On the Energy Consumption of CPython

Rolf-Helge Pfeiffer[1][0000−0003−2585−6473]

IT University of Copenhagen, Rued Langgaards Vej 7, 2300 Copenhagen ropf@itu.dk

Abstract. Interpreted programming languages, like Python, are amongst the most popular programming languages. This, combined with high developer efficiency leads to many web-application backends and web-services that are written in Python. While it is known that interpreted languages like Python are way less energy efficient compared to compiled languages like C++, Rust, etc., little is known about the energy efficiency of various versions of Python interpreters. In this paper, we study via a controlled lab experiment the energy consumption of various versions of the Python interpreter CPython when running a server-side rendered web-application. Our results indicate that currently the most energy efficient version is CPython 3.12. Energy consumption of CPython 3.12 can drop by more than 8% compared to previous versions.

Keywords: Software engineering · Energy consumption · CPython.

1 Introduction

Interpreted languages are amongst the most popular programming languages. Python is ranked to be the most popular programming language[1,2] in the early 2020s. The Python programming language is used in almost all domains of software development ranging from small scripts of "glue code", over scientific computing to development of web-applications.

Big corporations and projects use Python due to its ease of use and understandability which allows for short development times[3] [21, 25]. For example, Airbnb created and relies on the Python-based workflow management platform Apache Airflow,[4] Google's YouTube is powered by Python [27], circa 20% of Facebook's infrastructure is written in Python,[5] Instagram is a Python application using the Django web-framework,[6] or circa 80% of Spotify's backend services are written in Python.[7] Since such big projects and in general evermore software is

1 `https://www.tiobe.com/tiobe-index/`

2 `https://spectrum.ieee.org/the-top-programming-languages-2023`

3 `https://computerhistory.org/profile/guido-van-rossum/`

4 `https://airflow.apache.org/`

5 `https://engineering.fb.com/2016/05/27/production-engineering/python-in-production-engineering/`

6 `https://instagram-engineering.com/web-service-efficiency-at-instagram-with-python-4976d078e366`

7 `https://engineering.atspotify.com/2013/03/how-we-use-python-at-spotify/`

written in high-level interpreted languages like Python and since we as software engineers have to contribute to reducing the carbon footprint of our sector [1], software qualities like sustainability in the sense of energy efficiency become more important.

Usually, interpreted programming languages are slower than programming languages that are compiled to native code. Therefore, the authors of CPython started to focus on improving performance of the interpreter,[8] e.g., by introduction of Just-In-Time compilation, removal of the so-called global-interpreter lock,[9] etc. CPython is the open-source reference implementation of the Python[10] programming language originally implemented by Guido van Rossum, which is now maintained and improved by a team of developers.[11]

Besides being less performant, interpreted programming languages are reported to be of low energy efficiency, see e.g., [2, 6, 19, 20]. But it seems that energy efficiency is of secondary interest for language designers and maintainers so far. For example, none of the current Python Enhancement Proposals (PEPs)[12] is about energy efficiency.

Additionally, it is unclear how current performance improvements of CPython[13] impact energy efficiency. Furthermore, there is little knowledge on how to assess energy efficiency of various versions of language interpreters like CPython.

In this paper, we investigate the energy consumption of various versions of CPython. Our research question is: *What is the energy consumption of different versions of the Python interpreter CPython?* To answer this question, we design and conduct a controlled lab experiment. Before executing our experiment, we assume that newer versions of Python are less energy efficient due to increasing language features and internal extensions. Our assumption is loosely inspired by "Wirth's law" [28] which states that *"software manages to outgrow hardware in size and sluggishnes"* [23].

To investigate our research question, we conduct a controlled lab experiment in which we measure energy consumption of a server running a web-application. Three clients execute long-running *scenarios* against the server, where a scenario is an automated sequence of actions that a potential user could perform, like requesting a page, registering a user, etc. The only variable in our experiment is the version of CPython running the application. The web-application, its dependencies, the operating system (FreeBSD), webserver (Gunicorn), etc. are all fixed to the same versions.

Contrary to our initial assumption, our results indicate that newer versions of CPython (versions CPython 3.11, 3.12, and 3.13) are not only more performant, they are also more energy efficient than previous versions. In particular, CPython 3.12 appears to be currently the most energy efficient CPython interpreter. On

[8] `https://github.com/markshannon/faster-cpython/blob/master/plan.md`
[9] `https://peps.python.org/pep-0703/`
[10] `https://www.python.org/`
[11] `https://devguide.python.org/core-developers/developer-log/#developers`
[12] `https://peps.python.org/`
[13] `https://www.python.org/downloads/release/python-3110/`

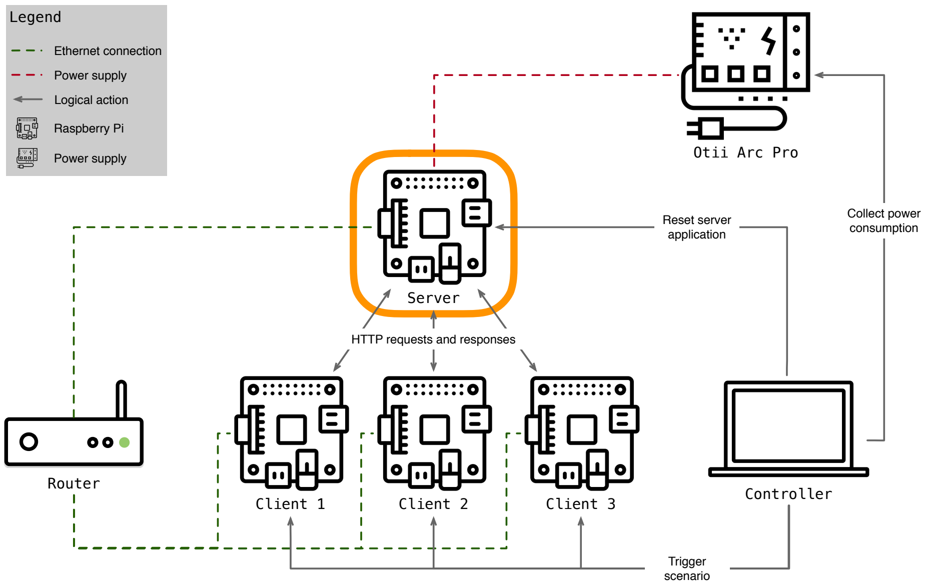


Fig. 1: Illustration of our experiment design. Highlighted in orange, is the webserver for which we study energy consumption.

average it consumes 8.41% less energy than the most energy consuming version 3.10. Consequently, software developers or maintainers can make their CPython-based products more sustainable plainly by updating to CPython 3.12.

Our contributions are:

- We present an experiment design to directly measure energy consumption of a server in a networked client-server architecture. Our design can serve as a blueprint for similar sustainability experiments.
- We demonstrate that recent versions of CPython are more energy efficient than older versions.
- We provide a complete replication kit with automated experiment together with this paper.[14]

2 Experiment Design

In this section, we describe how we measure and evaluate the energy consumption of various versions of CPython in a controlled lab experiment.

Fig. 1 illustrates our experiment setup with one webserver (center in Fig. 1) and three clients (bottom center in Fig. 1). The three clients and the webserver are all Raspberry Pis (Model B+ V1.2) that are connected via ethernet to a

[14] https://github.com/HelgeCPH/cpython_energy

Wi-Fi router in a network. The webserver is powered by an Otii Arc Pro.[15] This is a power supply with integrated meter that allows to accurately profile and analyze voltage, current, power, etc. That is, we collect power draw of the server directly compared to indirect estimation of energy consumption, e.g., via Intel's Running Average Power Limit [7] or similar.

The power supply is connected to a computer which we call *controller* via USB (bottom right in Fig. 1). On that computer runs the Otii software suite[16] that allows us to collect measurement data about power draw, current, voltage, etc. Besides used as data collector, the *controller* orchestrates single runs of the experiment.

The webserver and clients are provisioned with FreeBSD 13.3. On the webserver, we install CPython 3.8, 3.9, 3.10, 3.11, 3.12, and 3.13. These are the versions of CPython for which we measure energy consumption in our experiment. CPython 3.8, 3.9, 3.10, and 3.11 are installed via their respective packages with the `pkg` tool.[17] These contain pre-compiled distributions of the latest versions of the respective CPython interpreter for the ARM architecture of the Raspberry Pi. Since no pre-compiled packages exist yet for CPython 3.12 and 3.13, we install these via `pyenv`.[18]

Previous work on assessment of energy consumption of software often relies on exemplary implementations of certain algorithms, see e.g. [2,6,10,17,19,20,26]. However, we decide that we want to conduct our experiment in an environment that resembles a more realistic software product, similar to [6, 9]. Therefore, the server will run a server-side rendered web-application called MiniTwit. It is a Twitter-like minimal micro-blogging web-application. The MiniTwit application is originally written in Python 2 by Armin Ronacher. It used to be an example application for his Python web-framework Flask.[19] Python 2 and 3 are incompatible versions of the programming language. Therefore, we update the original MiniTwit application to be compatible with Python 3 and contemporary versions of the Flask framework.

The MiniTwit Flask application is served via the WSGI HTTP server Gunicorn,[20] which is a Python application too. For this experiment, we do not serve the MiniTwit web-application via a reverse proxy server like nginx[21] or the Apache HTTP Server[22] as likely done in a production setup. The main reason is that we want to prevent caching of responses, so that the effect of the Python interpreters on energy consumption is better represented.

Consequently, to be operational, the MiniTwit application requires Flask and Gunicorn to be installed on the server. Both of these are distributed as Python

[15] `https://www.qoitech.com/otii-arc-pro/`

[16] `https://www.qoitech.com/software/`

[17] `https://man.freebsd.org/cgi/man.cgi?query=pkg&sektion=8&format=html`

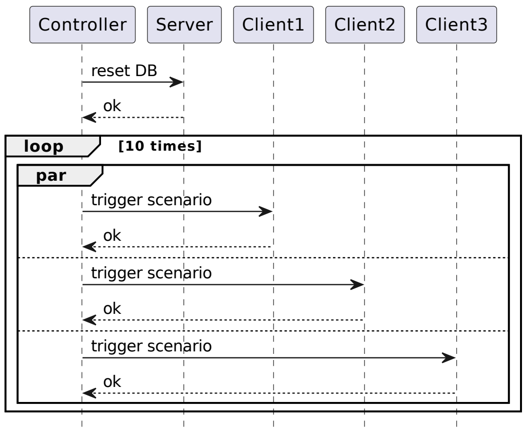
[18] `https://github.com/pyenv/pyenv`

[19] `https://github.com/pallets/flask/tree/1592c53a664c82d9badac81fa0104af226cce5a7/examples/minitwit`

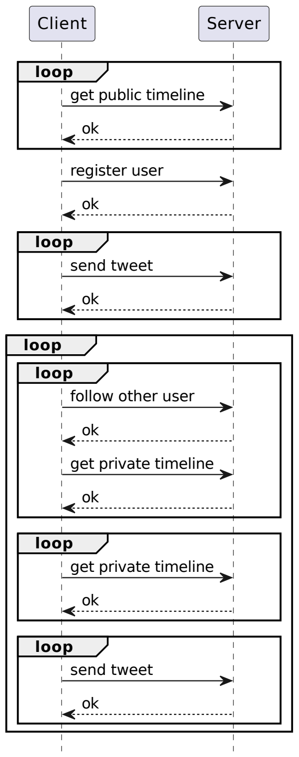
[20] `https://gunicorn.org/`

[21] `https://nginx.org/`

[22] `https://httpd.apache.org/`



(a) Illustration of ten *experiment runs*, i.e., how the *controller* executes ten iterations of the experiment scenario in parallel across the three clients.



(b) Illustration of a single scenario that a client executes against the web-application on the server.

Fig. 2: Illustration of ten *experiment runs* (left) and a *scenario* (right).

packages on the Python Packaging Index (PyPI).[23] We manually download the latest versions of these packages, together with their transitive dependencies, from PyPI. We do that so that we can vendor pinned versions of dependencies with our experiment setup to facilitate reproduction and replication. Note, a replication kit is provided online.[14] It contains complete descriptions on how to prepare the experiment computers, provisioner scripts for the server and clients, etc. To sum up, we make sure that the experiment variable in our experiment is the precise version of CPython running the web-application.

The technical setup described above, see Fig. 1, provides the environment for experiment execution. The experiment consists of so-called *scenarios*. In a scenario, a client performs a sequence of actions against the MiniTwit web-

[23] https://pypi.org/

application on the server. These are actions like loading the application's front-page (which lists latest tweets), registering a new user, sending tweets, following other users, and requesting a user's private timeline (which displays only a selection of tweets). Fig. 2b sketches such a scenario as UML sequence diagram. The exact scenario can be inspected in the replication kit.

Per version of CPython (3.8 to 3.13), we execute ten iterations of the same scenario in parallel from the three clients, as illustrated in Fig. 2a. That is, each of the three clients executes ten times the same scenario against the server. We execute ten iterations of the same scenario per version of CPython to minimize the effect of potentially running other processes on the multi-tasking operating system FreeBSD.

Note, before the ten parallel scenarios are executed for a version of CPython, the *controller* resets the server's web-application. That is, the database of users, their tweets, and their follower structure is reset to a predefined state so that the MiniTwit application operates in the completely same environment for each version of CPython. Once that step is completed, the *controller* starts the execution of the scenario on each of the three clients. We call the execution of a scenario in parallel over three clients an *experiment run*. Per experiment run, we record the power draw of the server (in Watt) with 10000 samples per seconds and the time it took to execute. That is, the reset of the web-application is not recorded in our measurements. Only the energy consumption of the body of the loop in Fig. 2a is recorded. All measurement data is stored automatically in CSV files, which are also distributed in our replication kit.

The experiment runs per version of CPython are completely automated using the Otii Automation Toolbox.[24,25] After recording energy consumption of one version of CPython, we manually log to the server from the controller, stop the web-application, and switch to the next version of CPython.

3 Results

The results of our measurements of the server's power draw for a single experiment run look as illustrated in Fig. 3. This plots the power draw of the first experiment run in which the MiniTwit web-application runs on CPython 3.10. It illustrates power draw over time with 10000 samples per second (x-axis). On the y-axis power draw in Watt (W) is plotted. It can be seen that this experiment run lasts for ca. $2min23s$. On average, the server's power draw is $\mu \approx 1.286$W ($\sigma \approx 0.034$ and $q_2 \approx 1.282$W). Minimum and maximum power draw are $q_0 \approx 1.133$W and $q_4 \approx 1.685$W respectively.

Note, for reporting we use the following notation: q_0 denotes the minimum, q_1 the 25th quartile, q_2 the median, q_3 the 75th quartile, q_4 the maximum, μ is the arithmetic mean (we use average and arithmetic mean synonymously), and the standard deviation is denoted by σ.

[24] `https://www.qoitech.com/docs/advanced/scripting-with-python`

[25] `https://github.com/HelgeCPH/otii-tcp-client-python`

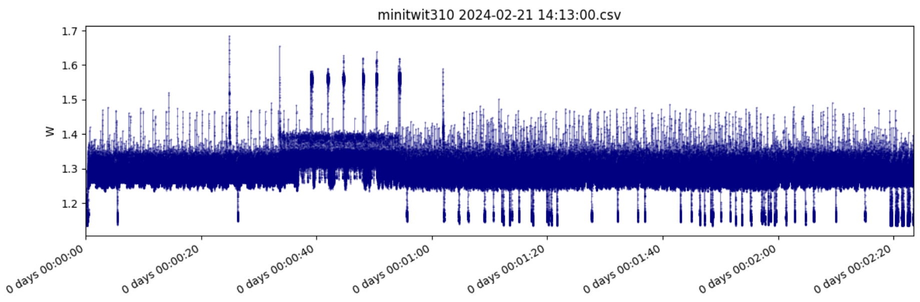
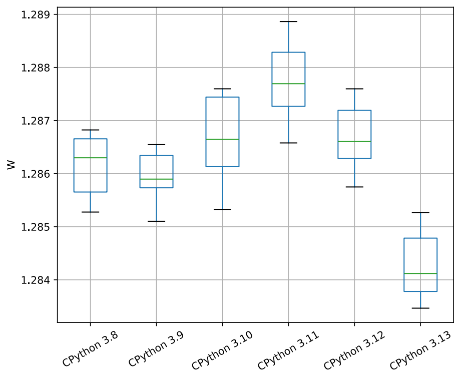
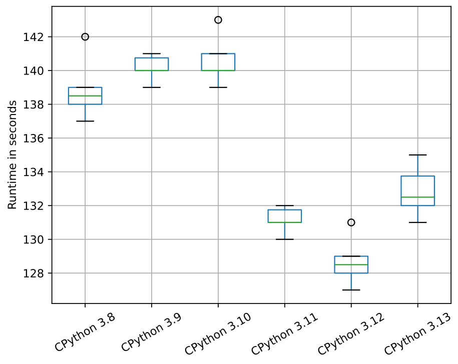


Fig. 3: Power draw of the first experiment run in which the MiniTwit web-application runs on CPython 3.10.

To facilitate statistical analysis and visual inspection, we compute the average power draw per experiment run and plot these in a box plot, see Fig. 4a. It shows that, CPython 3.11 has the highest power draw for all ten experiment runs ($q_0 = 1.286579$W, $q_1 = 1.287268$W, $q_2 = 1.287694$W, $q_3 = 1.288286$W, $q_4 = 1.288864$W, $\mu = 1.287683$W, $\sigma = 0.000770$) and CPython 3.13 the lowest power draw per experiment run ($q_0 = 1.283466$W, $q_1 = 1.283782$W, $q_2 = 1.284120$W, $q_3 = 1.284786$W, $q_4 = 1.285268$W, $\mu = 1.284268$W, $\sigma = 0.000667$). The other versions of CPython have power draws in between these respective maximum and minimum.



(a) Average power draw in Watt (W).



(b) Runtimes in seconds.

Fig. 4: Power draw and runtimes of the experiment runs per version of CPython.

Note, the y-axis of Fig. 4a. It shows that differences between power draw of the various versions of CPython are small, in the range of milliWatts (mW). The difference between the highest and lowest power draw of all experiment runs is ca. 4.13mW.

Besides power draw of the server, we record the runtimes per experiment run. Fig. 4b illustrates these in seconds (s) per version of CPython. This box plot shows that, in general, runtimes for CPython 3.12 are the lowest ($q_0 \approx 127.84$s, $q_1 \approx 128.39$s, $q_2 \approx 128.89$s, $q_3 \approx 129.75$s, $q_4 \approx 131.25$s, $\mu \approx 129.09$s, $\sigma \approx 1.06$) and runtimes for CPython 3.10 are the highest ($q_0 \approx 139.63$s, $q_1 \approx 140.11$s, $q_2 \approx 140.68$s, $q_3 \approx 141.61$s, $q_4 \approx 143.48$s, $\mu \approx 140.94$s, $\sigma \approx 1.20$). That is, the difference between the fastest and slowest experiment run over all versions of CPython is ca. 15.64s. All other runtimes are between these extremes.

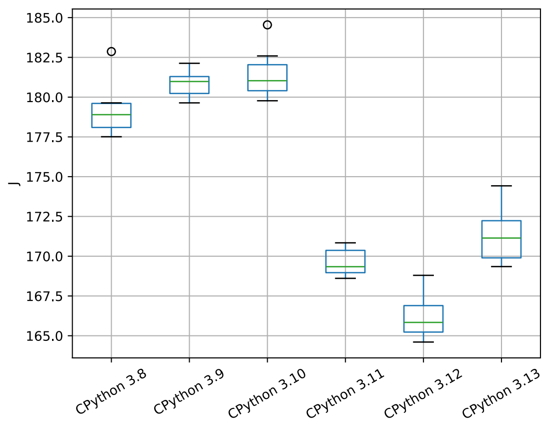


Fig. 5: Energy consumption in Joule (J) of all experiment runs per version of CPython.

Based on these measurements for power draw and runtimes, we compute energy consumptions of the experiment runs via $E_{nergy}[\text{J}] = P_{ower}[\text{W}] \times t_{ime}[\text{s}]$. Fig. 5 illustrates energy consumption of the experiment runs. It shows that CPython 3.10 consumes most energy ($q_0 \approx 179.77$J, $q_1 \approx 180.40$J, $q_2 \approx 181.03$J, $q_3 \approx 182.04$J, $q_4 \approx 184.54$J, $\mu \approx 181.35$J, $\sigma \approx 1.47$) and that CPython 3.12 consumes the least energy ($q_0 \approx 164.60$J, $q_1 \approx 165.23$J, $q_2 \approx 165.84$J, $q_3 \approx 166.90$J, $q_4 \approx 168.80$J, $\mu \approx 166.10$J, $\sigma \approx 1.30$). Energy consumption of CPython 3.8 and 3.9 are all between ca. 177.51J and ca. 184.54J, whereas newer versions of CPython (3.11 to 3.13) generally consume less energy (between ca. 164.60J and 174.42J respectively). For completeness, Tab. 1 lists the descriptive statistics for the energy consumption of all versions of CPython.

Visual inspection of all experiment runs' energy consumptions (Fig. 5) in combination with the descriptive statistics in Tab. 1 show that CPython 3.10 and 3.12 have the biggest difference in energy consumption. In general, CPython 3.10 consumes the most and 3.12 consumes the least.

Tab. 1: Descriptive statistics of energy consumption in Joule (J) rounded to two digits.

	CPython 3.8	CPython 3.9	CPython 3.10	CPython 3.11	CPython 3.12	CPython 3.13
min (q_0)	177.51J	179.64J	179.77J	168.61J	164.60J	169.35J
q_1	178.09J	180.23J	180.40J	168.96J	165.23J	169.90J
median (q_2)	178.90J	180.98J	181.03J	169.34J	165.84J	171.14J
q_3	179.60J	181.29J	182.04J	170.37J	166.90J	172.23J
max (q_4)	182.87J	182.13J	184.54J	170.84J	168.80J	174.42J
μ mean	179.12J	180.84J	181.35J	169.58J	166.10J	171.21J
σ	1.52	0.84	1.47	0.83	1.30	1.60

4 Analysis & Discussion

We conduct a Kruskal-Wallis test [11] to analyze if energy consumption differs significantly across versions of CPython. Unlike an ANOVA test, a Kruskal-Wallis test has fewer preconditions, such as, that our recorded energy consumptions come from a normally distributed population or that the standard deviations of the groups are all equal (homoscedasticity). Since we only conduct ten experiment runs per version of CPython, we have too few samples to test for these two preconditions. Visual inspection of the histograms of energy consumptions per version of CPython suggest they are not from a normal distribution. Therefore, we settle on a Kruskal-Wallis test, for which we formulate the following null and alternative hypotheses.

Null Hypothesis (H_0) There is no significant difference in energy consumption depending on the version of CPython. In other words, energy consumption of the server is more or less equal no matter which version of CPython is used to run the web-application.

Alternative Hypothesis (H_a) At least one of the group means differs significantly from at least one other group mean. That is, at least one version of CPython interpreter consumes considerably less or more energy than the others.

For our recorded energy consumptions, the Kruskal-Wallis H statistics is ≈ 53.07 (p-value $\approx 3.25 \times 10^{-10}$). Since the value of the H statistics is quite large and the p-value is below 0.05, we have to reject the null hypothesis and settle on the alternative hypothesis. That is, the energy consumption of at least one version of CPython is significantly different than the others. This result is supported by visual inspection of Fig. 5.

In general, one can observe a drop in energy consumption between CPython versions 3.8, 3.9, and 3.10 compared to CPython versions 3.11, 3.12, and 3.13, see Fig. 5. The biggest difference is between CPython 3.10 and 3.12. The median energy consumption of CPython 3.12 is $\approx 8.39\%$ lower than the one of CPython 3.10. The average energy consumption drops by $\approx 8.41\%$.

Remember, our initial assumption was that newer versions of Python are less energy-efficient. The results above (and as plotted in Fig. 5) suggest that this is generally not the case. To the contrary, newer versions of CPython are actually more energy efficient than older versions. If only considered for CPython 3.8 to 3.10, our initial assumption seems to be true as there is a continuous increase in energy consumption across these three versions of CPython. However, the more recent versions CPython 3.11, 3.12, and 3.13 consume all less energy than the previous versions in our experiment. Only the energy consumption of CPython 3.13 appears to increase again compared to the previous two versions. Notably, the maximum energy consumption of CPython 3.13 ($q_4 \approx 174.42$J) is still circa 3.09J lower than the minimal energy consumption of CPython 3.8 ($q_0 \approx 177.51$J).

CPython 3.11 was the first in a series of releases in which the language developers focused on improving performance of the Python interpreter. The release notes state: *"The Faster CPython Project is already yielding some exciting results. Python 3.11 is up to 10-60% faster than Python 3.10. On average, we measured a 1.22x speedup on the standard benchmark suite."*[13]

A performance improvement is visible in our experiment too, see Fig. 4b, though to a lower degree than stated by the language developers. On average, in our scenario, CPython 3.11 performs 6.56% faster than 3.10 and CPython 3.12 is 8.41% faster than 3.10.

Interestingly, the power draw of CPython 3.11 is the highest of all investigated versions of CPython, where CPython 3.12 is comparable to 3.10, and 3.13 sports the lowest. That is, it seems that the implemented performance improvements come at the cost of slightly higher power draw. However, the relative difference between the versions of CPython with highest and lowest power draw are marginal. The average power draw drops by only 0.26% between CPython 3.11 and 3.13 (q_2 drops ca. 0.28%).

Consequently, there is a sweet spot between performance optimizations and how they are implemented. In future, the Python language developers might face the situation, in which further performance improvements decrease energy efficiency in case power draw becomes more dominating. While some researchers describe that performance improvements suffice to decrease energy consumption, see e.g., Yuki et al. [29], others find that in certain cases power draw has an impact too, see e.g., Koedijk et al. [10]. Since the increases in power draw are comparably small, it seems however that a focus on performance optimizations is advisable for the Python project.

4.1 Threats to Validity

Conclusion Validity: Since we cannot conclusively test for normal distribution or homoscedasticity due to small sample size of energy consumptions, we can only conduct the less powerful Kruskal-Wallis test instead of the more powerful one-way ANOVA test. However, the results of Kruskal-Wallis test together with the plots that we provide in this paper, make us confident that the energy consumption of the various versions of CPython actually differ in a statistically significant way.

Internal Validity: To minimize the influence of other running processes, garbage collector runs, etc., we execute ten iterations of the same experiment per version of CPython. A higher number of experiment runs could increase the confidence in our results.

We chose to conduct our experiment on the FreeBSD operating system, since the amount of processes and daemons running in a default installation is quite small compared to modern Linux, MacOS, or Windows setups. Besides controlling the amount of running processes, one could opt to replicate our experiment on a unikernel operating system like Unikraft [12]. We are confident that our reported results are caused by the version of CPython and that similar patterns of energy consumption would occur in such a setup.

Construct Validity: At the time of setting up our experiment, FreeBSD does not have pre-compiled packages for CPython 3.12 and 3.13. We install them via `pyenv`. Thereby, CPython gets build from sources on the experiment server before installation. We install these two versions of CPython with `pyenv`'s defaults. We did not inspect how precisely the pre-compiled FreeBSD packages are built. There might be a risk that different optimization switches are used to build FreeBSD CPython packages compared to `pyenv`. In case `pyenv` optimizes more aggressively for performance, then our results might overemphasize the more favorable energy efficiency especially of CPython 3.12.

External Validity: We conduct our experiment only on one platform, the Raspberry Pi 1 Model B+ V1.2, which sports a single core Broadcom BCM2835 ARM 6 CPU. Currently, we do not know to which degree our results can be generalized to other or multi-core (ARM) processors.

5 Related Work

With the increased availability of mobile applications, researching the energy impact of various aspects of software became more relevant to larger parts of the software engineering community. On mobile devices, power hungry software is directly perceivable by end-users via quick battery drain. Consequently, researchers focused attention on energy impact of mobile applications on Android or iOS, see e.g., [3,4,8,14,16].

Olivera et al. [16] compare the energy consumption of applications using the three mobile application development frameworks Flutter, React Native, and Ionic to native Java applications. They find that generally, Flutter imposes the least energy overhead. We believe that our work, can provide the backend counter part to Olivera et al.'s work. Even though possible,[26] CPython applications are rarely deployed to mobile platforms but more often as backend software to data centers in "the cloud". In such environments, software engineers *"rarely have requirements or goals about energy usage"* [15]. That might explain why high-level interpreted languages like Python have been of lower priority for the green software engineering community, which we attempt to change with this work.

[26] `https://beeware.org/`

In general, software engineers care about the energy impact of their code [15]. However, they lack knowledge and support on how to improve energy efficiency of their applications [15, 18, 22]. The results of our experiment yield an easy to apply high-level solution to increase energy efficiency of CPython applications by just executing them on the "right" version of the interpreter. This differentiates our work from previous work, see e.g., [3, 4, 13, 14, 24] which usually focuses on application-level modifications to improve energy efficiency. Our experiment is more coarse-grained. We consider the application a black-box and investigate the impact of the execution environment, the CPython interpreter. To the best of our knowledge, we report the first results of an experiment of such kind.

A common experiment design to investigate energy efficiency of software, is that a set of benchmarks (often the Computer Language Benchmark Game[27] or Rosetta Code,[28]) which implement certain algorithms in various programming languages, are executed. Then energy consumption is measured indirectly via power profiling tools like Intel's Power Gadget,[29] Running Average Power Limit [7], Performance Counter Monitor,[30] PowerJoular,[31] or similar estimation tools.[32] Subsequently, programming languages are compared to each other with regards to energy consumption, see, e.g., [2, 6, 10, 17, 19, 20, 26]. In all of these experiments, only a single language version, i.e., interpreter or compiler is considered. Our experiment design differs in that we investigate only one case application, the MiniTwit web-application, where the experiment variable is the version of CPython, the language interpreter.

More rare are experiments that meter power draw or energy consumption directly via a physical power meter, see, e.g., [5, 6, 9]. Our experiment design is similar to these works. However, we conduct our direct measurements via the Otii Arc Pro, which we are not aware of as reported in similar experiments.

Kirkeby et al. [9] suggest to investigate energy consumption of the live software system, Edora A/S's Work Force Planner. Admittedly, our software is not a production grade system and our three clients are executing predefined scenarios. However, our case application (MiniTwit) is a complete web-application that resembles a more production ready application than the more "artificial" benchmarks used in previous research [2, 6, 10, 17, 19, 20, 26].

Kwon et al. [14] investigate the effect of various distributed programming abstractions like sockets, remote-procedure calls, messages, etc. They experiment with a client-server setup with two separate computers. However, they only have a single client and they measure energy consumption indirectly on the server via the process-level power profiling tool pTopW for a single version of Java. Their experiment variable are the respective distributed programming abstractions.

[27] https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html
[28] https://rosettacode.org/wiki/Rosetta_Code
[29] https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html
[30] https://github.com/intel/pcm
[31] https://github.com/joular/powerjoular
[32] https://luiscruz.github.io/2021/07/20/measuring-energy.html

We are not aware of any experiment that reports assessment of energy efficiency of multiple versions of the same programming language. Wagner et al.'s experiment [26] is most similar to ours in that they compare two web-assembly runtimes (Wasmer and Wasmtime) against each other. These runtimes could be compared to the versions of CPython in our experiment. Unlike in our results, the authors do not find a statistically significant difference in energy consumption when running the same programs on different web-assembly runtimes.

Currently, research results on the impact of runtime to energy efficiency are mixed. For example, Yuki et al. [29] confirm that programs that are optimized for performance are also more energy efficient. On the other hand, e.g., Koedijk et al. [10] identify certain cases where slower programs are more energy efficient compared to faster ones. The results of our experiment show that runtime has the most impact on energy consumption. However, our results also illustrate that runtime optimizations come at the cost of power draw. Future work has to identify the sweet-spot between these two.

6 Future Work

In this paper, we report the first of a planned series of experiments. We will extend this experiment by executing the Python Performance Benchmark Suite.[33] We want to investigate more thoroughly the relation between performance improvements and power draw. Additionally, we plan to conduct these measures not only on one model of the Raspberry Pi but on various newer models and other architectures too. The purpose is to increase confidence in that our results are generalizable over different versions of the ARM processor architecture and over other architectures like Intel/AMD x86.

In the farther future, we plan to substantially extend this work by implementing the same MiniTwit web-application in various programming languages and with different web-frameworks. Our goal is to create more nuanced and more directly applicable results that support practitioners in choosing the "right tool for the job" when implementing sustainable web-applications.

7 Conclusions

In this paper, we investigate the research question: *What is the energy consumption of different versions of the Python interpreter CPython?* Initially, we assume that more recent versions of Python are less energy efficient than older ones. To investigate our research question, we conduct a controlled lab experiment in which we determine the energy consumption of a webserver sporting various versions of CPython. Three clients execute parallel scenarios against a web-application running on a server. Contrary to our initial assumption, our results show that recent versions of CPython are actually more energy efficient than older versions. Currently, the most energy efficient, i.e., sustainable, version is

[33] `https://pyperformance.readthedocs.io/`

CPython 3.12. Energy consumption of a web-application can drop by more than 8%, only by exchanging older versions of CPython with CPython 3.12.

Impact for practitioners: For developers and maintainers, we recommend to switch to CPython 3.12 since our results indicate that it is currently the most energy efficient CPython interpreter. In our experiment, energy consumption drops by more than 8% compared to the most energy consuming CPython 3.10.

For the developers of CPython, our results indicate that the performance improvements that were implemented in recent versions of CPython are not only beneficial from a performance point of view, they also render CPython more sustainable compared to previous versions. We suggest to execute the Python Performance Benchmark Suite[33] in a setup similar to ours to continuously monitor also energy efficiency of future CPython versions.

Impact for researchers: Our work raises multiple questions for researchers to address. Unanswered by our current work, but relevant for future language improvements are questions like a) *Which implementation details cause a Python interpreter to draw more power or to become faster and thereby impact energy consumption?* or b) *What is the precise reason for that power draw increases for certain versions of CPython that are optimized for performance?* We hope that our work inspires other researchers to investigate such questions and thereby help to increase energy efficiency of the reference implementation of the currently most popular programming language.

References

1. Caballar, R.D.: We need to decarbonize software: The way we write software has unappreciated environmental impacts. IEEE Spectrum **61**(4), 26–31 (2024)
2. Couto, M., Pereira, R., Ribeiro, F., Rua, R., Saraiva, J.: Towards a green ranking for programming languages. In: Proceedings of the 21st Brazilian Symposium on Programming Languages. pp. 1–8 (2017)
3. Cruz, L., Abreu, R.: Performance-based guidelines for energy efficient mobile applications. In: 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). pp. 46–57. IEEE (2017)
4. Cruz, L., Abreu, R.: Catalog of energy patterns for mobile applications. Empirical Software Engineering **24**, 2209–2235 (2019)
5. Eder, K., Gallagher, J.P., López-García, P., Muller, H., Banković, Z., Georgiou, K., Haemmerlé, R., Hermenegildo, M.V., Kafle, B., Kerrison, S., et al.: Entra: Whole-systems energy transparency. Microprocessors and Microsystems **47**, 278–286 (2016)
6. Georgiou, S., Kechagia, M., Spinellis, D.: Analyzing programming languages' energy consumption: An empirical study. In: Proceedings of the 21st Pan-Hellenic Conference on Informatics. pp. 1–6 (2017)
7. Khan, K.N., Hirki, M., Niemi, T., Nurminen, J.K., Ou, Z.: Rapl in action: Experiences in using rapl for power measurements. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS) **3**(2), 1–26 (2018)
8. Kholmatova, Z.: Impact of programming languages on energy consumption for mobile devices. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1693–1695 (2020)

9. Kirkeby, M.H., Gallagher, J.P., Thomsen, B.: An approach to estimating energy consumption of web-based it systems. In: CERCIRAS Workshop-01: Abstracts. p. 18 (2021)
10. Koedijk, L., Oprescu, A.: Finding significant differences in the energy consumption when comparing programming languages and programs. In: 2022 International Conference on ICT for Sustainability (ICT4S). pp. 1–12. IEEE (2022)
11. Kruskal, W.H., Wallis, W.A.: Use of ranks in one-criterion variance analysis. Journal of the American statistical Association **47**(260), 583–621 (1952)
12. Kuenzer, S., Bădoiu, V.A., Lefeuvre, H., Santhanam, S., Jung, A., Gain, G., Soldani, C., Lupu, C., Teodorescu, Ş., Răducanu, C., et al.: Unikraft: fast, specialized unikernels the easy way. In: Proceedings of the Sixteenth European Conference on Computer Systems. pp. 376–394 (2021)
13. Kumar, M., Li, Y., Shi, W.: Energy consumption in java: An early experience. In: 2017 Eighth International Green and Sustainable Computing Conference (IGSC). pp. 1–8. IEEE (2017)
14. Kwon, Y.W., Tilevich, E.: The impact of distributed programming abstractions on application energy consumption. Information and Software Technology **55**(9), 1602–1613 (2013)
15. Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., Clause, J.: An empirical study of practitioners' perspectives on green software engineering. In: Proceedings of the 38th international conference on software engineering. pp. 237–248 (2016)
16. Oliveira, W., Moraes, B., Castor, F., Fernandes, J.P.: Analyzing the resource usage overhead of mobile app development frameworks. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering. pp. 152–161 (2023)
17. Oliveira, W., Oliveira, R., Castor, F.: A study on the energy consumption of android app development approaches. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). pp. 42–52. IEEE (2017)
18. Pang, C., Hindle, A., Adams, B., Hassan, A.E.: What do programmers know about software energy consumption? IEEE Software **33**(3), 83–89 (2015)
19. Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: 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. pp. 256–267 (2017)
20. Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J.: Ranking programming languages by energy efficiency. Science of Computer Programming **205**, 102609 (2021)
21. Perez, F., Granger, B.E., Hunter, J.D.: Python: an ecosystem for scientific computing. Computing in Science & Engineering **13**(2), 13–21 (2010)
22. Pinto, G., Castor, F., Liu, Y.D.: Mining questions about software energy consumption. In: Proceedings of the 11th working conference on mining software repositories. pp. 22–31 (2014)
23. Reiser, M.: The Oberon system: user guide and programmer's manual, chap. Preface, p. v. ACM (1991)
24. Reya, N.F., Ahmed, A., Islam, T.: Greenpy: evaluating application-level energy efficiency in python for green computing. Ann. Emerg. Technol. Comput.(AETiC) **7**(3), 93–110 (2023)
25. Van Rossum, G., et al.: Computer programming for everybody. CNRI: Corporation for National Research Initiatives **1999** (1999)

26. Wagner, L., Mayer, M., Marino, A., Soldani Nezhad, A., Zwaan, H., Malavolta, I.: On the energy consumption and performance of webassembly binaries across programming languages and runtimes in iot. In: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering. pp. 72–82 (2023)
27. Winters, T., Manshreck, T., Wright, H.: Software engineering at google: Lessons learned from programming over time, chap. 24, p. 507. O'Reilly Media (2020)
28. Wirth, N.: A plea for lean software. Computer **28**(2), 64–68 (1995)
29. Yuki, T., Rajopadhye, S.: Folklore confirmed: Compiling for speed compiling for energy. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 169–184. Springer (2013)