

# Energy Leak! Profiling Programs for Increased Energy Consumption

Patrick May

Department of Computer Science, College of Wooster

April 24, 2023

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Goals . . . . .	4
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	Energy Consumption . . . . .	5
2.1.1	Between Languages . . . . .	5
2.1.2	Within a Language . . . . .	5
2.1.3	Across Platforms . . . . .	6
2.2	Profiling . . . . .	7
2.2.1	Instrumental Profiling . . . . .	7
2.2.2	Sampling Profilers . . . . .	8
2.2.3	Profile Analysis . . . . .	8
<b>3</b>	<b>Methodology</b>	<b>9</b>
3.1	Existing Tools . . . . .	9
3.1.1	Profilers . . . . .	9
3.1.2	Energy Measurement Software . . . . .	10
3.2	Research Procedure . . . . .	11
<b>4</b>	<b>🐮 CowProf 🐮, a Simple Energy Profiler</b>	<b>11</b>
4.1	Implementation . . . . .	11
4.1.1	Higher-Order Functions . . . . .	12
4.1.2	Metaprogramming . . . . .	12
4.1.3	Interpolation . . . . .	13
4.2	Use . . . . .	14
4.3	Limitations . . . . .	16
4.3.1	System Polling . . . . .	16
4.3.2	Specific Implementation Limitations . . . . .	17
4.3.3	Fringe Concerns . . . . .	18

<b>5</b>	<b>Analysis</b>	<b>19</b>
5.1	Understanding Overhead . . . . .	21
<b>6</b>	<b>Conclusion</b>	<b>22</b>

# 1 Introduction

As technology has advanced, computers and software too have improved in a multitude of ways. When creating new software, developers tend to focus on correctness, speed, and reliability of their code. Additionally, as software has grown, more and more levels of abstraction in code have been created. These abstractions are beneficial to developing broad applications quickly with not significant prior knowledge required prior to beginning, however they also disconnect the developer from what their code is truly *doing*. Memorization of syntax and faith in online solutions could very likely leave room for optimization on the table without the developer being aware of it.

Recently, energy consumption has become a more mainstream concern of developers. This comes from a multitude of reasons. Large business that do their computing in colossal datacenters are facing the issue of rising energy costs to maintain them. Portable technology (laptops, smart phones, wearables, etc.) also have a fundamental concern about energy consumption, as optimizing power draw will increase battery life. An additional interest to us as researchers is the underlying environmental factor, and benefit that could be gained from general reduction in energy used.

## 1.1 Goals

Energy efficiency is a difficult metric to track and understand. In the context of modern day development, the many layers of abstraction also serve as a form of obfuscation: is it code the end developer that has written that is causing energy to be wasted (an energy leak)? Or does it lie someplace else in the development stack, in open source packages, containers, dependencies, etc? As discussed by others, should energy consumption only be the concern of compiler writers, systems scientists, and hardware engineers [10]? Therefore, the main goal of this project is shed light into the inner energy workings of a program. We aim to understand the energy consumption of a program at a end-developer function-level.

## 2 Background

### 2.1 Energy Consumption

Energy consumption in software is a difficult task to fully understand. The first difficulty is of physical measurement. Energy metrics are taken in two different ways: by **multimeter**, which broadly is an external tool that sits between a system and its energy source to accurately measure consumption, or by **software** that is able to query various existing sensors on the CPU, RAM, graphics, and storage units. External measurement is *generally* slightly more accurate, yet significantly more difficult to configure correctly. Internal measurement (through software) is typically quicker to configure for a specific system, however it faces its own difficulties, discussed further in section 2.1.3.

#### 2.1.1 Between Languages

Existing work has shown there to be differences in energy consumption of different programming languages solving the same problem [8]. Their procedure involved taking solutions for 10 existing, well defined programming problems from 27 different programming languages, and run each of them, keeping track of time, memory consumption, and energy consumption. They utilized a software tool to log energy consumption over the time of a process. It was found that energy consumption and energy efficiency are a common connection within a programming language (for example, C tended to execute quickly, with less energy draw, while Perl completed slowly with significant energy draw). *However*, this is not a general rule. Faster execution **does not always** mean less power draw overall.

This leaves space for exploration of energy consumption within a programming language. At a high level, this may mean that it is more energy efficient to refactor fast code to something slower.

#### 2.1.2 Within a Language

Other work by CERN scientists have been working towards understanding energy utilization within a given language, similar to the goals of our project. They were able to create extend an existing profiler, IgProf, to utilize newer x86 Intel Processors **Running Average**

**Power Limit** (RAPL) interface to track energy consumption throughout the lifetime of a process [7]. They discovered there to be a correlation between execution time and energy consumption within their simple tests for single-threaded processes. However, they were not able to establish any correlation between multi-threaded processes. This is a challenge of profiling moreso than measuring energy itself.

### 2.1.3 Across Platforms

In addition to internal and external energy measurement tools, the platform itself influences greatly the results of a given measurement.

- **Architecture** of the hardware. x86 based processors are distinct, and do not overlap with energy tracking tools for ARM based processors.
- **Manufacturer** of the hardware. Intel has some existing, software tools for tracing energy consumption, such as *Intel Power Gadget* [3]. AMD does not have an existing, easily accessible, corporate endorsed energy consumption tool that works the AMD Ryzen 3000 chip that our tester has access to. Energy and voltage sensors used to be more accessible, however AMD removed their original energy measurement software due to its use in exploiting security vulnerabilities [11]. More recent AMD processors are compatible with *AMD  $\mu$ Prof* [1].
- **Operating System** of the testing bench. Various flavors of Mac and Linux are more easily able to access existing linux development tools to trace power consumption, such as powerTop and specific MSRs. Windows has less lightweight options for tracking power consumption.
- **Containment** of the system. Virtual Machines make existing energy tools generally ineffective, as they purposefully are having the code within interact with *virtual* hardware. This means that it is difficult to get underlying hardware sensor information instead of some virtualized counter. Some enterprise grade energy consumption tools exist, however they are primarily focused on consumption in the context of a data center, and also well outside the scope of this project.

## 2.2 Profiling

Profiling is the process of software analysis through taking various measurements throughout the system's execution. Profiling allows development teams to know where to work next to improve their code [5]. Frequently, profiling is done from an efficiency perspective. Within efficiency profiling, measurements of time will be taken throughout the software's lifetime at various key points within the process. Once profiling execution is complete, the development team can analyze the software based off of measurements such as how much time was spent in I/O, in computing certain values, in communicating over the network, etc. to determine what subsection of the code should be optimized first.

Profilers fall into two different, broad categories. Similar to energy tracking tools that are either *external* multimeter-esque tools or *internal* software tools, profilers can perform measurements through *instrumentation* or by *sampling*.

### 2.2.1 Instrumental Profiling

Profiling through **instrumentation** occurs when code is *added* to the software being profiled in order to measure the desired metrics [5]. At a very simple level, correctness profiling through instrumentation could be as simple as adding `print()`, `cout <<`, or `console.log()` statements to the code. A more advanced example of profiling through instrumentation would be to wrap every function within within a software project with a call to log sensor data about the state of the program.

While this is a tedious process of injecting profiling code into existing software, many profilers can do this automatically. A real concern with the instrumentation approach is that of introduced overhead and interference. If the granularity of profiling is small (meaning that every step of a program is tracked, down to the fundamental operator level), then the number of additional lines of code could introduce a real risk of influencing the measurements that it is trying to trace to begin with [5].

```

fn func_to_profile(*args, **kwargs){
    // "instrumental" code that takes and log metrics
    datafile.log(timestamp,
        measure1(),
        measure2(),
        measure3(),
    )
    // pre-existing code of function here
    ...
}

```

Figure 1: Profiling by Instrumentation

### 2.2.2 Sampling Profilers

The other broad approach to profiling is through the use of **sampling**. Sometimes called **statistical profiling**, a sampling profiler exists outside of the software it is trying to rely on. Instead, it uses the operating system to periodically interrupt the CPU to take measurements of the desired underlying metrics. The benefit of profiling by sample is that it is less invasive to the software, instead allowing it to execute "normally" within the view of itself, with no internal modifications. The downside of profiling through external sampling is that the measurements are at best an approximation. The exact data found will not be as accurate as the same program that is profiled by instrumentation. [5]

### 2.2.3 Profile Analysis

After a successful profiling of a software system, the next step is to analyze the results. Ideally, **bottlenecks** will be apparent. If one were profiling for speed, a bottleneck could be a slow algorithm that takes the majority of the time of the overall program's execution time. Then the development team focus on that specific module/portion of the overall software to gain the most benefit. For efficient energy profiling, bottlenecks will appear as portions of code that cause an uncharacteristically large influx of power draw. The scope our research will end at the discovery of bottlenecks.



## 3 Methodology

Software energy measurement tools exist, and, separately, instrumental and statistical profilers exist. To gain insight into the inner workings of a program, such as at an individual method level, profiling through instrumentation is superior because of its higher specificity. Provided the timeline of this project, acquiring an external energy meter and configuring to work with a given system will also introduce needless complexity to the process.

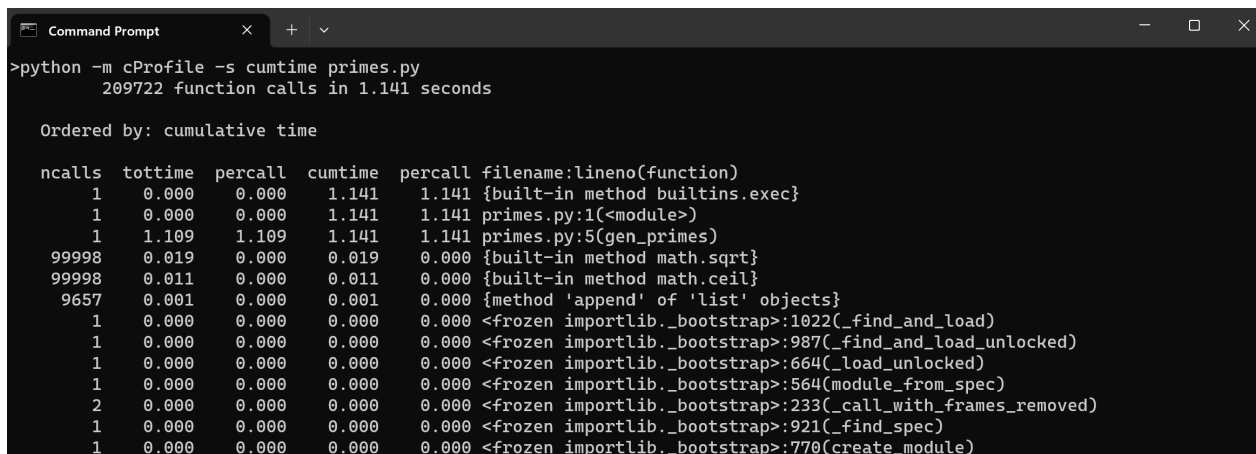
### 3.1 Existing Tools

#### 3.1.1 Profilers

Many programming languages have their own existing profilers that work through instrumentation. Python, C, Go, Rust, and Java are a few examples of built-in language profilers [9]. (c)Python can be profiled for time using built-in modules. For an existing python project, this can be done with the following:

```
$> python -m cProfile script_to_profile.py
```

Which produces output along the lines of the following (sorted for clarity):



```
Command Prompt
>python -m cProfile -s cumtime primes.py
209722 function calls in 1.141 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      0.000    0.000    1.141    1.141 {built-in method builtins.exec}
1      0.000    0.000    1.141    1.141 primes.py:1(<module>)
1      1.109    1.109    1.141    1.141 primes.py:5(gen_primes)
99998   0.019    0.000    0.019    0.000 {built-in method math.sqrt}
99998   0.011    0.000    0.011    0.000 {built-in method math.ceil}
9657    0.001    0.000    0.001    0.000 {method 'append' of 'list' objects}
1      0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:1022(_find_and_load)
1      0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:987(_find_and_load_unlocked)
1      0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:664(_load_unlocked)
1      0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:564(module_from_spec)
2      0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:233(_call_with_frames_removed)
1      0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:921(_find_spec)
1      0.000    0.000    0.000    0.000 <frozen importlib._bootstrap>:770(create_module)
```

Figure 2: Profiling with cProfile Module

which will return data about times when within a function, the number of function calls, etc. In Figure 2, a simple, naive prime number generator is used as the program to be

profiled. We see that in generating prime numbers from Refer to Appendix A for code examples. A standard profiler will provide information about overall time spent within each function in a script. **However**, the most accessible language profilers condense all function usages into a “cumulative time within function” value, which means the profiler obfuscates the interval data. While we will know we spent 50% of execution time in `foo()`, we will not know if this 50% was all from a call at the beginning of the script, or `foo()` was called 10 times, with each execution taking 5%. This is not ideal when trying to interweave energy utilization data, because energy readings are of the form *Voltage* throughout *time*.

### 3.1.2 Energy Measurement Software

Intel’s Power Gadget is a GUI interface that tracks multiple power metrics of the system it is running within when it begins. Additionally, it has a CLI tool, *PowerLog3.0*, that can be piped a command to execute. It will automatically start and stop logging the system metrics when the subcommand begins and terminates.

```
$>PowerLog.exe
  -resolution <msec> {msec between sensor queries}
  -verbose {extra information}
  -file <logfile> {output file of command}
  -cmd {command to evaluate here}
```

Figure 3: Using Intel’s PowerLog3.0 [3]

After a PowerLog command completes, a large csv file is created that contains various power readings from intel sensors. For every time interval since a command starts, an entry in the csv will have a measurement of CPU Utilization, CPU Frequency, Processor Power, GPU Power, etc. Like other power measurement tools, *Power Log* is only able to sense the various states and draw-rates of the entire system.

Various Linux utilities are also able to accomplish similar data results to Intel’s PowerLog, however many of the tools such as **turbostat** and **powerTOP** are more focused on diagnostic style snapshots, instead of information throughout a program’s execution.

## 3.2 Research Procedure

The goal of this research is to verify that our energy profiling tool works, identify limitations, and investigate differences within a specific programming language.

These questions are answered through running various tests through CowProf. The testing programs generally fall into three different categories:

- **Stress Tests** aim to observe how much a program influences the testing system's energy as a whole. Some examples would be computationally stressful programs - generating all prime numbers within a large range, sorting a large list of numbers, etc.
- **Practical Tests** aim to see if differences appear in energy consumption throughout a program's execution. For example, would less energy be required to await the return of an API call then to parse the 1 GB of returned `json`.
- **Improvement Tests** aim to compare code snippets that produce the same output, but achieve it differently, to determine if one method is more energy efficient than another. For example, comparing different sorting algorithms on the same input, or utilizing a library function compared to a self-written solution.

The profiler and power-logging tool create two separate files containing information about the current working point within the program and the energy draw of the system, respectively. This data is then aligned and interwoven to paint a more explicit picture of the location of a program and its energy consumption at various points in its lifecycle.

## 4 🐮 CowProf 🐮, a Simple Energy Profiler

### 4.1 Implementation

CowProf is our energy profiler, built off of Intel's *PowerLog 3.0* CLI tool, language specific wrappings (currently we have created C++ and Python wrappers), data analysis using `polars`, glued together into a python script and command line tool.

CowProf is built to work on modern Intel x86 devices (Sandybridge or later) and a Windows operating system. The most important requirement to successfully running CowProf

is that *Intel Power Gadget* is installed and that the path to the bundled CLI tool, *Intel PowerLog* is found and edited in the `cowprof.py` script.

The **Power Consumption** statistics are provided by Intel’s PowerLog tool, and dumped into a csv file that is titled the current date/time stamp as of starting the script.

The **Temporal Data** that is provided through *injective profiling* is created through adding wrapper code to the programs that the developer desires to test. Within python, this requires minimal end-developer overhead through the use of decorators, a pythonic form of **higher order functions**. Our C++ wrapper requires an instance of a CowProf class wherever profiling should begin and a `finish()` call wherever that opened timestamp should terminate. The temporal data is then dumped into a different csv file containing function interval data.

#### 4.1.1 Higher-Order Functions

Are a construct within various programming languages. They are “higher order” in that their input is other functions, instead of static data. In Python, higher order functions are easily accessible with *decorators*. While in C, higher-order functions are accessible with *void pointers*. Our specific C++ wrapper implementation did not utilize higher order functions, instead focusing on a truly primitive time stamping formula.

Regardless of any specific programming language’s wrapper implementation, all wrappers need to have their output path changed prior to program execution. This is done with very some simple **metaprogramming** techniques that ensure temporal data is output to the correct location.

#### 4.1.2 Metaprogramming

Metaprogramming, while sounding complex, is simply using a program to generate other programs [6]. Some forms of metaprogramming exist internal to a language, however for CowProf, we stepped outside of any individual language entirely.

The problem was that although the each wrapper does not change how it works, every single time the CowProf Script executes, we need to rewrite where the output data is being

dumped. While this could likely be done with output redirection, we employed this meta-programming instead to avoid standard out pollution.

Prior to profiling the desired project, CowProf rewrites the entire wrapper for the language of the project in question. Then the entire project is either compiled and then executed (or simply executed for interpreted languages).

As the tested project is running, during its own execution it is reporting timelogs for its location, and additionally *PowerLog3.0* is reporting powerlogs throughout the command's lifetime.

### 4.1.3 Interpolation

Upon the completion of the test project, the {Time, Power}Log paths are piped to an analysis script for wrangling and visualization. The most significant step prior to constructing the visualization is to normalize all temporal intervals from `system time` (number of seconds since 01/01/1970) to cumulative process time (i.e. the first timestamp is 0s).

This allows Power data to be aligned with Time data based off of their aligned 'system time since program beginning'.

Next, various simple mapping and hashing tools are used to identify repeated function calls. Then along the shared time axis, map both the wattage consumption of the system and an interval of each function.

When run on something with multiple recursive calls, such as a naive Fibonacci calculation of the 13th Fibonacci number, we get an output like the following:

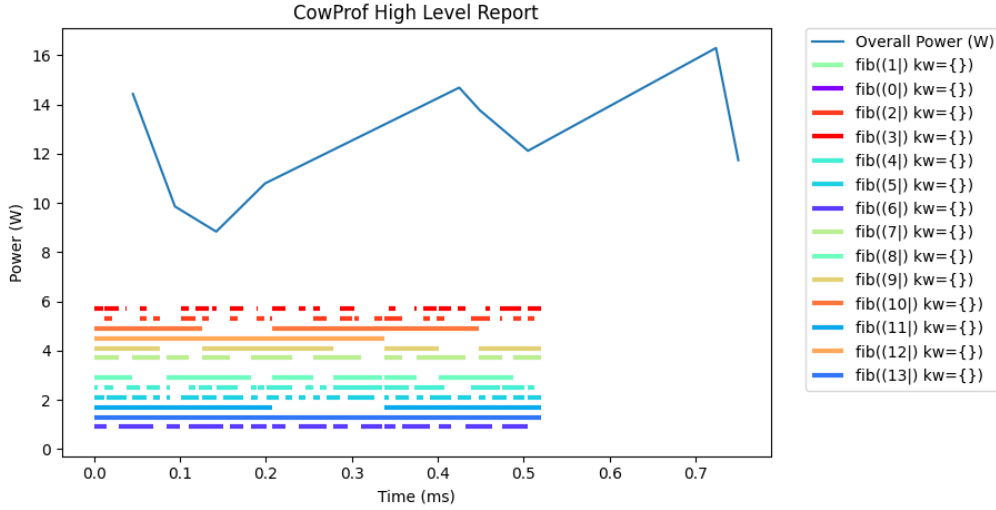


Figure 4: CowProf of Naive-Fibonacci(13)

The intervals do not last the entire program lifetime due to pieces of the program not being tracked by CowProf currently, and hidden timings that are not traceable at this high-level of profiling. For further information on usage and interpreting CowProf output, refer to section 4.2 for further specifics.

## 4.2 Use

CowProfiler is currently configured to work within a folder hierarchy similar to that of what it is working is currently.

To profile a single-file python script, place the python file in the `stress-tests\python\` folder. Second, follow the python folder's README instructions on how to 'inject' wrapper code into the desired program to profile.

Then, a command invoking the script, with at least the `-t` flag invoked, which is the python files you want to pass to the profiler.

The command line tool has the following flags for your convenience:

```

$>python cowprof.py --help
-o <OUTFILE> {What to title output file}
-ctype {CMDTYPE, i.e. python, c++, etc.}
-r <RESOLUTION> {Set time between powerlog writes}
-v, --verbose {output extra powerlog information}
-t INFILE {A string of the command to {compile, run}}

```

Figure 5: CowProf Argument Flags

At its most abstract, CowProf functions along the lines of Figure 6. It has been designed to require as little end-developer tinkering as possible. The Programmer's efforts are highlighted in blue, while CowProf's internal steps are colored green.

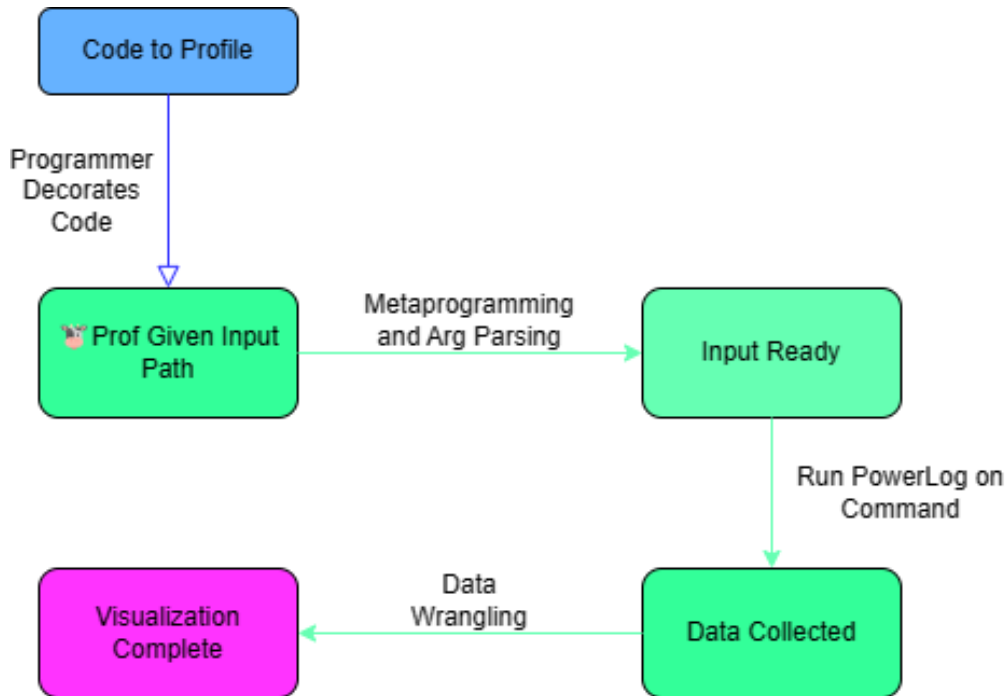


Figure 6: Abstracted Steps of CowProfiler

## 4.3 Limitations

CowProf is an imperfect tool, with numerous flaws or sub-optimal design decisions that are justified within the scope of this being an undergraduate single-semester project.

### 4.3.1 System Polling

Power sensors can only measure the energy being used by the *entire* CPU. It is (unfortunately) physically impossible to specifically identify what proportion of energy is attributable to each process the CPU is currently running. Instead, the energy data will provide a snapshot of the entire system at specified points in time.

Thus, when running this program, it is important to minimize intensive programs that may be running in the background. For example, Figure 7 shows CowProf analyzing the exact same script, except one of them our tester was streaming a Youtube video in the background and going to various different web-pages.

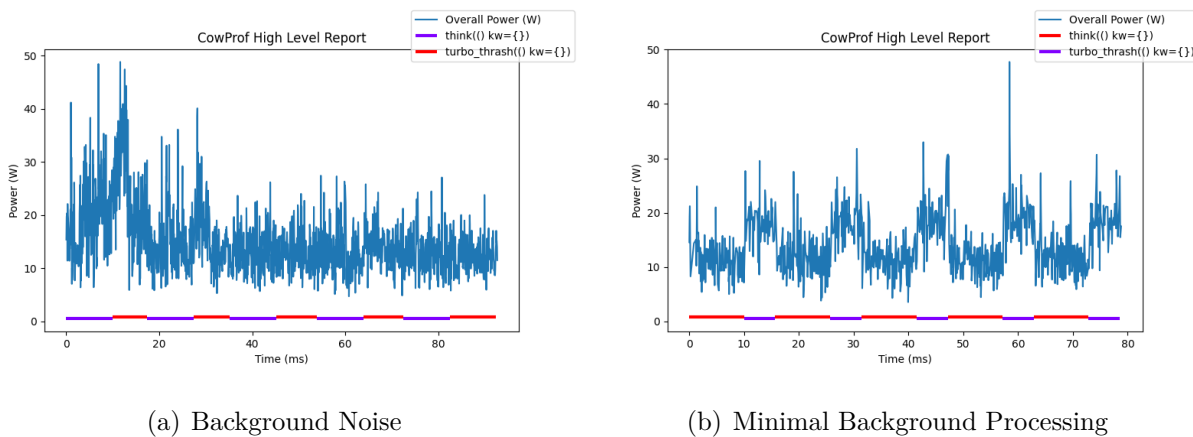


Figure 7: Cost of Background Programs

In this above example, CowProf is analyzing a simplified `mixed.py` script that either thrashes or sleeps, so we should see very clear energy consumption increases and decreases during thrashing or sleep time. However, with background interference in the form of complex other software running, CowProf loses the ability to nicely demonstrate energy utilization at a function call level.

This is a limitation with any energy consumption analysis, as are unable attribute all



energy readings to a specific program with absolute certainty, especially since this testing was done on Windows and its suite of random surprise background software updates. However, for the remainder of this work, anytime CowProfiler was used, all intensive applications were closed manually before running the profiler.

We briefly investigated if there was additional CPU cost to executing the script from a VSCode integrated terminal compared to a plain CMD prompt, however no significant overhead was found and thus we disregarded this concern.

### 4.3.2 Specific Implementation Limitations

While CowProfiler does provide relatively easy access to the underlying TimeLog and PowerLog dataframes, they are not the automatic end result of the script. We instead chose to focus on a visualization over time. For functions with a very high amount of function calls, a different approach to presenting function intervals is necessary, as currently CowProfiler will end up covering the energy data with timing intervals of unique function calls. Displaying so many individual datapoints both abuses the underlying `Matplotlib` library and also obliterates immediate value of the visualization. For an example of a visualization with 20 distinct function calls, refer to Figure 8. If CowProf were to attempt to profile something with multiple hundred unique functions, it would quickly get bogged down in trying to render all the different function intervals on a graph.

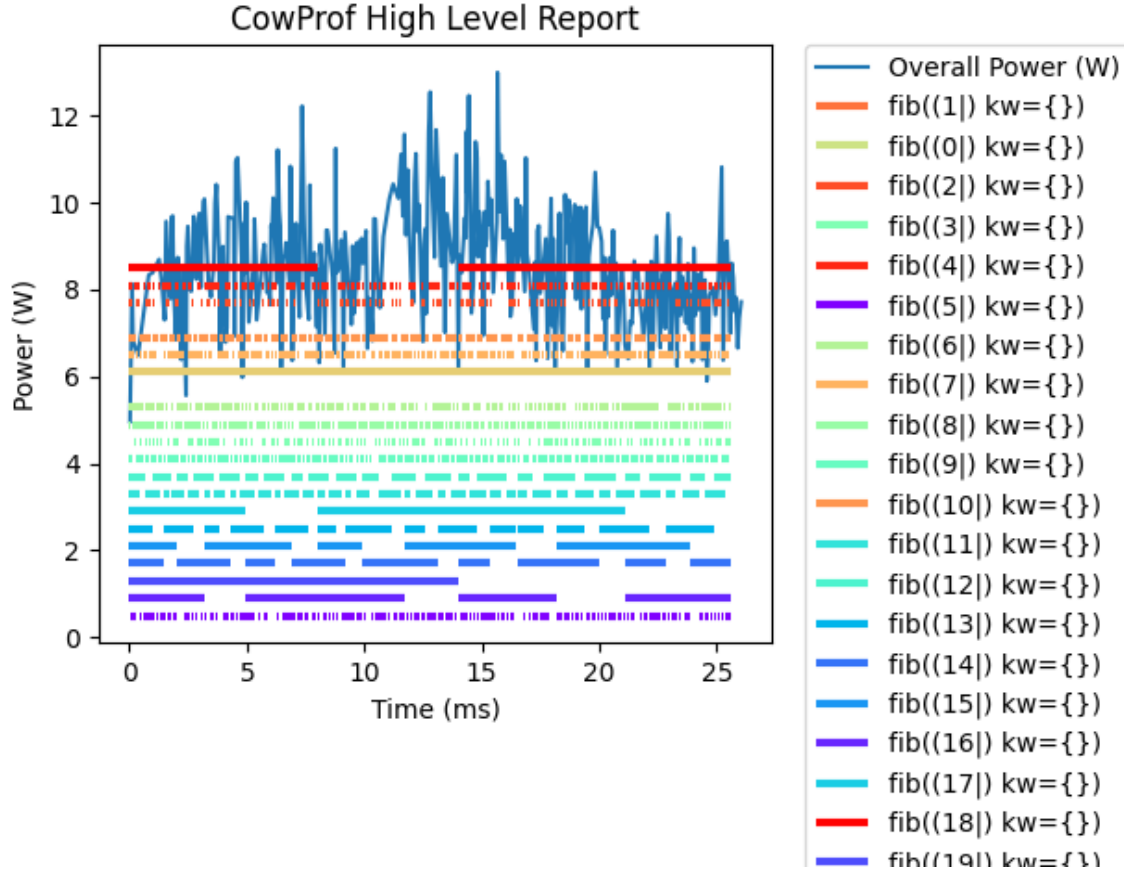


Figure 8: Too Many Distinct Function Calls

For our testing, we stuck to simple programs with ideally 2 to at most 10 unique function calls. The same function with the same arguments can be called repeatedly just fine, as it will just appear as another horizontal bar on its already reserved space in the visualization.

### 4.3.3 Fringe Concerns

As CowProf is, at its heart, a dynamic analysis tool, there are a few additional liabilities that we feel important to mention when using.

When repeatedly running the same program with the same inputs (such as in a testing scenario), it is entirely possible that the **same code** will perform **better** subsequent times it is ran. This could be attributable to many different things, typically because of some form of cache (branch prediction, level 0, etc.)

When comparing different tests that are trying to produce the same output with different

approaches, simpler approaches does not always mean faster code execution. Nor does faster execution mean that the least amount of energy will be consumed in that solution [8]. This is because of the vagueness of a word such as “simple”. For a computer, wouldn’t individual vectorized instructions be the simplest to compute? But for a human, the simplest may be the program with the least amount of characters. Even this is not always the best approach, as fancy optimization techniques such as assembly language loop unwrapping can make “more complex” code faster and/or less energy intensive. [8]

## 5 Analysis

As CowProf developed, we first wanted to heuristically verify that it was functioning. To do this, we first tested with a simple python script, that alternates between sleeping for 10 seconds (`think()`) and spinning 16 threads that each count to 10 million (`thrash()`). When decorated and run through CowProf, our analysis graph should show higher energy consumption while thrashing and lower energy consumption while sleeping.

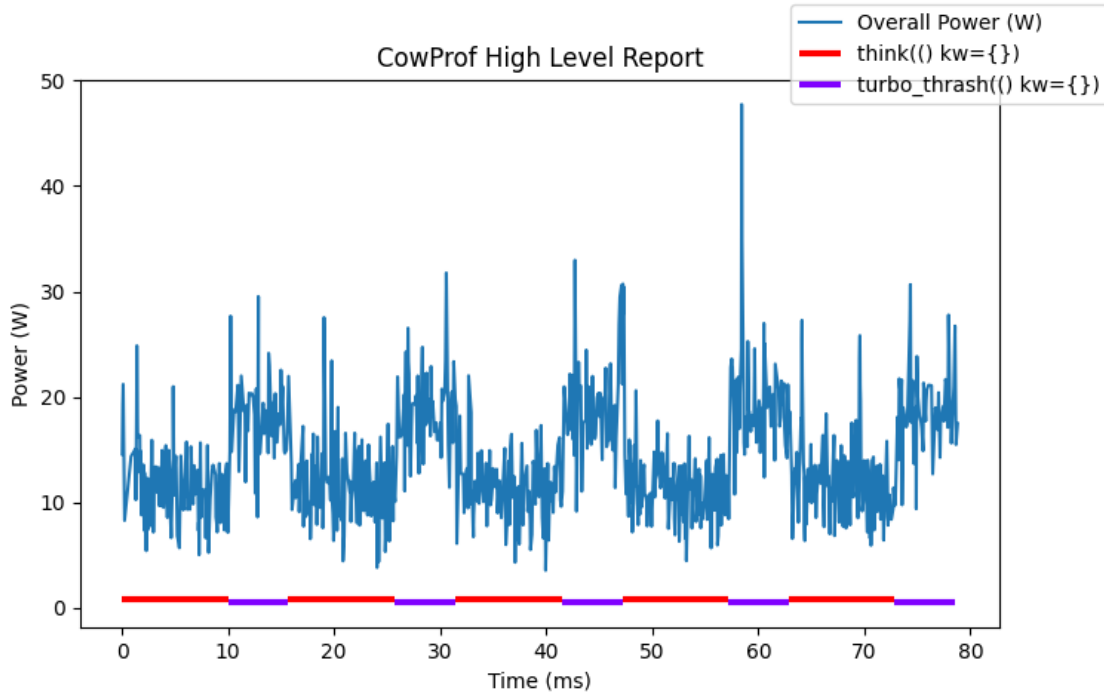


Figure 9: Simple Thrash/Think Sanity Check

Looking at Figure 9, this *is* the kind of energy consumption output that we want. Without statistical rigor, we find that CowProf is able to measure changes in energy consumption of a program.

Next, CowProf was used to analyze a few different sorting algorithms and see if they were more or less energy efficient.

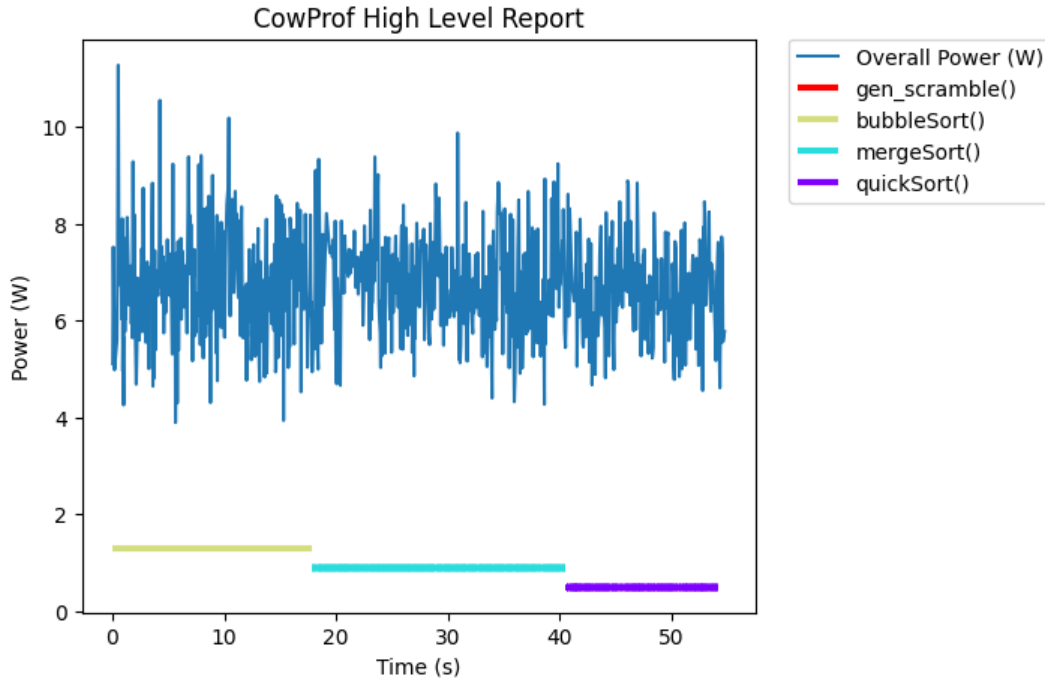


Figure 10: CowProf on Sorting Algorithms

In Figure 10, we see that **Bubble**, **Quick**, and **Merge** sort all seem to have similar energy consumption, with no significant increases or decreases compared to the visible steps within a thrashing workload.

Interestingly, all sorts investigated seemed to have similar running time, which at first seems counter intuitive based on their different asymptotic run-times [2]. However, we have to be careful about the **overhead** introduced by CowProfiler’s wrapper code. If the wrapper is called a significant number of times, those additional epoch and I/O calls do become nontrivial. Especially if bubbleSort only had one wrapper call, while mergeSort and quickSort performed CowProf wrapper functions after every recursive call.

After refactoring the sorting testcode, we arrived to a solution that seems much more

asymptotically accurate:

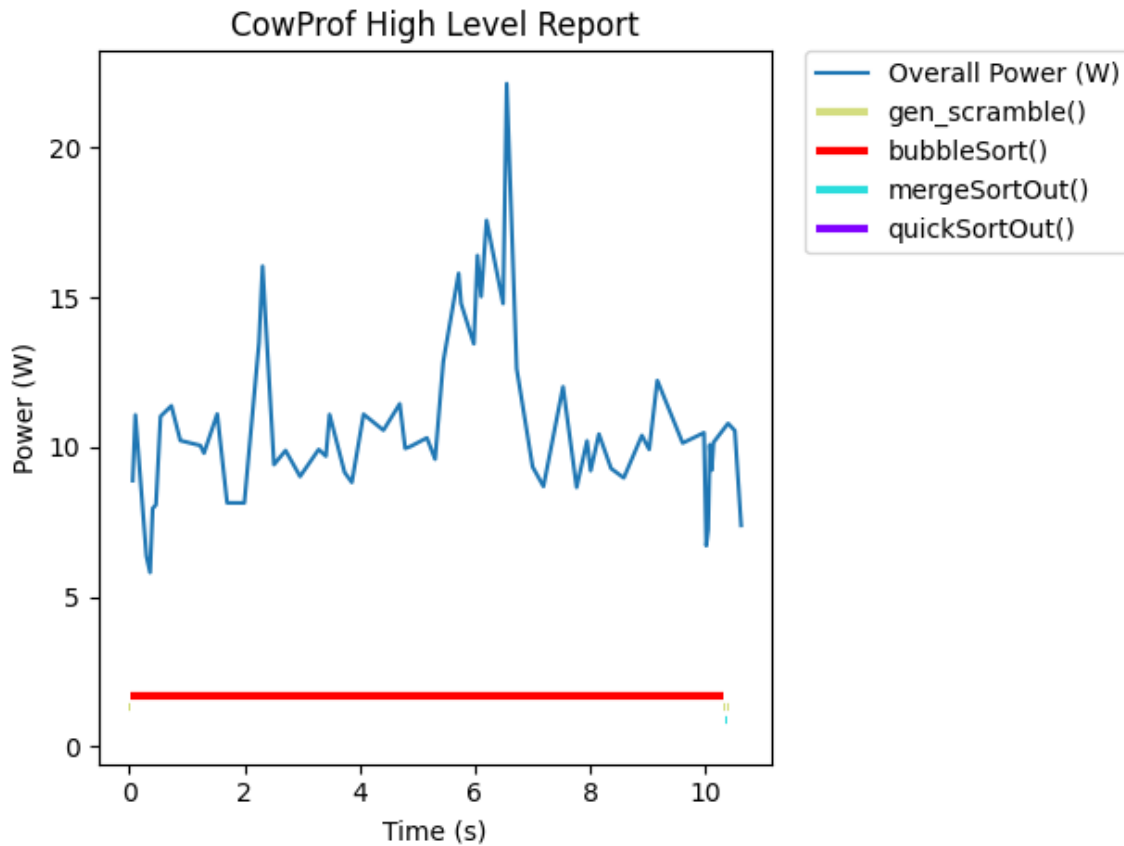


Figure 11: Reduced Overhead Sorting Refactor

## 5.1 Understanding Overhead

As discussed in section 2.2.1, instrumental profiling injects overhead into the underlying software project. As CowProfiler is a simplified instrumental profiler we must be wary of the additional code injected messing with the readings we are trying to acquire. To do this, we need to look at the ‘cost’ of a wrapper call compared to the underlying code it is surrounding.

Without getting bogged down in details, CowProf’s wrappers essentially timestamp when a function has started, timestamp when a function has ended and write function & timestamp information to a data file.

Getting Unix epoch time is an extremely quick operation, generally converting to only a

few lines of assembly, regardless of the specific library function being called [4]. Writing to a file is not quite as trivial, however we still consider an individual append to be a constant time operation.

Significant temporal overhead *can* be introduced with excessive CowProf wrapper calls. We discovered this by simply timing the execution of a `naive-fib(20)` with profiler code injected and without profiler code injected.

Func	Time to Complete
Fib(20)	0.00245s
Fib(20) w/ Wrapper	19.04s

Table 1: Temporal Cost of Wrapper Calls

We believe this overhead is introduced because the underlying fibonacci function is extremely quick math compared to the filewriting we are doing. Additionally, throughout the execution of a fibonacci program, for every  $2^n$  `fib()` calls, it is also making  $2^n$  epoch and I/O calls. Compare this to a much comparatively trivial overhead for the program `mixed.py`, where 10 functions are called that either thrash or sleep, allowing for us to observe the overhead introduced with time grabs and file I/O for a program with an unrelated amount of profiling code compared to underlying code.

Func	Time to Complete
Mixed.py	90.00s
Mixed.py w/ Wrapper	97.87s

Table 2: Temporal Cost of Wrapper Calls on Nonlinear I/O

## 6 Conclusion

Through this Junior Independent Study, I have discussed existing types of energy measurement tools, program profilers, and the concept of green software. Additionally, I have shown that a simple energy consumption profiler can be created that can work with multiple different programming languages. I then demonstrated that CowProfiler does work, however

it has limitations that must be understood, in the form of overhead, background processes, and tooling it utilizes.

Power measurement is fickle for computers. An immediate extension of this work is to perform similar tests on other systems with differing architectures, operating systems, and programming languages. We have found that this tool can identify sections of energy intensive code in Python and C++. Another step would be to construct a testing framework to try and investigate if certain ways of doing a programming task are *more energy efficient* and additionally, if *temporal efficiency* is always the most *energy efficient* solution. Beyond that, in the field of dynamic program analysis, optimizations could be made to reduce the overhead of CowProfiler, or switch implementations entirely to a statistical profiler.

## References

- [1] AMD. AMD  $\mu$ Prof. <https://www.amd.com/en/developer/uprof.html>. Accessed: April 24, 2023.
- [2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [3] I. Corporation. Intel Power Gadget. <https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html>. Accessed: April 24, 2023.
- [4] G. GCC. std::chrono assembly decomposition. <https://gcc.gnu.org/legacy-ml/gcc-help/2019-07/msg00068.html>. Accessed: April 24, 2023.
- [5] J. Ingeno. *Software architect's handbook: Become a successful software architect by implementing effective architecture concepts*. Packt, 2018.
- [6] P. Joshi. What is Metaprogramming? Part 2/2. <https://prateekvjoshi.com/2014/04/05/what-is-metaprogramming-part-22/>. Accessed: April 24, 2023.

- [7] K. N. Khan, F. Nybäck, Z. Ou, J. K. Nurminen, T. Niemi, G. Eulisse, P. Elmer, and D. Abdurachmanov. Energy profiling using igprof. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CC-GRID '15, page 1115–1118. IEEE Press, 2015.
- [8] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, and J. Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.
- [9] Python Software Foundation. Python Profiler Documentation. <https://docs.python.org/3/library/profile.html>. Accessed: 01-27-2023.
- [10] C. Sahin, F. Cayci, I. L. M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winblad. Initial explorations on design pattern energy usage. In *Proceedings of the First International Workshop on Green and Sustainable Software*, GREENS '12, page 55–61. IEEE Press, 2012.
- [11] Y. Wang, R. Paccagnella, E. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner. Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In *Proceedings of the USENIX Security Symposium (USENIX)*, 2022.