Tópicos de Desenvolvimento de Software

Simão Cunha, João Saraiva
University of Minho

February 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 [3]. As described in [7], RAPL provides accurate energy consumption estimates In fact, RAPL has been successfully used to measure and estimate software energy consumption in numerous research studies [5, 4, 6].

We will adopt a different approach than that reported in [1, 5]: 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 [5].

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 [2], which incorporates temperature sensors 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.

1.2.1 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`.

```
int main(int argc, char **argv) {
    ...
    if (WHATTSCAP != -1) {
        if(initializeRapl(&rc))
            return -1;
    }

    if(programWorks(command)){
        performMeasurements(command, language, program, ntimes, core);
    } else {
        writeErrorMessage(language,program);
    }

    if (WHATTSCAP != -1) {
        if (raplcap_destroy(&rc)) {
            printf("Error destroying CAP\n");
            perror("raplcap_destroy");
        } else {
            printf("Successfully destroyed CAP\n");
        }
    }
    return 0;
}
```

1.2.2 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);
    }

    ...
}
```

1.3 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.

1.4 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)

2 Exercises

Download the `RAPL_Measurements.zip` file 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: Fibonacci Numbers

Consider the following definition of the Fibonacci function:

$$fib(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ fib(n-1) + fib(n-2), & \text{if } n \geq 2 \end{cases}$$

Please perform the following tasks.

1. Write a Fibonacci function in your favorite programming language.
2. Create a main function that calculates a Fibonacci number, resulting in an execution time of approximately one minute.
3. Measure the time and energy consumption of the program using the provided energy-based C main function.
4. Implement an optimized version of the Fibonacci function that runs in linear time. Perform the same measuremts as in the previous task.
5. Compare the performance in terms of *Speedup*, *Greenup* and *Powerup* of the non-optimized and optimized versions of the Fibonacci functions.

6. Consider several inputs for both functions and draw the *GPS-UP Software Energy Efficiency Quadrant Graph*

7. Test the impact of various compiler optimizations on energy consumption. For example, use '`-O0`' (no optimizations) and '`-O2`' (full optimizations).

8. Repeat the previous exercises for different Fibonacci numbers (a minimum of two different sequences).

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

10. Select a program of your own (or one from any software repository) to conduct a similar energy consumption analysis. **Suggestion**: use a CLI (Command Line Interface) program.

Exercise 2: Sorting Algorithms

1. Implement three sorting algorithms in your favorite programming language, ensuring that each algorithm has a different time complexity.

2. Use arrays of different sizes for sorting.

3. Use different data types for each array element. Suggestions: *int*, *float*, *double*, and *string*.

4. Test with both sorted and unsorted arrays, and compare the energy consumption and execution time results.

5. Test the impact of various compiler optimizations on energy consumption. For example, use '`-O0`' (no optimizations) and '`-O2`' (full optimizations).

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.