Tópicos de Desenvolvimento de Software

Simão Cunha, João Saraiva
University of Minho

March 18, 2025

1 Introduction

1.1 Energy Consumption Monitoring

In order to monitor the energy consumption of a given program, we will use Intel's Running Average Power Limiting (RAPL) interface [4]. As described in [9], RAPL provides accurate energy consumption estimates and supports fine-grained power capping [5]. In fact, RAPL has been successfully used to measure and estimate software energy consumption in numerous research studies [7, 6, 8].

We will adopt a different approach than that reported in [2, 7]: using a C program that interfaces with the RAPL C library and executes the target program via a system call while measuring its energy consumption. The primary advantage of this method is that it enables us to compile and run the original programs without requiring the addition of intrusive code to track energy consumption. Additionally, the runtime and energy consumption overhead introduced by using a system call to run the target program is negligible, as reported in [7].

The implemented C program accepts the executable program to run and the number, N, of times the program should be executed as command-line arguments. It outputs a `CSV` file with energy and runtime measurements obtained from each of the specified N executions.

We will use the C main function from [3], which incorporates temperature sensors, enforces power caps and measures the peak memory used by the program during its execution.

1.2 Temperature Sensors

When benchmarking a program, it is essential to execute it multiple times to eliminate outlier measurements, for example. Each execution should also be performed under similar conditions to avoid influencing the results.

When measuring energy consumption, it is crucial to ensure that each program execution starts with the CPU at the same temperature. To maintain these conditions across all executions, the main C program uses the *lm-sensors* library [1] to monitor the CPU temperature. Each execution begins once the CPU reaches the same temperature, which is calibrated before starting the benchmarking process. This calibration defines the CPU temperature in an idle state.

Since different computers may have varying numbers of cores, we consider the average temperature across all cores and monitor it during program execution using the C main function.

[1] Available at `https://github.com/lm-sensors/lm-sensors`

In the provided C main function, the temperature of the CPU cores is first calibrated by sleeping for 30 seconds and saving the results to `core_temperatures.txt`. The average temperature of all cores is then used as a threshold to ensure that the CPU temperature during execution does not exceed this value (with a variance of $\pm 5°C$). If `core_temperatures.txt` exists, the script reads the previously recorded average temperature. Otherwise, it performs the calibration process as described.

1.3 Limiting the Energy Consumption with PowerCap

We can reduce the energy consumed during a task by lowering the power used to perform it. A key feature of our energy measurement framework is its support for power capping.

To effectively limit the energy consumption of the processor during execution, we rely on RAPL's *PowerCap* library [2]. This powerful tool provides fine-grained control over the processor's power usage.

PowerCap operates by continuously monitoring and managing the processor's power consumption in real time. It ensures that the processor's energy usage stays within the specified limits, preventing excessive power draw and potential wastage.

It is important to note that, rather than instrumenting every program in any language, we use a C program that calls the "pure" (non-modified) programs.

1.3.1 Energy Monitoring Main Function

The energy monitoring `main` function serves as the entry point of the program. This function begins by checking the value of `WHATTSCAP`, which represents the limit on the `raplcap` we will be using. If `WHATTSCAP` is equal to -1, then there is no `raplcap` defined, otherwise the `raplcap` will be initialized using the `initializeRapl` function. After the `raplcap` initialization, the function proceeds to execute the `programWorks` which runs the program and verifies its exit status to check if there is any problem in its execution and if not, the measurements are taken in `performMeasurements`. Finally, if RAPL was initialized earlier, then the main function destroys the RAPL configuration using `raplcap_destroy`.

```
1  int main(int argc, char **argv) {
2      ...
3      if (WHATTSCAP != -1) {
4          if(initializeRapl(&rc))
5              return -1;
6      }
7
8      if(programWorks(command)){
9          performMeasurements(command, language, program, ntimes, core);
10     } else {
11         writeErrorMessage(language,program);
12     }
13
14     if (WHATTSCAP != -1) {
15         if (raplcap_destroy(&rc)) {
16             printf("Error destroying CAP\n");
17             perror("raplcap_destroy");
```

[2] Available at `https://github.com/powercap/powercap`

```
        } else {
            printf("Successfully destroyed CAP\n");
        }
    }
    return 0;
}
```

1.3.2 PerformMeasurements Function

The `performMeasurements` function handles the process of performing measurements related to performance metrics and resource consumption. It initializes RAPL for a specific measurement using `rapl_init`. Subsequently, it iterates `ntimes`, performing measurements each time. It checks the temperature and waits if it exceeds a certain threshold, ensuring measurements are taken under stable conditions. Afterward, it begins by performing pre-measurement setup, including RAPL measurements using `rapl_before`. Following this setup, it runs `measureMemoryUsage`, which is the function that executes the program using the Linux `time` command to compute the program memory consumption. It then measures the time taken by the command to execute using `gettimeofday`, recording the elapsed time. Finally, it performs post-measurement cleanup, including RAPL measurements using `rapl_after`. The results are then written into the measurements file.

```
void performMeasurements(...) {
    ...

    rapl_init(core);
    ...

    for (int i = 0; i < ntimes; i++) {
        fprintf(fp, "%s, %s, %d,", language, program, WHATTSCAP);
        temperature = getTemperature();

        while (temperature > (TEMPERATURETHRESHOLD + VARIANCE)) {
            printf("Sleeping\n");
            sleep(1);
            temperature = getTemperature();
        }

        rapl_before(fp, core);

        gettimeofday(&tvb, 0);

        int mem = measureMemoryUsage(command);

        gettimeofday(&tva, 0);
        time_spent =
            ((tva.tv_sec - tvb.tv_sec) * 1000000 + tva.tv_usec - tvb.tv_usec) / 1000;

        rapl_after(fp, core);
        sprintf(str_temp, "%.1f", getTemperature());
        fprintf(fp, "%G, %s, %d\n", time_spent, str_temp, mem);
    }

    ...
```

```
33 }
```

1.4 Obtaining the memory usage by each program

Considering the memory usage of a particular program is crucial for green software. Therefore, to determine the memory usage of the program, we will use the "maximum resident" provided by the Linux `time` command. It refers to the maximum amount of peak memory (in KBytes) used by a process during its execution. It represents the peak memory usage throughout the execution of the command.

1.5 CSV files columns

The following columns will be presented in the CSV file `measurements.csv`:

- Language: Name of the programming language;
- Program: Name of the program;
- PowerLimit: Power cap of the cores (in Watts);
- Package: It measures the energy consumption of the entire socket. It includes the consumption of all the cores, integrated graphics and also the uncore components (last level caches, memory controller);
- Core: the energy consumed by all cores and caches;
- GPU: the energy consumed by the GPU;
- DRAM: this domain estimates the energy consumption of the random access memory (RAM);
- Time: Execution time (measured in milliseconds);
- Temperature: Mean temperature in all cores (in Celsius);
- Memory: Peak memory usage throughout the execution of the command (in KBytes)

1.6 CodeCarbon

CodeCarbon (`https://codecarbon.io/`, accessed: Mar 17, 2025) [1] is a lightweight software package that estimates the carbon footprint associated with computational energy consumption. It supports Windows, macOS, and Linux operating systems, as well as Intel and AMD CPUs and NVIDIA GPUs. CodeCarbon employs a scheduler that, by default, records measurements every 15 seconds, introducing minimal overhead.

For CPU energy consumption, CodeCarbon tracks Intel processor energy usage via the Intel Power Gadget on Windows and macOS. On Linux, it monitors both Intel and AMD processors through Intel RAPL files. The reported energy consumption represents the usage of the entire machine rather than an isolated process.

For GPU energy consumption, CodeCarbon leverages the *pynvml* library (compatible only with NVIDIA GPUs). Similar to CPU tracking, the reported consumption reflects the total energy usage of the machine rather than that of a specific process.

To estimate carbon emissions, CodeCarbon utilizes regional data for the United States and Canada. For other countries, it relies on the national energy mix (i.e., the proportion of energy sourced from carbon, solar, wind, etc.) to calculate carbon intensity. This methodology enhances the accuracy of CodeCarbon's estimates, yielding results that closely align with those obtained from a wattmeter.

More details on CodeCarbon usage can be found at `https://mlco2.github.io/codecarbon/index.html` (accessed: Mar 17, 2025). The tool provides multiple ways to integrate energy tracking in Python. Metrics such as execution time, CPU energy, RAM energy, GPU energy, ..., can be measured using the `EmissionsTracker` (Listing 1). If no Internet connection is available, the `OfflineEmissionsTracker` (Listing 2) should be used. In this case, an ISO country code must be provided to estimate the regional carbon intensity of electricity used.

```
from codecarbon import EmissionsTracker
tracker = EmissionsTracker()
tracker.start()
try:
    # Code under measurement
finally:
    tracker.stop()
```

Figure 1: Example of EmissionsTracker usage

```
from codecarbon import OfflineEmissionsTracker
country_code = None # Define ISO country code
tracker = OfflineEmissionsTracker(country_iso_code=country_code)
tracker.start()
# Code under measurement
tracker.stop()
print(tracker) # Object containing all measurement details
```

Figure 2: Example of OfflineEmissionsTracker usage

2 Exercises

Download the `RAPL_Measurements.zip` file from the previous worksheet and extract its contents. Follow the instructions in the `README.md` file to install the required dependencies and analyze the folder structure.

Once the setup is complete, proceed with the following exercises:

Exercise 1: Powercap

1. Assuming you have completed the previous worksheet on the Fibonacci sequence, consider the two implementations you have done: one unoptimized (recursive) and one optimized (linear).

Present both implementations along with the Fibonacci numbers you have tested. Ensure that you choose relatively high numbers to guarantee substantial computation.

2. Calibrate the CPU temperature of the machine you are using with the RAPL C main function. Identify the temperature of each CPU core and calculate the average CPU temperature after calibration. **Suggestion:** Remove the file `Utils/cores_temperature.txt` if it is present.

3. Calibrate the optimal power cap value. **Suggestion:** Choose a specific Fibonacci number and execute it for a large set of power cap values (e.g., from 2W to 20W). Determine which power cap value corresponds to the lowest energy consumption during execution; this will be your optimal power cap value. From this point on, when we refer to power cap, we will use the value you found. When power cap is not mentioned, assume it is set to -1.

4. Compare the performance in terms of *Speedup*, *Greenup*, and *Powerup* of the non-optimized and optimized versions of the Fibonacci functions under both power cap values.

5. Draw the *GPS-UP Software Energy Efficiency Quadrant Graph* for every execution under both power cap values.

6. Using the provided RAPL C code, measure the energy consumption and execution time of both Fibonacci implementations for all previously tested Fibonacci numbers using the optimal powercap value from Exercise 3. Note that setting the power cap to -1 disables power capping during measurement.

7. Create energy consumption and runtime graphs to illustrate the results.

Exercise 2: CodeCarbon

1. Install CodeCarbon on your own laptop by following the instructions provided at `https://mlco2.github.io/codecarbon/installation.html`. **Suggestion:** install codecarbon v2.4.2

2. Refer to the usage documentation at `https://mlco2.github.io/codecarbon/usage.html` and execute both Fibonacci implementations from Exercise 1 and the previous worksheet using CodeCarbon. Print relevant metrics collected by CodeCarbon through the `tracker` object. **Suggestion:** Use `os.system()` to execute your program if you chose a programming language other than Python.

3. Create a single CSV file containing all your CodeCarbon measurements. **Suggestion:** After each execution, choose which values you want to collect from CodeCarbon and append them in a new line to the CSV file.

4. Compare the results obtained from the RAPL C main function (when it was not under power capping) with those from CodeCarbon. Create energy consumption and runtime graphs to illustrate the results.

3 Submission Details

In groups of three, complete **Exercise 1** and **Exercise 2**, presenting your findings in a non-report format of your choice (e.g., PowerPoint). Support your answers with graphs or other materials and provide brief explanations in bullet points.

Submit your completed exercises on **Blackboard** under the assignment `Powercap & CodeCarbon` by **March 25, 2024, at 23:59 (Portugal time)**.
Upload a **single file** named after your student numbers. For example, if your student numbers are `pg12345`, `pg12346`, and `pg12347` and you are submitting a PowerPoint file, the filename must be `pg12345-pg12346-pg12347.pptx`

References

[1] Benoit Courty, Victor Schmidt, Sasha Luccioni, Goyal-Kamal, MarionCoutarel, Boris Feld, Jérémy Lecourt, LiamConnell, Amine Saboni, Inimaz, supatomic, Mathilde Léval, Luis Blanche, Alexis Cruveiller, ouminasara, Franklin Zhao, Aditya Joshi, Alexis Bogroff, Hugues de Lavoreille, Niko Laskaris, Edoardo Abati, Douglas Blank, Ziyao Wang, Armin Catovic, Marc Alencon, Michal Stechly, Christian Bauer, Lucas Otávio N. de Araújo, JPW, and MinervaBooks. mlco2/codecarbon: v2.4.1, May 2024.

[2] Marco Couto, Rui Pereira, Francisco Ribeiro, Rui Rua, and João Saraiva. Towards a green ranking for programming languages. In *Proceedings of the 21st Brazilian Symposium on Programming Languages*, SBLP '17, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Simão Cunha, Luís Silva, João Saraiva, and João Paulo Fernandes. Trading runtime for energy efficiency: Leveraging power caps to save energy across programming languages. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*, SLE '24, page 130–142, New York, NY, USA, 2024. Association for Computing Machinery.

[4] Matthias Hahnel and et al. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Performance Evaluation Review*, 40(3):13–17, January 2012.

[5] Kashif Khan, Mikael Hirki, Tapio Niemi, Jukka Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3, 01 2018.

[6] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Spelling out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software*, 161:110463, 2020.

[7] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 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, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery.

[8] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Ranking programming languages by energy efficiency. *Science of Computer Programming*, 205:102609, 2021.

[9] Zhenkai Zhang, Sisheng Liang, Fan Yao, and Xing Gao. Red alert for power leakage: Exploiting intel rapl-induced side channels. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, May 2021.