Leveraging performance profiling to increase energy efficiency in Python applications

Project Report

Group cs-22-it-7-02

Aalborg University
Department of Computer Science

AALBORG UNIVERSITET
STUDENTERRAPPORT

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Selma Lagerløfs Vej 300
http://www.cs.aau.dk/

Title:
Leveraging performance profiling to increase energy efficiency in Python applications

Theme:
Scientific Theme

Project Period:
Fall Semester 2022

Project Group:
cs-23-it-9

Participant(s):
Martinus Nel

Supervisor(s):
Bent Thomsen

Copies: 1

Page Numbers: 46

Date of Completion:
June 10, 2024

Abstract:

This report aims to tackle the issue of energy usage in Python programs by exploratively studying methods for energy profiling that developers may use to identify energy inefficient patterns in their applications. Throughout the course of this report we look for ways to apply these methods to retrieve information that can be used to predict energy usage.

0.1 Summary

This report aims to explore different methods of profiling Python applications, with the goal of identifying which pieces of information are required to predict energy usage to a sufficient degree of accuracy. The intent is that once a reliable method has been established, it can be implemented with minimal effort from developers.

0.1.1 Monitoring

To have a better understanding of the energy usage over time, we chose to create a systemd daemon service that wrote readings of the powercap energy interface, and sensors temperature to a file every 0.5 seconds. This has a marginal effect on the energy usage of the system, and was enough of a task that the fan activity could be clearly seen on the resulting graphs, but was chosen as a way to easily demonstrate the profile throughout the course of the experiment

0.1.2 Algorithms

We chose to profile 5 algorithms: sleeping; the nbody problem; the mandelbrot set; a binary tree generator; and a benchmark that compares dataframe technologies. The intent of these choices was to try and cover a broad spectrum Python patterns.

0.1.3 Performance Profiling

For this report, we chose to examine Bash time, the Python sys settrace functionality, and the python cProfile module. We found that the settrace functionality had too much overhead to be feasible in a realistic environment, with a minimum of 50x increase in execution time. The cProfile method had better overhead but unfortunately did not expose fundamental enough information to be useful for cross-functionality. The Bash time command was the most useful, however the relationship has not been fully understood yet: running the Python sleep command appeared to have no energy cost, which suggested that energy would be proportional to CPU time; this was disproven by multithreaded algorithms surprisingly having lower energy cost overall. We found that despite this, all algorithms appeared to correlate with real time, furthermore that repetitions of each algorithm would consistently scale with this number - suggesting that the algorithm itself would have a consistent energy footprint.

0.1.4 Summarised Conclusion

In conclusion, we believe that the information required to predict energy usage has not been fully covered in this report, and suggest that further investigation into CPU loads would be required to get a full picture of energy costs.

Contents

Chapter 1

Introduction

1.1 Problem Statement

1.1.1 Context

Energy usage is a growing concern of the computing industry[8, 17], with Python being a widely used language[41], it is a prime candidate for energy optimisation research.

Energy optimisation has incentivised approach in several computing sectors, including but not limited to server hosting and mobile technology.

Data centres account for a significant portion of global electricity demand[7], which presents a promising avenue for reducing costs of long-term running services in an effort to reduce such costs[4, 6].

Similarly, mobile devices require limited batteries to operate, leading to a rather obvious incentive to optimise their energy usage for longer uptime[42].

1.1.2 Issue

Surveys on conventional knowledge on energy consumption show a general lack of understanding for avenues to improving power saving[24], this means that even if developers choose to optimise their code for energy efficiency, they may not know how to do so. This issue is exacerbated by the opacity of the variety of modern hardware (which can be complex to learn the details of) and the layers of abstraction that modern computers operate on - leading to a lack of understanding of the energy costs of specific operations.

The concept that this report hinges on is the fact that performance profiling is relatively a lot easier, as there are more readily available system agnostic tools that simply measure the time taken for a specific operation to execute. This is in contrast to energy profiling, which requires system level tools or physical hardware to measure the energy readings. Developers who want to measure the energy profile

of their system must find and implement such tools which require knowledge of the system that the software is running on, information which may not even be readily accessible in situations such as automated testing or cloud computing.

In summary, while both performance and energy profiling have their own challenges, performance profiling is more accessible and easier to implement than energy profiling.

1.1.3 Objectives

The objective of this report is to study the relationship of Python performance against its energy usage, with the goal of leveraging performance profiling to predict the energy profile of a Python application. To do this, we will have to find patterns that affect performance and energy in consistent ratios, such as specific system calls, memory interactions, or higher level concepts. Furthermore, we must test that these patterns scale, as specific concepts involving memory management and caching may require more advanced approaches to make accurate predictions. A completed product of this research would be a process that can be implemented with minimal developer effort or performance overhead, to aid in the adoption of energy efficient programming practices. The questions we aim to answer are:

- How does the performance of a Python application relate to its energy usage?
- What techniques are required to accurately infer the energy profile of a Python application?
- How can these techniques be applied in a convenient and non-invasive manner?

1.1.4 Justification

The positive correlation between performance optimisation and energy optimisation is rather intuitive, as many strategies for performance optimisation (such as reducing unnecessary execution) are congruent with energy optimisation. This correlation appears to transcend programming languages themselves, with studies into energy usage of languages coming up with similar ratios of energy against time for the same algorithms[25].

In comparison to performance profiling, energy profiling can be far more cumbersome to execute - as there is no truly accurate way to attribute energy usage to any specific process. This has lead to the suggestion of using performance profiling as a proxy to loosely optimise energy cost[2] with papers suggesting that execution time can be used to directly infer energy cost[3].

1.2 Related Work

Scaphandre

Scaphandre[36] is a tool that is designed to monitor the energy usage of a system, similar to the purpose of this report they seek to "*enable the tech industry to shift towards more sustainability*". As of the writing of this report, Scaphandre has had updates as recent as 4 months ago, and 1.5k stars on GitHub, which are good indicators of a healthy project.

Intel Power Gadget/Monitor

Intel Power Gadget[14] is a tool that is designed to monitor the energy usage of Intel CPUs, as of October 13, 2023, the tool has been deprecated in favour of the more general Intel Performance Counter Monitor(PCM) [13] - which is designed to analyze CPU resource utilisation on Intel processors.

1.3 Performance Profiling

This section will explore accepted ways to profile the speed of Python applications. Microsoft Visual Studio recognizes three forms of profiling: sampling, instrumentation, and tracing[27]. Profiling is a balance, as the more detailed the profiling, the more overhead is introduced to the script, in this section we will explore all the aforementioned methods of profiling, and find the most appropriate balance between detail and overhead.

Bash Time

Assuming the script is being run in a bash terminal, the time command[19] is a typical simple approach to finding execution time; as per the documentation this outputs 3 values: real, user, and sys time. Real time refers to the total time taken for the script to execute, user time refers to the time spent executing instructions on the CPU in user space, while sys time refers to the time spent executing calls in kernel space with elevated privileges. This is immediately more useful than simply timing the experiment, as it gives us insight into the execution time at different levels of processing. The Bash time function returns results in milliseconds, with both user and sys time being translated from clock ticks within the CPU - this makes it difficult to judge the accuracy of such measurements, as "*the value of HZ varies across kernel versions and hardware platforms*"[20], for the purpose of this report we will assume it is sufficient. Lastly, as the timer is not connected to the script being tested, there is no way to create a profiler that can locate exact points of inefficiency, simply indicate that they exist if the program is running slower than

expected - this can be partially circumvented by creating profiles for units of code that can be applied using static profilers.

```
from time import sleep
sleep(5)

>>> real    0m5,022s
>>> user    0m0,018s
>>> sys     0m0,004s
```

Listing 1.1: Measuring the Bash time output of running the sleep command from the Python time module to maximise the real time

```
x = 0
for i in range(100000000):
    x += i

>>> real    0m6,539s
>>> user    0m6,509s
>>> sys     0m0,010s
```

Listing 1.2: Measuring the Bash time output of iteration in Python to maximise the usr time

```
writer = open('dumpfile.txt', 'w')
large_text = 'x' * 100000000
writer.write(large_text)
writer.close()

>>> real    0m1,307s
>>> user    0m0,210s
>>> sys     0m0,797s
```

Listing 1.3: Measuring the Bash time output of writing to a file in Python to maximise the sys time

Perf

Perf is a complex Linux tool designed to collect a myriad of different events, for this section we will focus on the *record* and *report* functionality, which gathers samples of the cycles event, this compiled information can be displayed as a report with various different tools, such as the gecko tool displayed in fig1.1. Perf has the advantage of (if enabled) describing the entire system in its report, which can be useful to ascertain whether the script is being affected by other processes, or causing extra load not accounted for in the script.

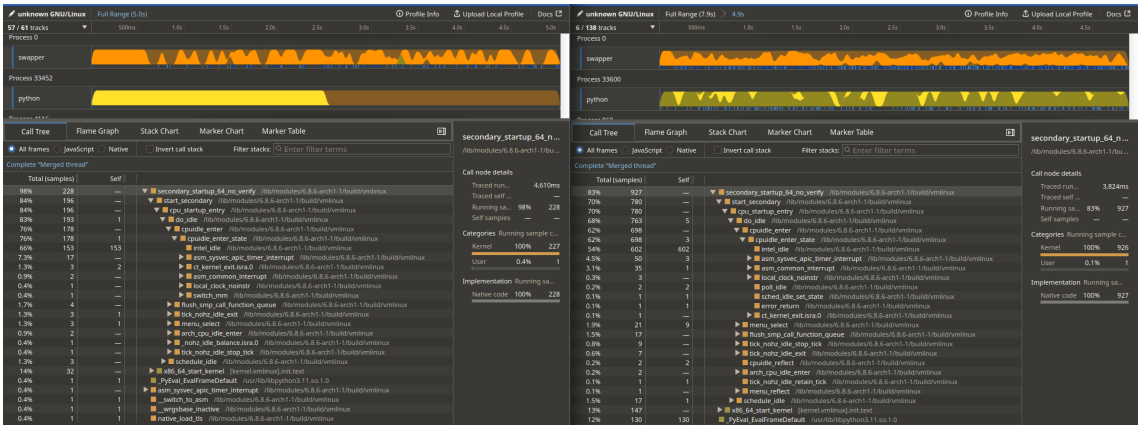

Figure 1.1: Comparing a Python script that sleeps for 5 seconds (left) to the busy calc script from 1.3 (right) note the increased amount of samples for busycalc, which may be used to predict its increased load

Python Profiler

The python profiler[31] is a standard tool for profiling Python applications, it improves on Bash time by providing information on the time within each call of the script, allowing for a more granular understanding of what might be causing performance bottlenecks. As per the documentation, the timer is limited by the underlying clock rate (1ms) - additionally due to the time between an event and the call for the clock state can introduce inaccuracies for calls that execute many times in a short period. The python profiler is split into two parts, the cProfile and profile modules, the cProfile module is a C extension with minimal overhead that is intended for our use-case, while the profile module is a pure Python implementation that is designed to be more approachable for tasks such as creating extensions - for our purposes we will focus on cProfile. While the Python profiler is a powerful tool, it is not without drawbacks, as calls themselves can be nebulous: 1.4 shows how the previously created *busycalc* script has only a single identifiable call. This suggests that simply attaching a value to each call is not enough, and that predicting an energy reading using this information will take a more complex approach.

```
$ time python -m cProfile busycalc.py
         3 function calls in 6.221 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function
    )
        1    0.000    0.000    6.220    6.220 {built-in method
    builtins.exec}
        1    6.220    6.220    6.220    6.220 busycalc.py:1(<module>)
```

```
9         1    0.000    0.000    0.000    0.000 {method 'disable' of '
    _lsprof.Profiler' objects}
10
11
12
13 real  0m6,303s
14 user  0m6,277s
15 sys 0m0,010s
```

Listing 1.4: Running cProfile on the busycalc script

Python Logging

The previous sections showed that there are many easy ways to record the time before and after script execution, so this section will rather focus on situations where this is not desired - namely in applications that are not expected to run for short periods of time.

Logging is a standard concept within programming, generally used to extract information about the state of a program from a user's perspective. The standard Python logger[30] is simple and allows timings to be attached to log messages, however this only operates up to a millisecond resolution, so for more precise calculations, this functionality has to be expanded on. When logging events, the desired effect is to retrieve the most accurate and high resolution time available, this is done by running the Python in-built time[33] utility - as of Python3.7, time provides six different functions at nanosecond resolution[34], preventing loss of precision due to floating point calculations. Similarly to Bash time, the different functions provide different insights into the execution of the script, these can be explored provided that logging is explored as a method of profiling.

The issue with logging is that it is not a standard method of profiling, and requires changes to the script to provide valuable insights into where time is being invested in the script. Lastly, as logging is an invasive process, it will undoubtedly affect the performance of the script, which introduces the decision of correct logging intervals to minimize overhead while still retrieving sufficiently precise data.

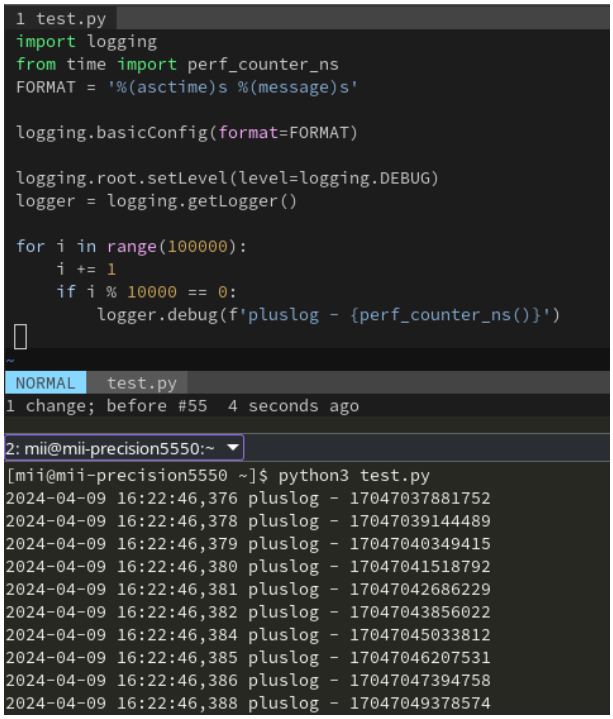

Figure 1.2: An example of logging being used to find the time after every 10,000 executions of adding to i. Note that while the logging library has a time function, there is no way to inject higher resolution timers without manually calling them. In the output there is also the perf counter, showing a far higher precision output of execution

PyTracer

As an alternative to logging, the Python sys package includes a settrace module[32] that allows for the injection of code into individual callbacks of the Python interpreter. This is a powerful tool, as it allows us to analyse individual stack frames, we can use this to analyse executed opcodes and derive energy usage metrics from predictions of their energy footprint - to understand the opcodes we will need to make further use of the disassembler module *dis*[29]. Furthermore, these frames contain contextual information that will allow a working profiler to pinpoint areas of contention to developers.

The major drawback of tracing is similar to logging, as any form of tracing is invasive, and will skew results, this can be mitigated if the footprint of the tracer is known and accounted for in the final results.

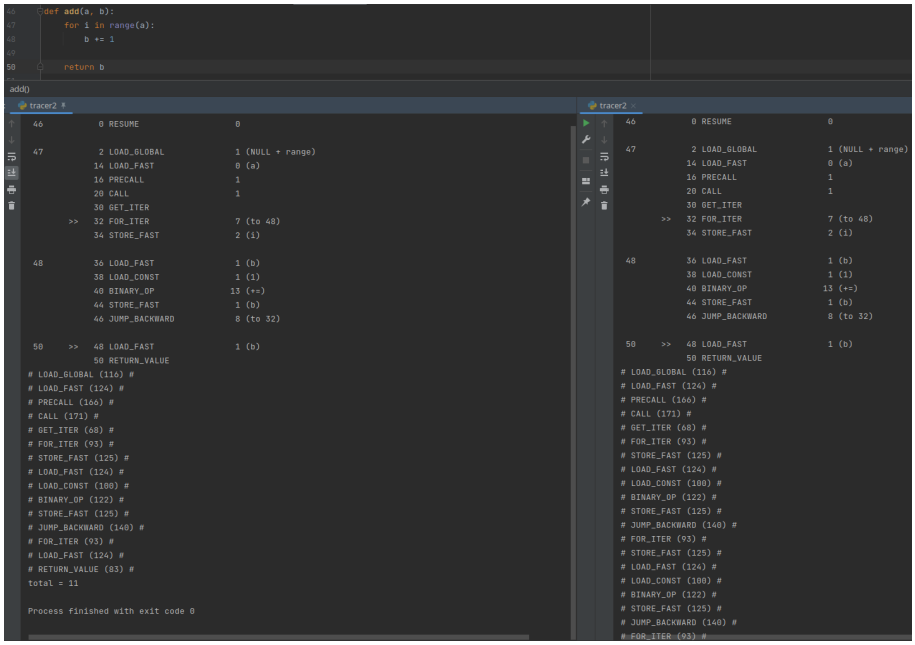

Figure 1.3: An example of using pytracer to trace the execution of a function that adds two values via iterating over the first, the output at the bottom is split into calling add(1, 10) (left) and *add(2, 10)* (right), first the bytecode of the function is printed using the diassembler module (with line numbers on the far left), followed by the individual opcodes being run - note that the output on the right is identical except for the repeated opcodes in the output on the right.

Chapter 2

Setup

2.1 Setup

The experimentation in this paper are being performed on a laptop with the battery removed, the specifications are as follows:

Laptop: Lenovo ThinkPad P1
CPU: Intel(R) Core(TM) i7-8850H CPU @ 2.60GHz
Memory: 32GB
Disk: 500GB SAMSUNG MZVLB512HAJQ-000L7
OS: Ubuntu 23.10
Kernel: 6.5.0-28-generic

2.2 Approach

It is unlikely that a single ratio can model the relationship between performance and energy, this report will attempt several methods at extrapolating contextual information of a script to predict its energy usage. Certain programming concepts such as memory management and may require more advanced instrumentation to make accurate predictions, such as multithreading, which has positive effects on performance at the cost of negative effects on energy usage[26]. The goal of this paper is to find a method that produces the most accurate predictions with the minimum effort of implementation. In reality, we do not expect to need incredibly accurate predictions, the aim is that the result of this paper can be used to find areas of greater energy cost, as long as we can expose these areas, we are not concerned with the exact energy values.

To find the most accurate model, this report will attempt to use a variety of forecasting techniques to predict energy readings from performance readings, then compared against actual energy readings to find the most accurate model.

Experiment Process

As even the most sanitised systems are unlikely to have a completely stable power usage, we will need to have multiple runs of each. For this report we have chosen to run each experiment 10 times, then average results deemed to not be outliers - outliers will be identified by unexplained rises in temperature, or large deviation from the mean. We choose 10 experiments as a maximum amount of tests that we can complete within the time constraints of this report, accounting for failed experiments and the time taken to set up each experiment. There will be a 10-minute gap for each experiment, this length of time has been chosen simply as a convenient length of time for plotting results, allowing for faster visual inspection of the data.

2.3 Data Collection

Energy Data Collection

Energy data collection will be done using intel's Running Average Power Limit(RAPL) interface, which is widely accepted as a sufficiently reliable method of measuring energy[39, 16, 12]. It provides a simple interface for retrieving energy usage, and breaks down energy usage into multiple domains, which can be examined separately. It should be noted that throughout this report we focus on the **PSys** domain, which contains a slightly more complete picture of the energy usage of the system that the other domains - this domain is unfortunately not widely available on all processor models[15], and so the results of this report may not be repeatable on other systems.

The RAPL interface has multiple points of access, which on Linux all depend on the *powercap* framework[21]. This section will explore the best approaches to collecting data from RAPL to get the best test results. For the purposes of our experiments we will retrieve the energy values in micro-Joules directly from the framework (this can be done simply by reading the files exposed in the powercap interface). Our method for retrieving this data is to manually find where each RAPL domain is stored, and retrieving the number that is stored there - later we can compare these numbers to retrieve a delta of energy usage. To have a better idea of the energy usage throughout a script's execution, we will retrieve the energy usage at regular intervals, this will also help identify any anomalous behaviour in the energy usage. The disadvantage of this is similar to the disadvantage of physical energy meters, as there is no way to guarantee that the beginning of the script is synced with the first energy reading, multiple experiments can mitigate this issue. It should be noted that the perf functionality mentioned in 1.3 can also be used to retrieve energy usage, however we have decided that manual retrieval is more reliable for our purpose.

The first and easiest way to measure the energy usage of a script is to measure the energy before and after, this functionality is provided quite simply via the perf tool. To understand the affect of individual constructs on execution time, there is a need for a more granular measurement of energy, this can be done one of two ways - either by injecting energy measurements into the scripts themselves, or by having a separate process measuring the energy level at specified intervals. The first approach has the advantage of being able to measure the energy usage of the components themselves, while the second approach is much less invasive and can be applied broadly. Due to the structured nature of RAPL interface writes, an outside polling approach seems to be more advantageous, as more granular measurements will not necessarily provide additional data. This does introduce the possibility of synchronisation issues between the script and the polling process.

Our first attempt at a polling process for energy usage is with the use of a cron job that runs every second, the job is simply a Bash script that saves data points to a logfile that can then be plotted.

Unfortunately cron does not expose the ability to run a job at a higher frequency than once a minute, so we must implement a script that executes the polling process in a loop at 1 second intervals. This has two disadvantages: firstly, if there is any interruption of the script, the polling process will be offset (if the script is delayed by a second, then there will be a 2-second gap between measurements, and consequently a duplicate value at the start of the next minute); additionally, as the script executes in non-zero time, every execution will delay the next poll, this means that the interval does not remain consistent even with perfect execution. Finally, in the event that the results of this report are to be reproduced, it should be noted that the cron service is not a guaranteed part of all Linux systems, and may have to be manually installed.

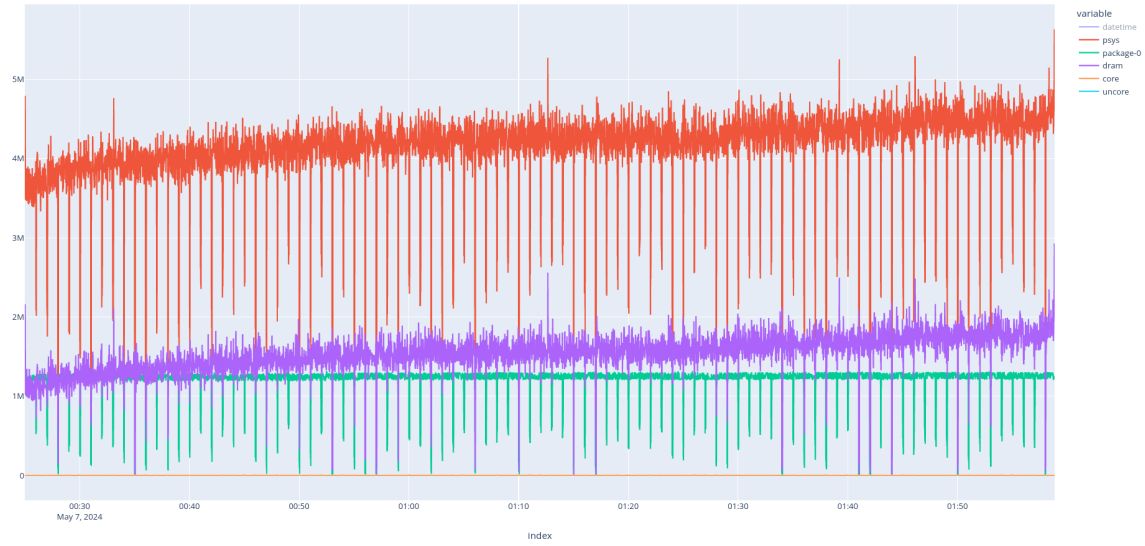

Figure 2.1: Using crontab to poll for energy usage every second, the inconsistency of intervals can be seen to cause spikes in the data that would have to be normalised for a more accurate representation of the data.

The next logical step is to attempt to take advantage of the systemd[11] service manager, which allows creation of custom daemon services, this includes a timer functionality that allows us to run a polling script every second. As we can see in fig.2.2, the systemd service provides a more consistent interval between polls, which reduces variance.

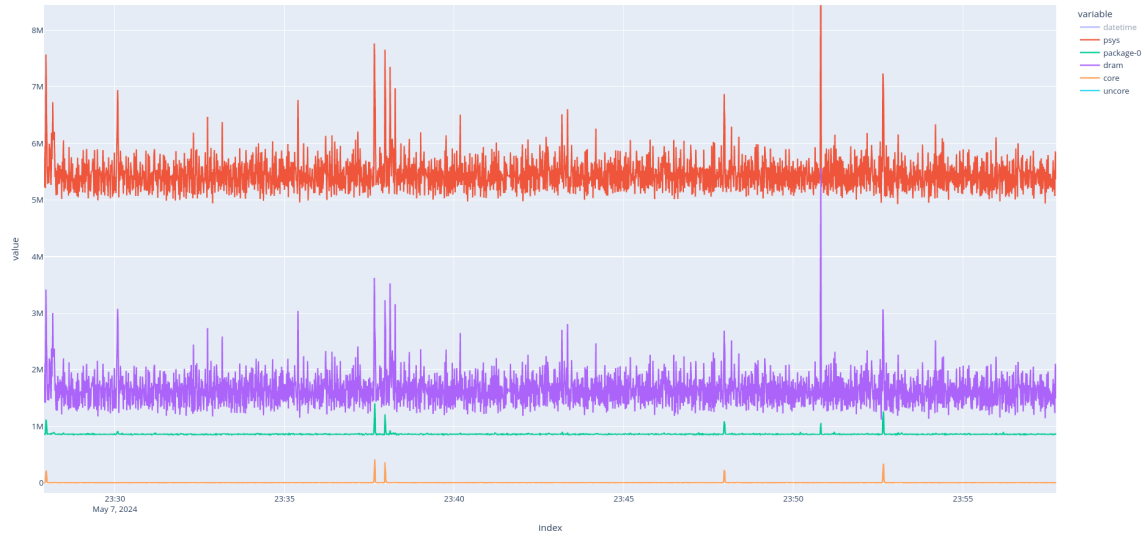

Figure 2.2: Using systemd to poll for energy usage every second, the consistency of intervals can be seen to provide a more consistent data set, however there is still clear visible variation in the data that may pollute results

Supplementary Data Collection

In the interest of a full picture; we also retrieve the temperature statistics of the machine, as it may help to explain anomalous behaviour in the results we find. This is simply done by parsing the output from the sensors command, which provides the temperature of the CPU cores. Unfortunately the sensors command does not provide functionality to request specific data, so some post processing is required to achieve plottable output - this is done by using the awk command to extract the relevant data, and can be seen on line 7 in listing 2.1. For plotting, we have chosen to simply average the temperatures of all cores, which may lead to an over-representation of multithreaded workloads, as the CPU will be hotter when all cores are in use.

For this report we have chosen not to log the workload of the CPU, a technique used by the Scaphandre tool [10] This has been done as a time constraint, as we would have to implement extra monitoring, parsing, and analysis to retrieve meaningful data from the workload of the CPU.

Polling Frequency

The first step in data collection is to determine the frequency at which we will poll the system, this is important as higher poll rates will affect the system more, while providing more accurate data. Intel claims an update frequency of approximately 1 millisecond[35], providing an upper limit for viable frequency, however the value that we choose will likely be lower than this, due to the time required to read from the RAPL interface. As it is impossible to guarantee exact intervals (to within microseconds), we will normalize all results to Joules/second, this will also prevent higher poll frequencies from having lower energy readings, allowing for easeier comparisons of the true invasiveness of the polling system.

In addition to finding the optimal polling frequency, this section will provide a baseline control dataset that can be compared against future data.

Rate - 1s A rate of 1 second produces unremarkable results, as there is a lack of comparison so far.

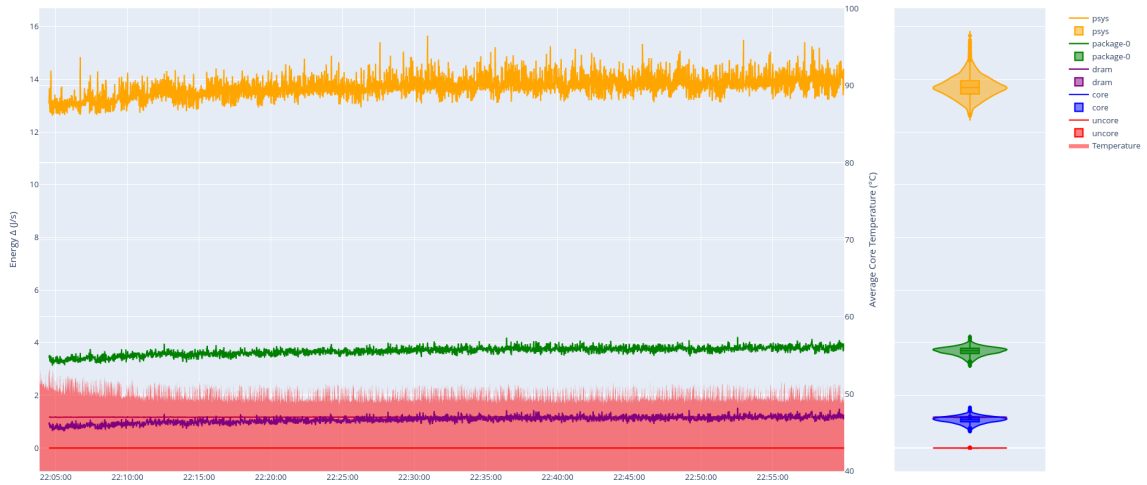

Figure 2.3: Polling the powercap interface at a rate of every second for an hour; psys stats in are: min - 12.61, max - 15.65, mean - 13.70, std - 0.41

Rate - 0.5s A rate of 0.5 seconds has a noticeable increase in power usage, however variance is slightly lower, this could either be due to random chance, or it may indicate that there is a pattern to the volatility of RAPL's power readings, which suggest the possibility of modeling the baseline pattern of the system to predict future power usage. This rate also exposes a cyclical pattern to temperature, which may be useful for future analysis.

Figure 2.4: Polling the powercap interface at a rate of every 0.5 seconds for an hour; psys stats are: min - 16.41, max - 21.24, mean - 17.97, std - 0.38

Rate - 0.2s A rate of 0.2 seconds uses almost double the energy of the 1 second rate, and has much higher variance, making it the worst choice for a polling rate.

Figure 2.5: Polling the powercap interface at a rate of every 0.2 seconds for an hour; psys stats are: min - 18.29, max - 26.30, mean - 20.61, std - 0.58

Conclusion

Unfortunately increasing the rate past 0.2 seconds appears to crash the systemd process, and has been deemed not viable for experimentation. As results are relative, we choose the polling rate of 0.5 seconds, as it provides a good balance between invasiveness and repeatable patterns - and better models temperature change. Readers repeating the process may also want to use these results to extrapolate the energy cost of the polling system - this can be done by taking the difference between the mean of the 0.5s rate and the 1s rate, and then multiplying this that difference by 2 (note that this is unlikely to be accurate, and is disproved by our results where the difference between 1 and 0.5 should be smaller than the difference between 0.5 and 0.2).

Lastly, RAPL occasionally resets the powercap interface to zero readings - this interferes with results, as it is no longer possible to simply compare the power reading before and after results. Due to the nature of powercap updates, we will never receive a zero value, so the current process is to subtract the first power usage value from the maximum, and then add the final value - it is important to note that this approach will not work for experiments that run long enough to have two or more updates.

The resulting collection logic is displayed in listing 2.1, this is executed by the systemd service in listing 2.2 which is in turn executed by the timer service in listing 2.3.

```
1 ENERGYOUT=################
2 TEMPOUT=##################
3 ENERGY_FILE=$(cat ################)
4
```

```
time=$(date +"%y-%m-%d %T")
energy=$(cat $ENERGY_FILE | sed -z 's/\n/,/g')
temp=$(sensors | awk -F '  ' '/^Core/{gsub("[[:space:]]+",""); printf
    "%s,", $1}')

echo "RUNNING POWERLOG"
printf "$time,$energy\n" >> $ENERGYOUT
printf "$time,$temp\n" >> $TEMPOUT
```

Listing 2.1: power and temperature logging script - variable names have been replaced with hash characters

```
[Unit]
Description=Run profiler every second
StartLimitIntervalSec=60
StartLimitBurst=61

[Service]
ExecStart=/usr/bin/bash #########/powerlog.sh
User=#######

[Install]
WantedBy=multi-user.target
```

Listing 2.2: systemd service

```
[Timer]
OnUnitActiveSec=1s
AccuracySec=1us
Unit=profiler.service
```

Listing 2.3: systemd timer

2.4 Experimental Algorithms

The intent of this section is to define and justify the algorithms we have chosen to test in our research, additionally, we can use preliminary tests of each algorithm as a baseline for their respective energy footprints in the Implementation chapter.

Experimental Space

To prevent bias in specific operations, this paper will attempt to use a variety of scripts that perform different purposes. As a time constraint, the scripts should be simple to execute, and ideally should have very little ambiguity on their intended execution - as an example, python libraries will not be considered, as they perform many functions, and it is difficult to determine an execution path without a full understanding of the library. Consequently, programs that operate on a resource,

or connect to the internet will also not be considered, to prevent time wasted on debugging network issues or misunderstandings of the operable resource.

Constraining the experimental space like this introduces an obvious bias, this can be addressed in continuations of this research by expanding the experimental space to include more complex scripts.

The most obvious approach is to start by running benchmarks, as they are often designed to test the limits of a system and are likely to explore a wide range of operations to draw conclusions. For this report we will use the scripts defined in the Debian computer languages benchmark game[40]. Relying on benchmarks has two major drawbacks: the first is that benchmarks necessarily are not a good representation of a typical use case of a system, rather a way to expose the flaws of the tested system; the second issue is that benchmarks must be designed to be comparable across the tested systems - in our case this means that all the algorithms we have taken from the benchmark games were chosen for their broad applicability and may not cover a sufficient amount of Python constructs to be useful for our purposes. Nevertheless, the benchmarks will provide a good starting point for our research. In a parallel approach, we can find example code to benchmark specific sectors by searching for research papers comparing technologies in that sector, this provides a more focused approach to targeting specific concepts in Python, as sufficiently well presented papers will provide control groups to eliminate technology bias.

We will attempt to keep algorithms at a baseline of 10 minute runs, this duration is chosen simply for convenience, as it is short enough to run multiple times, and long enough to aid visual analysis of the data. Each experiment will be run 10 times, to inspect variance in run time and energy usage, outliers will be identified by unexplained rises in temperature, or large deviation from the mean.

Sleep

The most trivial process to profile is the sleep command, profiling a sleeping Python script should give us a good baseline for the energy usage of the Python interpreter, and the footprint of the various profiling tools.

As can be seen in fig 2.6, the Python interpreter has a completely negligible energy impact, and can not be distinguished from the noise of the system. For this reason, when forecasting future results, we can ignore baseline costs.

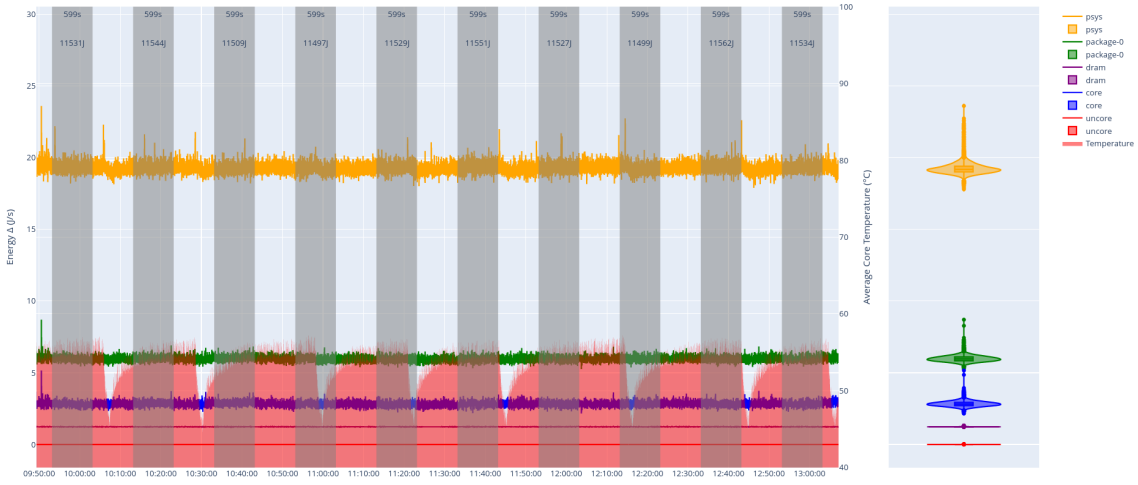

Figure 2.6: Running sleep within Python for 10 minutes, experiment durations are highlighted in grey

Bash Time The sleep command is important for understanding the Bash profiling tool, as we can see that sleeping has no visible effect on overall power consumption, we will attempt to make the assumption that when Bash profiling, time not spent in usr or sys accounts for zero energy usage. This assumption is likely incorrect, as simply executing the script will incur an energy cost, however we believe that the value is low enough that it can be safely ignored.

N-Body

The n-body problem is a classic problem in physics, and is used to simulate the motion of celestial bodies in a system. In order to come close to the 10-minute testing time that we have previously implemented for sleep, we are running this algorithm with an input of 50 million. As seen in 2.7, the algorithm is running at an average of 652 seconds, which is acceptably close to the 10-minute mark, with an average of 15211 joules of energy used. Visual analysis of the graph shows that energy usage varies highly (with a range of 13448 to 17119 joules), though this seems to be directly proportional to the variance of time - this suggests that more promising results will be seen from approaches that test for time, rather than the actual execution of the script.

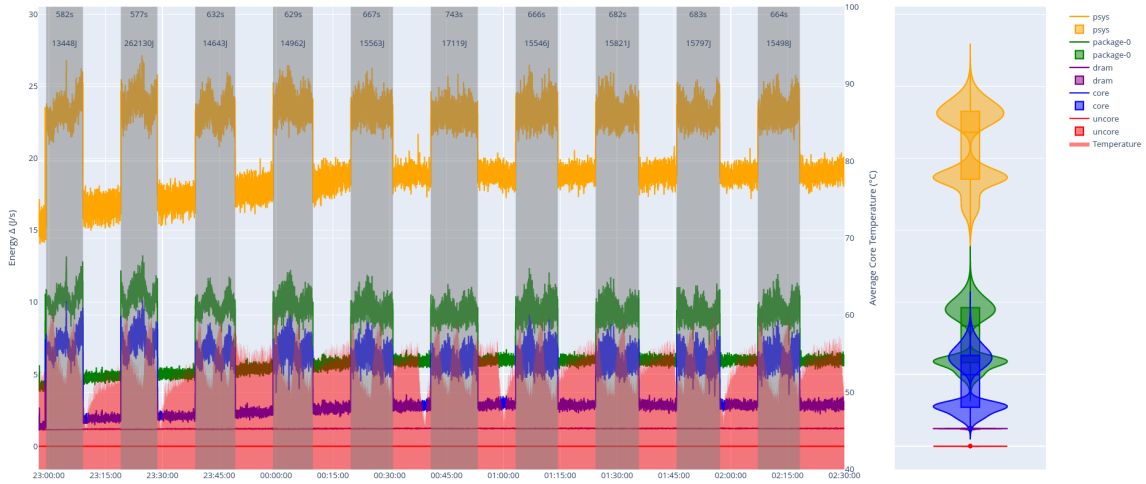

Figure 2.7: Running the n-body problem, experiment durations are highlighted in grey

Bash Time The n-body problem is a good example of a pure user space script, after 10 runs the mean of each time is as follows: **real** - *10m:48s* (std - 55s) **user** - *10m:48s* (std - 55s) **sys** - *0.02s* (std - 0.01s)

Binary Trees

Binary trees are a simple data structure that are classically used to as an excercise in computer science - for this experiment we have specifically chosen an implementation that avoids multithreading, as we are interested in having multiple algorithms that are sequential. As seen in 2.8, the algorithm is running at an average of 457 seconds, with an average energy usage of 10860 Joules, this is a promisingly close result to the n-body problem (an average of 23.3J/s compared to 23.8J/s), and suggests that the energy usage of the Python interpreter is consistent across different sequential algorithms.

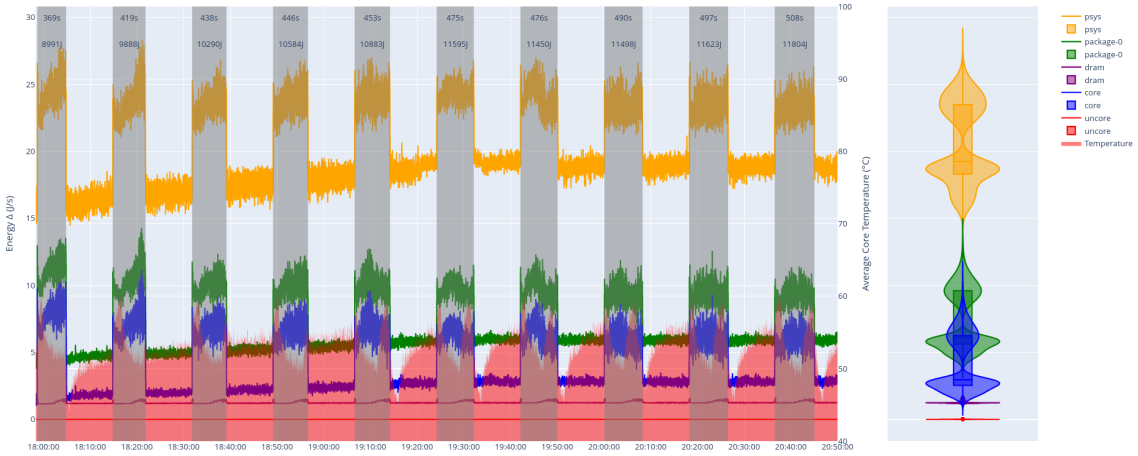

Figure 2.8: Running the binary tree algorithm, experiment durations are highlighted in grey

Mandelbrot

Mandelbrot is a classic algorithm used to generate fractal images, and is often used as a benchmark for performance, we have chosen this specific implementation as it is highly multithreaded, as can be seen in 2.9, the core and package-0 domains are not only far lower than previous sequential algorithms, they are also far more stable; this suggests that further research must be done to understand how the energy usage of multithreaded applications affect energy usage.

In our preliminary experiments, the Mandelbrot algorithm has shown to execute at an average execution time of 585 seconds, with an average energy usage of 13341 Joules (averaging 22.8J/s).

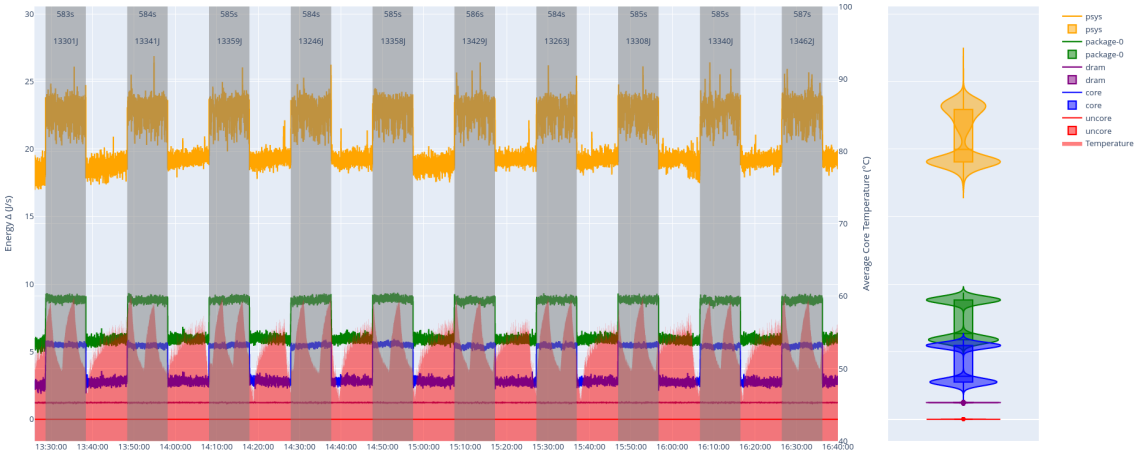

Figure 2.9: Running the mandelbrot algorithm, experiment durations are highlighted in grey

DataFrame Tester

Python DataFrames are a powerful tool used for data manipulation and analysis, and is being used extensively to generate the data seen in this report, we believe that DataFrame benchmarking is a useful avenue as they are often heavily involved in running underlying C binaries, together with the fact that there may be some level of multithreading implemented. For this experiment we have found an article looking to compare and benchmark the performance of different dataframe libraries [1] - unfortunately two of the four benchmarking libraries were too cumbersome to implement, this is ultimately not an issue as the validity of the test is not a concern of this report.

As seen in 2.10, the algorithm is running at an average of 217 seconds, with an average energy usage of 5200 Joules, unfortunately, it has been difficult to find an input to reach an average 10-minute mark.

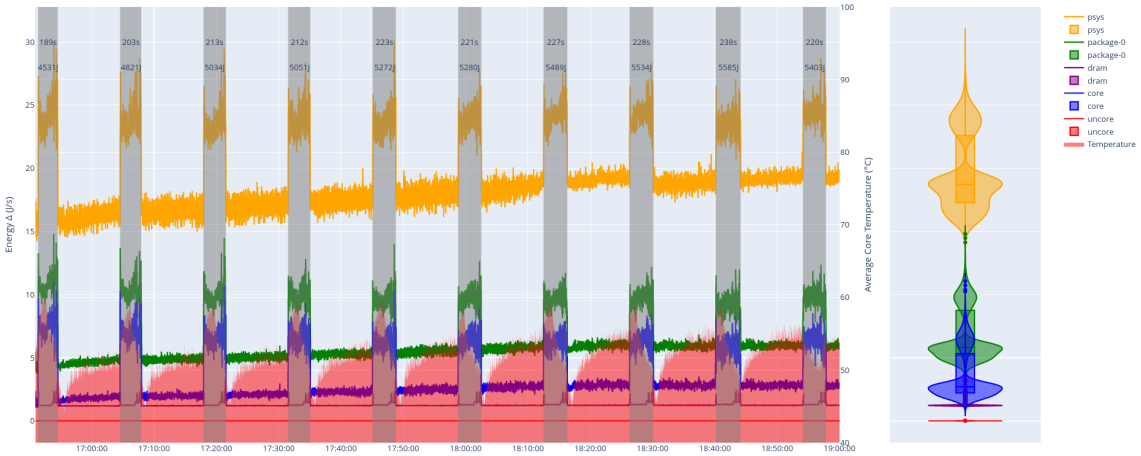

Figure 2.10: Running the dataframe tester, experiment durations are highlighted in grey

Bash Time The DataFrame tester brings two facets, firstly, the real time is lower than the sum of user and sys time, this indicates that the process must be multithreaded - when looking at the sleep experiment, we suggested that time spent outside the user and sys time would incur zero energy cost, while this experiment suggests that it might be pertinent to ignore the value completely.

The statistics of the exploratory run are as follows: **real** - *3m:37s* (std - 15s) **user** - *5m:9s* (std - 14s) **sys** - *44s* (std - 1.5s)

Chapter 3

Tests

3.1 Bash Time

The first profiling method we will test is simply Bash time, the assumption to make for this section is that the energy profile can be accurately predicted by simply knowing the amount of time a process has spent in real, user, and kernel time.

Internal Consistency

The first step in our process is to verify that running an algorithm multiple times remains consistent in terms of the amount of energy used over the time taken. From the previous section, it is clear that there is a large amount of variance in time taken for algorithms to complete - this section seeks to demonstrate that the variance of time predictably scales with the variance of energy usage. If we can prove that the energy usage of an algorithm is consistent over multiple runs, this would suggest that time measurements are a required component of predicting energy usage. For this section, we simply use the results of the baseline experiments performed earlier, the average times are as follows:

	Sleep	NBody	BTree	DataFrames	Mandelbrot
real	10m	10m54s	7m38s	3m38s	9m46s
usr	0.02s	10m53s	7m28s	5m09s	1h46m
sys	0.01s	0.01s	10.7s	45.0s	16.5s

Next we compare the energy usage of each algorithm to the real time taken to complete each algorithm (Joules/sec):

	Sleep	NBody	BTree	DataFrames	Mandelbrot
mean	19.212	23.285	23.715	23.820	22.782
std	0.038	0.261	0.411	0.333	0.077
range	0.109	0.742	1.143	1.071	0.243

From this data, we can see that the energy usage of each algorithm is consistent over multiple runs, with each algorithm using 3.5-4.5J/s (normalising against the results of sleep). Surprisingly despite its multithreaded nature, the Mandelbrot algorithm is by far the most efficient, this unfortunately invalidates the theory that we could correlate CPU time with energy usage, as the Mandelbrot algorithm uses by far the most CPU time. For single threaded applications, this correlation may still be possible to draw, however we do not have a sufficient sample size to make this claim. In the following table we compare the energy usage per second of each algorithm to the CPU time taken to complete them (the values for sleep have been subtracted to prevent CPU time from accounting for regular background noise).

	Sleep	NBody	BTree	DataFrames	Mandelbrot
mean	0	4.07	4.56	2.83	0.33
std	0	0.29	0.38	0.20	0.01
range	0	0.83	1.03	0.54	0.02

This further proves that while the time spent executing the algorithm remains consistent, using execution time is simply not enough information to predict the energy usage of an algorithm.

Scaling Consistency

The next step is to verify that the energy usage of an algorithm scales with the time taken to complete the algorithm, as all the algorithms we have chosen rely heavily on repetition. In the previous section we have shown that algorithms appear to strongly correlate with their run-time, based on this evidence we can test that the energy usage will scale linearly for higher work-loads (under the assumption that running more of a similar set of instructions will lead to a linear increase in energy usage). If this also shows a strong correlation then analysis on the underlying process of the algorithms are likely to be the second component needed for an accurate prediction. For the following table we have taken increments of each algorithm and measured their energy usage over time.

Iteration	1	2	3	4	5	6	7	8	9	10
Sleep Time (s)	399	419	439	459	479	499	519	539	559	579
Sleep Energy (J/s)	15.83	16.78	17.17	17.94	18.54	18.89	18.97	18.87	18.93	18.87
NBody Time (s)	474	488	515	548	579	616	618	648	655	676
NBody Energy (J/s)	22.94	23.29	23.28	23.37	23.55	23.05	23.18	23.07	23.07	23.07
BTree Time (s)	0	0	1	3	7	21	45	106	209	504
BTree Energy (J/s)	0.0	0.0	19.1	19.45	21.64	22.61	22.44	23.23	23.47	23.21
Dataframes Time (s)	52	111	172	225	285	346	416	468	535	576
DataFrames Energy (J/s)	11.12	23.6	23.49	23.97	24.2	23.65	24.08	23.69	23.58	24.32
Mandelbrot Time (s)	154	191	229	271	318	366	415	469	526	585
Mandelbrot Energy (J/s)	21.58	22.7	22.83	22.92	23.09	22.97	22.81	22.81	22.78	22.68

We can see from the results that at lower time sample sizes, energy usage appears to be inconsistent, but stabilises rapidly for longer lived algorithms - this is likely due to fluctuations having a greater effect on faster run-times.

Temperature Adjusted Readings

Visually it can be noticed that as experiments continue, the average baseline temperature of the CPU appears to increase slightly, unfortunately, directly correlating this to energy usage is not possible, as the fans of the CPU reduce temperature while maintaining higher energy usage. Unfortunately due to time constraints we could not find the true relationship between temperature and energy usage

Conclusion

From the results of this section we can conclude that time is a required component for predicting energy usage, as we have deemed fluctuations in execution to correspond to changes in energy usage. We have found that there is very little correlation with the CPU time taken to complete an algorithm and its energy cost - disproven by the mandelbrot algorithm unexpectedly out-performing the other algorithms, this phenomenon is further explored in the 4.2 section. For a most basic prediction of energy usage, we can simply attach a value to the time taken to complete an algorithm, this unfortunately may be too basic of an approach for user's attempting to fully understand the energy usage of their applications.

3.2 Opcode Tracing

As purely looking at time doesn't appear to be a reliable method of predicting energy usage, we will now attempt to look at the opcodes being executed by each script to see if a conclusion can be drawn between executed opcodes and energy

usage. We have already noted that opcodes can be retrieved by using the sys module, unfortunately there does not appear to be a process that can also retrieve the operands provided, which have been suggested to affect energy usage, but possibly not by a significant amount [38].

Opcode Gathering

In the exploration of profiling tools 1.3 we simply took each opcode and printed it to console. This is a good start, but fails to gather data in a meaningful way, and will be useless for larger programs that execute more opcodes.

For testing opcode gathering in this section we will use the nbody algorithm, as it scales easily and has a large number of operations involved. The primary problem with any kind of granular tracing is that the operations involved in the trace naturally interfere with the execution of the script itself, as opcodes increase in amount this becomes a very real problem, as we will see in following examples.

We are apprehensive of the cost of tracing, and so will choose a very short run of the nbody algorithm for this test to begin with, and monitor effects on performance using Bash time as we introduce more invasive tracing.

To begin with, we will simply run the nbody algorithm with no tracing, we have chosen an input of 50000 as it runs for approximately half a second (we are also continuing to print the output of the algorithm to ensure that there is no effect on the value). It should be noted that the tests in this section are ad-hoc, and are not intended to be used as a benchmark.

```
1 offset_momentum(BODIES['sun'])
2 report_energy()
3 advance(0.01, 50000)
4
5 >>> -0.169075164
6 >>> real    0m0,583s
7 >>> user    0m0,672s
8 >>> sys     0m0,988s
```

Listing 3.1: Running the nbody problem at an input of 50000

Next we add basic tracing with no logic involved to see the cost of tracing itself.

```
1 def trace(frame, event, arg):
2     return trace
3
4 sys.settrace(trace)
5
6 offset_momentum(BODIES['sun'])
7 report_energy()
8 advance(0.01, 50000)
9
10 sys.settrace(None)
11
```

```
>>> -0.169075164
>>> real    0m1,915s
>>> user    0m1,987s
>>> sys     0m0,973s
```

Listing 3.2: Tracing the nbody problem with no trace logic

Lastly, to retrieve the opcodes executed by the system, we must also enable the tracer to trace them, we test the effect of this by adding a boolean assignment inside the trace function.

```
def trace(frame, event, arg):
    frame.f_trace_opcodes = True
    return trace

sys.settrace(trace)

offset_momentum(BODIES['sun'])
report_energy()
advance(0.01, 50000)

sys.settrace(None)

>>> -0.169075164
>>> real    0m8,320s
>>> user    0m8,249s
>>> sys     0m1,057s
```

Listing 3.3: Tracing the nbody problem with opcodes enabled

As we can see by these tests, enabling opcodes in the trace alone has an effect an entire order of magnitude greater than the original run-time, this likely already makes the approach unviable for the purpose of the report, which is to create a tool that can be used to predict energy usage in a reasonable amount of time. Despite this, we will continue to explore the concept of opcode tracing, as it may be useful for smaller scripts, or for gathering snippets of larger algorithms.

Finding the least logic required to trace opcodes is a difficult process, and there are likely other ideas to attempt in achieving smaller workloads between opcodes, here we document our attempts: First we attempt to simply print the opcodes directly to console, this is a naive approach, but fully functional and gives us a starting point to optimise from.

```
def trace(frame, event, arg):
    frame.f_trace_opcodes = True
    print(dis.opname[frame.f_code.co_code[frame.f_lasti]])
    return trace
sys.settrace(trace)

offset_momentum(BODIES['sun'])
report_energy()
```

```
advance(0.01, 50000)

sys.settrace(None)

>>> RESUME
>>> LOAD_FAST
>>> LOAD_FAST
>>> GET_ITER
>>> ...
>>> -0.169075164
>>> ...
>>> LOAD_CONST
>>> RETURN_VALUE
>>> RETURN_VALUE

>>> real    20m29,021s
>>> user    2m29,438s
>>> sys     1m20,866s
```

Listing 3.4: Naive opcode printing approach

To create an efficient tracer in Python space (compiled solutions will be reserved for future work), we have identified two potential approaches towards reducing the overhead of the tracer:

Aggregation - Instead logging each opcode, we can aggregate the opcodes into a map structure that gets incremented, providing a count of each opcode at the end and reducing the amount of data being stored; the downside of this approach is that it may not be possible to determine the order of opcodes executed.

Streaming - On Linux the Python print statement sends data to the STDOUT stream[22], which is then flushed to the console, the print statement unfortunately also has significant overhead[28], so to make this solution work, we will have to bypass the regular logic Python uses and write directly to either a stream or a file, whichever is more efficient. This solution will allow us to preserve the order of the opcodes, and find patterns in the order of their execution, but it creates a lot of extra data that must then be processed and analysed.

```
events = np.zeros(200)
def trace(frame, event, arg):
    frame.f_trace_opcodes = True
    events[frame.f_code.co_code[frame.f_lasti]] += 1
    return trace
sys.settrace(trace)
offset_momentum(BODIES['sun'])
report_energy()
advance(0.01, 50000)

sys.settrace(None)

events = pd.Series(events)
```

```
print('\n'.join([f"{dis.opname[x]} - {y}" for x, y in events[events > 0].items()]))

>>> -0.169075164
>>> POP_TOP - 1.0
>>> BINARY_SUBSCR - 3750000.0
>>> ...
>>> PRECALL - 500002.0
>>> CALL - 500002.0

>>> real    0m29,549s
>>> user    0m29,574s
>>> sys     0m1,001s
```

Listing 3.5: Tracing the nbody problem using aggregation

The most efficient aggregation solution we found was to generate a numpy[23] array of zeros, and then increment them, this solution unfortunately still requires indexing, which is likely the cause of the 4 times increase in execution time (we consider the loss of memory on unused opcodes to be negligible, as the array is only 256 elements long).

```
def trace(frame, event, arg):
    frame.f_trace_opcodes = True
    opout.write(str(frame.f_code.co_code[frame.f_lasti])+',')
    return trace
sys.settrace(trace)
offset_momentum(BODIES['sun'])
report_energy()
advance(0.01, 50000)

sys.settrace(None)

>>> -0.169075164
>>> real    0m28,407s
>>> user    0m28,290s
>>> sys     0m1,075s
```

Listing 3.6: Tracing the nbody problem using streaming

The most efficient streaming solution we found was to append to a file on the system, while it is slightly faster than the aggregation solution, it requires more post-processing to be useful, and notably generated 305MB of data in a script that normally has a 0.5 second runtime. As opcode numbers are not uniform length, a separator is required to split the opcodes, this action accounts for approximately 1/3 of the overhead.

Conclusion

While the opcode tracing approach is interesting, and may be useful for smaller scripts, all of our efforts to reduce the overhead of the tracer have failed, resulting in a minimum of 50 times overhead for our most efficient attempts. A continuation of this report may involve a compiled solution, or a more efficient method of opcode tracing.

3.3 Python Profiler

As opcodes do not appear to be a viable approach in the current state, we will attempt to use a more traditional approach in this section.

Implementation

Similarly to the opcode section, we will use the NBody algorithm as a test case to understand the general overhead of the Python profiler. As can be seen in the code snippet 3.7, we must do some extra work to capture the output of the profiler, as it directs output to STDOUT. We then use a regex string to capture the individual values outputted.

```
# Prepare test
pr = cProfile.Profile()
pr.enable()

# Run test
offset_momentum(BODIES['sun'])
report_energy()
advance(0.01, 500000)

# End test
pr.disable()
s = io.StringIO()
ps = pstats.Stats(pr, stream=s)
ps.print_stats()

# Parse values
text_output = s.getvalue().split('\n')
total_run = text_output[0].split()[-2]  # get total run value
restring = r"^\s*(\d*)\/?\d*\s*(\d*\.?\d*)\s*(\d*\.?\d*)\s*(\d*\.?\d*)\s*(.*)$"
values = [list(re.match(restring, x).groups()) for x in text_output[5:-3]]
pandas_values = pd.DataFrame(values, columns=['ncalls', 'tottime', '_', 'cumtime', 'percall', 'method'])
pandas_values = pandas_values.drop(columns=['_'])

```

```
24 # Truncate method length for printing to console demo
25 pandas_values['method'] = pandas_values['method'].str[:20]
26 print(pandas_values.sort_values('tottime', ascending=False))
27
28 >>>     ncalls  tottime  cumtime  percall                 method
29 >>> 0        1    6.531    6.793    6.793  ####################
30 >>> 5  5000000    0.261    0.261    0.000  {built-in method mat
31 >>> 1        1    0.000    0.000    0.000  {method 'disable' of
32 >>> 2        1    0.000    0.000    0.000  ####################
33 >>> 3        1    0.000    0.000    0.000  ####################
34 >>> 4        1    0.000    0.000    0.000  {built-in method bui
```

Listing 3.7: Running the NBody problem with the cProfile profiler (file paths have been replaced with hash values)

```
1 ncalls  tottime  percall  cumtime  percall           method
2 1    0.000    0.000    0.000    0.000 {built-in method
3 11    0.001    0.000    0.001    0.000 {built-in method
4 1    0.000    0.000    0.000    0.000 {method 'disable
5 3.../1...  222.276    0.000  222.276    0.000 ..:15(make_tree)
6 3.../1...  230.958    0.000  230.958    0.000 ..:20(check_tree)
```

Listing 3.8: Running the Btree problem with the cProfile profiler, we changed the truncation style slightly, line 5 and 6 originally have a path from root to the method

Our preliminary tests have shown that the above method increases execution time by approximately 200%, which we consider an acceptable decrease in performance for a test that only needs to be run once.

Unfortunately listing3.8 shows that the profiler is not as useful as we had hoped, as the methods that take up the majority of the time are both specific to the algorithm, so attaching a value will not be applicable for all algorithms. In conclusion, we do not believe that the Python profiler is a viable method for predicting energy usage, as the information given is not fundamental enough, as disproven by the BTrees experiment.

Chapter 4

Conclusion

4.1 Conclusion

How does the performance of a Python application relate to its energy usage?

As was expected when we began this project, there are many factors that can affect the energy usage of a Python application, our exploratory findings are as follows:

- The energy usage of a running the Python interpreter is not measurable against the noise of the system itself, though running the interpreter with no instructions repeatedly appeared to increase the ambient temperature of the processor, suggesting that there are unaccounted for processes in our monitoring. This discovery led us to believe there was a connection between the time spent within the CPU and energy usage, as programs with low CPU time had negligible energy usage.
- Following our previous findings, we attempted to tie CPU time to energy usage, but found that largely multithreaded processes - that used far more CPU time - were far more energy efficient, and more stable. This leads us to believe that the energy usage of a process is more tied to the CPU load, as evidenced by other studies highlighted in the Threat to Validity4.2 section. More research is required to confirm this hypothesis.

What techniques are required to accurately infer the energy profile of a Python application?

Due to time constraints this report has not found a definitive prediction model for energy usage. Research of smaller scale approaches to energy usage have shown that analysis of underlying instructions can be used to infer energy costs[9], suggesting that accurate predictions require these instructions to be documented and understood. In this report we have shown that some of these techniques can be

cumbersome to build and apply, in reality this may not be an issue as the techniques become automated, however we pose that such metrics are not required for a general estimate of energy usage.

How can these techniques be applied in a convenient and non-invasive manner?

In this report we have attempted several methods to profile energy usage, rather expectedly the only non-invasive method we have achieved is simply retrieving time, the overhead of which could not be distinguished from the fluctuations of the run-times themselves. In contrast, both opcode and call tracing proved to have significant overhead; it may be argued that as these tracers only have to run once, there is a higher tolerance for lower efficiency monitoring. We believe that implementing these methods will however negatively affect development time, we believe developers will be de-incentivised from experimenting, as testing new code will require a significant amount of time to run.

Conclusion

In conclusion, we believe that counter-intuitively, the most effective method of profiling energy does not lie in the Python profiling tools, but rather by monitoring metrics that are already ubiquitous in the development process. Due to lack of experimentation results, we can not definitively state the culprit of energy usage in Python applications, but the efficiency of our multithreaded experiments, paired with research into similar fields has led us to believe that CPU load is a major factor in energy usage.

4.2 Threats to Validity

CPU Load

The CPU load of a system is a measure of how much of the CPU is being used at any given time, in section2.3 we noted that monitoring CPU load would be too cumbersome for the time requirements of this report. Unfortunately this appears to have been a mistake, as multiple studies have shown a non-linear relationship between CPU load and energy usage[37, 5].

Temperature

Visual inspection of the average CPU temperature throughout experiments shows a clear increase in temperature as experiments progress, this suggests that a 10-minute gap was not sufficient for a true reset, this is also shown by the increased run-times for later repetitions of each algorithm. Furthermore, experimentation

was performed in a room with standard commercial air conditioning, which increases the risk of environmental factors such as temperature and humidity affecting results differently at different times. Fortunately, this mistake led to the discovery of the correlation between real time and energy usage, as the increased run-time of each algorithm is directly proportional to the increased energy usage.

Experimental Width

Due to time constraints, we could only test a certain amount of algorithms on a specific set of hardware, with such a narrow experimental width there is a chance that specific algorithms or hardware quirks could have affected the results.

4.3 Future Work

More Algorithms

An exhaustive test of all Python applications is infeasible, however it can be argued that there are many sectors that are not represented by the algorithms in this report. Future studies could look into finding if there are specific implementations that produce different patterns to those found in this report, and if so, what the cause of these differences are.

Opcode Gathering

In the opcode section3.2 we deemed that opcode tracing was not a viable approach to predicting energy usage due to its large overhead; an extension of this report could be to attempt to create a compiled approach to opcode gathering, which may reduce overhead sufficiently for an applicable solution.

Hardware

Difference combinations of hardware will have different effects on energy, a worthwhile future contribution could repeat similar experiments for different popular hardware configurations, to see if the patterns found in this report are consistent across different systems.

Chapter 5

Appendix A - Experimental Algorithms

The code used for each experimental algorithm.

```
import time, sys

time.sleep(int(sys.argv[1]))
```

Listing 5.1: Sleep Algorithm

```
# The Computer Language Benchmarks Game
# https://salsa.debian.org/benchmarksgame-team/benchmarksgame/
#
# contributed by Joerg Baumann

from contextlib import closing
from itertools import islice
from os import cpu_count
from sys import argv, stdout

def pixels(y, n, abs):
    range7 = bytearray(range(7))
    pixel_bits = bytearray(128 >> pos for pos in range(8))
    c1 = 2. / float(n)
    c0 = -1.5 + 1j * y * c1 - 1j
    x = 0
    while True:
        pixel = 0
        c = x * c1 + c0
        for pixel_bit in pixel_bits:
            z = c
            for _ in range7:
                for _ in range7:
                    z = z * z + c
                if abs(z) >= 2.: break
```

```
            else:
                pixel += pixel_bit
            c += c1
        yield pixel
        x += 8

def compute_row(p):
    y, n = p

    result = bytearray(islice(pixels(y, n, abs), (n + 7) // 8))
    result[-1] &= 0xff << (8 - n % 8)
    return y, result

def ordered_rows(rows, n):
    order = [None] * n
    i = 0
    j = n
    while i < len(order):
        if j > 0:
            row = next(rows)
            order[row[0]] = row
            j -= 1

        if order[i]:
            yield order[i]
            order[i] = None
            i += 1


def compute_row(p):
    y, n = p

    result = bytearray(islice(pixels(y, n, abs), (n + 7) // 8))
    result[-1] &= 0xff << (8 - n % 8)
    return y, result

def ordered_rows(rows, n):
    order = [None] * n
    i = 0
    j = n
    while i < len(order):
        if j > 0:
            row = next(rows)
            order[row[0]] = row
            j -= 1

        if order[i]:
            yield order[i]
            order[i] = None
            i += 1

```

```
def compute_rows(n, f):
    row_jobs = ((y, n) for y in range(n))

    if cpu_count() < 2:
        yield from map(f, row_jobs)
    else:
        from multiprocessing import Pool
        with Pool() as pool:
            unordered_rows = pool.imap_unordered(f, row_jobs)
            yield from ordered_rows(unordered_rows, n)

def mandelbrot(n):
    write = stdout.buffer.write

    with closing(compute_rows(n, compute_row)) as rows:
        write("P4\n{0} {0}\n".format(n).encode())
        for row in rows:
            pass
            #write(row[1])

if __name__ == '__main__':
    mandelbrot(int(argv[1]))
```

Listing 5.2: Mandelbrot Algorithm

```
import sys
from math import sqrt

def combinations(l):
    result = []
    for x in range(len(l) - 1):
        ls = l[x+1:]
        for y in ls:
            result.append((l[x][0],l[x][1],l[x][2],y[0],y[1],y[2]))
    return result

PI = 3.14159265358979323
SOLAR_MASS = 4 * PI * PI
DAYS_PER_YEAR = 365.24

BODIES = {
    'sun': ([0.0, 0.0, 0.0], [0.0, 0.0, 0.0], SOLAR_MASS),

    'jupiter': ([4.84143144246472090e+00,
                 -1.16032004402742839e+00,
                 -1.03622044471123109e-01],
                [1.66007664274403694e-03 * DAYS_PER_YEAR,
                 7.69901118419740425e-03 * DAYS_PER_YEAR,
                 -6.90460016972063023e-05 * DAYS_PER_YEAR],
                9.54791938424326609e-04 * SOLAR_MASS),

```

```
    'saturn': ([8.34336671824457987e+00,
                4.12479856412430479e+00,
                -4.03523417114321381e-01],
               [-2.76742510726862411e-03 * DAYS_PER_YEAR,
                4.99852801234917238e-03 * DAYS_PER_YEAR,
                2.30417297573763929e-05 * DAYS_PER_YEAR],
               2.85885980666130812e-04 * SOLAR_MASS),

    'uranus': ([1.28943695621391310e+01,
                -1.51111514016986312e+01,
                -2.23307578892655734e-01],
               [2.96460137564761618e-03 * DAYS_PER_YEAR,
                2.37847173959480950e-03 * DAYS_PER_YEAR,
                -2.96589568540237556e-05 * DAYS_PER_YEAR],
               4.36624404335156298e-05 * SOLAR_MASS),

    'neptune': ([1.53796971148509165e+01,
                 -2.59193146099879641e+01,
                 1.79258772950371181e-01],
                [2.68067772490389322e-03 * DAYS_PER_YEAR,
                 1.62824170038242295e-03 * DAYS_PER_YEAR,
                 -9.51592254519715870e-05 * DAYS_PER_YEAR],
                5.15138902046611451e-05 * SOLAR_MASS) }

SYSTEM = tuple(BODIES.values())
PAIRS = tuple(combinations(SYSTEM))

def advance(dt, n, bodies=SYSTEM, pairs=PAIRS):
    for i in range(n):
        for ([x1, y1, z1], v1, m1, [x2, y2, z2], v2, m2) in pairs:
            dx = x1 - x2
            dy = y1 - y2
            dz = z1 - z2
            dist = sqrt(dx * dx + dy * dy + dz * dz);
            mag = dt / (dist*dist*dist)
            b1m = m1 * mag
            b2m = m2 * mag
            v1[0] -= dx * b2m
            v1[1] -= dy * b2m
            v1[2] -= dz * b2m
            v2[2] += dz * b1m
            v2[1] += dy * b1m
            v2[0] += dx * b1m
        for (r, [vx, vy, vz], m) in bodies:
            r[0] += dt * vx
            r[1] += dt * vy
            r[2] += dt * vz

def report_energy(bodies=SYSTEM, pairs=PAIRS, e=0.0):
    for ((x1, y1, z1), v1, m1, (x2, y2, z2), v2, m2) in pairs:
        dx = x1 - x2
```

```
        dy = y1 - y2
        dz = z1 - z2
        e -= (m1 * m2) / ((dx * dx + dy * dy + dz * dz) ** 0.5)
    for (r, [vx, vy, vz], m) in bodies:
        e += m * (vx * vx + vy * vy + vz * vz) / 2.
    print("%.9f" % e)

def offset_momentum(ref, bodies=SYSTEM, px=0.0, py=0.0, pz=0.0):
    for (r, [vx, vy, vz], m) in bodies:
        px -= vx * m
        py -= vy * m
        pz -= vz * m
    (r, v, m) = ref
    v[0] = px / m
    v[1] = py / m
    v[2] = pz / m

def main(n, ref='sun'):
    offset_momentum(BODIES[ref])
    report_energy()
    advance(0.01, n)
    report_energy()

if __name__ == '__main__':
    main(int(sys.argv[1]))
```

Listing 5.3: NBody Algorithm

```
# The Computer Language Benchmarks Game
# https://salsa.debian.org/benchmarksgame-team/benchmarksgame/
#
# contributed by Antoine Pitrou
# modified by Dominique Wahli
# modified by Heinrich Acker
# 2to3
# *reset*

import sys

def make_tree(depth):
    if not depth: return None, None
    depth -= 1
    return make_tree(depth), make_tree(depth)

def check_tree(node):
    (left, right) = node
    if not left: return 1
    return 1 + check_tree(left) + check_tree(right)

min_depth = 4
max_depth = max(min_depth + 2, int(sys.argv[1]))
```

```
stretch_depth = max_depth + 1

print("stretch tree of depth %d\t check:" %
    stretch_depth, check_tree(make_tree(stretch_depth)))

long_lived_tree = make_tree(max_depth)

iterations = 2**max_depth

for depth in range(min_depth, stretch_depth, 2):

    check = 0
    for i in range(1, iterations + 1):
        check += check_tree(make_tree(depth))

    print("%d\t trees of depth %d\t check:" % (iterations, depth),
    check)
    iterations //= 4

print("long lived tree of depth %d\t check:" %
    max_depth, check_tree(long_lived_tree))
```

Listing 5.4: BTrees Algorithm

```
import sys
import pandas as pd
import polars as pl

data_URL = "https://raw.githubusercontent.com/keitazoumana/
    Experimentation-Data/main/diabetes.csv"
original_data = pd.read_csv(data_URL)

amount = int(sys.argv[1])
# Duplicated each row [amount] times
benchmarking_df = original_data.loc[original_data.index.repeat(amount)
    ]
benchmarking_df.drop(['Outcome'], axis=1, inplace=True)
file_name = "benchmarking_data.csv"
# Save the final benchmarking data
benchmarking_df.to_csv(file_name, index=False)


def read_csv_with_time(library_name, file_name):

    final_time = 0

    if library_name.lower() == 'polars':
        df = pl.read_csv(file_name)
    elif library_name.lower() == 'pandas':
        df = pd.read_csv(file_name)
    else:
```

```
        raise ValueError("Invalid library name. Must be 'polars', 'pandas', 'vaex', or 'datatable'")


    return {"library": library_name, "execution_time": final_time}

pandas_time = read_csv_with_time('pandas', file_name)
polars_time = read_csv_with_time('polars', file_name)

def group_data_with_time(library_name, df, column_name='Pregnancies'):

    final_time=0


    if library_name.lower() == 'polars':
        df_grouped = df.groupby(column_name).first()
    elif library_name.lower() == 'pandas':
        df_grouped = df.groupby(column_name)
    else:
        raise ValueError("Invalid library name. Must be 'polars', 'vaex', or 'datatable'")



    return {"library": library_name, "execution_time": final_time}

pandas_df = pd.read_csv(file_name)
polars_df = pl.read_csv(file_name)

pandas_time = group_data_with_time('pandas', pandas_df)
polars_time = group_data_with_time('polars', polars_df)

def sort_data_with_time(library_name, df, column_name='Pregnancies'):

    final_time = 0

    if library_name.lower() == 'polars':
        df_sorted = df.sort(column_name)
    elif library_name.lower() == 'pandas':
        df_sorted = pd.DataFrame(df).sort_values(column_name)
    else:
        raise ValueError("Invalid library name. Must be 'polars', 'vaex', 'datatable', or 'pandas'")


    return {"library": library_name, "execution_time": final_time}


pandas_time = sort_data_with_time('pandas', pandas_df)
polars_time = sort_data_with_time('polars', polars_df)

```

```
def offload_data_with_time(library_name, df):
    final_time = 0


    if library_name.lower() == 'polars':
        array = df.to_numpy()
    elif library_name.lower() == 'pandas':
        array = pd.DataFrame(df).values
    else:
        raise ValueError("Invalid library name. Must be 'polars', 'vaex', 'datatable', or 'pandas'")



    return {"library": library_name, "execution_time": final_time}


pandas_time = offload_data_with_time('pandas', pandas_df)
polars_time = offload_data_with_time('polars', polars_df)
```

Listing 5.5: Dataframe Benchmark Algorithm

Bibliography

[1] *Benchmarking High-Performance pandas Alternatives*. URL: https://www.datacamp.com/tutorial/benchmarking-high-performance-pandas-alternatives.

[2] Kwame Chan-Jong-Chu et al. "Investigating the Correlation between Performance Scores and Energy Consumption of Mobile Web Apps". In: *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*. EASE '20. Trondheim, Norway: Association for Computing Machinery, 2020, pp. 190–199. ISBN: 9781450377317. DOI: 10.1145/3383219.3383239. URL: https://doi.org/10.1145/3383219.3383239.

[3] Luis Corral et al. "Can execution time describe accurately the energy consumption of mobile apps? an experiment in Android". In: *Proceedings of the 3rd International Workshop on Green and Sustainable Software*. GREENS 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 31–37. ISBN: 9781450328449. DOI: 10.1145/2593743.2593748. URL: https://doi.org/10.1145/2593743.2593748.

[4] *Costs of a Data Center*. URL: https://www.kio.tech/en-us/blog/data-center/costs-of-a-data-center.

[5] Waltenegus Dargie. "A Stochastic Model for Estimating the Power Consumption of a Processor". In: *IEEE Transactions on Computers* 64 (Apr. 2014). DOI: 10.1109/TC.2014.2315629.

[6] *Data Centre Costs*. URL: https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks.

[7] *Data Centres and Data Transmission Networks*. URL: https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks.

[8] *Fact file: Computing is using more energy than ever.* URL: https://frontiergroup.org/resources/fact-file-computing-is-using-more-energy-than-ever/.

[9] Philipp Gschwandtner et al. "Modeling CPU Energy Consumption of HPC Applications on the IBM POWER7". In: *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2014, pp. 536–543. DOI: 10.1109/PDP.2014.112.

[10] *How scaphandre computes per process power consumption*. URL: https://hubblo-org.github.io/scaphandre-documentation/explanations/how-scaph-computes-per-process-power-consumption.html.

[11] *Systemd Wiki*.

[12] Henry Hoffmann Huazhe Zhang. *A Quantitative Evaluation of the RAPL Power Control System*. URL: https://newtraell.cs.uchicago.edu/files/tr_authentic/TR-2014-11.pdf.

[13] *Intel Power Counter Monitor*. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/performance-counter-monitor.html.

[14] *Intel Power Gadget*. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html.

[15] Kashif Nizam Khan et al. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3.2 (Mar. 2018). ISSN: 2376-3639. DOI: 10.1145/3177754. URL: https://doi.org/10.1145/3177754.

[16] Kashif Nizam Khan et al. *RAPL in Action: Experiences in Using RAPL for Power Measurements*. URL: https://dl.acm.org/doi/10.1145/3177754.

[17] Jonathan Koomey et al. "Implications of Historical Trends in the Electrical Efficiency of Computing". In: *Annals of the History of Computing, IEEE* 33 (Apr. 2011), pp. 46–54. DOI: 10.1109/MAHC.2010.28.

[18] Gabriele Lanaro. "Python High Performance". In: second. Packt Publishing, 2017, p. 7.

[19] *Linux man page time(1)*. URL: https://man7.org/linux/man-pages/man1/time.1.html.

[20] *Linux man page time(7)*. URL: https://man7.org/linux/man-pages/man7/time.7.html.

[21] *Linux Power Cap*. URL: https://www.kernel.org/doc/html/latest/admin-guide/pm/powercap.html.

[22] *Linux Standard Output*. URL: https://www.putorius.net/linux-io-file-descriptors-and-redirection.html#standard-output-stdout.

[23] *NumPy Org*. URL: https://numpy.org/.

[24] Candy Pang et al. "What Do Programmers Know about Software Energy Consumption?" In: *IEEE Software* 33.3 (2016), pp. 83–89. DOI: 10.1109/MS.2015.83.

[25] Rui Pereira et al. "Energy efficiency across programming languages: how do energy, time, and memory relate?" In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*. SLE 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 256–267. ISBN: 9781450355254. DOI: 10.1145/3136014.3136031. URL: https://doi-org.zorac.aub.aau.dk/10.1145/3136014.3136031.

[26] Gustavo Pinto, Fernando Castor, and Yu David Liu. "Understanding energy behaviors of thread management constructs". In: *SIGPLAN Not.* 49.10 (Oct. 2014), pp. 345–360. ISSN: 0362-1340. DOI: 10.1145/2714064.2660235. URL: https://doi-org.zorac.aub.aau.dk/10.1145/2714064.2660235.

[27] *Profiling Techniques*. URL: https://docs.microsoft.com/en-us/visualstudio/profiling/profiling-techniques?view=vs-2022.

[28] *Python Built-in Functions*. URL: https://github.com/python/cpython/blob/v3.12.0/Python/bltinmodule.c#L2006.

[29] *Python Disassembler*. URL: https://docs.python.org/3/library/dis.html#analysis-functions.

[30] *Python Logging*. URL: https://docs.python.org/3/library/logging.html.

[31] *Python Profiler*. URL: https://docs.python.org/3/library/profile.html.

[32] *Python Sys Settrace*. URL: https://docs.python.org/3/library/sys.html#sys.settrace.

[33] *Python Time*. URL: https://docs.python.org/3/library/time.html#module-time.

[34] *Python Time PEP*. URL: https://peps.python.org/pep-0564/.

[35] *RAPL Interface*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/running-average-power-limit-energy-reporting.html.

[36] *Scaphandre Documentation*. URL: https://hubblo-org.github.io/scaphandre-documentation/index.html.

[37] Yakun Sophia Shao and David Brooks. "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor". In: *International Symposium on Low Power Electronics and Design (ISLPED)*. 2013, pp. 389–394. DOI: 10.1109/ISLPED.2013.6629328.

[38] *Software-based Power Side-Channel Attacks on x86*. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=9519416.

[39] Vincent M. Weaver Spencer Desrochers Chad Paradis. *A Validation of DRAM RAPL Power Measurements*. URL: https://dl.acm.org/doi/abs/10.1145/2989081.2989088.

[40] *The Computer Language Benchmarks Game*. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[41] *TIOBE Index*. 2024. URL: https://www.tiobe.com/tiobe-index/.

[42] *What Smartphone Buyers Really Want*. URL: https://www.statista.com/chart/5995/the-most-wanted-smartphone-features/.