IT UNIVERSITY OF COPENHAGEN

# Energy consumption of different programming languages

Authors:
**Andreas Severin Hauch Trøstrup** – **atro@itu.dk**
**Frederik Petersen** – **frepe@itu.dk**
**Lucas Frey Torres Hanson** – **luha@itu.dk**

**Course code:**
KSCOSYP1KU

IT University of Copenhagen
Copenhagen, Denmark

Group 42
May 23th 2025

# Contents

# Chapter 1

# Introduction

## Background and Motivation

"Ranking Programming languages by Energy Efficiency" Pereira et al. (2021) is a massively cited and virally shared paper highlighting the energy efficiency of different programming languages, when performing the same tasks. Despite the great amount of attention the paper has received, some of the results seem questionable, and have been taken out of context in many a LinkedIn post. This is partly due to lacking explanations in the paper as well as questionable results – consider as an example the Fannkuch-redux results where TypeScript (a preprocessor to JavaScript) is 15 times slower than JavaScript, despite it having very similar results in every other case.
Our goal of this project is to try to recreate the results of the initial paper, as well as see how newer languages fit the observations made by the original paper. Specifically we'd like to benchmark Zig, a self-proclaimed high-performance compiled language, to see if it gets similar low energy results to other compiled languages like C, as projected by the initial paper.

# Chapter 2

# Methodology

For this project we used a Raspberry Pi 5 (henceforth Pi). This allowed for both virtual power monitoring using the `vcgencmd` command, as well as physical power monitoring if so desired. The Pi comes equipped with a Broadcom BCM2712 processor.

Table 2.1: Hardware specification for the Broadcom BCM2712 processor - specifications partly taken from Appendix A.1 and Raspberry Pi Foundation (2023)

| Processor | Broadcom BCM2712 |
|---|---|
| #Cores | 4 |
| #Threads per core | 1 |
| Clock speed | 2.4GHz |
| Main memory | 8 GB |
| L1i | 64 KB |
| L1d | 64 KB |
| L2 | 512 KB |
| L3 | 2 MB |
| Operating system | Debian 12 |

We settled for collecting power consumption using the `vcgencmd` command. This command reports the voltage and current as measured by sensors on the Pi. The output of this command is, however, not very human-readable, so we used the metrics collection and monitoring tool Prometheus to manage the data. We used a simple Python webserver which on receiving a GET request responded with the output of `vcgencmd` formatted as Prometheus metrics, which allowed us to query the voltage and current data using the Prometheus query language (PromQL). From this, we derived the power as the product of the voltage and current. By multiplying the reported power usage with the time used for the implementation process to finish, we got the total amount of energy consumed by the program. We measured consumption in Joules, the same measurement used in Pereira et al. (2021). This setup did mean that there were processes running on the Pi used for the measuring and logging, which of course had the

| Language | Compiler / Runtime | Build / Run Flags[1] |
|---|---|---|
| C (**c**) | gcc (Debian 12.2.0-14) 12.2.0 4 | `-pipe -Wall -O3 -fomit-frame-pointer -march=native` |
| Java (**c**) | javac 24.0.1 (graalvm-24) / native-image 24.0.1 | `--silent --gc=G1 -cp . -O3 -march=native` |
| JavaScript (**i**) | deno 2.3.3 (v8 13.7.152.6-rusty) | None |
| TypeScript (**i**) | deno 2.3.3 (typescript 5.8.3) | None |
| Ruby (**i**) | ruby 3.1.2p20 | `--yjit -W0` |
| Zig (**c**) | zig-linux-aarch64-0.14 | `-O ReleaseFast` |

Table 2.2: Compiler- and Runtime versions used in the benchmarks. (**c** = compiled language, **i** = interpreted language)

side effect of adding noise to the reported energy usage of the tested implementations. We mitigated this by subtracting a constant amount of Watts from the measured power usage, which we derived by letting the Pi run for a while to log an average idle power. Instead of developing the programs used for benchmarking ourselves, we resorted to a mixture of 1st and 3rd party developed benchmarking programs for the same tasks. Most of the implementations are taken from or based off of implementations found at the Computer Language Benchmark Game website (Gouy (nd)). Here we also found instructions for how to compile and run the various algorithms. The C implementations often used x86 SIMD instructions. For the time constraints it was prioritized to not rewrite the code from x86 SIMD to ARM SIMD – thus affecting the end result of the languages energy usages. This represents a side effect of using implementations we haven't made ourselves; we can't ensure that the implementation chosen is *the* most efficient one. All code we run will be referenced directly in this paper to ensure direct reproducibility.

We structured the code in subdirectories of languages and implementations, with the implementation folder containing the source code and a Makefile specifying the build step, if needed, and the run step. The compilers and runtime versions for the different languages can be found in Table 2.2.

It is worth noting that while Java is typically compiled to Java bytecode and then executed using the Java Virtual Machine (JVM), making it more alike an interpreted language, we used the Oracle GraalVM JDK to compile the Java code to a *native image*, a binary file that can be executed natively. This is in accordance with the compile instructions found on the Computer Language Benchmark Game webpage.

All the source code can be found on our created GitHub repository Trøstrup et al. (2025).

[1]The build flags of compiled languages may also include specific libraries needed for each implementation. The full flags can be found in our GitHub repository.

# Chapter 3

# Results

Energy consumption and time duration were both measured for the build step, as well as the run step, for each programming language.

## 3.1 Build

The build energy was not measured for Ruby, JavaScript, and TypeScript, as they are not compiled languages.

As seen in Figure 3.1 Java uses a lot more energy compared to the languages C and Zig. On average it uses about 1000 times more energy to compile as compared to C, as shown in Table 3.1.

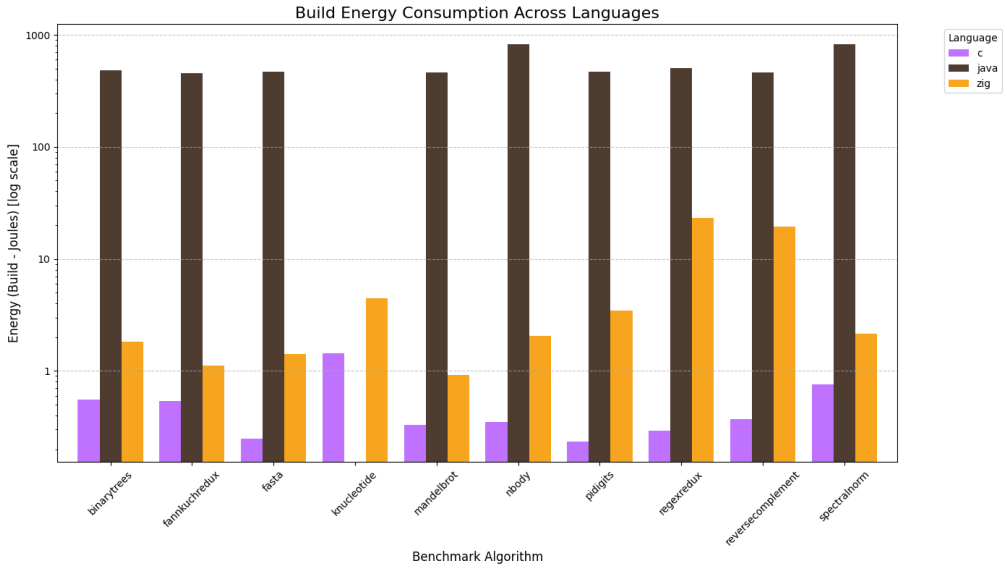



Figure 3.1: The energy used to create the native executables (Y-axis uses log scale)

It is important to remember that the Java code is being compiled to a native executable – the results would have likely looked different if compiled to a JAR-file to be run by the JVM. The compilation to a native image is slightly different from the compilation of C or Zig since they compile directly to bytecode, and are designed to do so. In contrast, the Java native image technology is a third-party tool that first has to analyze the Java source code to be able to compile it to a native bytecode.

Table 3.1: Average build time and energy (normalized)

| | Energy (C eqv.) | Duration (C eqv.) |
|---|---|---|
| C | 1.00 | 1.00 |
| Zig | 11.72 | 12.55 |
| Java | 1076.12 | 647.19 |

## 3.2 Run

In Figure 3.2 a seemingly lower energy consumption for native executed programs can generally be observed. In *binarytrees* this does not hold as Zig uses more than TypeScript and JavaScript.

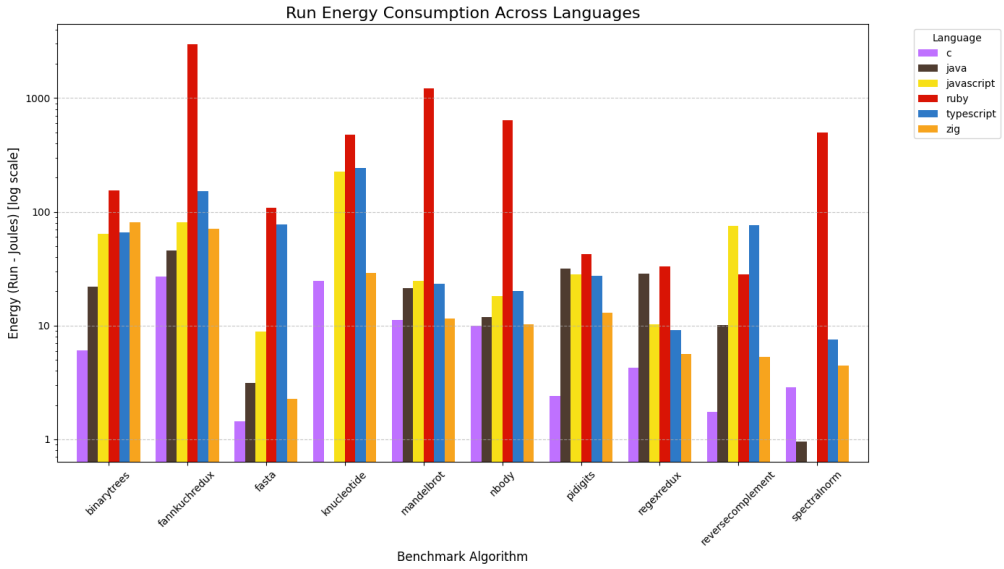



Figure 3.2: The amount of energy it takes to run the algorithms for each language (a few languages does not have a bar for algorithms as they they either could not run or implementations did not exist)

In Table 3.2b the overall energy can be seen to be less for the natively executed languages, with Ruby having the absolute largest energy usage. This can be explained by the sheer duration of the programs, but it is also worth noting that the average power is much higher than for the other languages, meaning that it is not only duration, but also the work done that increases the energy usage (See Appendix A.2). This seems to indicate

that the Ruby interpreter in itself is less efficient at executing the code. It could also be explained by parallelisation in the implementations, thus using more cores, and by the fact that we use just-in-time compilation in the form of YJIT.

Generally, there also seems to be high correlation between duration and power as displayed in the two tables in Table 3.2b. This could be attributed to running the algorithms on the Pi, as the implementations will likely be limited by the hardware. If run on better hardware, the correlation might be less obvious as more optimizations can be made for each language, especially in regards to multithreading as we only had four cores available – though this would be a whole separate research paper.

Table 3.2: Energy and time usage on runtime

(a) Average runtime (C baseline)

| | Energy (C eqv.) | Duration (C eqv.) |
|---|---|---|
| C | 1.00 | 1.00 |
| Java | 2.13 | 2.50 |
| Zig | 2.56 | 4.23 |
| JavaScript | 6.54 | 10.11 |
| TypeScript | 7.68 | 12.29 |
| Ruby | 67.54 | 48.70 |

(b) Average runtime

| | Energy (J) | Duration (s) | Power (W) |
|---|---|---|---|
| C | 9.14 | 3.36 | 2.72 |
| Java | 19.46 | 7.91 | 2.46 |
| Zig | 23.41 | 14.25 | 1.64 |
| JavaScript | 59.78 | 34.00 | 1.76 |
| TypeScript | 70.19 | 41.33 | 1.70 |
| Ruby | 617.21 | 163.82 | 3.77 |

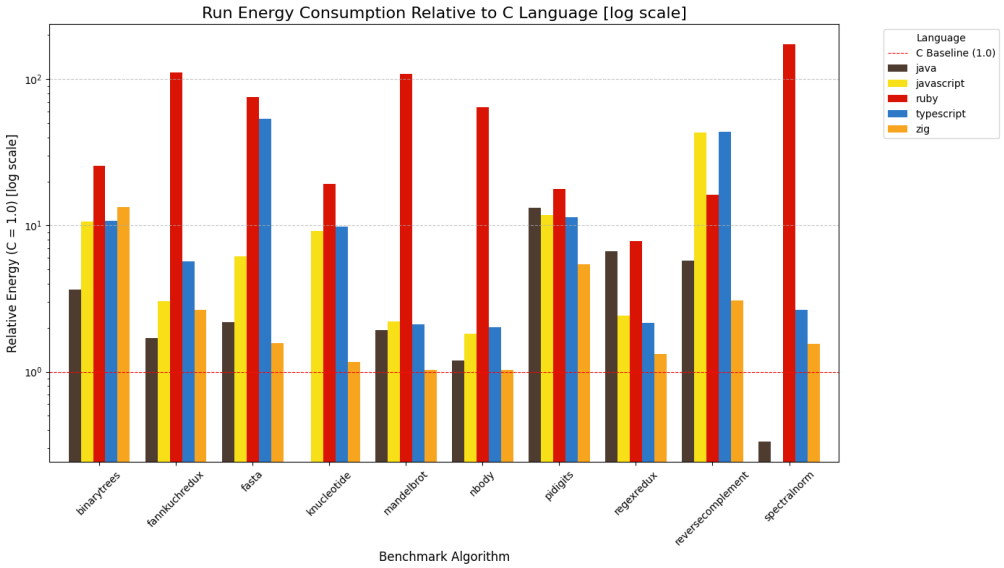



Figure 3.3: The relative amount of energy it takes to run and build each algorithm for each language (a few languages does not have a bar for algorithms as they they either could not run or implementations did not exist)

If all the results are shown in relation to C it is clear that some native executables come close to the same energy consumption. Java even outperforms C in *reversecomplement*. This might be due to a lucky outcome in testing, and not that it would outperform each time. However as clearly displayed in Table 3.2a, C uses the least amount of energy of the languages benchmarked.

## 3.3 Total

When looking at the total energy, in Figure 3.4, Java has the highest energy consumption for most algorithms.

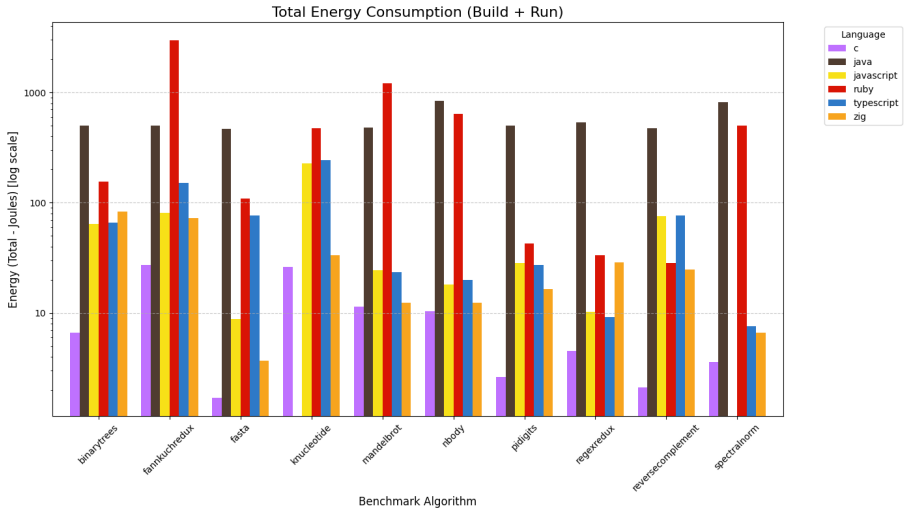



Figure 3.4: The total amount of energy it takes to run and build each algorithm for each language

This is mainly due to its compilation step. JavaScript and TypeScript are often comparable in energy consumption, with Zig often being lower but occasionally having a slower implementation. It is important to remember that in production most code is compiled once, and run thousands of times, and Figure 3.4 essentially adds the price of compiling and running an algorithm once for a specific language. So in most cases it would make more sense to ignore the consumption of compilations mostly, to figure out which language to use from an energy consumption perspective.

# Chapter 4

# Discussion

As it can be seen from the results, we've achieved some of the same general trends as Pereira et al. (2021). Compiled languages are generally faster, and process run times correlate very strongly with energy usage.
It is however also clear that our results for specific languages vary greatly when compared to the original paper. This can both be attributed to using different compiler versions, compilation flags, different runtimes for interpreted languages, and especially, different implementations of algorithms.

This highlights one of the main issues with this kind of study – that, to compare languages in this way is a complex matter, setup and implementations vary greatly on a language by language basis. Not only in terms of the speed of executing the languages, but also simply because the implementations that are *possible* vary greatly. Languages like C can take advantage of low-level memory manipulation and highly specialized and optimized data structures, not to mention intrinsic functions allowing the compiler to create extensively optimized code for techniques like SIMD. In higher-level languages these tools are simply not possible, so even with the same algorithmic implementation, lower-level languages will often be much more efficient. Even the implementations used in this paper, and the original paper, do not utilize the full functionalities of the languages used. Problems with processor architecture made it difficult to use intrinsics, and, for example, few of the TypeScript implementations actually use the TypeScript-specific typing system, mostly rendering the type pre-processing redundant. A fully typed implementation would likely show different results that what we are currently seeing. However, whether or not this would reduce or increase the energy usage could be interesting for further work.

We chose to also include Zig in this study, a new language compared to the languages used in Pereira et al. (2021), but the results we see are interesting. We would expect that a compiled, low-level language would generally see lower energy usage and faster runtimes than higher-level languages, but as we can see in Figure 3.2, this is not always the case. We can see that in the *binarytrees* benchmark, it is actually beaten by not

only Java, but also JavaScript and TypeScript. In the *fannkuchredux* benchmark, Zig is also beaten by Java. Though this is not the case in all benchmarks, it does show that it is not always the case that low-level languages are more efficient. It is of course worth noting that Zig is a relatively new language, and this does mean that there are simply fewer efficient implementations of algorithms readily available, and it was not in the scope of this project to improve upon them, if possible.

This also does highlight another discrepancy in our results; looking at the average running time in Table 3.2, we see that Java is actually the second most efficient and fast language. While Java is generally fast in our results, we are missing an implementation in the *knucleotide* problem, which does mean that the Java average is taken over a smaller amount of samples, possibly improving it falsely. There is also a questionable Java result in the *spectralnorm* algorithm seen in 3.2, as it seems unlikely that Java would be as much faster than C as is reported.

In general, a standardization for all variables in testing an algorithm in different languages would be needed to create a valuable comparison between them. Without heavy standardization, the value provided by the results is minimized.

With as many variables as is the case with this type of work, it is difficult to draw any safe conclusions. Therefore it seems dangerous to present the data so matter-of-factly as is done in the original paper – without emphasising the shortcoming. This data can, and has been, taken out of context and shared virally, leading to misinformation. In particular, results from the study were used to critique an industry move to TypeScript, as it was shown to use almost 5 times more energy than JavaScript[1]. No place in the paper is this result mentioned, and as seen from our results, was a questionable conclusion.

[1]Like on popular message boards such as Reddit and LinkedIn, as seen on `https://www.reddit.com/r/ProgrammerHumor/comments/x0lnj4/greenest_programming_languages_a_reason_to/` and `https://www.linkedin.com/posts/wvdaalst_sustainability-activity-7223303687266336768-0X0r/?rcm=ACoAADVUt8MBL6-FGSDX41X11AOV5Udk4eeAuAs`

# Chapter 5

# Conclusion

For this project we wanted to recreate and validate the results of the paper from Pereira et al. (2021), as well as see how the general trends they found, apply to languages not in the original study. We aimed to create a more transparent study, as the original paper lacked information about methodology, for example about the compilers and runtimes, as well as the implementations used. Additionally, we aimed to reflect more on our results, as the original paper lacked much discussion about possibly questionable outcomes.

We used a Raspberry Pi 5 to create a controlled environment for running our benchmarks and used the built-in `vcgencmd` command to report voltage and current. Along with a customized, but reproducible Prometheus setup, we used this to derive energy usage while running our benchmarks.

Using the most efficient implementations able to run on the Pi for a small set of problems we benchmarked the performance of six different languages. Using the same normalized standard used in the paper, we compared them and discussed the results.

While the overall trends of our findings were equivalent to those of the original paper, the main findings of this paper is that it is very complex to compare languages. There are extremely many variables in how a language runs. These include; compiler versions, compilation flags, runtimes for interpreted languages, and their direct implementations.

All results from this paper should be taken with a grain of salt (a large one). Of course, languages compiled to native executables generally use less power during runtime, but each language has its use case and they cannot be compared one-to-one without a lot of standardizations for testing.

# Bibliography

Gouy, I. (n.d.). clbg-webpage. https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html. Accessed: 2025-04-29.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., and Saraiva, J. (2021). Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609.

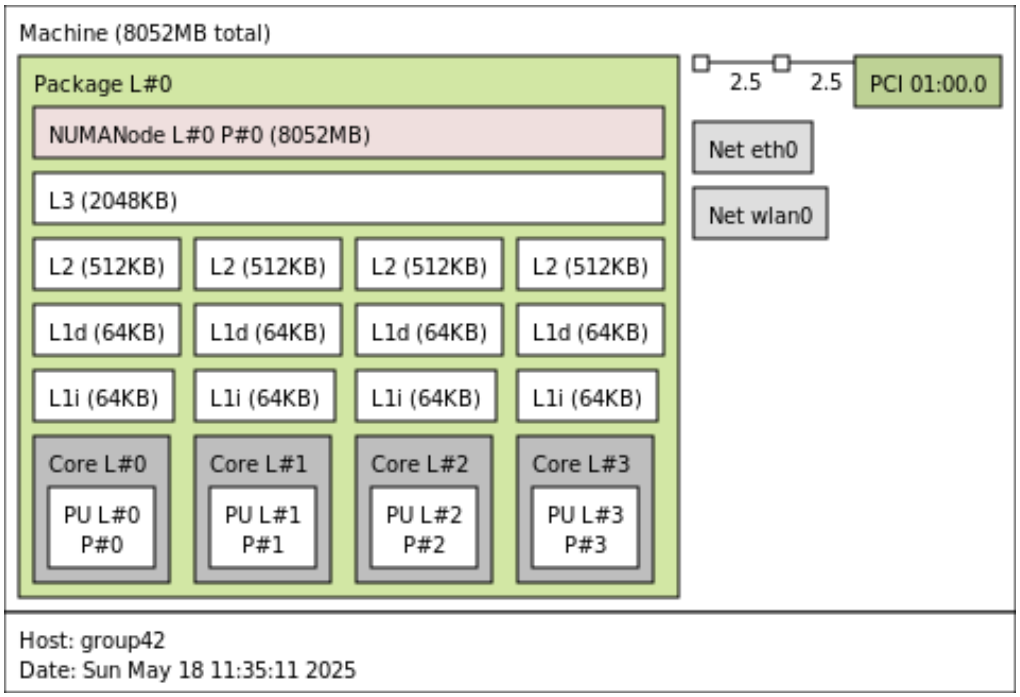
Raspberry Pi Foundation (2023). Raspberry pi 5. https://www.raspberrypi.com/products/raspberry-pi-5/. Accessed: 2025-05-18.

Trøstrup, A., Petersen, F., and Hanson, L. (2025). language-energy-efficiency. https://github.com/lucasfth/language-energy-efficiency. Accessed: 2025-05-22.

# Appendix A

# Appendix

## A.1 Raspberry Pi 5 lstopo




The figure shows the hardware specifications shown by running *lstopo*

## A.2 Power usage before and after ruby


The figure shows the power consumption during benchmarking, after ruby begins, as well as before (when the other languages were running)