Bachelor's Thesis Computing Science


Radboud University Nijmegen

---

**Measuring the energy overhead of computer language features**

---

*Author:*
Eline Stehouwer
s1084507

*First supervisor/assessor:*
dr. Bernard van Gastel

*Second supervisor:*
MSc Jordy Aaldering

*Second assessor:*
dr. Sjaak Smetsers

June 22, 2024

**Abstract**

This thesis explores the energy overhead of computer language features by measuring their energy consumption. The goal is to gain a better understanding of the energy dynamics in a programming language. The features we have investigated are garbage collection, bounds checking, and dynamic typing. To investigate this, we implemented these features in a set of numeric code problems written in C and measured what costs they add in terms of energy consumption and execution time. From our measurements, we conclude that bounds checking has a minimal effect on these two factors in numeric code problems. Garbage collection increases them, making them up to twice as inefficient in such problems. Dynamic typing has the most substantial impact among the three tested features, potentially increasing energy consumption and execution time by up to 30 times. These findings explain the performance gap between a number, but not all, programming languages. Therefore, it is likely that other language features also play a significant role in these languages. These other features and their impact should be analysed further in future research.

# Contents

# Chapter 1

# Introduction

In the world of software development, the choice of programming language is a crucial decision that companies, researchers, and developers have to make when starting a new project. Beyond considerations like familiarity and functionality, the efficiency of a language in terms of energy consumption is increasingly important [15]. Especially, in a time when sustainability is a growing concern in all parts of society. As a consequence, there has been a growing body of research into the energy efficiency of various programming languages [14].
An example of this is the study by Periera et al. [14]. It compares 27 programming languages based on energy consumption, runtime, and memory usage. By comparing these factors, the study offers an insight into their relative efficiencies. Notably, the study categorizes languages by their execution type: interpreted, virtual machine-based, or compiled. While this categorization highlights important distinctions, it also raises the question of whether other features impact a language's performance.

Building on the foundation laid by Periera et al., this thesis aims to delve deeper into the nuances of programming language efficiency by examining additional features beyond execution type. Specifically, we investigate how the factors memory management, typing system, and memory safety impact energy consumption and runtime. For the category memory management, we focus on the difference between garbage collection and explicit memory management. For typing, we look at the difference between a static and dynamic typing system. For memory safety, we compare automatic bounds checking and no bounds checking.

To be able to analyze these different features, we implement or add them to certain code examples. Then we compare these adapted programs to their original version on energy consumption and execution time. The goal is to see where the added energy costs, found in the research of Periera et al.,

come from and to get a more comprehensive understanding of the energy dynamics in a programming language.

We look at code in the language C. We choose this language because, among the languages tested by Periera et al., only C and Rust allow for the individual inclusion or exclusion of each feature. Many other languages either already have one or more of them implemented, or do not allow users to implement them, making it impossible to turn them off and on. Since this study is inspired by the results of Periera et al., we use the code repository they used to verify their studies, to implement and test the different features on, namely the Rosetta Code repository (Section 3.1 has more information on this repository). The code problems we take from there are all numerical problems.

We start the research by looking into good practices for benchmarking time and memory (Chapter 2). After that we specifically look into measuring energy (Chapter 3). Next, we implement the bound checks, garbage collection, and dynamic typing in the chosen programs (Chapter 5). Then we run the code with and without these expansions and measure their energy consumption and execution time (Chapter 4). We visualize the results using tables and graphs (Chapter 6), followed by an analysis (Chapter 7). From our results and analysis, we draw conclusions on what part of the language has the biggest impact on the energy consumption of numerical programs (Chapter 8).

# Chapter 2

# Benchmarking execution time

Benchmarking is an important part of this thesis. Benchmarking the execution time of programming languages has been extensively researched, part of which we discuss in this chapter.
The research in this area can be divided into two main categories. The first category consists of the optimal procedures for configuring the benchmarking system and selecting the appropriate benchmark method. The second encompasses studies that used these methods to benchmark one or more programming languages on their time efficiency.

## 2.1 Designing good benchmarks

The first important aspect to consider when designing a benchmark is its relevance [17]. Meaning, relevance of what is being benchmarked. It seems like an obvious requirement, yet is often forgotten, resulting in wasted research time. We deem the benchmarking of energy consumption of the features bounds checking, garbage collection, and dynamic typing relevant for the following reasons. Understanding these features can provide more or less reason to include them in a language. It also sheds light on the energy consumption of different programming languages.

Another aspect of increasing importance when designing good benchmarks is the replicability of a study [1]. According to Beyer, Löwe, and Wendler a conducted study is replicable, when another research team can later obtain the same results under the same conditions by running the benchmarks again. This means using the same hardware and software versions. For this to be the case, it is important that the experiment satisfies several properties, both on the reporting and technical side.

### 2.1.1 Transparent reporting for replicable benchmarking

When reporting the research choosing the right metrics, documenting the experimental setup, making the data available, and making sure the experiments are repeatable are essential. Benchmarks need to carefully follow a well-defined set of goals and rules [10].

What metrics the benchmark measures is one crucial point. Dufour et al. [4] established five guidelines for the dynamic metrics that a benchmark can measure. The metrics, when designed according to these guidelines, should concisely and informatively summarize a benchmark [10].

1. The metrics need to be unambiguous. This means that what is measured is clearly defined.

2. The metrics need to be dynamic. This means that they need to relate to runtime aspects. This requirement is there to make sure that the metric characterizes the benchmark behavior.

3. The metrics need to be robust. This means that they should not be too dependent on program input.

4. The metrics need to be discriminating. This means that when the behavior of the program changes this is reflected in the metrics.

5. The metrics need to be machine independent.

When applying these guidelines to our metrics of energy consumption and execution time, we see that the metrics are unambiguous, dynamic, and discriminating in nature. They are not (always) robust and machine independent. For some programs, when the input changes drastically, the time it takes and the energy it consumes will change, this cannot be avoided. They are also not machine independent, every machine consumes resources differently. Even with the same hardware there can be a major difference in energy consumption of that hardware [9]. Still these measurements do give interesting insights, even though they may not hold for every machine.

In an experiment, documenting the experimental setup entails sharing all important aspects of how it was carried out. These aspects are: the hardware used (e.g. CPU model, memory size), software dependencies (e.g., compiler versions, runtime environments), configurations (e.g., optimization settings, BIOS settings, parallelization techniques), source code, scripts for gathering data, and other steps that were followed to obtain the results [17, 1]. Comprehensively documenting this allows others to understand exactly how the experiments were conducted. When this is known, researchers can try to replicate this study under the same conditions and verify the results. Without clear documentation of these factors, research becomes harder to

validate and experiments harder to accurately reproduce. In Section 4.2, we share all the hardware and software dependencies, and we provide a link to the scripts and code we used.

The next reporting aspect that is key for replicability, is making the data used in the benchmarking study available [1]. This includes: the input data and the final performance metrics. Access to this data allows other researchers to verify the computations, perform additional analyses, or compare the findings with alternative approaches. We have applied this to this thesis. The source of the input data is shared in Chapter 4 and the final performance metrics are shared in Chapter 6.

Furthermore, the experiments should be repeatable. It should not be the case that the results of a published paper are obtained by doing only one benchmark run, without any checks whether the resulting findings were a random occurrence or not [1]. Some random variations or isolated incidents in the benchmarks may lead to results that appear to show a clear relationship. However, repeating the experiment several times would likely reveal that these results were merely coincidental. Hence, it is important to make sure the relationship seen in the results is real by checking if the experiments are repeatable. To ensure that this is the case for our experiment, we measure every benchmark 9 times. Because of this, we can verify whether our results are relatively consistent over these 9 runs by looking at the spread of the measurements.

### 2.1.2 Technical requirements for replicable benchmarking

Various technical aspects can invalidate benchmarking results [1].
The first thing to consider is whether the measurements are actually valid and reliable. This is the case if the random and systematic measurement error is small (this means that there is no bias and there are no "volatile" effects) and there is adequate precision [1]. There are several places in the measuring process where this can go wrong.

To start off, it can go wrong in the measuring tool used. When measuring the execution time, it should be measured using a reliable tool [1]. Some measuring tools can be influenced, for example by changes to the system clock, resulting in invalid results. A good tool to measure the time the execution of a program took is the Unix `time` utility. We will be using this utility in our measurements and look at the real time this tool outputs.

In addition, when running a benchmark, no other processes should be able to have a performance influence on the benchmark [17]. This can be avoided by only executing one benchmark at a time and ensuring that out-

side of the benchmark a minimal amount of other processes are active. We run our experiments on a server that is ensured to run minimal processes in the background of the one running at that moment. This ensures that the performance influence is minimal.

There are other considerations, but these are not applicable to this thesis. They are only relevant under certain conditions, for example when benchmarking concurrent code or benchmarking memory.

## 2.2 Benchmarking in practice

When reading on how researchers have handled benchmarking programming languages, we encountered several procedures:

- Nanz and Furia [12] benchmarked different programming languages by running code from the Rosetta Code repository. This is a database with programming problems and their solutions implemented in as many languages as possible (Section 3.1 has more information on this repository). For benchmarking the code, they had a script for every language. This script takes an executable name, executes it, and logs the results. The results were gotten by repeating the execution of the code 6 times, and discard the first of these in the case of bytecode languages that had to load virtual machines from disk. Next, the remaining 5 results were taken only if they were within one standard deviation of the mean. If that was the case they logged the mean, otherwise they ran the execution again.

- Fourment and Gillings [5] benchmarked certain bioinformatics algorithms implemented in different programming languages. During the benchmarks they ensured that unnecessary processes were disabled. They used the user time measured with the GNU program `time` for the speed of processing. Every program was run three times. Their final measurement for each program was determined by taking the minimum execution time recorded.

- Nanz, West, Silveira, and Meyer [13] benchmarked four different programming languages for parallel programs. This study looked at the code size, coding time, execution time, and speedup. Their setup for measuring the execution time was relatively simple. They ran all their performance tests 30 times, and the mean of the results was taken.

As is the scientific standard when measuring a variable, in all these examples, they measure the time $x$ times and then take a statistic that summarizes these measurements to be the final value for that variable. The

mean is the most commonly used statistic for this purpose [16]. Using the minimum, as Fourment and Gillings did, is unusual.

As mentioned in the previous section, our procedure for measuring execution time and energy consumption is similar to the method used by Nanz, West, Silveira, and Meyer.

# Chapter 3

# Measuring energy

In this chapter, we discuss measuring energy. The topics covered include related work on measuring energy, the test set-up that we use when measuring our energy, and the validity of our energy measurements.

## 3.1 Related work on measuring energy

As mentioned in the introduction, in 2017 Periera et al. wrote an interesting paper on a method to rank programming languages on time, memory consumption, and most importantly energy consumption [14]. In this study, they took the code from the Computer Language Benchmarks Game (CLBG) [6], a repository containing the optimized implementations of a number of programming problems. These implementations are made in as many different languages as possible. In 2021, they did a validation study that took less optimized code from the Rosetta Code repository. This also has implementations of the same task in many different languages [11], but these implementations have to uphold to looser guidelines. This makes them more representative of day-to-day programming practices [14].

Their measurements on both sets of programming problems led to similar conclusions. Of the different conclusions that they drew, there were two most interesting for this thesis. First, they found that a faster language is not always the most energy efficient. Second, they made a ranking of programming languages based on their energy consumption, execution time, and memory usage. This ranking is shown in Figure 3.1.

In this figure it can be seen that the language C is, overall, the fastest and most energy efficient. Although the actual results nuance this view slightly—for example, in some benchmarks, it is neither the fastest nor the most energy efficient—it is still consistently in the top three [14].

| Total | | | | | |
|---|---|---|---|---|---|
| | Energy (J) | | Time (ms) | | Mb |
| (c) C | 1.00 | (c) C | 1.00 | (c) Pascal | 1.00 |
| (c) Rust | 1.03 | (c) Rust | 1.04 | (c) Go | 1.05 |
| (c) C++ | 1.34 | (c) C++ | 1.56 | (c) C | 1.17 |
| (c) Ada | 1.70 | (c) Ada | 1.85 | (c) Fortran | 1.24 |
| (v) Java | 1.98 | (v) Java | 1.89 | (c) C++ | 1.34 |
| (c) Pascal | 2.14 | (c) Chapel | 2.14 | (c) Ada | 1.47 |
| (c) Chapel | 2.18 | (c) Go | 2.83 | (c) Rust | 1.54 |
| (v) Lisp | 2.27 | (c) Pascal | 3.02 | (v) Lisp | 1.92 |
| (c) Ocaml | 2.40 | (c) Ocaml | 3.09 | (c) Haskell | 2.45 |
| (c) Fortran | 2.52 | (v) C# | 3.14 | (i) PHP | 2.57 |
| (c) Swift | 2.79 | (v) Lisp | 3.40 | (c) Swift | 2.71 |
| (c) Haskell | 3.10 | (c) Haskell | 3.55 | (i) Python | 2.80 |
| (v) C# | 3.14 | (c) Swift | 4.20 | (c) Ocaml | 2.82 |
| (c) Go | 3.23 | (c) Fortran | 4.20 | (v) C# | 2.85 |
| (i) Dart | 3.83 | (v) F# | 6.30 | (i) Hack | 3.34 |
| (v) F# | 4.13 | (i) JavaScript | 6.52 | (v) Racket | 3.52 |
| (i) JavaScript | 4.45 | (i) Dart | 6.67 | (i) Ruby | 3.97 |
| (v) Racket | 7.91 | (v) Racket | 11.27 | (c) Chapel | 4.00 |
| (i) TypeScript | 21.50 | (i) Hack | 26.99 | (v) F# | 4.25 |
| (i) Hack | 24.02 | (i) PHP | 27.64 | (i) JavaScript | 4.59 |
| (i) PHP | 29.30 | (v) Erlang | 36.71 | (i) TypeScript | 4.69 |
| (v) Erlang | 42.23 | (i) Jruby | 43.44 | (v) Java | 6.01 |
| (i) Lua | 45.98 | (i) TypeScript | 46.20 | (i) Perl | 6.62 |
| (i) Jruby | 46.54 | (i) Ruby | 59.34 | (i) Lua | 6.72 |
| (i) Ruby | 69.91 | (i) Perl | 65.79 | (v) Erlang | 7.20 |
| (i) Python | 75.88 | (i) Python | 71.90 | (i) Dart | 8.64 |
| (i) Perl | 79.58 | (i) Lua | 82.91 | (i) Jruby | 19.84 |

Figure 3.1: The normalized global results for energy, time, and memory of Periera et al.

A shortcoming of this study and their ranking is that it does not actually look what the underlying features of the programming languages are. They distinguish only between the execution types: compiled, virtual machine, and interpreted. This makes it seem like these are the only determining factors of the energy efficiency, runtime, and memory consumption of a programming language. But a programming language has other features, which can all have an impact on its consumption of resources, like its typing system, or whether or not it has a garbage collector. In this thesis, we examine other possible features and their impact. Through the lens of these features, we also aim to gain a better understanding of Figure 3.1.

In addition to the results of this study by Periera et al., their method for measuring the energy is relevant. They got their energy consumption results through measurements using RAPL [14]. Intel's Running Average Power Limit (RAPL) tool was introduced to allow a user or an OS to specify maximum power limits for the processor, GPU, and DRAM. To adhere to these power limits, the processor has to be aware of its power usage at every moment [3]. RAPL does not directly measure this, but uses a power model. As an added feature, the user can also access these power estimations for CPU, GPU, and DRAM. This is a useful feature, because not many people

have access to hardware that can measure the energy consumption of a whole machine. RAPL makes estimating the energy consumption of code more accessible. Since it does not do hardware measurements but uses an energy model, its accuracy is around 20% [3], which can be accurate enough depending on the purposes.

## 3.2 The test set-up

Instead of using RAPL, we use a hardware setup to measure the energy consumption of our code. The hardware setup used is the machine of the Software Energy Lab. The Software Energy Lab is a set-up that is part of a research project aiming to get better insight in the energy consumption of software. With this set-up come several servers on which people can run code with hardware that measures the energy consumption. Our code runs on an ODROID-H3+ server.[1] The energy consumption of this machine is measured with an INA260 chip. This chip is positioned in between the ODROID's power source and the device itself.
The test set-up makes its measurements available by responding to HTTP POST requests in the form of a JSON, an example of this response and explanation of the metrics involved can be found in Appendix A. The INA260 chip measures power input and calculates the energy consumption of the entire machine with a sampling frequency of 806 Hz.

## 3.3 Validity energy measurement

To have actual valid energy measurements with this device, we need to ensure that its accuracy is within reasonable bounds. In data sheets from Texas Instruments [8], it can be seen that the maximum error of the INA260 chip is 0.15%. This is the maximum error and with normal behavior it is lower, around 0.02%. It is our conviction, that this percentage is sufficiently small, such that we can get valid conclusions from the measurements of the device.

[1] This contains an Intel N6005 quad-core processor, a NVMe SSD, and 16 GB DDR4 RAM.

# Chapter 4

# Methodology

In this chapter, we describe the methods we used in this thesis. Firstly, we discuss noise factors and their mitigation. Then we discuss the selection of the code problems. Lastly, we talk about the measuring process as a whole.

## 4.1 Noise factors and mitigation

When measuring the energy consumption of our code, it is important to identify possible noise factors and account for them. In this section, we name our noise factors and explain our way of handling them.

### 4.1.1 System load

We run the energy measurements on a system, and we measure the energy consumption of the whole system during the run of our program. Hence, it is important to ensure that there is a minimal amount of other processes running during the whole duration of the energy measurements. The set-up of the Software Energy Lab that we use for the measurements ensures that the process we are running is the only process running on the system at that time. It does this by using the Gitlab CI/CD interface for running code, the runner can only execute one job at a time, because of this two jobs cannot be running at the same time, competing for resources.

### 4.1.2 Idle energy consumption

Whenever a computer, or server, performs any action or is simply powered on, it always consumes some amount of energy to keep everything running. This minimal amount of energy, that is always consumed, is called the idle energy consumption. When benchmarking the energy consumption of our programs, we are only interested in the energy that the actual program consumed, not in the energy that went into keeping the underlying system running. This actual energy is important for two reasons. The first reason

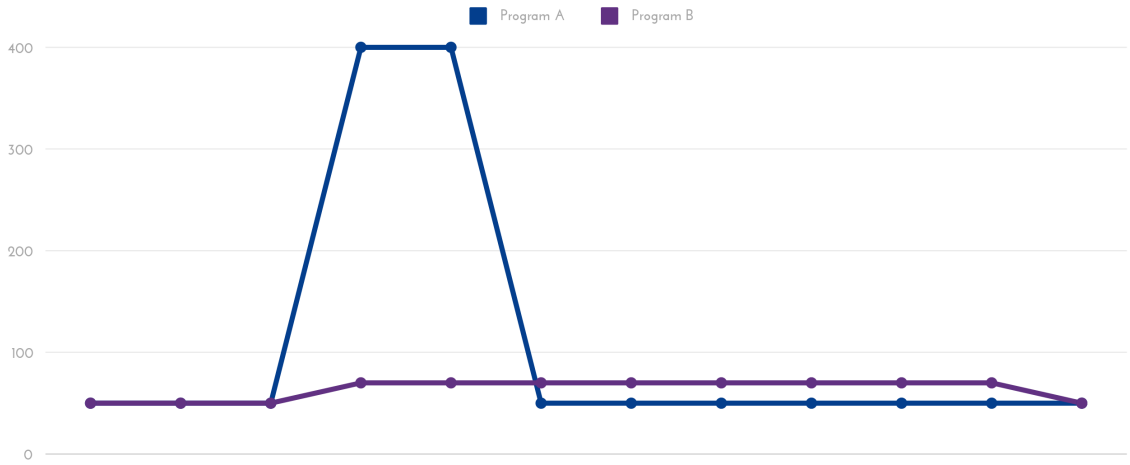


Figure 4.1: Example of skewed energy consumption between program A and B

is that changes in the energy that is being consumed in the background can greatly affect the results. The second reason is that it makes sure that the results are not naturally skewed towards the faster programs.

For instance, take two programs: A and B. A is very fast and uses up a lot of energy, B is a lot slower and uses little energy. Now, say we measure the energy consumption of both these programs. Then Figure 4.1 shows the resulting energy measurements over time, where program A corresponds to the blue line and program B to the purple line. The graph shows that the idle power consumption is 50 W. It also shows that program A should be considered the most expensive, with a big gap to program B. But if we were to include the idle energy consumption this gap would be much smaller, since the idle consumption that is present during their whole duration has to be included for both A and B, which is a lot more for B than it is for A.

For these reasons, we subtract the idle energy consumption from the energy measurement to prevent unfairly skewing the outcomes towards faster programs. The idle power consumption is measured before each measurement and afterwards. When these values do not differ more than 5%, the measurement is considered to be valid and the idle energy consumption is computed and subtracted from the measurement. Otherwise, the measurement is discarded, because then it is possible that during the duration of the code there was still a background process using up energy. The server also has a known normal idle power consumption which remains relatively steady between major updates. With our measured power consumption we also check whether it does not differ from this normal idle power consump-

tion by more than 10%, otherwise the measurement is also found invalid and discarded.

The measuring of the idle power consumption goes as follows. First, we measure the energy consumption of the command `sleep 60`. Then we divide the resulting value by 60 to get the idle wattage. Now to compute the idle energy consumption during the run of a program, we take this idle wattage and multiply it with the amount of seconds the program ran, this gives the amount of Joules that were idly consumed.
Given this idle energy consumption we can then compute the amount of Joules that was actually consumed by subtracting this idle energy consumption from the energy measurement of the program.

## 4.2 Code selection

The code Periera et al., used for the validation of their research [14], is the starting point of the code we use in our measurements. For this validation part of their study, they used the following problems from the Rosetta Code repository[1] `MergeSort`, `QuickSort`, `Hailstone`, `Fibonacci`, `Ackermann`, `N-Queens Problem`, `100-doors`, `Remove duplicates`, `Sieve of Eratosthenes`. They have a Github repository[2] containing the code and input for these and some other problems from the Rosetta Code respository. In this thesis, the goal is to take as many of the same problems as in their validation study and implement as many of our features on them. We need to add three new problems from Rosetta Code, since only one of the problems in this list (`MergeSort`) had a significant amount of `malloc()` calls, which are important for testing the garbage collection. These three problems were chosen from the other problems in the GitHub repository of Periera et al. We also discard two problems: `Fibonacci` and `Ackermann`, since their code is so simple that we cannot do anything interesting with any of the three features (i.e. no garbage collection, no arrays, only use of one type).

After this selection, we end up with the code problems that can be seen in Table 4.1. In this table, the "Implemented" column describes what of the features was implemented, so: garbage collection (GC); dynamic typing (DT); bounds checking (BC); bounds checking and garbage collection (GC+BC); and garbage collection and dynamic typing (GC+DT). The "Input" column defines the input for the program, these inputs are kept the same as the input Periera et al. used in their research.

[1] The whole Rosetta Code repository can be found here: https://rosettacode.org/wiki/Rosetta_Code

[2] This repository can be found at: https://github.com/greensoftwarelab/RosettaExamples/tree/master

| Benchmarks | Description | Input | Implemented |
|---|---|---|---|
| **Mergesort** | To sort a collection of integers using merge sort | 10k random integers | BC, DT, GC, GC+BC, GC+DT |
| **Quicksort** | To sort a collection of integers using quick sort | 10k random integers | BC, DT |
| **Hailstone sequence** | Generate the hailstone sequence for specific numbers | Rosetta Code | BC |
| **N-queens problem** | Solve the n-queens puzzle | 12-queens | BC, DT |
| **100-doors** | Solve the 100 doors problem | Rosetta Code | BC, DT |
| **Remove duplicates** | Remove duplicated elements in a sequence | $2^{17}$ random elements | BC |
| **Sieve of Eratosthenes** | Compute algorithm that finds the prime numbers up to a given integer | 10k | BC, DT |
| **Binary digits** | Create and display the sequence of binary digits for a given non-negative integer. | 1024 | BC, DT, GC, GC+BC, GC+DT |
| **Factors of an integer** | Compute the factors of a positive integer | Rosetta Code | BC, GC, GC+BC |
| **AVL tree** | Make an AVL tree of a given number of random elements | 20k | GC |

Table 4.1: The chosen set of programs from Rosetta Code with a description of what they do, the chosen inputs and the features we implemented for them.

When we have the chosen code problems, we start the implementation of the three features (bounds checking, garbage collection and dynamic typing). Their implementation is described in Chapter 5. The actual code we used, as well as the full implementations of these features, are available on GitHub.[3]

## 4.3 Measurements

Before beginning the measurement process, it is important to ensure that all programs run long enough, at least two minutes. This ensures the measurements' accuracy. All code is implemented in C, which is fast, hence to get the running times up to two minutes we execute the programs many times. The amount of times it actually runs for one specific measurement is written to a csv file.

To compare the different implementations, we need to collect the energy consumption of each of them. To do this we make a shell script that we can call from the Gitlab CI/CD pipeline. This script follows the same measurement process every time it is called. The process is shown in Listing 1, in which it is assumed that the variable `$executable_path` holds the path of

[3]This code can be found in a public GitHub repository: https://github.com/ElineStehouwer/features-energy-analysis/tree/main

the executable that is to be run.

```
idle_energy_before=10
idle_energy_after=0
diff=$(get_difference "$idle_energy_before" "$idle_energy_after")
ten_percent=0
while (( $(echo "$diff > $ten_percent" | bc -l) ));
do
    idle_energy_before=$(get_idle ...)
    energy_before=$(get_energy ...)
    time_output=$(run_program "$executable_path" )
    energy_after=$(get_energy ...)
    idle_energy_after=$(get_idle ...)
    diff=$(get_difference "$idle_energy_before" "$idle_energy_after")
    ten_percent=$(echo "$idle_after * 0.10" | bc -l)
done
elapsed_seconds=$(get_actual_elapsed_time "$time_output")
energy_consumed=$(echo "$energy_after - $energy_before" | bc -l)
idle_energy_consumed=$(echo "$total_seconds
                        * $idle_power_consumption" | bc -l)
actual_energy_consumed=$(echo "$energy_consumed
                        - $idle_energy_consumed" | bc -l)
```

Listing 1: Overall process of an energy measurement.

When we find the `actual_energy_consumed` to be negative, we restart this measurement process, since that indicates an invalid measurement.
Each benchmark implementation is executed and measured 9 times. Afterwards, we take the average of the measured energy consumption and execution time to summarize those values for that implementation.
Since we do not run all implementations the same number of times to get 2-minute measurements, we divide the resulting averages by the number of times it is repeated to get the actual energy consumption per run. Section 6.1 has a more detailed explanation of this process.

We conduct all benchmarks on the device detailed in Chapter 3, and we compile the code using `GCC` version 13.2.0 with optimization flage `-O3`

# Chapter 5

# Code modifications

In this chapter, we outline the specific changes we make to the original codebase. These modifications emulate certain programming language features in C to investigate the impact of these features on the performance of the code.

## 5.1 Overview of the original codebase

Originally the codebase consisted of, as described in the methodology in Chapter 4, normal C code gotten from the Rosetta repository, chosen mainly from programs that were pre-selected by Periera et al. Except for one program, all programs have some form of arrays in it, most use different types and some have `malloc()` statements in them.

We then transform this code by adding the previously mentioned features. As can be seen in Table 4.1, we do not implement all features in every program, this is why:

- For the garbage collection, the reason for this is that these programs do not have `malloc()` statements in them (or not a significant amount). Since a garbage collector would not have any significant impact there, we do not add garbage collection to them.
- For dynamic typing, the reason is that for some programs implementing it is too complicated for the time limits of the thesis. We do not want to invest too much time into one problem, since all the 10 problems need an implementation. This means that when we get really stuck on one problem, we quickly move on to the next program.
- For bounds checking, this is implemented in all programs but one. The reason it is not implemented in that one program is that it does not make use of arrays.

- We have been able to make implementations for the combinations: bounds checking and garbage collection, and dynamic typing and garbage collection. We did not make any for the combination of bounds checking and dynamic typing, since in the dynamic typing the array already had bounds checks incorporated. This means that the combination of dynamic typing and garbage collection actually incorporates all three features.

## 5.2 Description of modifications

In this section, we provide for every feature a description of how it works and of the modifications and additions made to the code.

### 5.2.1 Garbage collection

In C, when a program needs dynamic memory, normally it gets allocated by calling `malloc()` and released by calling `free()`, when the memory is not needed anymore. Now a program with a garbage collector allows allocating memory as normal, but freeing it is no longer necessary, as the garbage collector does that. Using a garbage collector can have benefits like preventing memory leaks that may occur when freeing is not done properly. When the collector sees that memory can no longer be accessed it recycles that memory. In this way, it is available for future `malloc()` calls.

Since it would be inefficient to implement a garbage collector ourselves and there is already a well-developed and tested garbage collector for C, we added the Boehm-Demers-Weiser conservative garbage collector to our code to do the garbage collection for us.

There are different algorithms underlying garbage collectors. This garbage collector uses a modified mark-sweep algorithm [2]. The algorithm has four phases that are now and then performed in the context of a memory allocation [2]:

1. Preparation: the process starts out by clearing all mark bits of all objects. These mark bits, which every object has, indicate whether an object is reachable. So, clearing these bits indicates they are potentially unreachable.

2. Mark phase: all objects that can be accessed by a series of pointer chains from variables get their mark bits set. Pointers are seen as any bit pattern that represents an address that is inside a heap object managed by the collector.

3. Sweep phase: in this phase the heap is scanned to look for objects that are unmarked, hence inaccessible. The collector puts the objects that it finds on a free list for reuse.

4. Finalization phase: the objects that have now been put in this free list and are to be collected can have a finalization function defined. When this is defined that code will be executed just before it gets collected, in the finalization phase. An example usage of this is to claim system resources or non-garbage-collected memory associated with the object.

We add the garbage collector to our code using C macro's. But first in the C file we have to add the header: `#include<gc.h>`. Then we define the macro's [7], as done here:

```
#define malloc(x) GC_malloc(x)
#define calloc(n,x) GC_malloc((n)*(x))
#define realloc(p,x) GC_realloc((p),(x))
#define free(x) (x) = NULL
```

These macros guarantee that whenever `malloc()`, `calloc()`, or `realloc()` is called in the code, the Boehm garbage collector equivalent function is called instead. For the `free` function, they ensure that no function is called (since the garbage collector handles cleanup) and they set the pointer to NULL, which can improve performance [7].

When the code has the correct macro's added to it, it still needs to be compiled correctly. The first step is to install the garbage collector library on our machine.[1] Then during the compilation, we link this library to the code on which we want to run the garbage collector. Before running the garbage collected program for the actual energy measurements it is important to check whether the garbage collector really is collecting garbage. This can be done by defining the `GC_PRINT_VERBOSE_STATS` environment variable. With this set the garbage collector will print a bit of descriptive output for each collection. We go over this output to check if it is acting as expected. After we verify that the collector actually works, we run the code with the garbage collector.

### 5.2.2 Dynamic typing

C is a statically typed language. This means that type checking happens at compile time, instead of at runtime, which is what happens with dynamic typing. For dynamically typed languages, this means that at compile time the compiler does not know of what type the variable they are handling is, hence they cannot make the same optimizations that it may be able to make

[1] Instruction on how to do this can be found here: https://hboehm.info/gc/#where.

for a statically typed language. It also has to perform more checks during runtime than a statically typed language. By implementing dynamic types in the C language, we try to determine the impact of having a dynamic and static typing system on the energy consumption of a programming language.

Since C has no ready to use library implementing a dynamic typing system, we implement it ourselves. We do this using the `union` type and a `tag`. The `union` type is, like a `struct`, a user-defined data type which can contain multiple elements of different types. In a `struct`, each element is stored in its own memory location, allowing all elements to be defined simultaneously. However, in a `union`, all elements share the same memory location, meaning only one element is defined at any given time. To keep track of which element is currently in use, `unions` are accompanied by a `tag`. This `tag` is a value of an `enum`, where each constant represents a possible data type that could be stored in the `union`. Essentially, the `tag` stores the constant corresponding to the data type currently stored in the `union`.
This `union-tag` combination emulates the idea of dynamic typing, since the compiler is not able to tell what value is stored in the union at compile time, and whenever an operation is performed at runtime a check needs to be performed to see if the types and the operation are compatible with each other.

We implement this idea in a C++ class. To ensure portability between programs, we put this class in a header that we can include. This header looks roughly like this:

```
class DynamicType {
    public:
        // First new names for each type used are defined so these
        // types can be easily changed
        typedef int NumberT;
        typedef std::string StringT;
        typedef vector<DynamicType> ArrayT;
        typedef char CharT;
        typedef short ShortT;
        typedef size_t SizeT;

        DynamicType(Number str);
        // And the other constructor declarations
        DynamicType operator+ (const DynamicType& other) const;
        DynamicType& operator[](int lookup);
        // And the other operator declarations
        bool isNumber() const;
        // And the other type check function declarations
```

```
        NumberT asNumber() const;
        // And the other conversion function declarations

        void print(std::ostream& out) const;

    private:
        enum {DTNull, DTBool, DTString, DTNumber, DTArray,
        DTChar, DTShort, DTSize} type; // The tag
        union {    // The union where one of the values is stored
            bool b;
            StringT str;
            NumberT number;
            ArrayT array;
            CharT c;
            ShortT s;
            SizeT size;
        };
};
```

In the implementationof the operators, we have a large switch statement going over the tag value and doing the addition operation if the tag has some number type in it.

Since this is C++ code, we need to make this accessible to C. We do this by adding another header file, this time representing our DynamicType class in a structure and having C accessible functions for every corresponding function on the class. In the C++ file corresponding to this header, we wrap all implementations of these functions in an extern block:

```
extern "C" {
// The code goes here
}
```

Through this `extern "C"` linkage these functions are made callable to our C code.
After defining this class, we rewrite all the code of our programs using the dynamic type we define and the functions corresponding to that type. When compiling our code, it is important to include all our headers and implementations of them and link them correctly.

### 5.2.3 Bounds checking

In a programming language, bounds checking means that whenever an array is accessed, a check is performed to ensure the access is within the bounds of the array. When a bounds check fails, it usually generates some exception

or error that exits the program. Many programming languages incorporate this in their array definition, but not all languages do (C for instance), because of efficiency reasons. However, one important reason why one might consider these checks is that not having them could cause bugs and security vulnerabilities, like buffer overflow.
So, for a programmer it is important that they take great care to deal with these array boundaries when writing code in a language without bounds checking.

We implement the bounds checks in C ourselves, because, just like with dynamic typing, we could not find an easily accessible library to do it for us. We implement it with the the following two macros:

```
#define setSafe(array, index, size, value) {\
    if (index < 0 || index >= size) {\
        fprintf(stderr, "Error: Index out of bounds\n");\
        exit(EXIT_FAILURE);\
    }\
    array[index] = value;\
}

#define getSafe(array, index, size) ({\
    if (index < 0 || index >= size) {\
        fprintf(stderr, "Error: Index out of bounds\n");\
        exit(EXIT_FAILURE);\
    }\
    array[index];\
})
```

Before including the bounds checks, the code would for example look like this:

```
int some_array = {1, 2, 3, 4};
int value = some_array[1];
some_array[3] = 10;
```

After including the bounds checks, this code would look like this:

```
int some_array = {1, 2, 3, 4};
int value = getSafe (some_array, 1, 4);
setSafe(some_array, 3, 4, 10);
```

With this definition of bounds checking, we rewrite all bounds checks into this format.

### 5.2.4 Combinations

The implementation of the combination features is straightforward. For both combinations, first follow the steps described in Section 5.2.1. Then follow

the steps as described in either Section 5.2.3 or Section 5.2.2, depending on the combination.

# Chapter 6

# Results

The aim of this thesis is to measure the energy impact of the programming language features garbage collection, bounds checking, and dynamic typing on C. These last two we implement ourselves. We ran measurements on the code problems that we implemented these three features on. In this chapter, we discuss the results gotten from these measurements. We start out by exploring all data, then we move on to explaining our calculations of the mean and spread, lastly we present our processed results in four tables.

## 6.1 Data preprocessing

To make the measurements as reliable as possible, we ran them until they took around 2 minutes. Different code implementations take different amounts of time. This means that the number of times they have to be run to run around 2 minutes also differs (e.g. 250 times and 61200 times). To be able to compare their mean energy consumption and execution time and the standard deviation of the energy consumption and execution time, we first calculate the energy and time per run for all nine measurements. We do this in the same way for both the time and energy. Different measurements of the same program might need slightly more or less runs to run the right amount of time. Hence, each of the nine measurements of every program might have a different number of times it was run.
So in the example of the n-queens problem: say in one of the nine measurements the code was executed 250 times and this measurement has measured energy consumption: $E_{measured} = 13.3524$ J. Now we can calculate the energy consumption of one run: $E_{run} = \frac{13.3524}{250} = 0.053409486$ J.
This same measurement has measured execution time: $t_{measured} = 114.093$ s. So using this we also calculate the execution time per run: $t_{run} = \frac{114.093}{250} = 0.456372$ s.
We do this for all measurements of all nine runs. We use the resulting values to calculate the mean and variability of the energy consumption and execu-

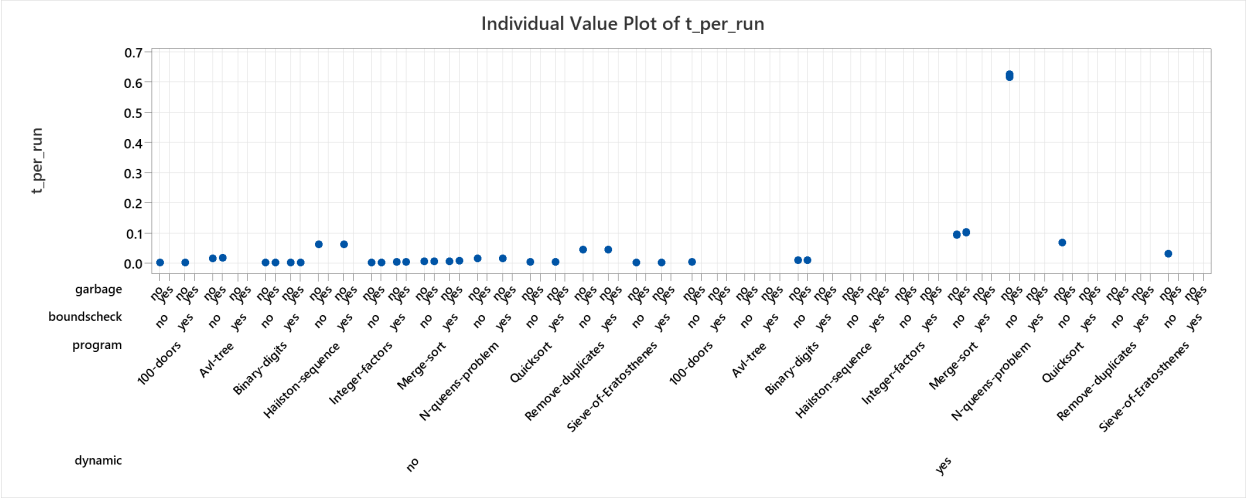


Figure 6.1: Graph plotting the measurements of the time in s, the data points are divided by program and feature they implement.

tion time of one run of every version of every program.
Before we look at these means and standard deviations, we first look all data of all measurements we have for the energy consumption and execution time per run.

## 6.2 Exploration of all data

All data that we got from the measurements can be seen in Figure 6.1 and Figure 6.2. The preprocessing step described in Section 6.1 is already applied to this data. These respectively hold the individual value plots for the time one run of every program takes and the individual value plots for the energy one run of every program consumes.

These plots give a rough overview of the data we gathered. The data points are divided into the category they belonged to, so by program and by which of the three features was implemented. So one vertical line of dots represents the 9 measurements of a single implementation.

In these plots, there are two things that stand out. The energy measurements have a bigger spread than the time measurement, mainly in the results implementing dynamic typing. In addition to this, there seems to be a correlation between the execution time and the energy consumption. This is not unexpected, because we know: $E = t \cdot P$. More evidence of this linear relation is seen in Figure 6.3, where we plotted the execution time in relation to the energy consumption.

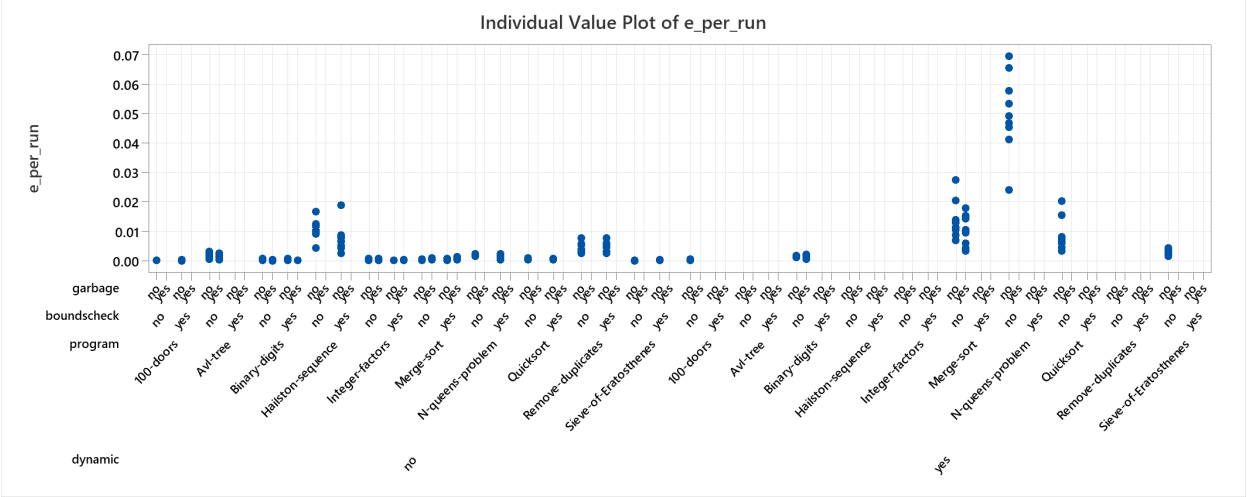


Figure 6.2: Graph plotting the measurements of the energy consumption in J, the data points are divided by program and feature they implement.

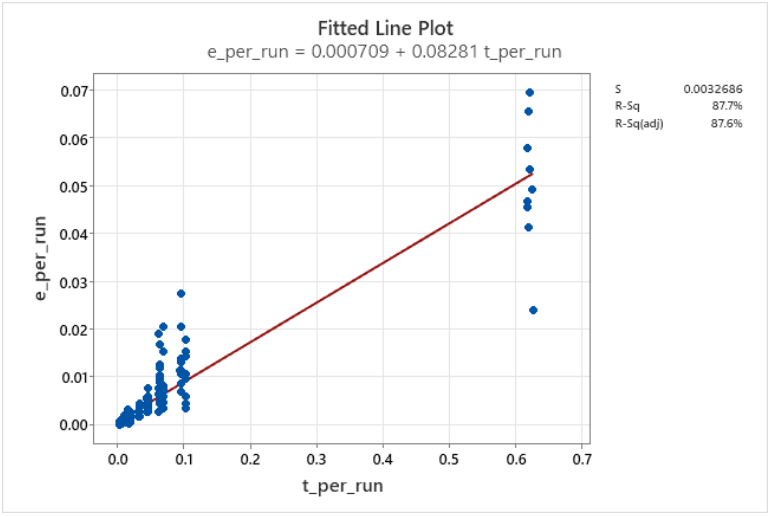


Figure 6.3: Scatter plot showing the linear relationship between execution time per run in s (t_per_run) and energy consumption per run in J (e_per_run) with a fitted regression line.

From the graph, we can conclude that although the energy measurements have a big spread, their averages are strongly correlated with the time measurements.

## 6.3 Processed results

For the measurements of time and energy we made two tables each. In Table 6.1, we put all means and standard deviations for the energy measurements. The first column states the program name. The next six columns list the mean energy consumption per run of the program: without additions (default), with bounds checking (BC), with bounds checking and garbage collection (GCBC), with garbage collection (GC), with garbage collection and dynamic typing (GCDT), and with dynamic typing (DT). The following six columns present the standard deviation of the energy consumption per run for the same respective categories. There are empty cells, because we did not implement all features in all programs.

In Table 6.2, we show slightly different data. In this revised table, actual values are not displayed; instead, normalized values are presented by dividing them by the default value. This format enables a direct comparison of different versions against the standard version. To enhance this comparison, color coding is used to indicate energy consumption differences: redder cells signify higher energy consumption than the default, whiter cells indicate energy consumption closer to the default, and greener cells represent lower energy consumption than the default.
There is also a slightly different subdivision of columns. The columns that previously showed the standard deviation now shows the coefficient of variation, which was calculated by dividing the standard deviation by the mean. While the standard deviation in the previous table helped identify potential data overlap across different features, the coefficient of variation is more beneficial in this table as it allows for comparing values relative to the default implementation.
For instance, without considering the coefficient of variation, one might mistakenly think the dynamic typing result for the n-queens problem is highly inaccurate due to its standard deviation being approximately 30 times greater than that of the default measurement. However, this difference is primarily because of the significantly higher energy consumption of the dynamic typing implementation, which the coefficient of variation adjusts for.

Table 6.3 and Table 6.4 are formatted in the same way as Table 6.1 and Table 6.2 respectively. The only difference is that these tables contain the data on the execution time of the programs instead of the energy consumption.

| Program | Mean | | | | | | Standard deviation | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | default | BC | GCBC | GC | GCDT | DT | default | BC | GCBC | GC | GCDT | DT |
| 100-doors | 0.266 | 0.333 | | | | 0.386 | 0.072 | 0.131 | | | | 0.196 |
| AVL tree | 1.598 | | | 1.619 | | | 0.807 | | | 0.622 | | |
| Binary digits | 0.272 | 0.298 | 0.233 | 0.329 | 1.198 | 1.695 | 0.189 | 0.148 | 0.081 | 0.148 | 0.510 | 0.231 |
| Hailstone sequence | 10.512 | 7.511 | | | | | 3.264 | 4.780 | | | | |
| Factors of an integer | 0.346 | 0.225 | 0.3 | 0.427 | | | 0.196 | 0.035 | 0.154 | 0.153 | | |
| Mergesort | 0.437 | 0.572 | 0.953 | 0.805 | 10.617 | 13.693 | 0.175 | 0.157 | 0.313 | 0.254 | 5.193 | 6.388 |
| N-queens problem | 1.882 | 1.342 | | | | 50.278 | 0.331 | 0.708 | | | | 13.551 |
| Quicksort | 0.551 | 0.643 | | | | 8.809 | 0.286 | 0.200 | | | | 5.497 |
| Remove duplicates | 4.513 | 4.609 | | | | | 1.678 | 1.729 | | | | |
| Sieve of Eratosthenes | 0.182 | 0.224 | | | | 3.246 | 0.061 | 0.109 | | | | 1.047 |

Table 6.1: Mean and standard deviation of energy (mJ) per single run of the code.

| Program | Mean | | | | | | Coefficient of variation | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | default | BC | GCBC | GC | GCDT | DT | default | BC | GCBC | GC | GCDT | DT |
| 100-doors | 1.0 | 1.2 | | | | 1.4 | 1.0 | 1.5 | | | | 1.9 |
| AVL tree | 1.0 | | | 1.0 | | | 1.0 | | | 0.8 | | |
| Binary digits | 1.0 | 1.1 | 0.9 | 1.2 | 4.4 | 6.2 | 1.0 | 0.7 | 0.5 | 0.6 | 0.6 | 0.2 |
| Hailstone sequence | 1.0 | 0.7 | | | | | 1.0 | 2.1 | | | | |
| Factors of an integer | 1.0 | 0.7 | 0.9 | 1.2 | | | 1.0 | 0.3 | 0.9 | 0.6 | | |
| Mergesort | 1.0 | 1.3 | 2.2 | 1.8 | 24.3 | 31.3 | 1.0 | 0.7 | 0.8 | 0.8 | 1.2 | 1.2 |
| N-queens problem | 1.0 | 0.7 | | | | 26.7 | 1.0 | 3.0 | | | | 1.5 |
| Quicksort | 1.0 | 1.2 | | | | 16.0 | 1.0 | 0.6 | | | | 1.2 |
| Remove duplicates | 1.0 | 1.0 | | | | | 1.0 | 1.0 | | | | |
| Sieve of Eratosthenes | 1.0 | 1.2 | | | | 17.8 | 1.0 | 1.5 | | | | 1.0 |

Table 6.2: Mean and coefficient of variation of energy per single run of the code normalized to the value of the default implementation.

| Program | Mean | | | | | | Standard deviation | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | default | BC | GCBC | GC | GCDT | DT | default | BC | GCBC | GC | GCDT | DT |
| 100-doors | 1.884 | 1.882 | | | | 4.711 | 0.021 | 0.018 | | | | 0.015 |
| AVL tree | 15.945 | | | 18.545 | | | 0.031 | | | 0.015 | | |
| Binary digits | 2.252 | 2.275 | 3.009 | 3.006 | 10.792 | 9.671 | 0.015 | 0.015 | 0.009 | 0.013 | 0.049 | 0.012 |
| Hailstone sequence | 62.948 | 62.273 | | | | | 0.310 | 0.233 | | | | |
| Factors of an integer | 2.303 | 3.329 | 3.340 | 2.294 | | | 0.012 | 0.008 | 0.010 | 0.011 | | |
| Mergesort | 5.369 | 5.983 | 7.732 | 7.110 | 102.381 | 94.773 | 0.012 | 0.014 | 0.014 | 0.019 | 0.499 | 0.515 |
| N-queens problem | 15.885 | 16.350 | | | | 620.124 | 0.030 | 0.014 | | | | 3.199 |
| Quicksort | 4.505 | 4.485 | | | | 68.849 | 0.013 | 0.013 | | | | 0.293 |
| Remove duplicates | 44.625 | 44.915 | | | | | 0.177 | 0.210 | | | | |
| Sieve of Eratosthenes | 1.880 | 1.893 | | | | 32.285 | 0.014 | 0.014 | | | | 0.196 |

Table 6.3: Mean and standard deviation of time (ms) per single run of the code.

| Program | Mean | | | | | | Coefficient of variation | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | default | BC | GCBC | GC | GCDT | DT | default | BC | GCBC | GC | GCDT | DT |
| 100-doors | 1.0 | 1.0 | | | | 2.5 | 1.0 | 0.9 | | | | 0.3 |
| AVL tree | 1.0 | | | 1.2 | | | 1.0 | | | 0.4 | | |
| Binary digits | 1.0 | 1.0 | 1.3 | 1.3 | 4.8 | 4.3 | 1.0 | 1.0 | 0.4 | 0.1 | 0.7 | 0.2 |
| Hailstone sequence | 1.0 | 1.0 | | | | | 1.0 | 0.8 | | | | |
| Factors of an integer | 1.0 | 1.4 | 1.5 | 1.0 | | | 1.0 | 0.5 | 0.6 | 0.9 | | |
| Mergesort | 1.0 | 1.1 | 1.4 | 1.3 | 19.1 | 17.7 | 1.0 | 1.0 | 0.8 | 1.2 | 2.2 | 2.4 |
| N-queens problem | 1.0 | 1.0 | | | | 39.0 | 1.0 | 0.5 | | | | 6.0 |
| Quicksort | 1.0 | 1.0 | | | | 15.3 | 1.0 | 1.0 | | | | 1.5 |
| Remove duplicates | 1.0 | 1.0 | | | | | 1.0 | 1.2 | | | | |
| Sieve of Eratosthenes | 1.0 | 1.0 | | | | 17.2 | 1.0 | 1.0 | | | | 0.8 |

Table 6.4: Mean and coefficient of variation of time per single run of the code normalized to the value of the default implementation.

# Chapter 7

# Discussion

In this chapter, we discuss and interpret the results as they are presented in Chapter 6.

## 7.1 Impact of added features

### 7.1.1 Bounds checking

We expected bounds checking to add a certain overhead due to the additional checks it performs during runtime. In Table 6.2, we see that the impact of bounds checking on the energy consumption differs per program. In some, it seems to make the program more energy efficient, in other programs, there is a slight increase in energy consumption. But when we look at the standard deviation in Table 6.1, we see that none of these differences are significant.

In Table 6.4, we see that no programs with bounds checking added are quicker than the default program. There are two programs which do become slightly slower with bounds checking added. These results are significant, this suggests that bounds checking could make some programs slower, but does not make a difference within other programs.

From these results, we can conclude that the impact of bounds checking is either minimal or non-existent on the type of numeric programs that we have tested on. One possible explanation for this finding lies in the operation of branch prediction in modern CPUs. Branch prediction involves the CPU estimating in advance which branch will be taken. With bounds checks, the CPU predicts that the check will succeed, which it does in all our programs, since these were written to not have any out-of-bounds errors, as is normal in well-written programs. This allows the CPU to continue execution efficiently without unnecessary interruptions.

Assembly instructions are executed in multiple stages through a pipeline. If the CPU correctly predicts the outcome of a branching, it can already partially execute instructions in the pipeline, which provides more efficient use of resources and less energy consumption. However, if the prediction is incorrect, all partially executed instructions must be discarded and restarted, which is costly in terms of time and energy. This almost never happens in correctly written programs, so there are rarely wrong predictions on these types of branches by the branch predictor. Since these predictions are correct, the only added costs of bounds checking is the if-statement checking whether it is in bounds. Thus, the few extra assembly instructions that bounds checking adds have no significant impact on pipelining efficiency, energy consumption, and execution time.

### 7.1.2 Garbage collection

We expected garbage collection to add some overhead due to the fact that it has to keep track of the malloced pieces of memory and manage them during runtime. In Table 6.2, we see that garbage collection has some kind of impact on the energy consumption of every program, except for AVL tree. The size of impact does differ per program. The energy consumption of mergesort goes up most. This is for several reasons. First, this program has the most mallocs relative to the amount of code, since every merge calls `malloc()`. These results show that the intensity with which the garbage collector has to allocate memory and clean it up is a factor that changes its energy consumption.
Second, compared to the other programs mergesort calls `free()` more frequently. A garbage collector only has to actually collect "garbage" if this garbage is no longer used. In mergesort, for every merge, memory has to be claimed through `malloc()` and after the merge the memory goes out of scope, which means it can be cleaned up by the garbage collector. The other programs often use the pointers they create with `malloc()` for a bigger part of the program and hence the garbage collector has nothing to clean up during those periods.
When we look at the standard deviation in Table 6.1, not all these differences are significant, but there does seem to be a trend toward a larger energy consumption with garbage collection enabled.

In Table 6.4, we see a similar pattern. The difference is that although the gap between the execution time for the default programs and the programs with garbage collection added to it is smaller, Table 6.3 shows that these numbers are actually significant.

From the data, we can conclude that garbage collection has an impact on the energy consumption and execution time of programming languages

in the type of numeric programs that we have tested on. We also find that calling `malloc()` more often leads to a higher energy consumption. It is notable that the runtime of mergesort with garbage collection increases less than the energy consumption.
Memory operations generally consume a lot of energy, and this consumption is not always linear with time. This could be a possible reason for this difference. While a standard algorithm is primarily CPU-intensive, garbage collection creates additional memory operations, resulting in higher overall hardware usage. Consequently, more frequent access to the entire memory leads to less efficient use of the cache and more direct access to RAM, triggering additional circuits and increasing power consumption.
A different explanation would be that many operations are cached, meaning that the initial cache operation is slow, but repeated accesses are faster. But the operations themselves can still consume a lot of energy, even though they can be quickly retrieved.

We cannot say for sure which of these explanations, if any, is the actual cause of this difference in impact on energy consumption and execution time.

### 7.1.3 Dynamic typing

We expected that the added runtime checks introduced by dynamic typing would result in an efficiency overhead. In Table 6.2, we see that dynamic typing has a big impact on all implementations. As we observed with the garbage collection the difference does depend on the program. For example, it only makes 100-doors consume 1.4 times more energy than the default implementation, while it makes N-queens problems consume 26.7 times more energy. This can be explained by the complexity of the program. 100-doors is a nested for-loop, which does one array access and one array assignment every run of the loop. In the n-queens problem, there are many more operations that are more energy intensive, like declaring and initializing big arrays and doing complicated computations. We see the same effect on execution time in Table 6.4. In Table 6.1 and Table 6.3, we can see that all these differences are significant, except for the difference in energy consumption of the 100-doors problem.

From these numbers, we can conclude that dynamic typing has a significant impact on the energy consumption and execution time of a programming language in the type of numeric problems we have tested on. However, the amount of influence can vary from program to program. The impact of dynamic typing is due to several factors.

First, dynamic typing reduces the effectiveness of branch prediction. In integer-only numeric programs, the processor can often successfully predict

which branch to follow, especially with code inlining. Dynamic typing makes these predictions less reliable, because the processor cannot predict as well what type of value is in a variable. This reduces efficiency, since a branch miss can be costly.

Second, dynamic typing hinders auto-vectorization. Normally for statically typed programs with many numeric operations, the compiler can perform optimizations, such as merging multiple operations into one vector operation. This is more efficient and reduces energy consumption. With dynamic typing, however, the compiler is not sure which data types are being used, so this optimization is not possible.

Third, the way we implemented dynamic typing, with a union type, causes inefficient memory management. Because variables are stored in `union`-types, the largest possible type determines the size of the `union`. For example, an integer that normally occupies 8 bytes is now stored in a 24-byte `union`-type if the largest possible variable within that `union` is 24 bytes. This means that caching is more inefficient, increasing the energy consumption and execution time.

In short, dynamic typing introduces several inefficiencies in both processing and memory management, leading to higher energy consumption and execution time. This is an important consideration when designing and optimizing programs that rely on dynamic typing.

### 7.1.4 Garbage collection and bounds checking

When combining garbage collection and bounds checking we generally see in Table 6.2 that the individual overheads of the energy consumption have an additive nature, except for for the binary digits program. However, from the data of that program we cannot draw any conclusions, because the standard deviation of the default, bounds checking and garbage collection program in Table 6.1 are too high.

This same additive relation can be seen in all programs when we look at the execution time measurements in Table 6.4. These results are significant, as the standard deviation in Table 6.3 shows. So in this case we can conclude that for the execution time and energy consumption, there is an additive relation between garbage collection with bounds checking in the type of numeric programs that we tested on.

### 7.1.5 Garbage collection and dynamic typing

When we look at the data on the energy consumption of garbage collection combined with dynamic typing in Table 6.2, there seems to be a different relation at first. This combination of features looks more energy efficient than dynamic typing on its own. As with the previous combination, when we look at Table 6.1, we can see that these differences are not significant. Therefore, we cannot draw any conclusions about the relationship between the energy consumption of garbage collection with dynamic typing.

In Table 6.4, we see a different relationship between the measurements of the execution time. When we add garbage collection to a dynamically typed program the execution time goes up more than when we add it to a program without. As opposed to the results on energy consumption, the standard deviation in Table 6.3 shows that these results are actually significant. This means we can conclude that the combination of garbage collection and dynamic typing takes more time than only dynamic typing in numeric programs. A possible explanation for the phenomenon that garbage collection adds a bigger load when using dynamic typing compared to without it, is that these dynamically typed variables all look like pointers to the garbage collector. Hence, it assumes it has to garbage collect all these possible pointers, while in the original program the difference between an integer (which did not need to be garbage collected) and a pointer was clear to the garbage collector.

## 7.2 Comparing with programming language ranking

Now that we know these numbers, we look back at the energy rankings in Figure 3.1 in Chapter 3 and try to explain the differences of languages that we see there.

The 27 languages Periera et al. investigated can all be divided into categories based on which of the three features they have. Every one of those categories has some average added energy consumption and execution time. We calculate each of these by taking the average of the values in Tables 6.2 and 6.4. For the averages of energy consumption, we discard the values for which the normalized coefficient of variation is higher than 1.5. For those of execution time we do not discard anything, because for those measurements all standard deviations were sufficiently small. We have listed these categories, the languages that belong to each category, and the average added costs:

- No features: C++, Pascal, Chapel, and Fortran. This category has no

added features, meaning the costs (in terms of energy and time) are the same as those of C.

- Bounds checking: Rust and Ada. This category consumes, on average, 1.1 times more energy and has a 1.06 times longer execution time.
- Garbage collection: Ocaml, TypeScript, and Hack. On average, this category consumes 1.55 times more energy and has a 1.20 times longer execution time.
- Bounds checking and garbage collection: Java, Swift, Haskell, C#, Go, Dart, and F#. This category consumes, on average, 1.33 times more energy and has a 1.40 times longer execution time.
- Garbage collection and dynamic typing: Lisp, JavaScript, Racket, PHP, and Lua. Although we did not explicitly measure this combination (since bounds checking was already incorporated in our dynamic typing system), we estimated the average cost by subtracting the average added costs of bounds checking from those of the combination of garbage collection and dynamic typing. Thus, this category consumes, on average, 14.25 times more energy and has an 11.90 times longer execution time.
- Bounds checking, garbage collection, and dynamic typing: Erlang, JRuby, Ruby, Python, and Perl. On average, this category consumes 14.35 times more energy and has an 11.95 times longer execution time.

We make a table with these values, comparing the normalized energy consumption and execution time that these languages would have if the calculations were correct and the normalized energy consumption and execution time that Periera et al. measured. This can be seen in Table 7.1, which has 27 languages ordered from most to least efficient in that category, either energy or time.

When we look at this table, for some languages we see that our calculated cost estimation comes pretty close to the actual costs that Periera et al. observed. These languages are Rust, C++, Ada, Java, and we can add Racket to this if we only consider the execution time. Here there are still some minimal differences, but these could be explained by different hardware. Here it stands out that almost all of these languages are either compiled or use a virtual machine.

Then there are two groups of languages that both have a significant gap of unexplained costs, the difference between these two groups is how big the part of the costs is that could not be explained by the three features. The first group contains the languages Pascal, Chapel, Ocaml, Fortran, Swift,

| Total | | | | | |
|---|---|---|---|---|---|
| | Energy (J) | | | Time (ms) | |
| | **Periera et al.** | **Calculated costs** | | **Periera et al.** | **Calculated costs** |
| C | 1.00 | 1.00 | C | 1.00 | 1.00 |
| Rust | 1.03 | 1.10 | Rust | 1.04 | 1.06 |
| C++ | 1.34 | 1.00 | C++ | 1.56 | 1.00 |
| Ada | 1.70 | 1.10 | Ada | 1.85 | 1.06 |
| Java | 1.98 | 1.33 | Java | 1.89 | 1.40 |
| Pascal | 2.14 | 1.00 | Chapel | 2.14 | 1.00 |
| Chapel | 2.18 | 1.00 | Go | 2.83 | 1.40 |
| Lisp | 2.27 | 14.25 | Pascal | 3.02 | 1.00 |
| Ocaml | 2.40 | 1.55 | Ocaml | 3.09 | 1.20 |
| Fortran | 2.52 | 1.00 | C# | 3.14 | 1.40 |
| Swift | 2.79 | 1.33 | Lisp | 3.40 | 11.90 |
| Haskell | 3.10 | 1.33 | Haskell | 3.55 | 1.40 |
| C# | 3.14 | 1.33 | Swift | 4.20 | 1.40 |
| Go | 3.23 | 1.33 | Fortran | 4.20 | 1.00 |
| Dart | 3.83 | 1.33 | F# | 6.30 | 1.40 |
| F# | 4.13 | 1.33 | JavaScript | 6.52 | 11.90 |
| JavaScript | 4.45 | 14.25 | Dart | 6.67 | 1.40 |
| Racket | 7.91 | 14.25 | Racket | 11.27 | 11.90 |
| TypeScript | 21.50 | 1.55 | Hack | 26.99 | 1.20 |
| Hack | 24.02 | 1.55 | PHP | 27.64 | 11.90 |
| PHP | 29.30 | 14.25 | Erlang | 36.71 | 11.95 |
| Erlang | 42.23 | 14.35 | JRuby | 43.44 | 11.95 |
| Lua | 45.98 | 14.25 | TypeScript | 46.20 | 1.20 |
| JRuby | 56.54 | 14.35 | Ruby | 59.39 | 11.95 |
| Ruby | 69.91 | 14.35 | Perl | 65.79 | 11.95 |
| Python | 75.88 | 14.35 | Python | 71.90 | 11.95 |
| Perl | 79.58 | 14.35 | Lua | 82.91 | 11.90 |

Table 7.1: Normalized energy consumption and execution time of 27 programming languages with both the values Periera et al. measured and the costs we calculated.

Haskell, C#, Go, Dart, and F#. For these languages it hence seems that their costs in the ranking of Periera et al. should be 1 to 3 points lower. These differences may be due to noise in our measurements or those of Pereira et al. However, it is also possible that other factors are at play. For example, the execution type (compiled, virtual machine, or interpreted), the programming paradigm (imperative, object-oriented, functional, or scripting), or other unidentified factors could influence the results.

The second group contains TypeScript, Hack, PHP, Erlang, Lua, JRuby, Ruby, Python, and Perl. These languages have much higher costs than the costs we calculated based on the three features. The differences are so big that there seem to be other factors in these language that play an big role in the energy consumption and execution time of the program. For future research, it is worth it to look at the differences between dynamically typed languages like JavaScript, which is relatively efficient, and Python, which is relatively inefficient. These two have all features in common that we researched, but there seem to be other factors that cause them to behave very differently in terms of energy consumption and execution time.
An interesting outlier in this group is TypeScript, which is the only statically typed language in this group. This is possibly, because at its core it is still a dynamically typed language. When TypeScript gets compiled it is first translated to JavaScript and then run as if it was JavaScript. This means we would expect its performance to be similar to JavaScript, but it is 5 times more inefficient. This is another difference that should be further looked at in future research.

Finally, there are two languages that performed a great deal better than than our calculated costs expected, Lisp and JavaScript. Note that both of these are dynamically typed languages. A possible reason for this could be that the implementation of their dynamic typing system under the hood is somehow more efficient than ours, making our implementation less representative for these languages. Or there could be other reasons that cause this difference in estimation, for example that the interpreter, or virtual machine of these languages is just better at optimizing the handling of dynamic types, where the C compiler is not made to do specifically that, since C is a statically typed language. Both of these possible explanations are interesting starting points for possible future research.

# Chapter 8

# Conclusions

In this thesis, we investigated the impact of the programming language features bounds checking, garbage collection, and dynamic typing on the energy consumption and execution time of a programming language in numeric programs. We did this by implementing these features in C and measuring these metrics on 10 different programs with and without these features. We implemented the features bounds checking and dynamic typing ourselves and used the Boehm-Demers-Weiser garbage collector for garbage collection.

Our measurements examined the impact of three features and their combinations on program performance. Firstly, bounds checking was found to have minimal effect due to effective branch prediction in the absence of out-of-bound errors. Secondly, garbage collection negatively impacts efficiency, especially with frequent `malloc()` calls, affecting energy consumption and execution time differently due to non-linear memory operations and inefficient energy use of cached data. Thirdly, dynamic typing greatly increases both energy consumption and execution time, likely because of inefficiencies in processing and memory management.

When combining features, garbage collection and bounds checking exhibited an additive effect, while the effect of garbage collection got bigger when combined with dynamic typing.

With these results, it is important to keep in mind that future research is needed to verify them and further investigate the findings. One important aspect to verify is that these results do not only hold for this specific language C with the compiler `gcc`. This can be tested by repeating these measurements and implementations in Rust (which just like C sits close to the physical machine) and by repeating them in C using the `clang` compiler. Another aspect that needs testing and verification is whether these results apply to all types of problems or only to the numeric problems we researched.

The reason for this research was the article by Periera et al. that made a ranking of 27 different programming languages based on their energy efficiency and execution time. We looked back at the main results of their article with the knowledge we gained from our own measurements. From this reflection, we found that for some languages them having some combination of the three features completely explains their energy consumption. For others, it did not, which is logical, since these three features are usually not the only factors on which programming languages differ, other factors might be execution type, programming paradigm, or something else entirely. There were big differences between the amount of effect that was explained by the existence of bounds checking, garbage collection, and/or dynamic typing. These differences, possible factors that could explain them, and what the impact is of those factors are all of interest for future research.

# Bibliography


[1] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Reliable benchmarking: requirements and solutions". In: *International Journal on Software Tools for Technology Transfer* 21 (2019), pp. 1–29.

[2] Hans-J. Boehm, Alan Demers, and Mark Weiser. *Conservative GC Algorithmic Overview*. 2014. URL: `https://hboehm.info/gc/gcdescr.html`.

[3] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. "A Validation of DRAM RAPL Power Measurements". In: *Proceedings of the Second International Symposium on Memory Systems*. MEMSYS '16. Association for Computing Machinery, 2016, pp. 455–470.

[4] Bruno Dufour et al. "Dynamic metrics for java". In: *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. OOPSLA '03. Association for Computing Machinery, 2003, pp. 149–168.

[5] Mathieu Fourment and Michael R. Gillings. "A comparison of common programming languages used in bioinformatics". In: *BMC Bioinformatics* 9 (2008), pp. 1–9.

[6] Computer Language Benchmarks Game. *Measure "Which programming language is fastest?"*. 2024. URL: `https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html`.

[7] Gianluca Insolvibile. *Garbage Collection in C Programs*. 2003. URL: `https://www.linuxjournal.com/article/6679`.

[8] Texas Instruments. *INA260 Precision Digital Current and Power Monitor With Low-Drift, Precision Integrated Shunt*. 2024. URL: `https://www.ti.com/document-viewer/ina260/datasheet`.

[9] Jóakim von Kistowski et al. "Variations in CPU Power Consumption". In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. ICPE '16. Association for Computing Machinery, 2016, pp. 147–158.

[10] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. "Cross-language compiler benchmarking: are we fast yet?" In: *SIGPLAN Not.* 52 (2016), pp. 120–131.

[11] Mike Mol. *Rosetta Code*. 2023. URL: `https://rosettacode.org/wiki/Rosetta_Code`.

[12] Sebastian Nanz and Carlo A. Furia. "A Comparative Study of Programming Languages in Rosetta Code". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 2015, pp. 778–788.

[13] Sebastian Nanz et al. "Benchmarking Usability and Performance of Multicore Languages". In: *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 183–192.

[14] Rui Pereira et al. "Ranking programming languages by energy efficiency". In: *Science of Computer Programming* 205 (2021), p. 102609.

[15] Gustavo Pinto, Fernando Castor, and Yu David Liu. "Mining questions about software energy consumption". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Association for Computing Machinery, 2014, pp. 22–31.

[16] StudySmarter. *Statistical Measures*. 2020. URL: `https://www.studysmarter.co.uk/explanations/math/statistics/statistical-measures/`.

[17] Mark Wallace et al. "On Benchmarking Constraint Logic Programming Platforms. Response to Fernandez and Hill's "A Comparative Study of Eight Constraint Programming Languages over the Boolean and Finite Domains"". In: *Constraints* 9 (2004), pp. 5–34.

# Appendix A

# Appendix

## A.1 Example JSON output from server


```
{
   "electricity_consumed_current":13.015040000000001,
   measurements":84141,
   "electricity_consumed_total":418145.88456979376,
   "power_draw":10.25
}
```


These variables have the following meaning:

- `electricity_consumed_current`: The electricity that was consumed between the last measurement and the one before. So the results of the measurements done right before this JSON was sent to output.
- `measurements`: The total number of measurements that have been done on the machine.
- `electricity_consumed_total`: The total amount of energy that has been consumed on the machine.
- `power_draw`: The current Wattage that the machine is using, so the amounts of Joules per second.