On the Impact of PowerCap in Haskell, Java, and Python

Luís Maia[1,2,†], Marta Sá[1,†], Inês Ferreira[1,†], Simão Cunha[1,†], Luís Silva[1,†], Paulo Azevedo[1,2,†] and João Saraiva[1,2,†]

[1]*Department of Informatics, University of Minho, Portugal*
[2]*HasLab/INESC TEC*

Abstract

Historically, programming language performance focused on fast execution times. With the advent of cloud and edge computing, and the significant energy consumption of large data centers, energy efficiency has become a critical concern both for computer manufacturers and software developers. Despite the considerable efforts of the green software community in developing techniques and tools for analysing and optimising software energy consumption, there has been limited research on how imposing hardware-level energy constraints affects software energy efficiency. Moreover, prior research has demonstrated that the choice of programming language can significantly impact a program's energy efficiency.

This paper investigates the impact of CPU power capping on the energy consumption and execution time of programs written in Haskell, Java, and Python. Our preliminary results analysing well-established benchmarks indicate that while power capping does reduce energy consumption across all benchmarks, it also substantially increases execution time. These findings highlight the trade-offs between energy efficiency and runtime performance, offering insights for optimising software under energy constraints.

Keywords

Energy Efficiency, Programming Languages, Benchmark, PowerCap

1. Introduction

Programming languages provide powerful mechanisms to improve the productivity of their programmers. Modern languages rely on powerful module and type systems, reusable libraries, IDE, testing frameworks, etc. Such mechanisms are built on top of advanced abstraction, thus relying on the language compiler to produce efficient code.

In the last century, good performance in programming languages was synonymous of fast code, that is to say, fast execution time. In this century, this reality has changed because the cost to run software became higher than the cost to build that same software [1]. In our age of cloud/edge computing and large data centers the energy consumption of software is a key concern for big tech and computer manufacturer companies, programming language designers, and programmers. Software systems are implemented using programming languages. As shown in [2], the energy efficiency of program can be drastically influenced by programming language used to develop it.

Although the green software community has done a considerable work on developing techniques and tools to analyse and optimise the energy consumption of software systems. Such techniques already provide knowledge on the energy efficiency of data structures [3, 4] and android language [5], the energy impact of different programming practices both in mobile [6, 7, 8] and desktop applications [9, 10], the energy efficiency of applications within the same scope [11, 12], or even on how to predict energy consumption in several software systems [13, 14], among several other works.

There is also recent work on analysing the energy efficiency of programming languages [15, 2, 16], where 28 of the most known and used programming languages are ranked according their energy performance. This ranking shows very interesting results, namely that slower/faster software languages consume less/more energy. The energy efficiency ranking, however, was produced assuming no limit on the energy usage by the operating system nor the hardware that executed the programs written in the 28 languages. However, we know that by limiting energy consumption of hardware we can reduce energy consumption. For example, a driver when operating a car (the hardware) can reduce its fuel consumption by defining a limit on the engine Revolutions Per Minute (RPM). Thus, an interesting question that immediately arises is whether *we can reduce the energy consumption of a program by just defining a limit on the energy used by the CPU.* Energy consumption does not depend only on execution time, as shown in the equation $E_{nergy} = T_{ime} \times P_{ower}$. In fact, this equation shows that we can reduce energy of program by reducing its execution time and/or the power it uses.

In this paper we analyse the impact of limiting the power of the CPU when executing programs in three different programming languages, namely, Haskell, Java and Python.[1] For each language we considered well established benchmark frameworks, namely *NoFib* for Haskell [17], *DaCapo* for Java [18] and *pyPerformance* for Python. We executed each benchmark in two situations: with and without a limit on energy consumption. In both cases we monitored and analysed the performance of each benchmark considering energy consumption and execution time. Our very first results show that by defining a power cap we do reduce energy consumption in all benchmarks. On the contrary, the execution time of the three benchmarks increases significantly.

This paper is organised as follows: Section 2 presents the methodology we follow to conduct our study. It briefly presents the benchmarks, the use of the Intel's RAPL and RAPLCap energy framework, the calibration of the powercap and how we executed and measured the benchmarks.

[1]We considered these three languages because this research originated from an exercise proposed in a MSc course on green software where those languages are studied.

RAW'24: 3RD WORKSHOP ON RESOURCE AWARENESS OF SYSTEMS AND SOCIETY, July 02–05, 2024, Maribor, SI

[†]These authors contributed equally.

pg47422@alunos.uminho.pt (L. Maia); pg54084@alunos.uminho.pt (M. Sá); pg53879@alunos.uminho.pt (I. Ferreira); simaopscunha@outlook.pt (S. Cunha); luis.m.peixoto@gmail.com (L. Silva); pja@uminho.pt (P. Azevedo); saraiva@uminho.pt (J. Saraiva)

In Section 3 we present and discuss the results of the three benchmarks. In Section 4 we present our conclusions.

2. Methodology

2.1. Benchmarks

In order to analyse the impact of limiting the energy consumption while executing Haskell, Java and Python programs, we consider well-established and widely used benchmark frameworks developed for those three programming languages. The benchmarks and the compilers/interpreters used in our study are:

- Haskell language (compiler ghc 9.4.8): *The nofib Benchmark Suite of Haskell Programs* [17].[2]
- Java language (compiler: Java Openjdk 11.0.22): *The DaCapo Benchmark suite* [18], version 23.11-chopin.[3]
- Python language (interpreter: python 3.12.0): *The Python Performance Benchmark Suite*, version 1.11.0.[4]

2.2. Energy Measurements and Cap via RAPL

For measuring the energy consumption, we used Intel's Running Average Power Limit (RAPL) tool [19], which is capable of providing accurate energy estimates at a very fine-grained level, as it has already been proven [20]. RAPL has been used in many studies on green software and was the energy measurement framework used in defining the ranking of programming languages [2, 16]. Moreover, the ranking of languages provides programs/scripts built on top of RAPL that makes very easy to execute and monitor the energy consumption of a (executable) program.

In our study we will reuse this (C-based) infrastructure to execute and monitor the benchmarks and their programs written in three languages.

Finally, RAPL also support a key ingredient for our study: the possibility of defining a power limit on the CPU while executing a program [21]. In fact, *RAPLCap*[5] provides a C interface for managing Intel RAPL power caps. Thus, we have extended the ranking energy monitoring infrastructure to allow the execution of a program under a given power cap.

2.3. RAPL's PowerCap calibration

The RAPL and RAPLCap can be configured with different values to limit the PowerCap of a specific CPU. Thus, when executing a program it is possible to define the (maximum) power used by the (intel) CPU. Different values for PowerCap can be used. Moreover, different intel CPUs may have the lowest energy consumption by using different PowerCap values. In order to know the value of the PowerCap that produces the lowest energy consumption in a specific CPU, we need to compute it. Thus, we consider a calibration phase where we execute with different RAPLcap values a computation intensive program and we compute the value that produce the most energy efficient execution [22, 23].

Thus, we consider the following implementation of the Fibonacci number in the C language:

[2]https://gitlab.haskell.org/ghc/nofib. Version available on January,2024.
[3]https://www.dacapobench.org/
[4]https://github.com/python/pyperformance
[5]https://github.com/powercap/raplcap

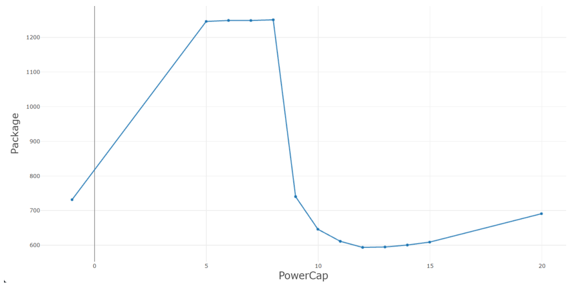


Figure 1: Power Cap Impact on Fibonacci's Energy Consumption.

```
long fib(int n)
{ if (n <= 0)      return 0;
  else if (n==1) return 1;
       else return fib(n-1) + fib(n-2);
}
```

In order to have significant energy and runtime measurements we execute this function to compute Fibonacci of 50. Furthermore, we executed (ten times) this function using specific power cap values.

The processor i7-7700hq has a TDP of 45Watts. With this in mind, there was a first set of power cap variables that went from -1 to 45 with a spacing of 5 between each value. The substantially better results were when power cap took a value between 5 and 20. To further analyse these results, a second attempt of trying to find the best power cap value was made, this time with values -1, 20 and all withing 5 and 15, where -1, which is when there is no power cap, and 20 served to compare results with the remaining power caps. In order to to make the results more robust, each iteration of a power cap was ran ten times, and were then averaged.

In figure 1 we confirm that the power cap that yields the best results for our purpose is power cap 12, as it proved to be the lowest energy consuming. From this moment, the only two power cap values that will in the following steps are -1 (No power cap) and 12 (with power cap).

2.4. Program Execution

As we mention before, we will use the ranking of programming languages infrastructure to execute an monitor the energy consumption of the three benchmarks. Such infrastructure has a key feature: there is no need to add intrusive code, for example calls to RAPL API, to the programs we would like to monitor energy consumption. This infrastructure includes a C program, named `energy`, that uses system calls to execute the unchanged programs while measuring the energy and time consumption via call to RAPL (and the `time` library). The overhead caused by the usage of a system call is insignificant as shown in [2, 24, 25].

Thus, to analyse the energy consumption of the benchmarks, we just use the benchmarks original makefiles to call the language compilers/interpreters with the defined optimisation's flags. Then, we use the `energy` C program to execute each of the executable program of each benchmark. This program is called from the command line as follows:

```
sudo ${custom_path}/RAPL/energy "./
     benchmark_executable" $(
     benchmark_language) $(benchmark) $
     (NTIMES) $(PowerCap)$
```

Its first argument is the executable program (of each benchmark) to be executed and monitored. The following two arguments are for naming purposes in the (csv) results output. Next argument is the number of runs for each program. The final argument is the value of the power cap (-1 or 12, as mentioned before).

In our study each benchmark's program was executed with and without power cap. Each program was executed a total of 10 times to grant a good starting set of results. Moreover, between each iteration, a cool down to the aforementioned mean CPU temperature is taken place to make sure each benchmark execution starts at the same CPU temperature. In the ranking of programming [2] it was used a 1 second sleep between executions. That may not be enough for the CPU to be at the same temperature at each execution [26].

In order to make sure every iteration of each benchmark starts on equal grounds, the mean temperature of the CPU is taken, and after every run of a benchmark a cooling process is executed until the temperature of the CPU cools down to the point of the mean temperature. Once the CPU is back to the mean temperature, it is able to execute another benchmark. As a consequence, each execution of a program starts with the CPU at the same temperature.

To obtain the mean temperature of the CPU, the library `lm-sensors`[6] was used in the `energy` C program.

3. Results

All studies were conducted on a desktop with the following specifications: Linux Ubuntu 2.04.4 LTS operating system, kernel version 4.8.0-22-generic, with 32GB of RAM, a Haswell Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz.

3.1. Treatment of the datasets

The RAPL framework is supported by all recent intel CPU architectures. However, not all intel architectures support the four energy estimations provided by RAPL [19], namely Core, DRAM, GPU and Package. In the intel architecture we conducted our studies, only the Core and Package estimations were available. Because the package estimation includes the consumption of the Core, GPU and other electronic devices on the chip, we will use this measurement as a reliable indication of the energy consumption of the CPU.

The energy monitoring C program developed for the energy ranking of languages and that we are reusing in our study, however, produces three more measurements: execution *time*, *memory* consumption and *temperature*. *Time* and (peak) *memory* consumption are provided by the linux operating system. The *temperature* measurement, provided by the `lm-sensors C library`, gives the value of the temperature of the CPU after the program execution. Thus it provides an indication of the energy dissipated as heat by the CPU. Thus, every single execution of a benchmark program produces five measurements: *core*, *package*, *time*, *memory* and *temperature*.

In our study we execute each benchmark program 10 times. In fact, we run each program 20 times: 10 times with power cap and 10 times without power cap. We also used box plots to analyse outliers. These box plots showed that the Java and Haskell benchmarks do not have outliers and although Python benchmarks have outliers they are not relevant in terms of number or value.

3.2. Haskell nofib Benchmark

To evaluate the impact of power cap on the variables *Package* and *Time* we started by creating two datasets: one with all the measurements of Haskell benchmarks without power cap and another with all the measurements of the Haskell benchmarks with a power cap of 12. Figure 2 presents the energy and runtime measurements for each program of the *nofib* benchmark.[7] Thus, for each program we associate two bars: energy (in joules) on the left and runtime (in seconds) on the right. Moreover, each bar uses two colours to indicate the use or not of power cap.

As we can observe in figure 2, all Haskell programs consume less energy when executed with power cap (dark blue bar) when compared to the same program executed without a power cap. We can also see that power cap has a considerable impact on execution time: all programs increase their execution time when using power cap (green bar).

Figure 3 shows in more detail the energy reduction obtained by each benchmark program with power cap.

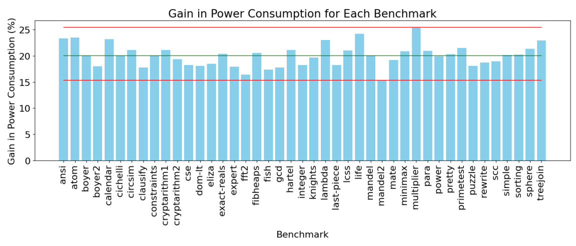


Figure 3: Reduction in *energy* consumption (*package*) with the use of power cap for Haskell benchmarks.

The green, red and dark red lines indicate the average, maximal and minimal energy gains. The precise values are:

- Average gain: **20.0%**
- Maximal gain: **25.4%**
- Minimal gain: **15.4%**

The reduction on energy consumption comes at a price: the increase in execution time. Indeed, figure 4 shows negative results for execution time, only. Thus, meaning that every benchmark had a worse execution time with a power cap.

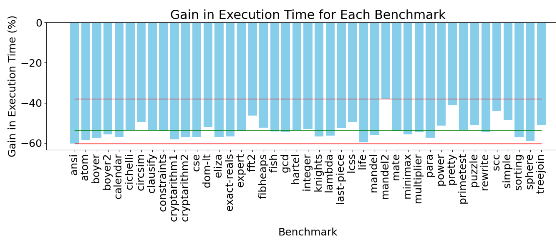


Figure 4: Increase in *time* with the use of power cap for Haskell benchmarks.

The average, maximal an minimal increase in execution time are the following:

[6]https://github.com/lm-sensors/lm-sensors

[7]Due to scale issues which will make it difficult to see the results in figure 2, we omit here the *hartel* program. That program, however, is included in the figure 17 available in appendix.

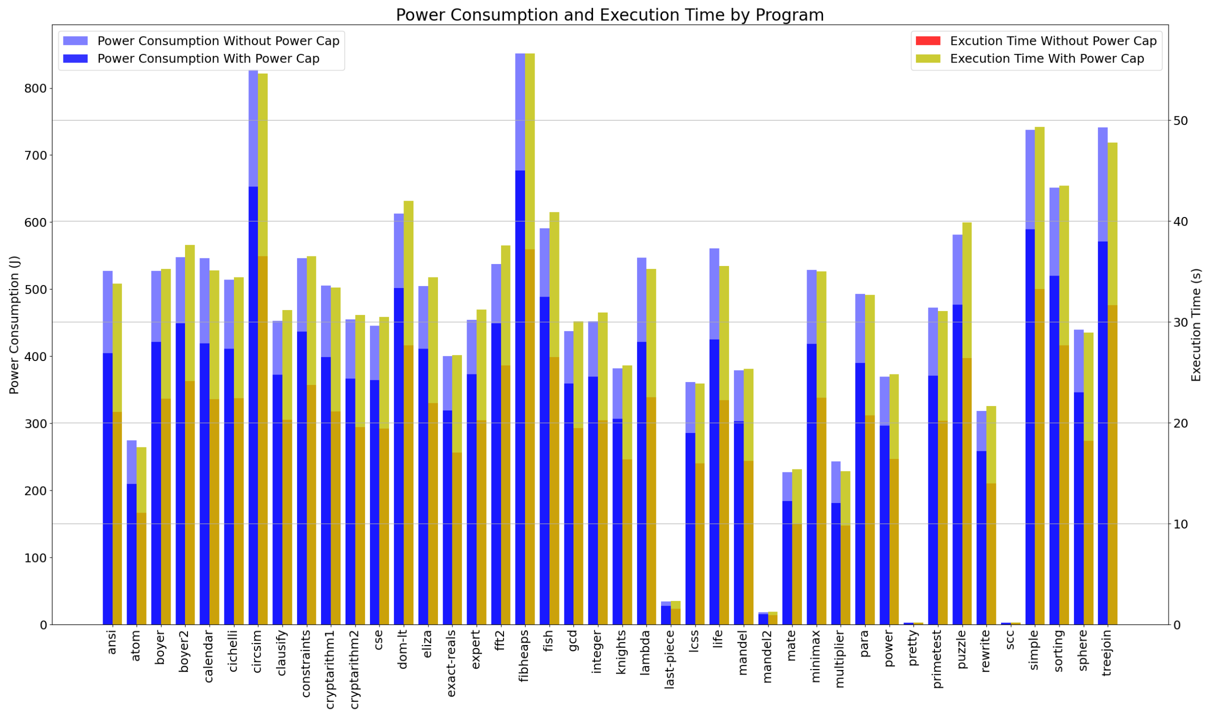


Figure 2: Energy consumption (left bar/legend) and execution time (right bar/legend) for each nofib benchmark program executed with and without power cap.

- Average loss: **53.7%**
- Maximal loss: **60.3%**
- Minimal loss: **38.0%**

Besides analysing the impact of the power cap in execution time and energy consumption and the gain in this two aspects, we decided that it would be interesting to analyse how the values of one column influence the values of other columns. To do so we used a correlation matrix. A correlation matrix is a statistical tool that shows how strong and in what direction two or more variables are related. The correlation coefficient ranges from -1 to +1, where -1 means a perfect negative correlation, +1 means a perfect positive correlation, and 0 means there is no correlation between the variables.

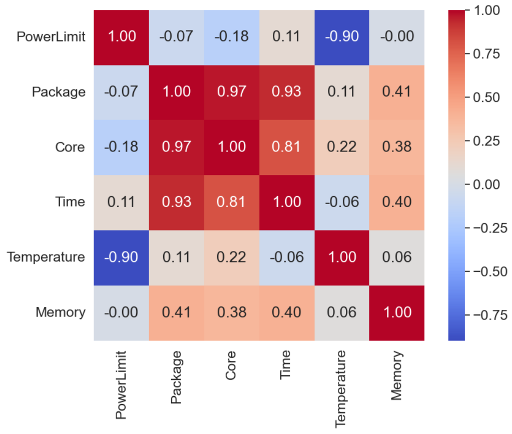


Figure 5: Correlation matrix for Haskell

By analysing the correlation matrix presented in figure 5, we can conclude that the two columns that present higher positive correlation are *Core* and *Package* followed by *Time* and *Package* and *Time* and *Core*. This means that if the values of a column presented in the pair increase, the values on the other column of the pair increase too. Is also relevant to mention that the columns *Temperature* and *PowerLimit* are negatively correlated which, in this case, means that if the values of a column present in the pair increase the values on the other column of the pair decrease.

Another aspect that we considered interesting to explore was how the power cap affects the relation between execution time and energy consumption. To do so we used the scatter plot presented in figure 6:

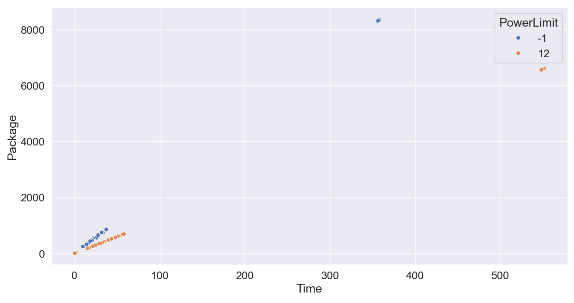


Figure 6: Correlation between Package and Time with and without power cap.

In figure 6 we observe that we have less energy consumption per time unit when using the power cap. Besides, we can once again confirm that the energy consumption and the execution time are positively correlated.

3.3. Java DaCapo Benchmark

The *DaCapo* benchmark, in its version *23.11-chopin*, consists of 20 real-world, open-source client-side Java benchmarks.

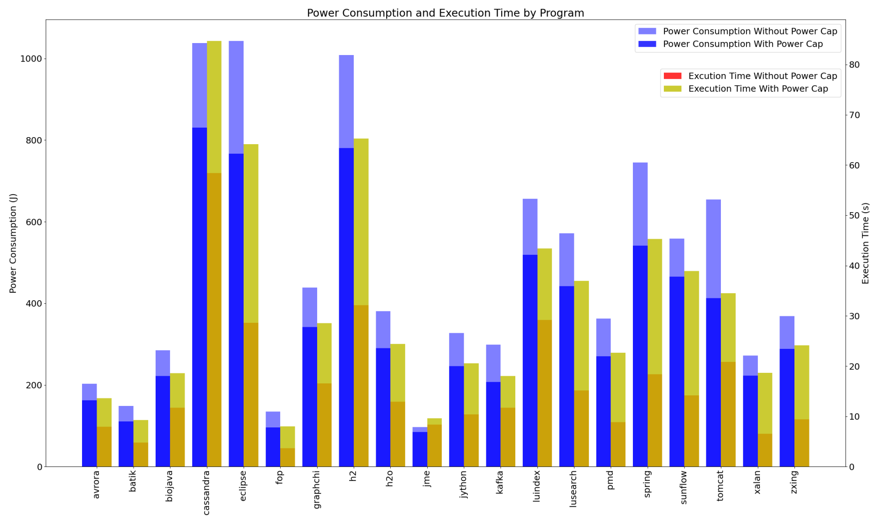


Figure 7: Energy consumption (left bar/legend) and execution time (right bar/legend) for each *DaCapo* benchmark program executed with and without power cap.

We executed each of this benchmarks with and without power cap. Figure 7 shows the energy and runtime of each *DaCapo* program. As we did for *nofib*, for each *DaCapo* program we associate two bars: energy (in joules) on the left and runtime (in seconds) on the right. Moreover, each bar uses two colours to indicate the use or not of power cap.

Figure 8 clearly shows that the use of power cap does decreases energy consumption, on average a 23.4% while increasing runtime, on average 101.1%, as shown in figure 9. This is the case for each of the 20 *DaCapo* programs, very much like in the *nofib* benchmark.

To have a better understanding on the impact that power cap has on energy consumption, we calculated the gain in percentage for each benchmark. The results are presented in figure 8.

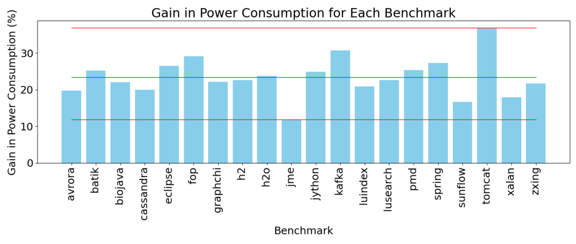


Figure 8: Reduction in energy consumption (package) with the use of power cap for Java benchmarks.

where the average, maximal and minimal values of energy consumption reduction are:

- Average gain: **23.4%**
- Maximal gain: **37.0%**
- Minimal gain: **11.9%**

The execution time of *DaCapo* benchmarks increases when using a power cap, as detailed in figure 9.

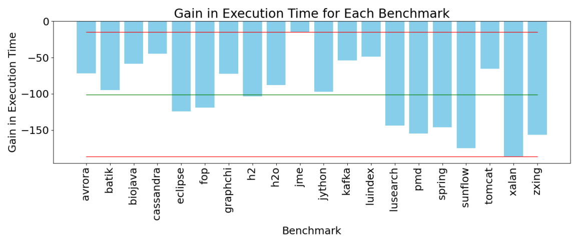
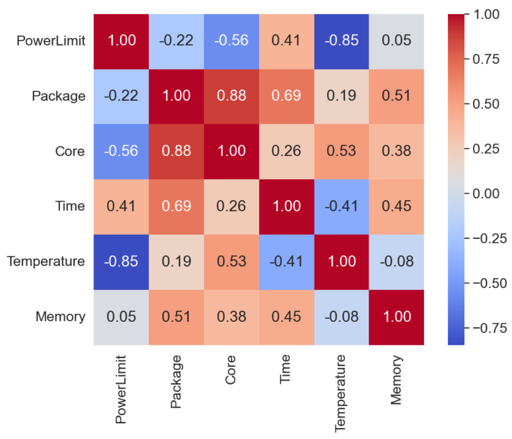


Figure 9: Increase in *Time* with the use of power cap for Java benchmarks.

In average the runtime in *DaCapo* increases 100%.

- Average loss: **101.1%**
- Maximal loss: **186.4%**
- Minimal loss: **14.6%**

In order to analyse the correlation between our five measurements, we use the correlation matrix presented in figure 10.



	PowerLimit	Package	Core	Time	Temperature	Memory
PowerLimit	1.00	-0.22	-0.56	0.41	-0.85	0.05
Package	-0.22	1.00	0.88	0.69	0.19	0.51
Core	-0.56	0.88	1.00	0.26	0.53	0.38
Time	0.41	0.69	0.26	1.00	-0.41	0.45
Temperature	-0.85	0.19	0.53	-0.41	1.00	-0.08
Memory	0.05	0.51	0.38	0.45	-0.08	1.00

Figure 10: Correlation matrix for Java benchmarks.

By looking at figure 10 we can conclude that *Package* and *Core* are the most positively correlated columns followed by *Time* and *Package*. Moreover, the attributes *Temperature* and *PowerLimit* are the most negatively correlated features.

Lastly we analyse the relation between *Package* and *Time* with and without power cap using a scatter plot and the results are displayed in figure 11,

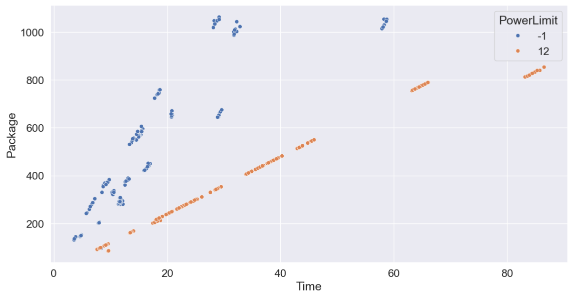


Figure 11: Correlation between *Package* and *Time* with and without power cap for Java *DaCapo* benchmarks.

By taking a look at figure 11 we can conclude that there is less energy consumption per time unit with the use of power cap. Moreover, with the use of power cap the columns present a stronger correlation.

3.4. Python pyPerformance Benchmark

The *pyPerformance* benchmark consists of 75 programs, which we executed with and without power cap by our energy framework. Of these 75 programs, only 57 were used due to some benchmarks requiring a python version above the executed one. This was the case when running python 3.8 on benchmarks that required python 3.10 or above. To be able to compare each compiler with each other, these benchmarks results in other version were discarded. Figure 12 shows the energy consumption and runtime for each python program.

As we can see in figure 12 with the use of power cap there is a decrease in the energy consumption of all programs. This is shown in more detailed in figure 13.

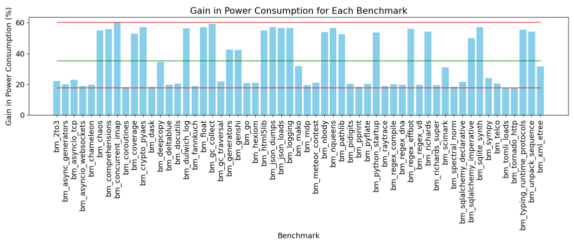


Figure 13: Reduction in energy consumption (package) with the use of power cap for Python benchmarks.

In fact, in average the use of power cap reduces energy consumption in 35%!

- Average gain: **35.3%**
- Maximal gain: **60.4%**
- Minimal gain: **17.9%**

In terms of runtime, however, the results are different. The red colour shown in several programs of figure 12 do indicate that some python programs execute faster when a power cap is defined!

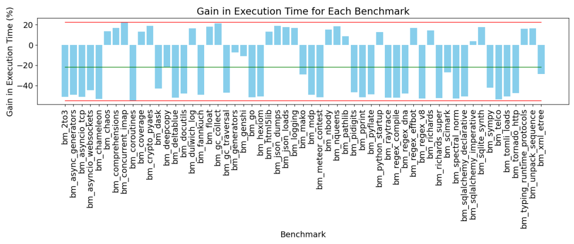


Figure 14: Increase in *Time* with the use of power cap Python benchmarks.

- Average loss: **21.5%**
- Maximal gain: **22.6%**
- Maximal loss: **54.8%**

Once we studied the impact of power cap on energy consumption and execution time, we opt to analyse the relations between different columns values. And we did so using a correlation matrix presented in figure 15

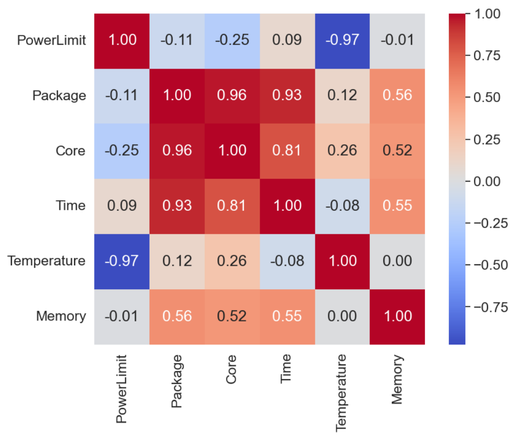


Figure 15: Correlation matrix for Python benchmarks version 3.12.

By analysing the correlation matrix presented above we can conclude that the two columns that present higher positive correlation are *Core* and *Package* followed by *Time* and *Package* and *Time* and *Core*. Is also relevant to mention that the columns *Temperature* and *PowerLimit* are negatively related once this relation in the correlation matrix presents such a small number.

Lastly we analysed the relation between energy consumption and execution time, and the results are presented in figure 16

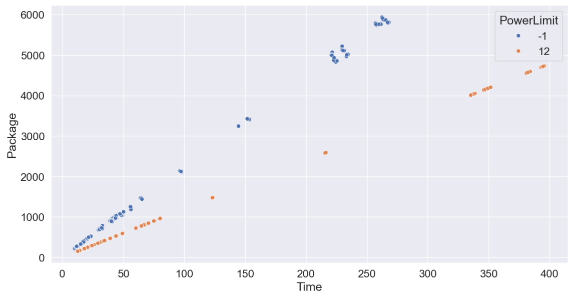


Figure 16: Correlation between *Package* and *Time* with and without power cap for Python benchmarks version 3.12.

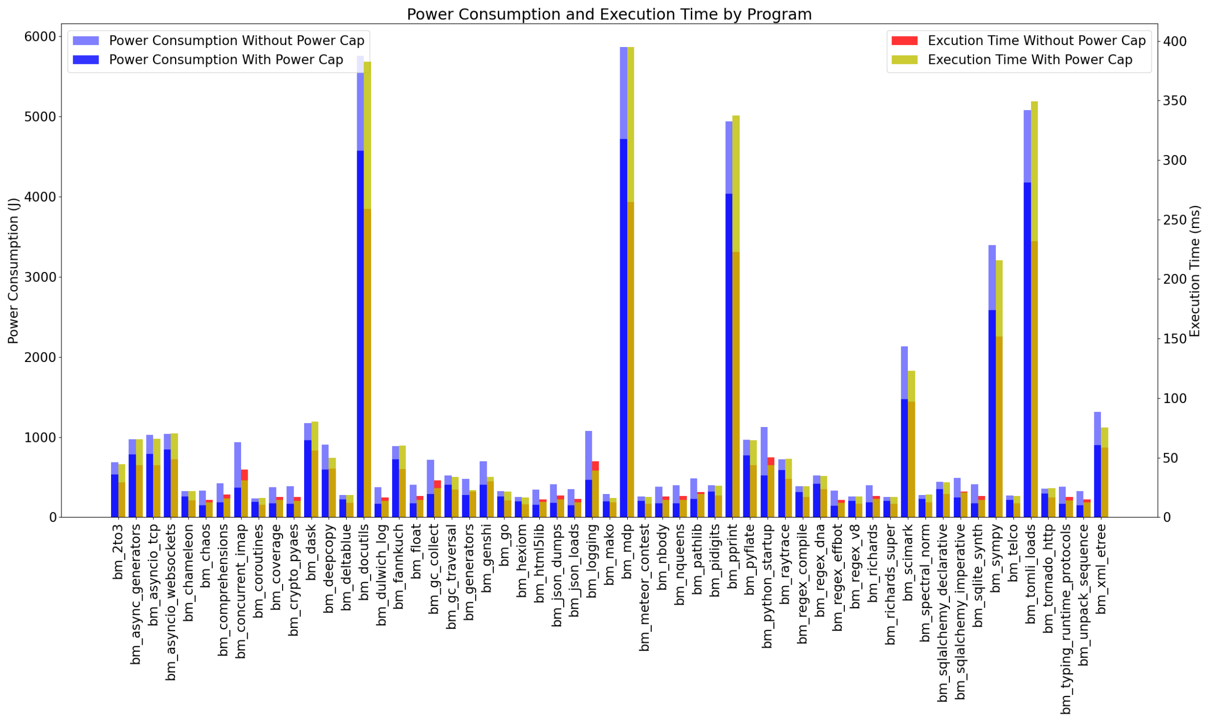


Figure 12: Energy consumption (left bar/legend) and execution time (right bar/legend) for each *pyPerformance* program executed with and without power cap.

Gain And Loss	Haskell	Java	Python
average energy gain	20.0%	23.4 %	35.3%
maximal energy gain	25.4%	37.0 %	60.4%
minimal energy gain	15.4%	11.9 %	17.9%
average runtime loss	53.7%	101.1%	21.5%
maximal runtime loss	60.3%	186.4%	54.8%
minimal runtime loss	38.0%	14.6 %	-
maximal runtime gain	-	-	22.6%
no. programs that increased energy	0	0	0
no. programs that decreased runtime	0	0	22

Table 1
Overall Results

After analysing figure 16 we can conclude that we have less energy consumption per time unit when using the power cap. Besides that, once again we can confirm that energy consumption and execution time are positively correlated.

3.5. Discussion

When comparing the correlation between *Package* and *Time* in Java and Haskell benchmarks, considering both correlation matrices, one can conclude that correlation is stronger in the Haskell benchmarks.

From table 1 we can observe that in terms of energy consumption, regardless the implementation language, every program improved their energy consumption. In fact, Python demonstrates the highest average energy gain (35.3%), followed by Java (23.4%) and Haskell (20.0%). Python also demonstrates a higher maximum energy gain (60.4%), followed by Java (37.0%) and Haskell (25.4%). In terms of minimal energy gain, Python has the highest (17.9%) compared to Haskell (15.4%) and Java (11.9%).

The impact on the runtime, Java shows a significant average runtime loss (101.1%), indicating that the programs are running much slower, on average, when comparing to Haskell (53.7%) and Python (21.5%). In fact, the maximum runtime loss is higher in Java (186.4%), followed by Haskell (60.3%) and Python (54.8%). In terms of minimal runtime loss, Haskell has the highest (38.0%) compared to Java (14.6%) and Python, which had a gain of 22.6%.

In contrast to the majority of the benchmarks, when a power cap is set, certain python benchmarks' runtime decreased, along with their energy consumption. The most notorious are *bm_concurrent_imap*, a concurrent model communication benchmark, which uses the *Pool* that handles CPU bound job and *ThreadPool* that handles *IO* bound jobs, from the *multiprocessing.pool* library. Also,*bm_logging* is a simple message logging benchmark, that utilises the in-memory text stream function *StringIO* from the *io* library. As a final example, we have *bm_gc_collect* benchmark, a node connected to another node traversal benchmark that is ran with *n* cycles, resembling a linked list collection traversal. The first two mentioned benchmarks handle *IO* related jobs. Thus, one could argue that execution of these type of jobs are more sensible to a power cap when compared to all the benchmarks. Hence, further testing and analysis is required to fully understand the reason behind these unexpected results.

In summary, Python shows the most balance performance with substantial energy gains and the potential for runtime reductions. However, Java programs tend to have a higher runtime loss, indicating a trade-off where energy savings may come at the cost of significantly increasing execution time. Also, Haskell shows a moderate performance in both energy gains and runtime losses, making it a middle ground between Python and Java in this analysis.

4. Conclusions

This paper presented a study on the impact of defining a power cap in three well-established benchmarks: *nofib* in Haskell, *DaCapo* in Java, and *pyPerformance* in Python. We executed all programs in these three benchmarks with and without defining a power cap.

Our first results show that power cap consistently decrease the energy consumption of programs written in all three languages. In average we obtained energy reductions of 20% in Haskell, 23% in Java, and 35% in Python. This energy reduction comes at a price: in all languages the execution times increases: 22% in Python, 54% in Haskell, and 101% in Java! Although Python seems to profit more from power cap it is worth to notice that Python is known for having poor runtime and energy consumption performances [2], and the speedup/greenup that achieves with power cap does not directly translate to speed/greenness.

Acknowledgements

This work is partially supported by CERCIRAS - Connecting Education and Research Communities for an Innovative Resource Aware Society - COST Action CA19135 funded by COST Association.

References

[1] W. Voegels, Keynote at AWS re:Invent 2023, 2023. URL: https://youtu.be/UTRBVPvzt9w?t=3718, accessed: 2024-05-19.

[2] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, J. 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, Association for Computing Machinery, New York, NY, USA, 2017, p. 256–267. doi:10.1145/3136014.3136031.

[3] R. Pereira, M. Couto, J. a. Saraiva, J. Cunha, J. a. P. Fernandes, The influence of the java collection framework on overall energy consumption, in: Proc. of the 5th Int. Workshop on Green and Sustainable Software, GREENS '16, ACM, 2016, pp. 15–21. doi:10.1145/2896967.2896968.

[4] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, A. Hindle, Energy profiles of java collections classes, in: Proc. of the 38th Int. Conf. on Software Engineering, ACM, 2016, pp. 225–236.

[5] W. Oliveira, R. Oliveira, F. Castor, A study on the energy consumption of android app development approaches, in: Proceedings of the 14th International Conference on Mining Software Repositories, IEEE Press, 2017, pp. 42–52.

[6] D. Li, W. G. J. Halfond, An investigation into energy-saving programming practices for android smartphone app development, in: Proceedings of the 3rd International Workshop on Green and Sustainable Software (GREENS), 2014.

[7] C. Sahin, F. Cayci, I. L. M. Gutierrez, J. Clause, F. Kiamilev, L. Pollock, K. Winbladh, Initial explorations on design pattern energy usage, in: Green and Sustainable Software (GREENS), 2012 1st Int. Workshop on, IEEE, 2012, pp. 55–61.

[8] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, D. Poshyvanyk, Mining energy-greedy api usage patterns in android apps: an empirical study, in: Proc. of the 11th Working Conf. on Mining Software Repositories, ACM, 2014, pp. 2–11.

[9] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, in: Proc. of 8th ACM/IEEE Int. Symposium on Empirical Software Engineering and Measurement, ACM, 2014, p. 36.

[10] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, J. Saraiva, Helping programmers improve the energy efficiency of source code, in: Proceedings of the 39th International Conference on Software Engineering Companion, ICSE-C '17, IEEE Press, 2017, p. 238–240. doi:10.1109/ICSE-C.2017.80.

[11] S. A. Chowdhury, A. Hindle, Greenoracle: estimating software energy consumption with energy measurement corpora, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR, 2016, 2016, pp. 49–60.

[12] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, P. Ammann, Ecodroid: An approach for energy-based ranking of android apps, in: Proc. of 4th Int. Workshop on Green and Sustainable Software, GREENS '15, IEEE Press, 2015, pp. 8–14.

[13] S. Hao, D. Li, W. G. J. Halfond, R. Govindan, Estimating mobile application energy consumption using program analysis, in: Proc. of the 2013 Int. Conf. on Software Engineering, ICSE '13, IEEE Press, 2013, pp. 92–101.

[14] M. Couto, P. Borba, J. Cunha, J. P. Fernandes, R. Pereira, J. Saraiva, Products go green: Worst-case energy consumption in software product lines, in: Proceedings of the 21st International Systems and Software Product Line Conference - Volume A, SPLC '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 84–93. doi:10.1145/3106195.3106214.

[15] M. Couto, R. Pereira, F. Ribeiro, R. Rua, J. Saraiva, Towards a green ranking for programming languages, in: Programming Languages: 21st Brazilian Symposium, SBLP 2017, Fortaleza, Brazil, September, 2017., 2017.

[16] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. P. Fernandes, J. Saraiva, Ranking programming languages by energy efficiency, Science of Computer Programming 205 (2021). doi:10.1016/j.scico.2021.102609.

[17] W. Partain, The nofib benchmark suite of haskell programs, in: J. Launchbury, P. Sansom (Eds.), Functional Programming, Glasgow 1992, Springer London, London, 1993, pp. 195–202.

[18] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The DaCapo benchmarks: Java benchmarking development and analysis, in: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2006, pp. 169–190. doi:http://doi.acm.org/10.1145/1167473.1167488.

[19] M. Dimitrov, C. Strickland, S.-W. Kim, K. Kumar, K. Doshi, Intel® power governor, https://software.intel.com/en-us/articles/intel-power-governor, 2015. Accessed: 2015-10-12.

[20] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, Z. Ou, Rapl in action: Experiences in using rapl for power measurements, ACM Trans. Model. Perform. Eval. Comput. Syst. 3 (2018). doi:10.1145/3177754.

[21] C. Imes, H. Zhang, K. Zhao, H. Hoffmann, CoPPer: Soft real-time application performance using hardware power capping, in: 2019 IEEE International Conference on Autonomic Computing (ICAC), 2019, pp. 31–41. doi:10.1109/ICAC.2019.00015.

[22] A. Krzywaniak, P. Czarnul, J. Proficz, Extended investigation of performance-energy trade-offs under power capping in hpc environments, in: 2019 International Conference on High Performance Computing & Simulation (HPCS), 2019, pp. 440–447. doi:10.1109/HPCS48598.2019.9188149.

[23] A. Krzywaniak, P. Czarnul, J. Proficz, Depo: A dynamic energy-performance optimizer tool for automatic power capping for energy efficient high-performance computing, SOFTWARE-PRACTICE & EXPERIENCE 52 (2022) 2598–2634.

[24] M. Hähnel, B. Döbel, M. Völp, H. Härtig, Measuring energy consumption for short code paths using RAPL, SIGMETRICS Performance Evaluation Review 40 (2012) 13–17.

[25] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, R. Geyer, An Energy Efficiency Feature Survey of the Intel Haswell Processor, Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2015 (2015) 896–904. doi:10.1109/IPDPSW.2015.70.

[26] B. Santos, M. H. Kirkeby, J. P. Fernandes, A. Pardo, Compiling Haskell for energy efficiency: Empirical analysis of individual transformations, in: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, ACM, 2024, pp. 1104–1113. doi:10.1145/3605098.3635915.

Appendice

Figure 17 includes all programs of the *nofib* benchmark.

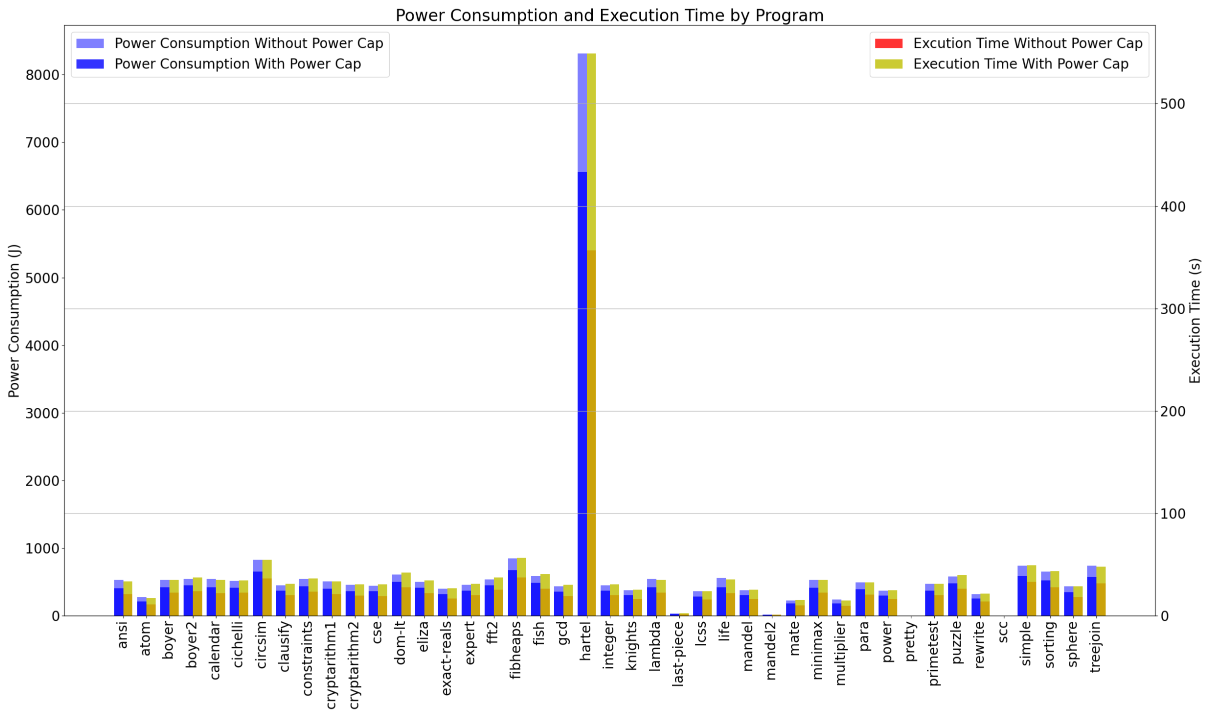


Figure 17: Relation between Haskell's benchmarks and the variables *Package* and *Time* with and without power cap.