kathail, Junilu Lacar, Ipek Ozkaya, and Alexandra Szynkarski. 2013. Technical debt: from source to mitigation. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity (SPLASH '13)*. ACM, 67–70.

[12] Stefanos Georgiou, Maria Kechagia, Panos Louridas, and Diomidis Spinellis. 2018. What are your programming language's energy-delay implications?. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, 303–313.

[13] Isaac Gouy. [n. d.]. *The Computer Language Benchmarks Game*. https://benchmarksgame-team.pages.debian.net/benchmarksgame/site.html

[14] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. 2012. Measuring Energy Consumption for Short Code Paths Using RAPL. *SIGMETRICS Performance Evaluation Review* 40, 3 (Jan 2012), 13–17.

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

[16] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. Energy profiles of java collections classes. In *Proc. of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, 225–236.

[17] Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. 2019. CoPPer: Soft Real-Time Application Performance Using Hardware Power Capping. In *2019 IEEE International Conference on Autonomic Computing (ICAC'19)*. 31–41.

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

'15). IEEE Press, 8–14.

[19] Timothy M. Jones, Michael F. P. O'Boyle, Jaume Abella, Antonio González, and Oğuz Ergin. 2009. Energy-efficient register caching with compiler assistance. *ACM Transactions on Architecture and Code Optimization* 6, 4, Article 13 (oct 2009).

[20] I. Kadayif, M. Kandemir, G. Chen, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. 2005. Compiler-directed high-level energy estimation and optimization. *ACM Transactions on Embedded Computing Systems* 4, 4 (nov 2005), 819–850.

[21] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3, 2, Article 9 (mar 2018).

[22] Maja Hanne Kirkeby, Bernardo Santos, João Paulo Fernandes, and Alberto Pardo. 2024. Compiling Haskell for Energy Efficiency: Empirical Analysis of Individual Transformations. In *Proceedings of the 39th ACM Symposium on Applied Computing (SAC'24)*. ACM, 1104–1113.

[23] Lukas Koedijk and Ana Oprescu. 2022. Finding Significant Differences in the Energy Consumption when Comparing Programming Languages and Programs. In *2022 International Conference on ICT for Sustainability (ICT4S)*. IEEE Press, 1–12.

[24] Adam Krzywaniak and Paweł Czarnul. 2020. Performance/Energy Aware Optimization of Parallel Applications on GPUs Under Power Capping. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Ewa Deelman, Jack Dongarra, and Konrad Karczewski (Eds.). Springer, 123–133.

[25] Adam Krzywaniak, Pawel Czarnul, and Jerzy Proficz. 2019. Extended investigation of performance-energy trade-offs under power capping in HPC environments. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*. 440–447.

[26] Adam Krzywaniak, Paweł Czarnul, and Jerzy Proficz. 2022. DEPO: A dynamic energy-performance optimizer tool for automatic power capping for energy efficient high-performance computing. *Software-Practice & Experience* 52 (2022), 2598–2634.

[27] Adam Krzywaniak, Paweł Czarnul, and Jerzy Proficz. 2023. Dynamic GPU power capping with online performance tracing for energy efficient GPU computing using DEPO tool. *Future Generation Computer Systems* 145 (2023), 396–414.

[28] Ding Li and William G. J. Halfond. 2014. An investigation into energy-saving programming practices for Android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software (GREENS 2014)*. ACM, 46–53.

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

[30] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. 2014. Mining energy-greedy API usage patterns in Android apps: an empirical study. In *Proc. of the 11th Working Conference on Mining Software Repositories (MSR'14)*. ACM, 2–11.

[31] Xinxin Mei, Ling Sing Yung, Kaiyong Zhao, and Xiaowen Chu. 2013. A measurement study of GPU DVFS on energy conservation. In *Proceedings of the Workshop on Power-Aware Computing and Systems (HotPower '13)*. ACM.

[32] Sparsh Mittal and Jeffrey S. Vetter. 2014. A Survey of Methods for Analyzing and Improving GPU Energy Efficiency. *ACM Comput. Surv.* 47, 2, Article 19 (aug 2014), 23 pages. https://doi.org/10.1145/2636342

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

[34] Ana Oprescu, Sander Misdorp, and Koen van Elsen. 2022. Energy cost and accuracy impact of k-anonymity. In *Proc. of the International Conference on ICT for Sustainability (ICT4S'22)*. IEEE, 65–76.

[35] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, and Joël Penhoat. 2021. Evaluating the Impact of Java Virtual Machines on Energy Consumption. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '21)*. ACM.

[36] Will Partain. 1993. The nofib Benchmark Suite of Haskell Programs. In *Functional Programming, Glasgow 1992*, John Launchbury and Patrick Sansom (Eds.). Springer London, 195–202.

[37] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. Energy efficiency across programming languages: how do energy, time, and memory relate?. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017)*. ACM, 256–267.

[38] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2021. Ranking programming languages by energy efficiency. *Science of Computer Programming* 205 (2021).

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

[40] Thomas M. Pigoski. 1997. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons.

[41] Hannah Ritchie and Max Roser. 2020. Energy. *Our World in Data* (2020). Available at: https://ourworldindata.org/energy.

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

[43] Rui Rua, Tiago Fraga, Marco Couto, and João Saraiva. 2020. Greenspecting Android virtual keyboards. In *Proceedings of the IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft '20)*. ACM, 98–108.

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

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

[46] Madhavi Valluri and Lizy K. John. 2001. *Is Compiling for Performance — Compiling for Power?* Springer, 101–115.

[47] Werner Voegels. 2023. Keynote at AWS re:Invent 2023. https://youtu.be/UTRBVPvzt9w?t=3656 Accessed: 2024-05-19.

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

[49] W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. 2004. Reducing instruction cache energy consumption using a compiler-based strategy. *ACM Transactions on Architecture and Code Optimization* 1, 1 (mar 2004), 3–33.

[50] Dan Zhao, Siddharth Samsi, Joseph McDonald, Baolin Li, David Bestor, Michael Jones, Devesh Tiwari, and Vijay Gadepally. 2023. Sustainable Supercomputing for AI: GPU Power Capping at HPC Scale. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23)*. ACM, 588–596.

Received 2024-06-14; accepted 2024-08-30