Energy-Efficient Computing: The Role of Memoization

Daniele Quartinieri
2767570
VU Amsterdam
d.quartinieri@student.vu.nl

Daniele Virzì
2859327
VU Amsterdam
d.virzi@student.vu.nl

Ivy Rui Wang
2801549
VU Amsterdam
r.wang3@student.vu.nl

Delong Yuan
2813751
VU Amsterdam
d.yuan@student.vu.nl

Qinhuihao Zeng
2815959
VU Amsterdam
q.zeng2@student.vu.nl

ABSTRACT

Goal. Analyze impact of memoization techniques on performance and energy efficiency of common python algorithms. Answering: RQ1) What is the difference in energy consumption between memoization and non memoization techniques when applied to various types of python functions? RQ2) How does the energy consumption of different memoization techniques relate to their performance(CPU utilization, execution time, memory usage) when applied to Python functions?

Method. In order to do so we have chosen 15 pure and common python functions in the field of CS. The functions have been used as a benchmark, each in 3 different ways, using: only the algorithm with no decorator and hence no caching applied, cached by key value and cached by Least Recently Used (LRU) key value. All caching has been obtained by using python functools cache decorators. In order to not contaminate the data collected by our benchmark with computations not related to benchmark, the code has been runned trough ssh on a laptop from a raspberry.

Results. For RQ1 where we checked the presence of difference in energy consumption between the memoization and non-memorization techniques, we have found no significant results form tests. For RQ2 where we checked relation between memoization techniques and performance metrics we have noticed a relation between energy consumption, cpu usage and execution time, but nothing statistically robust.

Conclusions. Given the background study that has been done we would have expected different results, with memoization techniques being much more efficient than standard function. Plenty of factors could have affected this study, ranging from the limited amount of data collected, to the lenght of tests used to more subtle factors such as the hardware on which has been run.

ACM Reference Format:
Daniele Quartinieri, Daniele Virzì, Ivy Rui Wang, Delong Yuan, and Qinhuihao Zeng. 2024. Energy-Efficient Computing: The Role of Memoization. In *Green Lab 2024/2025 - Vrije Universiteit Amsterdam, September–October, 2024, Amsterdam (The Netherlands).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Green Lab 2024/2025, September–October, 2020, Amsterdam, The Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the rapid growth of cloud computing and the exponential increase of mobile devices, optimizing software to reduce energy consumption has become essential. As mobile devices become more powerful, their energy demands grow, leading to faster battery depletion and reduced device lifetimes. Similarly, large-scale data centers, which power cloud computing, can consume vast energy. For instance, the forecasted electricity usage of data centers in 2030 is estimated at around 8000 TWh [1]. Energy-efficient computing is a part of *Green IT*, a practice that aims to reduce the energy footprint of software programs [2].

In this experiment, we focus on optimizing energy consumption through *Memoization*, a widely used *Dynamic Programming* technique to improve the computational efficiency of programs [3]. Memoization can significantly decrease algorithm execution time by saving the results of expensive function calls in a cache and reusing them when the same inputs are used again. This technique works only for pure functions with overlapping sub-problems, where the same calculations are performed multiple times (see Listing 1).

To our knowledge, almost no papers in the literature empirically explore the relationship between memoization and energy consumption. Indeed, while there are many studies about the effects of memoization on execution time, its impact on energy consumption has not been deeply examined. For this reason, the main goal of our experiment is to evaluate how memoization techniques affect energy usage. Accordingly, in this experiment, we compare the energy usage of a set of pure functions to their memoized implementation. To achieve this goal, we perform a controlled experiment involving different types of memoization techniques on several well-known algorithms in CS with different I/O and complexity, such as PCA, DFT, and Dijkstra's algorithm. Indeed, we generate these functions with AI for robustness and ensure the correctness of the code through a suite of tests written by us accompanying each function.

For the experiments, we use Python, since it is the most versatile and used programming language for TIOBE Index (#1 in Sep 2024 [4]). To implement memoization we use 2 different types of decorators from the Python *functools* module [5], *cache* and *Least-Recently-Used cache* (LRU) [6]. We utilize *Experiment Runner* [7] a framework designed to automatically execute measurement-based

experiments on any platform, to compute the metrics. This framework lets us measure execution time, energy consumption, CPU, and memory utilization.

The experiment's possible challenges are reliably assessing energy use and isolating the effects of memoization from other aspects influencing our evaluation, such as hardware variability, background system processes, and fluctuating workloads. Despite these challenges, the experiment opens many possibilities to developers. Understanding the trade-offs between computational efficiency and energy consumption enables developers to improve their code for both performance and energy efficiency. The outcomes of this study helps developers understand when memoization is beneficial for energy and performance and how to implement it to get a net quality optimization.

Listing 1: Memoized Fibonacci Function in Python

```python
def memoize(f: callable) -> callable:
    memo = {}

    def wrapper(x):
        if x not in memo:
            memo[x] = f(x)
        return memo[x]

    return wrapper

@memoize
def fibonacci(n: int) -> int:
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

2 RELATED WORK

Various related works have explored the performance improvement memoization could offer, but there has been limited focus on its potential to improve energy efficiency. This section reviews several studies focused on performance and energy efficiency and compares their objectives and experiments with our own.

Many researchers have focused their efforts on studying automation frameworks. They sought to identify pure functions that do not depend on external state and produce no side effects.

Loïc Besnard et al.[8] proposed an automated approach to memoization that shares the goal with our project. They focus on C/C++ using source-to-source compilation, while our project centers on Python. However, they inspire a similar idea for our Python environment, potentially using decorators or metaprogramming techniques to enhance function identification and memoization application. Specifically, they optimized C/C++ applications by automating memoization with Clava compiler, identifying memoizable functions, and inserting wrappers. Using LARA DSL, users specify targeted memorized functions; the compiler automatically generates and links a library containing the memoization logic. Hashtable allows caching the results. For float computations, certain unimportant input bits will be discarded. According to four benchmarks, larger cache tables show better energy efficiency; the not-updating-the-cache policy was more efficient than updating.

Marco Bessi et al. [9] employ memoization for energy efficiency but focus on Java bytecode and financial functions. Inspiration from this work is to apply the static analysis to identify pure functions that might be adapted for our Python project, using modules like AST (Abstract Syntax Tree) to strip out non-functional aspects of the code, such as comments and docstrings, to improve memoization. Specifically, they propose an energy optimization framework using memoization without altering code. The GreMe framework identifies pure functions, constructs a cache table for each, and caches results. Operating on Java bytecode, GreMe statically analyzes code for deterministic functions with no side effects. Tested on frequently used financial functions, it achieved up to 30% energy reduction.

Agosta et al.[10] developed a trade-off module for dynamic memory allocation based on function call frequency and input parameter distribution. They inspired that in our Python project, we could improve the efficiency of Python's memoization by adjusting cache sizes based on runtime data. Their strategy adjusts memory allocations for functions based on their call frequency and cache hit rates, reallocating memory from less efficient to more efficient functions to boost overall energy efficiency. They tested this approach with four computationally intensive financial functions, employing Gaussian-distributed random inputs to mimic real-world conditions. The use of memoization led to significant reductions in energy consumption (up to 99.9%) and execution time (up to 98.8%).

Rui Rua et al.[11] and Adriano Pinto et al.[12] both evaluated the performance of memoization across various functions in Android applications. Inspired by their experiment, in our Python project, we can use function-specific profiling to determine where memoization reduces energy consumption and where it introduces overhead, adapting caching strategies based on function complexity. Detailedly, they utilized a HashMap to cache function results and selected 18 Android functions from applications such as Pixate Freestyle, android-demos, and Chanu for testing. These functions covered common application scenarios such as graphics processing, string manipulation, and file handling. They tested these functions up to 50 times, measuring energy consumption with an energy analysis tool. Memoization reduced energy usage by up to 90% in most cases, though in complex functions, it increased due to memory management overhead.

Georgios Tziantzioulis et al.[13] introduced TAF-Memo, an output-based approximate memoization method that allows output quality trade-offs for reduced runtime and energy use. In our Python project, an output-based memoization can be used to tolerate minor inaccuracies, enhancing performance in compute-intensive applications. TAF-Memo deviates from traditional input-based memoization by permitting a margin of error in outputs to boost efficiency. It caches function outputs within a "memoization window," using these approximated results for subsequent calls if they fall within a defined error threshold. This strategy has proven effective across various domains, achieving up to 75% faster execution and quadrupling energy efficiency. Particularly, TAF-Memo excels in scenarios with gradual output changes, marking significant gains in areas requiring frequent, intensive computation cycles.

Here we further compare some methodologies with our own experiment setup, objectives, and measurement techniques. Loïc Besnard et al.[8] proposed a memoization framework using a compilation approach. Our experiment evaluates different memoization strategies (e.g., DFT, DFT_cache, DFT_lru_cache) to see their impact

on energy consumption and performance, using Python decorators to implement memoization. Marco Bessi et al.[9] improved energy efficiency through memoization without altering original application code, similar to our experiment that preserves function logic and uses the caching mechanism for performance improvement. What's different is that their GreMe framework operates on Java bytecode and uses static analysis to identify pure functions, while our experiment works directly on Python source code. Agosta's[10] dynamic memory allocation strategy based on runtime data, like call frequency and input distribution, is running on a desktop environment, while our experiment evaluates the caching strategies in terms of energy consumption and performance using remote profiling on the Raspberry Pi, rather than a desktop environment, and primarily focuses on Python's dynamic and interpreted nature. Rui Rua et al.[11] and Adriano Pinto et al.[12] selected 18 Andriod functions to measure memoization's impact which is similar to our experiment of profiling different benchmarks. Georgios Tziantzioulis et al.[13] introduced approximate memoization, whereas we focus on exact and complete memoization strategies rather than approximate methods. Our experiment measures exact energy usage to provide deterministic performance evaluations.

In summary, the papers reviewed provide various approaches to implementing memoization, each offering valuable insights for our Python project. Automated memoization techniques, such as those in C/C++ and Java bytecode, can be adapted to Python through decorators, metaprogramming, or static analysis to identify pure functions. Adjusting cache sizes and memory allocation dynamically could further optimize performance based on runtime data and function behavior. Profiling techniques suggest the importance of tailoring caching strategies to specific function complexity to avoid unnecessary overhead. Lastly, approximate memoization opens the possibility of trading output precision for performance, which can be applied in Python for compute-heavy tasks where slight inaccuracies are acceptable.

3 EXPERIMENT DEFINITION

We use the **Goal-Question-Metric (GQM) framework** [14] to systematically structure the research and define the experiment as shown in Fig 1.

3.1 Goal

Memoization techniques can improve program performance in tasks with repeated calculations, such as the Fibonacci sequence, where memoization can be used to reduce execution time from exponential complexity to linear complexity. However, it can also introduce additional energy consumption and system memory/-cache overhead. Hence, the **GOAL** of our experiment is: Analyze the impact of the memoization technique for the purpose of evaluating the impact on performance and energy efficiency from the point of view of Software developers in the context of Python applications(see Table1).

3.2 Research Questions

From the goal, we derived the following research questions:

RQ1 : To what extent does memoization impact the energy consumption of Python functions?

Analyze	Impact of memoization
for the purpose of	Evaluation
with respect to	Performance and Energy Efficiency
from the point of view of	Software developers
in the context of	Python applications

Table 1: Goal description following the GQM framework.

RQ2 : How does the energy consumption of different memoization techniques relate to their performance(CPU utilization, execution time, memory usage) when applied to Python functions?

RQ1 focuses on the differences in energy consumption when applying different memoization techniques(no memoization, @cache, and @lru_cache) across different types of Python functions. In CPU-intensive and recursion-intensive tasks, memoization can significantly reduce the computational load from repeated recursive calls, which in turn substantially decreases CPU runtime and energy consumption. In contrast, for memory-intensive tasks, where performance bottlenecks primarily stem from high memory usage and management, memoization may not have a significant positive impact on energy consumption.

RQ2 explores the relationship between energy consumption and performance when applying different memoization techniques. While memoization can enhance the performance of a program, particularly in reducing execution time and improving CPU utilization, it may also introduce additional resource overheads, such as increased memory usage. This question aims to analyze how energy consumption correlates with various performance metrics—memory usage, execution time, and CPU utilization.

3.3 Metrics

To address the research questions, we will test different types of Python functions (recursive, CPU-intensive, and I/O-intensive) by comparing the performance and energy efficiency with memoization before and after. Throughout these tests, we will gather data concerning both performance and energy efficiency. The following metrics will be used:

Energy Efficiency - The energy consumed by the function during execution will be measured in joules (J).

CPU Utilization - The average CPU usage during function execution, recorded as a percentage(%).

Execution Time - The execution time of each function will be measured in seconds (s).

Memory Usage - The average memory consumption during the execution of each function in bytes(B).

To answer RQ1, we use the energy consumption metric to analyze the impact of different memoization techniques on the energy efficiency of various types of Python functions. For RQ2, we utilize memory usage, CPU utilization, execution time, and energy consumption to examine whether performance improvements such as faster execution come at the cost of higher energy consumption.

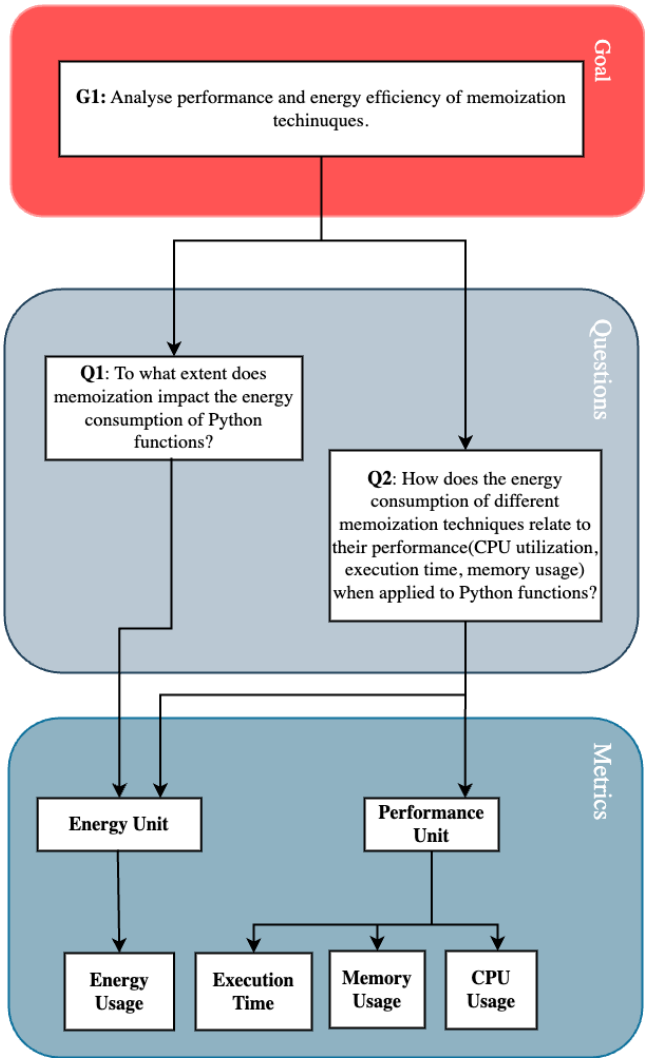

Figure 1: Complete visualization of the GQM

4 EXPERIMENT PLANNING

4.1 Subjects Selection

We focus on a specific class of functions, which "consistently produce the same output for a given set of inputs" and "do not produce external state changes or side effects." These types of functions follow strict rules of determinism, making them ideal subjects for algorithmic analysis and optimization. Pure functions are particularly useful for energy assessment because their predictable behavior eliminates variability caused by side effects, leading to more consistent energy measurements. As demonstrated in the study [15][16], pure functions have been used to evaluate both performance and energy efficiency in algorithmic testing.

Recursive functions solve problems through self-referential calls. They are suitable for testing how memoization optimizes recursive call stacks and reduces redundant calculations. In tasks with deep recursion and frequently repeated calculations, memoization can significantly improve efficiency by caching previous results [17].

Non-recursive functions usually rely on iterative or linear calculations. For these functions, memoization mainly improves performance by caching intermediate results, such as subproblem solutions in dynamic programming.

We chose not to rely on external benchmarks, like CLBG[18] and Rodinia[19], because benchmark functions often include hardware-specific which can conflict with caching strategies, making it challenging to integrate memoization directly[20]. Many benchmark functions are structured to perform all computations in a single execution loop[18], which limits the ability to apply memoization effectively.

Also, avoid using external library functions because some of them may invoke other dependencies or complex operations, functioning as a black box, which could make it harder to isolate the pure effects of memorization.

Table 2: Classification of 15 Selected Functions

Task Type	Function Name and Description
Recursive	fibonacci.py: Computes **Fibonacci** numbers.
	hanoi.py: Solves the **Tower of Hanoi** problem.
	edit.py: Compute **Edit distance** between two strings.
	reverse.py: **Reverses** a list or string.
	uniquepaths.py: Computes **unique paths** in a grid using recursion.
	knapsack.py: Solves the **knapsack problem** using dynamic programming.
	mergesort.py: Implements the **Merge Sort** algorithm.
Non-Recursive	dft.py: Performs Discrete Fourier Transform (**DFT**).
	hessian.py: Implements **Hessian** computation.
	lemmatization.py: Performs **text lemmatization**.
	pca.py: Performs Principal Component Analysis (**PCA**).
	convolve.py: Performs **convolution** operation.
	dijkstra.py: Solves the shortest path problem using **Dijkstra's algorithm**.
	floyd.py: Solves all-pairs shortest paths with **Floyd-Warshall algorithm**.
	permutation.py: **Generates permutations** of a set of elements.

4.2 Experimental Variables

The implementation of memoization techniques is the independent variable in this experiment, which includes no memoization, implementation using @cache, and implementation using @lru_cache.

The @cache[21] decorator introduced in Python 3.9 is used to cache function results without any eviction policy, making it useful for functions that always return the same result for the same inputs. In contrast, @lru_cache[22] (Least Recently Used) decorator, introduced in Python 3.2, caches a limited number of function results and evicts the least recently used entries when the cache limit is reached. This makes @lru_cache more memory-efficient for applications where memory constraints are a concern. We select these decorators because they represent two common strategies for caching: one without limitations and one with a memory-aware strategy. Additionally, as part of Python's built-in functools module, they are widely adopted due to their ease of use and integration into various tasks like Web Applications and Machine Learning.

We select energy consumption per function execution, execution time (total time taken for function execution), CPU utilization (average CPU usage during execution), and memory usage (average memory consumption during execution) as the dependent variables.

For RQ1, we focus on how memoization techniques affect the energy efficiency of different Python functions. The main dependent variable in this case is the energy consumption of the hardware, which is measured in joules. For RQ2, we explore the relationship between power consumption and performance (including memory usage, execution time, and CPU utilization).

4.3 Experimental Hypotheses

Firstly, we check if the memorization has statistically significant effects on execution time to ensure the correctness of the experiment. In order to achieve this result we perform ANOVA test (Normality Assumptions) and Kruskal–Wallis test (No-Normality Assumption) to the execution time of the first calls and the second calls of each implementation (base, cache, lru_cache). The first calls should be equal in theory so we expect a high p-value, the second calls instead should be significantly different so we expect a p-value ≈ 0. After ensuring the correctness of the experiment we can provide results.

$$H_0: \quad \bar{x}_{\text{non-memoized}/1} = \bar{x}_{\text{cache}/1} = \bar{x}_{\text{lru\_cache}/1} \text{ (Joules)} \tag{1}$$

$$H_1: \quad \bar{x}_{\text{non-memoized}/1} \neq \bar{x}_{\text{cache}/1} \neq \bar{x}_{\text{lru\_cache}/1} \text{ (Joules)} \tag{2}$$

$$H_0: \quad \tilde{x}_{\text{non-memoized}/1} = \tilde{x}_{\text{cache}/1} = \tilde{x}_{\text{lru\_cache}/1} \text{ (Joules)} \tag{3}$$

$$H_1: \quad \tilde{x}_{\text{non-memoized}/1} \neq \tilde{x}_{\text{cache}/1} \neq \tilde{x}_{\text{lru\_cache}/1} \text{ (Joules)} \tag{4}$$

$$H_0: \quad \bar{x}_{\text{non-memoized}/2} = \bar{x}_{\text{cache}/2} = \bar{x}_{\text{lru\_cache}/2} \text{ (Joules)} \tag{5}$$

$$H_1: \quad \bar{x}_{\text{non-memoized}/2} \neq \bar{x}_{\text{cache}/2} \neq \bar{x}_{\text{lru\_cache}/2} \text{ (Joules)} \tag{6}$$

$$H_0: \quad \tilde{x}_{\text{non-memoized}/2} = \tilde{x}_{\text{cache}/2} = \tilde{x}_{\text{lru\_cache}/2} \text{ (Joules)} \tag{7}$$

$$H_1: \quad \tilde{x}_{\text{non-memoized}/2} \neq \tilde{x}_{\text{cache}/2} \neq \tilde{x}_{\text{lru\_cache}/2} \text{ (Joules)} \tag{8}$$

This experiment addresses two main concerns: energy consumption (RQ1) and the relationship between energy efficiency and performance (RQ2).

Since our subjects are functions with different complexity, we expect a non-normal distribution of the data so we cannot perform a two-sample T-test because we do not meet the assumptions. So, to address RQ1, we perform a one-tailed Mann-Whitney U test (or Wilcoxon rank-sum test) since this test does not assume that the data are distributed normally. Indeed, without making strong assumptions, we formulate the following hypothesis with the respective test statistic:

$$H_0: \quad \tilde{x}_{\text{memoized}} = \tilde{x}_{\text{non-memoized}} \text{ (Joules)} \tag{9}$$

$$H_1: \quad \tilde{x}_{\text{memoized}} < \tilde{x}_{\text{non-memoized}} \text{ (Joules)} \tag{10}$$

$$U = n_1 n_2 + \frac{n_1(n_1+1)}{2} - R_1 \tag{11}$$

where:

n_1 = number of observations in group 1
n_2 = number of observations in group 2
R_1 = sum of ranks for group 1

The null hypothesis (9) states that the median energy consumption of the memoized functions is equal to the median energy consumption of not memoized ones. The alternative hypothesis (10), instead states that the median of the memoized functions is smaller than the median of their base implementations. With RQ1 we want to measure the impact of memoization on energy consumption. We accomplish this by analyzing the p-value of this test. A high p-value results in the failure to reject the null hypothesis. In contrast, a small p-value results in a significant impact of memoization on energy consumption; the smaller the p-value the higher the energy efficiency of memoization. Since we use 2 different methods to implement memoization and 3 different input sizes, we perform 6 different tests. In this way, the results are more robust and the hypothesis tests are:

$$H_0: \quad \tilde{x}_{\text{cache/small}} = \tilde{x}_{\text{non-memoized/small}} \text{ (Joules)} \tag{12}$$

$$H_1: \quad \tilde{x}_{\text{cache/small}} < \tilde{x}_{\text{non-memoized/small}} \text{ (Joules)} \tag{13}$$

$$H_0: \quad \tilde{x}_{\text{lru\_cache/small}} = \tilde{x}_{\text{non-memoized/small}} \text{ (Joules)} \tag{14}$$

$$H_1: \quad \tilde{x}_{\text{lru\_cache/small}} < \tilde{x}_{\text{non-memoized/small}} \text{ (Joules)} \tag{15}$$

$$H_0: \quad \tilde{x}_{\text{cache/medium}} = \tilde{x}_{\text{non-memoized/medium}} \text{ (Joules)} \tag{16}$$

$$H_1: \quad \tilde{x}_{\text{cache/medium}} < \tilde{x}_{\text{non-memoized/medium}} \text{ (Joules)} \tag{17}$$

$$H_0: \quad \tilde{x}_{\text{lru\_cache/medium}} = \tilde{x}_{\text{non-memoized/medium}} \text{ (Joules)} \tag{18}$$

$$H_1: \quad \tilde{x}_{\text{lru\_cache/medium}} < \tilde{x}_{\text{non-memoized/medium}} \text{ (Joules)} \tag{19}$$

$$H_0: \quad \tilde{x}_{\text{cache/large}} = \tilde{x}_{\text{non-memoized/big}} \text{ (Joules)} \tag{20}$$

$$H_1: \quad \tilde{x}_{\text{cache/large}} < \tilde{x}_{\text{non-memoized/big}} \text{ (Joules)} \tag{21}$$

$$H_0: \quad \tilde{x}_{\text{lru\_cache/large}} = \tilde{x}_{\text{non-memoized/large}} \text{ (Joules)} \tag{22}$$

$$H_1: \quad \tilde{x}_{\text{lru\_cache/large}} < \tilde{x}_{\text{non-memoized/large}} \text{ (Joules)} \tag{23}$$

If the results are not statistically significant, we perform the same type of test isolating each function, to understand locally which function shows a significant difference in energy consumption due to the different implementation. So we perform the following test for each function:

$$H_0: \quad \tilde{x}_{\text{cache/function\_name}} = \tilde{x}_{\text{non-memoized/function\_name}} \text{ (Joules)} \tag{24}$$

$$H_1: \quad \tilde{x}_{\text{cache/function\_name}} < \tilde{x}_{\text{non-memoized/function\_name}} \text{ (Joules)} \tag{25}$$

$$H_0: \quad \tilde{x}_{\text{lru\_cache/function\_name}} = \tilde{x}_{\text{non-memoized/function\_name}} \text{ (Joules)} \tag{26}$$

$$H_1: \quad \tilde{x}_{\text{lru\_cache/function\_name}} < \tilde{x}_{\text{non-memoized/function\_name}} \text{ (Joules)} \tag{27}$$

RQ2 is about the relationship between energy efficiency and performance, so our task is to find a pattern between the measurements of the memoized function and their base implementation. To accomplish this, we plot scatterplots between each variable and compute the Pearson Correlation Coefficient (28) to find a linear trend. If there is no relevant linear relation we perform the Spearman Rank Correlation (29) checking for complex non-linear patterns.

$$r = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \sum(y_i - \bar{y})^2}} \tag{28}$$

- x_i: Individual value of variable x
- y_i: Individual value of variable y
- $\bar{x}$: Mean of variable x
- $\bar{y}$: Mean of variable y
- $\mathbf{x}$: Vector of values for variable x
- $\mathbf{y}$: Vector of values for variable y

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{29}$$

- d_i: Difference between the ranks of corresponding values
- n: Number of observations

For all the statistical tests we perform, we use a significance level $\alpha = 0.05$ since, for decades, has been conventionally accepted as the threshold to discriminate significantly from non-significant results [23].

4.4 Experiment Design

This experiment adopts a multi-factor and multi-processing design and investigates the interaction effect between the memoization strategy and different types of functions. The factors and treatments of the experiment include:

- Factor 1: memoization strategy (from **functools** [24])
  - no-cache
  - normal cache (without old value eviction)
  - LRU cache
- Factor 2: function type
  - CPU-intensive
  - memory-intensive
  - recursive
- Input size
  - small
  - medium
  - large

This experiment is designed with multiple factors multiple treatments, and all treatments of each factor will be tested in combination. The specific experimental combination is as follows: 3 memoization strategies × 3 input scales × 3 type of functions (5 functions in each class) = 135 configurations. Then, run each configuration to get the result.

4.5 Data Analysis

In this section, we focus on the techniques we use to perform the data analysis.

Data Preprocessing: Before, starting the EDA we merge the output of Experiment Runner in one *master.csv* to group among all the functions in the data analysis phase. So we concatenate all the run tables and add columns that are useful for our tests. The columns we add are Function_Name with the name of the algorithms (i.e. DFT, pca, etc...), Implementation with the type of implementation (base, cache, lru_ cache), and Category that show if a function is recursive or not.

Data Exploration: Initially, we compute the summary statistics of the variables to check their distribution and range to see if we should apply data transformation techniques like Log-Transformation. Then we use boxplots and KDE plots grouping by Implementation and Function_Name to see the distribution of the data that we have to test. We look for normality to understand the right test to choose because T-test and ANOVA test assume normality. In our case, it is likely to not have normality since our subjects are functions with different complexity. For this reason, by just looking at the distributions we can robustly say if they are normal or not, without performing any diagnostic test like the Shapiro-Wilk test.

Since our analysis focuses deeply on the correlation between the variables, the goal is to find some pattern using plots like corrplot and scatterplots.

5 EXPERIMENT EXECUTION

5.1 Setup

5.1.1 *Hardware Setup.*

- **Control Unit:** A Raspberry Pi 4 running Ubuntu Server 20.04.5 LTS is used to execute the Experiment Runner framework. The Raspberry Pi is responsible for initiating and controlling the benchmarks remotely via SSH. This setup minimizes the computational interference of the Experiment Runner on the energy measurements taken during the experiments.
- **Computing Platform:** The benchmarks are executed on a MacBook Pro (16-inch, 2019 model) with a 2.4 GHz 8-core

Table 3: Measurements and Their Definitions

Measurement	Unit	Description
Execution_Time_1	Second (s)	Execution time for the first function call.
Execution_Time_2	Second (s)	Execution time for the second function call.
Execution_Time	Second (s)	Total execution time elapsed from the beginning of the first function execution to the end of second function execution within a single benchmark run.
Memory_Usage	Megabyte (MB)	Memory used during a single benchmark run.
Average_CPU_Usage	Percentage (%)	Average CPU usage across all CPU cores during a single benchmark run.
Energy_Consumption	Joule (J)	Energy consumption during a single benchmark run.

Intel Core i9 processor. The laptop runs macOS and hosts the benchmark functions, as well as Energibridge, a profiling tool that records energy consumption data.

5.1.2 *Software Setup.*

- **Experiment Runner Framework:** The Experiment Runner, running on the Raspberry Pi, automates the execution of benchmarks. It ensures that each benchmark is run with consistent parameters, including different input sizes and caching strategies. The Experiment Runner's control is facilitated through SSH, ensuring a seamless interaction with the MacBook Pro.
- **Energibridge:** Energibridge is installed on the MacBook Pro and is used to measure the energy consumption during each benchmark run. This tool provides a detailed breakdown of energy usage, which is crucial for analyzing the efficiency of different implementations. The data generated by Energibridge is stored locally on the MacBook Pro. The Raspberry Pi 4 runs the Experiment Runner framework, which remotely initiates and controls the benchmarks on the MacBook Pro. This setup ensures that the measurements remain unaffected by additional power consumption or CPU loads potentially introduced by the Experiment Runner itself. We take this precaution to account for any variability in the framework's resource usage across different benchmarks. However, no such adjustment is made for the Energibridge profiler, as prior studies have shown that its energy consumption remains low and consistent, ensuring minimal influence on comparative analyses[25].The sampling rate is set to 200 milliseconds, which means that Energibridge captures the power usage at intervals of 200 milliseconds throughout the duration of the benchmark execution, providing a detailed measurement of energy usage over time. The max execution is set to 20 seconds, defining the maximum duration for which the command is executed and energy data is collected. This limit is set to ensure efficient experiment execution, given the multiplication effect of having 15 problems combined with various factors such as input sizes and caching strategies.

5.2 Preparation

5.2.1 Benchmark Problems. The experiments focus on two categories of problems: **Non-Recursive** and **Recursive**, with a total of 15 problems. Each problem is implemented using three different strategies: a basic implementation without any caching mechanism,

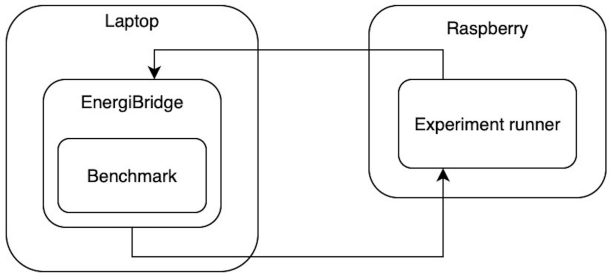

Figure 2: Scheme of hardware and software setup

and two implementations that leverage caching through Python decorators—@cache and @lru_cache. These variations allow for a detailed performance analysis, comparing the impact of different caching strategies on computational efficiency and energy consumption.

5.2.2 *Experiment Workflow.*

(1) **Setup of the Raspberry Pi:** The Raspberry Pi is set up using the official Raspberry Pi Imager tool, with Ubuntu Server configured as the operating system. After flashing, the network settings in the network-config file are adjusted to have stable WiFi connectivity.
(2) **SSH Configuration and GitHub Access:** An SSH key pair is generated without a passphrase to ensure that the Raspberry Pi can run the Experiment Runner and EnergiBridge without interruption. This setup prevents the need for manual passphrase entry, which could disrupt the process of EnergiBridge writing energy consumption data to the CSV files during the experiments.
(3) **Execution of Benchmarks:** The Experiment Runner, running on the Raspberry Pi, is programmed to initiate the benchmarks by writing custom code within the framework to remotely call the benchmark functions and execute the EnergiBridge command. This code is then executed remotely on the MacBook Pro via SSH. This setup allows the benchmarks to run on the MacBook Pro while the Experiment Runner on Raspberry Pi manages the orchestration, ensuring that EnergiBridge accurately measures the energy consumption during the benchmark executions.
(4) **Data Collection and Transfer:** During each benchmark run, EnergiBridge generates energy consumption data, which

is stored on the MacBook Pro. After each run, this data is copied back to the Raspberry Pi using SSH. The Experiment Runner on the Raspberry Pi manages the data transfer process, ensuring that each set of energy measurements is retrieved correctly. Once all benchmark runs are completed, the Experiment Runner processes the collected data and computes the final run table, which includes a summary of energy consumption for each benchmark problem.

This setup, as shown in Figure 2, effectively separates the control process from the benchmark execution. The Raspberry Pi manages the experiment execution, while the MacBook Pro focuses on running the benchmarks and recording energy data, ensuring that the measurements remain unaffected by the control overhead. This design enables a clear and accurate evaluation of the energy efficiency of different algorithms across various input sizes and caching strategies.

5.3 Measurements

We measured various performance metrics including *execution time*, *average CPU usage*, *memory usage*, and *energy consumption*. These metrics were recorded for each benchmark run, comparing computational performance and energy consumption between the basic and memorization versions.

Throughout the process, the Raspberry Pi initiates and manages the remote executions via SSH, while Energibridge performs real-time monitoring of the benchmarks running on the laptop. After each run, the collected data, including execution times, CPU usage, memory consumption, and energy metrics, is transferred back to the Raspberry Pi. Once all benchmark runs have been completed, the Experiment Runner processes this data to generate a comprehensive results table, `run_table.csv`, which aggregates the performance data across various runs.

Execution Time. Execution time is recorded as three metrics: *execution_time_1*, *execution_time_2*, *execution_time*.

The *execution_time_1*, *execution_time_2* represent the duration of the first and second function call, respectively. For instance, the DFT_cache function of the target benchmark DFT is consecutively executed twice on the remote laptop. For each execution, the start time is recorded using `time.perf_counter()`, and the end time is captured immediately after the function completes. The total *execution_time* is the cumulative time from the beginning of the first execution to the end of the second one.

The reason for executing the function twice is to capture the effect of caching mechanisms such as `@cache` and `@lru_cache`. These decorators cache the input during the first execution to optimize subsequent calls. By measuring the second execution, we can observe how the caching mechanism reduces the function's runtime. To ensure a fair comparison with the basic version, which does not use caching, we also execute it twice, even though there is no caching involved. This approach allows us to directly compare the performance of the basic implementation with the cached versions.

Average CPU Usage. During the benchmark run, Energibridge collects detailed data on the CPU usage of the laptop executing the functions. Once the data is transferred back to the Raspberry Pi, the `populate_run_data` method in Experiment Runner processes the collected data (`energibridge.csv` files), computing the mean CPU usage across all available cores throughout the run. This value reflects the computational load experienced during the function's execution, providing a measure of how intensively the CPU resources were utilized.

Memory Usage. Memory usage is another key metric tracked by Energibridge during each benchmark run. The data gathered is processed by the Raspberry Pi after the run, where the `populate_run_data` method reads the recorded memory usage values from the `energibridge.csv` file. It calculates the average memory usage, which is then converted into megabytes (MB) for easier interpretation.

Energy Consumption. During each run, Energibridge continuously monitors the energy usage of the system and logs the power consumption data. The recorded metric, *energy_consumption*, represents the total energy consumed by the system throughout the duration of the benchmark execution.

———————————————

Link to timelog click here

———————————————

6 RESULTS

To answer RQ2 we should find some pattern and relationship between the variables, so our analysis is focused on data visualization instead of measures of centrality that do not help us to achieve our goal.

6.1 Exploratoy Data Analysis (EDA)

We start our analysis by looking at the boxplots of our variables grouped by `Implementations` (figure 3):

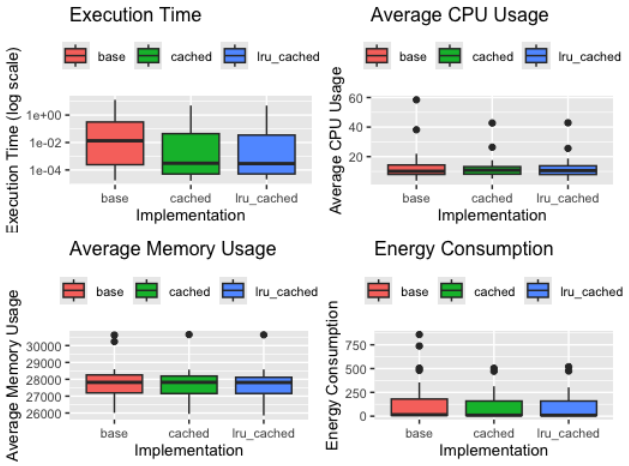

Figure 3: Boxplots grouped by implementation

As we can see there is a difference in total `Execution_time` but not in `Energy_Consumption` or `Average_CPU_Usage` and `Memory_Usage`, so we focus more on energy consumption grouping by `Function_Name` (figure 4):

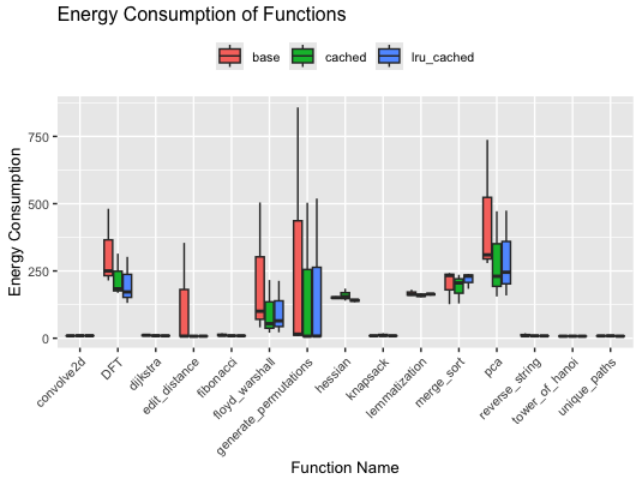

Figure 4: Boxplot of Energy Consumption grouped by Function Name

From this plot, it looks like there is a slight difference in Energy_Consumption between the Implementation, but it is noticeable only for some functions and this could be related to the complexity of the function or a problem in memoization implementation. By checking the the effect of memoization on Execution_time, we can ensure that is not a problem of implementation (figure 5).

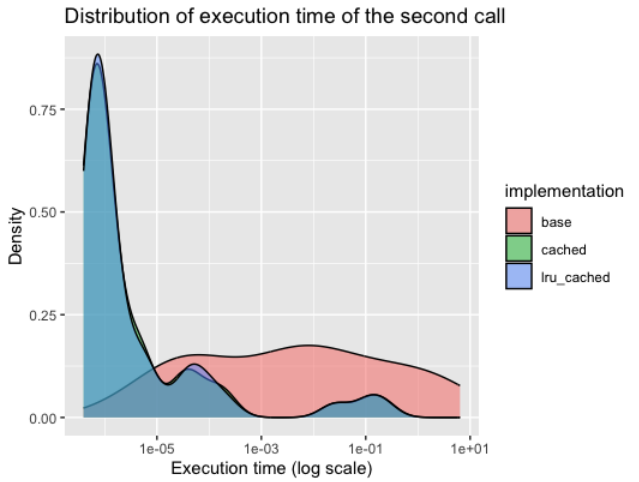

Figure 5: KDE of the Execution time of the second calls of the functions

Then we analyze if there is a statistical correlation between Energy_Consumption and Execution_time to answer RQ2. As we can see in the plot of the distributions grouped by Category it seems there is a pattern but it is not enough (figure 6).

To assess this correlation we plot the scatterplots and the correlation matrix (figure 7 & 8).

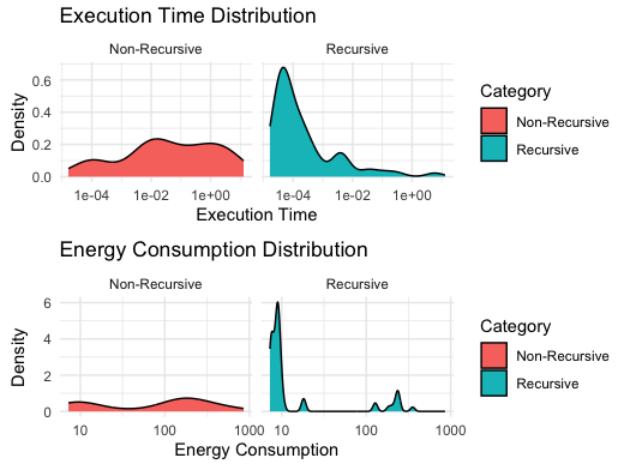

Figure 6: KDE of Execution time and Energy consumption grouped by category

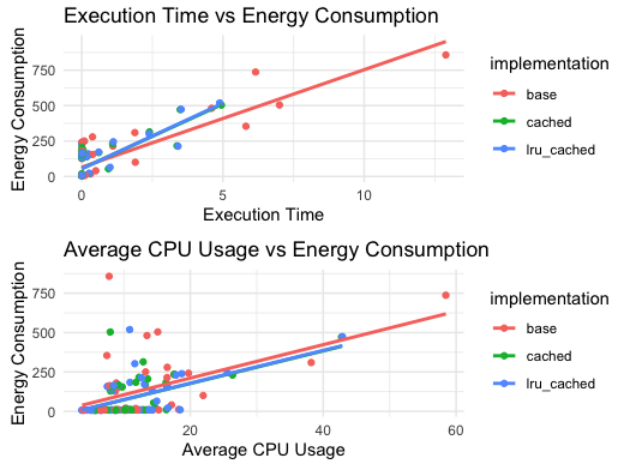

Figure 7: Scatterplot of Execution Time vs Energy Consumption and Average CPU Usage vs Energy Consumption

By looking at the scatterplot, there is a clear pattern that shows a positive correlation between the Execution_time and the Average_CPU_Usage vs Energy_Consumption and this is confirmed by the corrplot that returns a Pearson correlation $\rho_1 = 0.83$ (Execution_time vs Energy_Consumption), $\rho_2 = 0.52$ (Average_CPU_Usage vs Energy_Consumption) and $\rho_3 = 0.33$ (Execution_time vs Average_CPU_Usage). Finally, we plot a scatterplot 3D to explore the relationship between these 3 variables grouping by implementation (figure 9).

As we can see in the plot we cannot prove any relationship between the Memory_usage and memoization, because the outputs cached by the memoized function were not that big to influence the memory usage significantly (figure 10) .

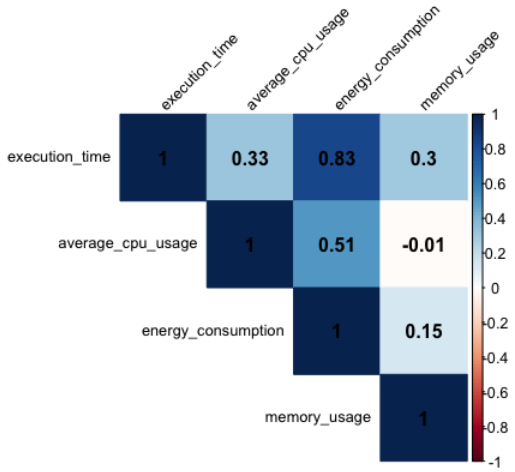

Figure 8: Corrplot

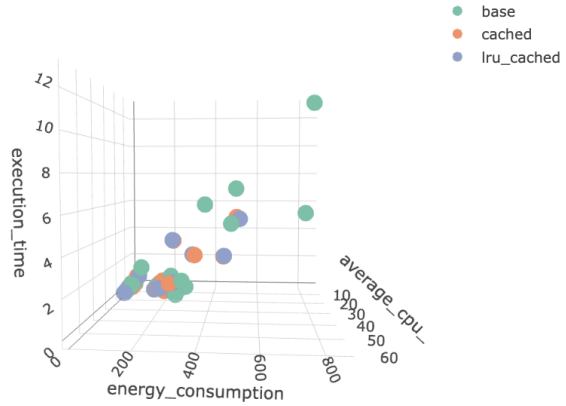

Figure 9: Execution Time vs Energy Consumption vs Average CPU usage

6.2 Hypothesis Testing

As cited in subsection 4.3, in order to ensure the correctness of the experiment we perform ANOVA and Kruskal–Wallis test to the execution time of the first calls and the second calls of each implementation (base, cache, lru_cache). These are the results of both test:

p-value	**first calls**	**second calls**
ANOVA	0.8845	0.002128
Kruskal–Wallis	0.2102	4.604e-14

Table 4: Results of the test on Execution Time.

The p-value of the test on the first calls is high, which means that there is no significant difference in execution time between the

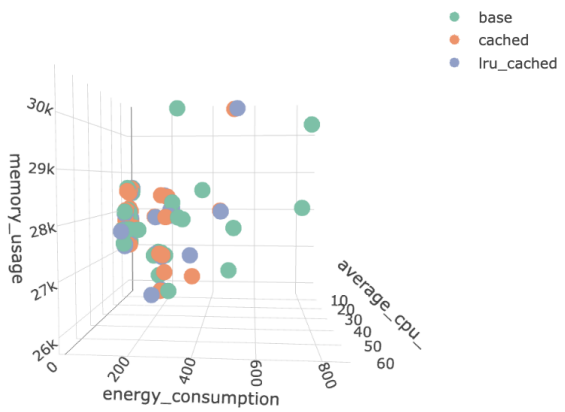

Figure 10: Memory Usage vs Energy Consumption vs Average CPU usage

3 implementations and this is correct because memoization effects work on the second calls of the memorized functions. The results of the test on the second calls instead, with a p-value near 0, show that there is a significant difference in execution time so we can robustly conclude that our memoization implementation works as it should. Once ensured the correctness of our experiment we performed the hypothesis test to answer the RQ1 by comparing the non-memoized functions with their memoized versions grouping by the input size. Firstly we confirm that we cannot assume that the data are normally distributed by plotting their sample distribution (figure 11).

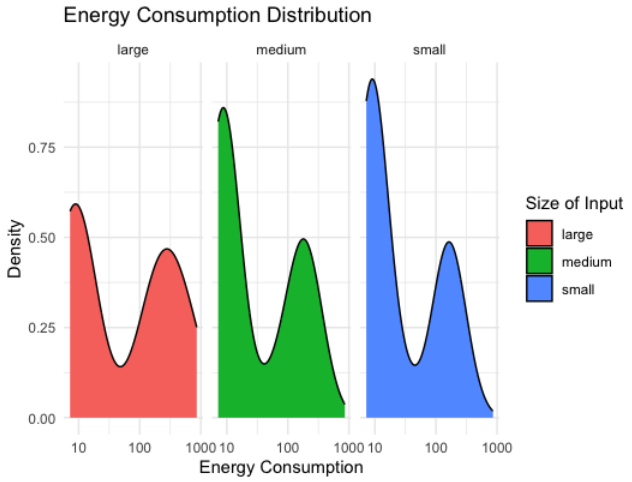

Figure 11: KDE of Energy Consumption grouped by Input Size

As we can see the data are not normally distributed, and we cannot use the one-tailed T-test. For this reason, we perform the one-tailed Mann-Whitney U test, and these are the results:

p-value	**base vs cache**	**base vs lru_cache**
small	0.3118	0.3565
medium	0.5809	0.4032
large	0.2062	0.1335

Table 5: Results of the test on Energy Consumption grouping by input size.

The tests show no evidence for an effect of memoization on energy consumption but these results could be biased because we are analyzing together different functions with different complexity. For this reason, we test the effect of memoization on each function stand-alone and provide the following results:

p-value	**base vs cache**	**base vs lru_cache**
DFT	0.2	0.2
merge_sort	0.5	0.5
pca	0.2	0.2
generate_permutation	0.5	0.5
reverse_string	0.2	***0.05***
unique_paths	0.8	0.35
dijkstra	0.35	0.2
edit_distance	0.2	0.35
fibonacci	***0.05***	***0.05***
floyd_warshall	0.35	0.35
tower_of_hanoi	0.8	0.5
hessian	0.65	***0.05***
knapsack	1	0.35
lemmatization	0.35	0.5
convolve2d	0.8	0.1

Table 6: Results of the test on Energy Consumption grouping by function.

As we can see from the table 6 some functions are significantly affected by memoization on energy consumption but these results are not statistically robust. We achieved evidence with a significance level of 5% on 4 out of 30 functions ($4/30 = 1.33 = 0.013\%$) so it is also possible that our results are Type 1 errors and so not statistically significant.

7 DISCUSSION

In this section, we discuss the results of our research, addressing the answers to our RQs.

RQ1 : To what extent does memoization impact the energy consumption of Python functions?

Answer : From our data, we do not have any statistically significant effect of memoization on energy consumption.

RQ2 : How does the energy consumption of different memoization techniques relate to their performance(CPU utilization, execution time, memory usage) when applied to Python functions?

Answer : From our data, we have discovered a positive correlation between execution time, energy consumption, and CPU usage.

We have proved that memoization has a statistical impact on execution time and that execution time has a positive correlation with energy consumption but, we cannot prove that memoization has a statistical impact on energy consumption. So, even though our experiment has not provided any evidence, it is still possible to prove that memoization affects energy consumption by improving how the experiment is conducted. Our experiments likely provided no result because of the lack of data and the complexity of the functions. Functions with higher execution time lead to higher energy consumption and so a bigger gap between the implementations. Moreover adding more data makes the hypothesis tests more effective and can lead to significant results.

8 THREATS TO VALIDITY

The following section describes our analysis of the four different threats to the experimental validity.

8.1 Internal Validity

- **Maturation**: Over time, system performance may fluctuate due to factors like hardware overheating or changing environmental conditions. We set a 1-minute interval between runs to prevent overheating and to avoid residual effects from previous computations. To further minimize interference, we executed only one benchmark at a time, running all benchmarks serially.
- **Selection**: All of the functions we selected were able to complete the experiment. So theoretically, we do not have a threat to validity in selection.
- **Reliability of Measures**: The accuracy of our measurements (e.g., energy consumption and execution time) could be affected by background processes or external factors such as system load. To mitigate these effects, we ensured that the system was in a stable state before each run. This included a stable power supply and ensuring no other processes were running.

8.2 External Validity

- **Interaction of Selection and Treatment**: The functions and inputs we selected for testing memoization strategies may not represent all possible real-world scenarios. We have selected fifteen functions from the most common functions with recursive and non-recursive types to cover a broad range of function types. However, the generalization of our results to all possible applications is limited and needs further discovery.
- **Interaction of Setting and Treatment**: Our experiments were executed on one of the most common laptops and OS. Results might differ if run on different machines, under various conditions (e.g., cloud servers or mobile devices). Thus, extending the experiment to different hardware setups in future research would help increase our findings' generalizability.

8.3 Construct Validity

- **Inadequate Preoperational Explication of Constructs**: We defined the research questions clearly before beginning the experiments. The independent variables (memoization strategies and function types) and the dependent variables (execution time, energy consumption, memory usage) were mapped to the research questions early in the planning phase, ensuring that they accurately reflect the memoisation aspects we will study.
- **Mono-operation Bias**: Our experiment focuses on three independent variables: memoization strategy, function type, and input size. Including these variables helps to provide a comprehensive picture.
- **Mono-method Bias**: We measured several metrics (execution time, energy consumption, memory usage) to reduce the risk of bias in the measurement process.

8.4 Conclusion Validity

- **Low Statistical Power**: To ensure our results are statistically significant, we ran each combination with different memoization strategies, function types, and input sizes. This large number of trials increases the credibility of the statistical analysis, making it more likely to detect the true effects of the memoization strategies on performance and energy consumption. However, some function types or memoization strategies might still yield insufficient data to detect significant differences.
- **Violated Assumptions of Statistical Tests**: We first examined the distribution of the data to ensure that we selected the appropriate statistical test. Initially, we assumed that the energy consumption data for the three cache strategies followed a normal distribution, and planned to use ANOVA. However, when we discovered that the data was not normally distributed, we applied the Kruskal-Wallis test for the three strategies. For pairwise comparisons between the baseline and memoization strategies, we performed Wilcoxon tests and drew our conclusions based on these results.

9 CONCLUSIONS

In this study, we have analyzed the impact of memoization techniques on the performance and energy consumption of Python functions. Our experiments showed that while memoization has a noticeable effect on reducing execution time, it did not demonstrate a statistically significant impact on energy consumption. We observed a clear correlation between execution time, CPU utilization, and energy consumption, with longer execution times resulting in higher energy usage, but no direct evidence that memoization affects energy consumption.

Looking forward, the experiment could be extended to investigate more complex functions and workloads, as well as alternative memoization strategies beyond the built-in Python decorators. Future teams could focus on expanding the dataset with more diverse functions, testing across different hardware platforms, or exploring hybrid caching mechanisms that combine memoization with hardware-level optimizations to further assess potential energy savings. This would provide a more comprehensive understanding of the trade-offs between performance improvements and energy efficiency in computing tasks.

REFERENCES

[1] A. Andrae and T. Edler, "On global electricity usage of communication technology: Trends to 2030," *Challenges*, vol. 6, pp. 117–157, 04 2015.

[2] E. M. Ben Lutkevich, "Green it definition." [Online]. Available: https://www.techtarget.com/searchcio/definition/green-IT-green-information-technology

[3] "Memoization - Wikipedia — en.wikipedia.org," https://en.wikipedia.org/wiki/Memoization, [Accessed 26-09-2024].

[4] "TIOBE Index - TIOBE — tiobe.com," https://www.tiobe.com/tiobe-index/python/, [Accessed 26-09-2024].

[5] "cpython/Lib/functools.py at 42336def77f53861284336b3335098a1b9b8cab2 · python/cpython — github.com," https://github.com/python/cpython/blob/42336def77f53861284336b3335098a1b9b8cab2/Lib/functools.py#L448, [Accessed 26-09-2024].

[6] C. Maiza, V. Touzeau, D. Monniaux, and J. Reineke, "Fast and exact analysis for lru caches," 2018.

[7] "GitHub - S2-group/experiment-runner: Tool for the automatic orchestration of experiments targeting software systems — github.com," https://github.com/S2-group/experiment-runner, [Accessed 13-09-2024].

[8] L. Besnard, P. Pinto, I. Lasri, J. Bispo, E. Rohou, and J. M. Cardoso, "A framework for automatic and parameterizable memoization," *SoftwareX*, vol. 10, p. 100322, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711018301559

[9] M. Bessi, "A methodology to improve the energy efficiency of software," 2014.

[10] G. Agosta, M. Bessi, E. Capra, and C. Francalanci, "Automatic memoization for energy efficiency in financial applications," *Sustainable Computing: Informatics and Systems*, vol. 2, no. 2, pp. 105–115, 2012, iEEE International Green Computing Conference (IGCC 2011). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210537912000066

[11] R. Rua, M. Couto, A. V. Pinto, J. Cunha, and J. de Sousa Saraiva, "Towards using memoization for saving energy in android," in *Conferencia Iberoamericana de Software Engineering*, 2019. [Online]. Available: https://api.semanticscholar.org/CorpusID:199373557

[12] A. Pinto, M. Couto, and J. Cunha, "Memoization for saving energy in android applications: When and how to di it," Submitted.

[13] G. Tziantzioulis, N. Hardavellas, and S. Campanoni, "Temporal approximate function memoization," *IEEE Micro*, vol. 38, no. 4, pp. 60–70, 2018.

[14] "GQM - Wikipedia — en.wikipedia.org," https://en.wikipedia.org/wiki/GQM, [Accessed 13-09-2024].

[15] D. T. N. E. D. B. Nicolas van Kempen, Hyuk-Je Kwon, "It's not easy being green: Exploring energy efficiency in computing," *arXiv preprint arXiv:2410.05460*, 2024. [Online]. Available: https://arxiv.org/abs/2410.05460

[16] J. Hughes, "How to specify it!: A guide to writing properties of pure functions," in *Lecture Notes in Computer Science, including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*, vol. 12053. Springer, 2020, pp. 58–83.

[17] O. A. Gbadamosi, A. E. Okeyinka, and I. Abdullahi, "Complexity analysis of recursive algorithms with and without memoization," in *2024 International Conference on Science, Engineering and Business for Driving Sustainable Development Goals (SEB4SDG)*, 2024, pp. 1–3.

[18] CLBG Team, "The computer language benchmarks game," https://benchmarksgame-team.pages.debian.net/benchmarksgame/, accessed: 2024-10-24.

[19] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54. [Online]. Available: https://ieeexplore.ieee.org/document/5306797

[20] T. Eftimov, P. Korosec, and I. F. Jr, "Benchmarking in optimization: Best practice and open issues," *Applied Sciences*, vol. 10, no. 22, p. 7942, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/22/7942

[21] "functools — higher-order functions and operations on callable objects," Python Documentation, 2024. [Online]. Available: https://docs.python.org/3/library/functools.html#functools.cache

[22] "functools — higher-order functions and operations on callable objects," Python Documentation, 2024. [Online]. Available: https://docs.python.org/3/library/functools.html#functools.lru_cache

[23] G. Di Leo and F. Sardanelli, "Statistical significance: p value, 0.05 threshold, and applications to radiomics-reasons for a conservative approach," *European Radiology Experimental*, vol. 4, no. 1, p. 18, Mar 2020.

[24] functools — higher-order functions and operations on callable objects. [Online]. Available: https://docs.python.org/3/library/functools.html

[25] J. Sallou, L. Cruz, and T. Durieux, "Energibridge: Empowering software sustainability through cross-platform energy measurement," *arXiv preprint arXiv:2312.13897*, 2023.