PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools

Adel Noureddine
Universite de Pau et des Pays de l'Adour, E2S UPPA, LIUPPA
Anglet, France
adel.noureddine@univ-pau.fr

***Abstract*—Monitoring the power consumption of applications and source code is an important step in writing green software. In this paper, we propose `PowerJoular` and `JoularJX`, our software power monitoring tools. We aim to help software developers in understanding and analyzing the power consumption of their programs, and help system administrators and automated tools in monitoring the power consumption of large numbers of heterogeneous devices.**

***Index Terms*—Power Monitoring, Measurement, Power Consumption, Energy Analysis**

I. Introduction and Related Work

Writing green software is a major concern for software developers and practitioners [5]. However, developers lack tools and knowledge in understanding the energy and power consumption of software and writing efficient ones [9]. In addition, the landscape of computing architecture is moving towards a heterogeneity of CPU and GPU architectures (*i.e.*, x86, ARM), and device types (*i.e.*, PCs, servers, single-board computers, mobile). Therefore, software developers are more often building multi-platform software. To help bridge the gap, we present, in this paper, `PowerJoular` and `JoularJX`, our multi-platform software power monitoring tools. `PowerJoular` monitors the power consumption of CPU and GPU for PCs, servers and single-board computers (such as Raspberry Pi). `JoularJX` uses the power data provided by `PowerJoular` to monitor the power consumption of methods and source code in Java applications.

In the last decade, multiple power monitoring tools were released with varying approaches and accuracy. Approaches range from hardware-based tools (using power meters) to software-based ones (using power estimation models). The former [6] uses a physical power meter or multimeter to measures the energy consumption of the device, and correlates the data with software-based monitoring. The main limitation of these approaches is the high installation cost, and the limited scalability as they require an additional hardware device. For the latter, earlier software tools used their own power estimation models, such as the first version of PowerAPI [1], Jolinar [7] or pTop [2]. These models are either based on CMOS power formulas, or on empirical experiments. However, newer approaches and tools are based on hardware manufacturers' APIs. In particular, most server tools use the Intel RAPL interface, either directly from the registers or through the Linux kernel's implementation (for example, using powercap interface). For instance, newer PowerAPI versions or Scaphandre [1] use Intel RAPL for power monitoring. However, these tools target server and cloud environments and provide features mostly used by their uses cases, such as monitoring virtual machines or exposing metrics to hypervisors and cloud dashboards. In addition, most tools focus on one particular platform (mainly Intel servers) and are aimed towards system administrators or automated monitoring platforms. In contrast, we aim with our approach, to provide a multi-platform power monitoring tool (starting with x86_64 servers and ARM single-board devices), and help software developers with up-to-date and easy-to-use tools to analyze the power consumption of software.

II. PowerJoular Design and Features

In this section, we present the power design and features of `PowerJoular`. Our tool allows runtime power monitoring of multiple hardware components of different devices and architectures. In particular, our initial version monitors the CPU and GPU power consumption in computers and servers, and the CPU in Raspberry Pi devices. `PowerJoular` is written in Ada in order to provide a low-impact tool as Ada is constantly ranked among the most energy efficient programming languages [8], while also improving code maintainability and safety in particular as we also target monitoring single-board computers and embedded devices. `PowerJoular` is aimed to software developers, system administrators and to automated tools, with a goal to help these users understand the power consumption of their devices and software, and to build more in-depth tools using our proposed platform.

A. Power Monitoring Approach

`PowerJoular` power monitoring is based on two modules:

- for PC/servers: the Intel RAPL through the Linux Power Capping Framework [2], and, optionally, NVIDIA's System Management Interface [3],

[1]https://github.com/hubblo-org/scaphandre
[2]https://www.kernel.org/doc/html/latest/power/powercap/powercap.html
[3]https://developer.nvidia.com/nvidia-system-management-interface

- for Raspberry Pis: our own empirical regression power models. [4]

PowerJoular automatically detects the computer configuration and supported modules, and provides power data accordingly. For the GPU, PowerJoular checks if NVIDIA SMI is installed, then uses it to verify if GPU power monitoring is supported to the specific graphic card, and to read GPU power consumption every second.

For the CPU, PowerJoular uses the Intel RAPL power data through the Linux powercap interface by reading the appropriate system files. It first detects which power domains are supported by the CPU:

- Pkg: which is supported since Intel Sandy Bridge CPUs, and provides energy consumption for the CPU cores, integrated graphics, memory controller and last level caches. PowerJoular also checks if DRAM power domain is supported (RAM attached to the memory controller) and adds its power readings to the total.
- Psys: which is supported since Intel Skylake CPUs, and provides energy consumption for the entire SOC (including Pkg along with other components, such as eDRAM, PCH, System Agent [3]). If Psys is supported, it will be exclusively used by PowerJoular for CPU power consumption instead of Pkg and DRAM, as it provides a more comprehensive power reading of the CPU SOC.

Finally, PowerJoular aggregates power readings from all supported components to provide an overall power consumption. For instance, if both Intel RAPL and NVIDIA SMI are supported, the tool will provide an aggregated power value for both CPU and GPU.

On Raspberry Pi devices, PowerJoular uses our own power polynomial regression models that maps the CPU utilization to power consumption. These models are accurate and have very low error rates, between 0.3% and 3.83%, far more accurate than the state-of-the-art models. The tool reads CPU cycles from /proc/stat system file, and calculates CPU utilization. The latter is then used in the polynomial models to provide an accurate estimation of the CPU power consumption. PowerJoular can also update power models from an online repository if new more accurate models are available. In addition to monitoring the power consumption of hardware components (CPU, GPU), PowerJoular can monitor the CPU power consumption of an individual process by providing its PID on runtime.

B. Features

We designed PowerJoular to be efficient, low on resources, flexible and intuitive to use. The interaction with the tool is achieved through a command-line interface. Such interface offers flexibility for scientific experimentations, headless monitoring in server environments, and can be easily incorporated into external frameworks or dashboards.

[4]The source code for the power models of Raspberry Pi ARM processors is soon to be published in the git repository of the tool as the approach and models are currently under review in a journal.

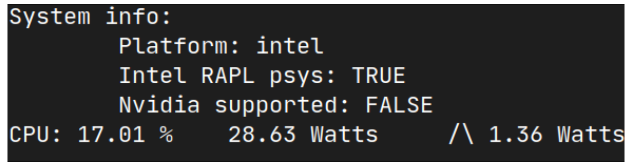

Fig. 1. Default output of the PowerJoular command-line interface

Runtime power monitoring can be displayed on the terminal and/or written to CSV files. Figure 1 shows the command-line interface of PowerJoular. The latter CSV option stores power data every second and can then be read to retrace the historical power consumption of a device or a specific process. If a PID is monitored, its power data will be displayed on the terminal (cf. Figure 2), and will be stored to a distinct CSV file, while the device's power is saved independently. At the end of each monitoring session, PowerJoular displays the total energy consumption (in Joules) of the session (and for the monitored PID).

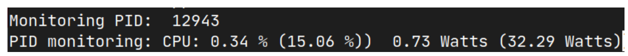

Fig. 2. Default output of the PowerJoular command-line interface when monitoring a PID

In addition, writing to a file can also be done in *overwrite* mode, *i.e.*, only the last power data is saved to the file. Therefore, the tool can run for long periods of time without generating a large CSV file. This mode allows external tools to connect to PowerJoular's power data in runtime to build dashboard or monitoring interfaces. For instance, a centralized dashboard can read and visualize power data of multiple servers or Raspberry Pi devices running PowerJoular.

Finally, PowerJoular provides a systemd service [5] that can be enabled and run automatically on Linux boot. The service monitors the computer's power consumption and stores data in a CSV file (with overwrite mode) in /tmp folder. This allows continuous and automated monitoring of servers and devices, and provides accurate runtime power data. Writing to /tmp while running automatically as a service, allows to bypass the added restrictions on reading powercap energy meters to non-privileged users in Intel CPUs [6]. The restriction was added due to the recently discovered PLATYPUS vulnerability [4]. However, the systemd service still requires privileged access (root/sudo) to be enabled, and the data provided is the runtime power consumption (every second) from aggregate sources (CPU and GPU when available).

III. Source Code Energy Monitoring with JoularJX

The flexibility of PowerJoular allows its integration and usage by other tools. In this section, we present JoularJX,

[5]https://systemd.io/

[6]https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=949dd0104c496fa7c14991a23c03c62e44637e71

our source-code power monitoring tool for Java programs that uses `PowerJoular`.

As we focus on monitoring the source code of Java applications, we build `JoularJX` as a Java agent that hooks to the Java Virtual Machine to monitor power consumption, on runtime, for every method in the monitored software. Java is among the top 10 of the most energy-efficient languages [8] with its modern on-the-fly optimizations in the JVM, ranking even fifth in the normalized global results for energy. The agent automatically starts `PowerJoular` with the appropriate parameters (monitoring the PID of the Java program, and writes power data to an overwritten CSV file in `/tmp` folder). CPU utilization is then monitored every second for every Java thread and power consumption is allocated accordingly. A second monitoring loop detects, for every thread, which method is currently being executed for every 10 milliseconds (by observing the first method in the thread stacktrace), and then power consumption is allocated statistically to each method. As this power monitoring also includes the Java JDK's own methods, we also have an option to monitor specific methods based on their full names. For instance, we can monitor all methods belonging for a certain package by allocating the power consumed by the first method of a thread stacktrace to the method that called it (by analyzing the stacktrace tree). In both cases, `JoularJX` will always generate power data for all methods in a separate file, and the specific methods in another file. Overall, our source-code statistical approach is similar to the one we developed in our old Jalen tool [7].

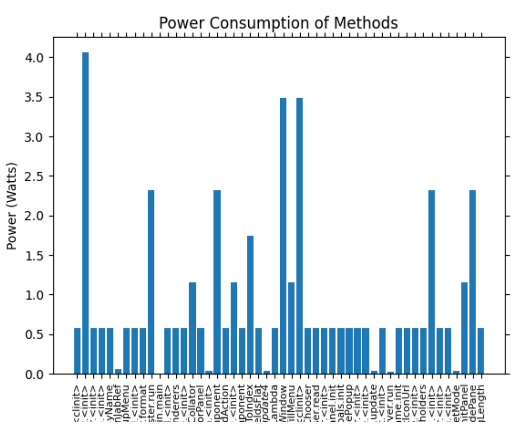

Fig. 3. Snapshot of runtime power consumption of methods with JoularJX, and a sample runtime Python GUI

However, `JoularJX` has two main differences with Jalen: first, it uses an external tool to monitor the software power consumption (*i.e.*, `PowerJoular`) and therefore runs on servers and Raspberry Pi devices. Second, `JoularJX` provides the overall energy consumption of methods for the program duration, but also provides power consumption for methods every second on runtime (which can be plotted with a visualization tool, such as a sample Python GUI as see in Figure 3), unlike Jalen which only provides overall energy consumption. This allows developers and system tools to follow power consumption for individual methods live throughout the execution of software.

IV. Multi-Platform Use Case

To showcase the multi-platform capabilities of `PowerJoular`, we measure the energy consumed by three implementations of the Ray-casting algorithm taken from Rosetta Code [7]. We run the Python implementation in a loop for 10 000 iterations, the C version for 100 000 iterations and the Java version for 5 000 iterations. We conduct our experiments on three devices: a Dell 5530 laptop (Intel Core i7-8850H) running Fedora Linux 34 with kernel 5.11.19, GCC 11.1, Java 11, and Python 3.9.5. A Raspberry Pi 3b+ revision 1.3, running Raspberry Pi OS 32 bits (based on Debian 10), and a Raspberry Pi 4B revision 1.2, running Raspberry Pi OS 64 bits, both running with kernel 5.10.17, GCC 8.3, Java 11, and Python 2.7.16.

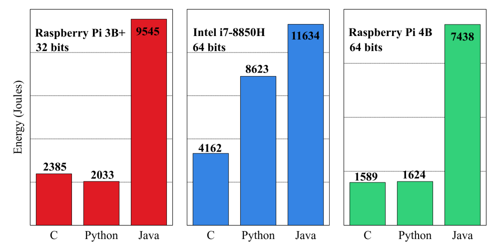

Fig. 4. Energy consumption of Ray casting algorithm on different programming languages and platforms

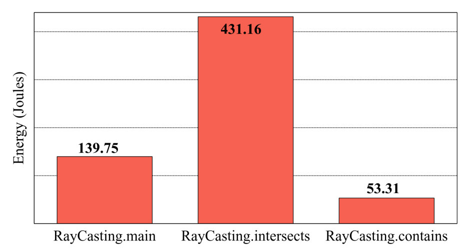

Fig. 5. Energy consumption of the Java implementation of the Ray casting methods

Figure 4 shows the total energy consumption of all experiments for both platforms. However, each implementation prints different output to the Linux terminal. In particular, the Java and Python version prints multiple lines of text, while the C version prints 3 numbers on each loop. Therefore, the results should not be used to compare programming languages, but rather to compare energy distribution across different platforms. In particular, we observe that the Python program consumed much more energy, compared to the C program, on the Intel-based computer, while it consumed less on the Raspberry Pi devices. This can be partially explained by the different software stack (Python and GCC versions), and the execution time of both programs: Python code on Intel took

[7]https://rosettacode.org/wiki/Ray-casting_algorithm

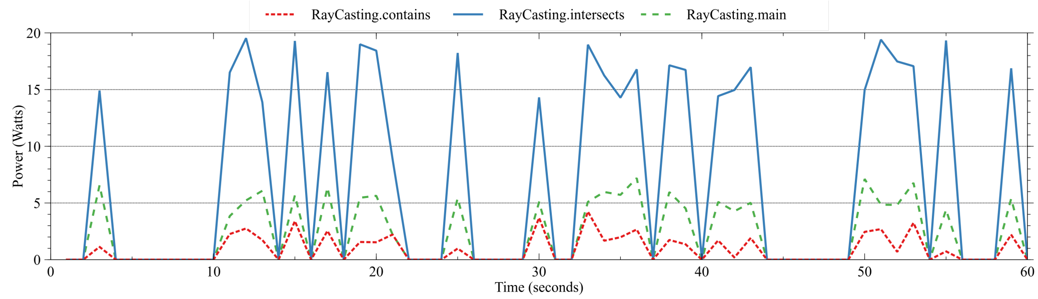

Fig. 6. Runtime power consumption of the Java implementation of the Ray casting methods

3 minutes and 28 seconds (for C: 3 min 14 sec), and on Raspberry Pi it took 12 min and 32 sec (RPi 3B+) and 7 min and 38 sec (RPi 4B) (for C: 14 min and 8 sec for RPi 3B+ and 7 min and 23 sec for RPi 4B). `PowerJoular` allows such comparisons and studies across multiple platforms. In this particular example, a software programmer might decide to use the C version on a server, and the Python version on a Raspberry Pi, instead of using the C version on both platforms if the programmer solely relied on energy consumption of only the Intel computer.

We also run `JoularJX` on the Java program in order to get insights on methods' energy consumption. We use a modified version of the Ray casting program where we added additional print commands in each method, and added a loop in the main method (for 50 000 iterations). We collect the total energy consumption of all methods (including those from the JDK), and the ones only from the program, and we collect runtime power consumption every second for them too. Figure 5 shows the total energy consumption of our Java program. This in-depth details help developers understand where are the energy hotspots of their programs [7]. However, `JoularJX` introduces power monitoring of methods in real time. Figure 6 outlines the power consumption of the methods for the duration of the program's execution. This insight allows developers and automated tools to detect power variations in real time, and understand power draws in different scenarios. For example, a developer might run a program with different input values, sequentially, and analyze the power draw automatically.

Our tools can, therefore, be incorporated into integrated development environments (IDEs), testing frameworks, or be used in pre-production servers, with a goal to help developers understand power consumption in software and write power-efficient multi-platform software.

V. Conclusion

In this paper, we presented `PowerJoular`, a mutli-platform tool that can monitor the power consumption of PCs, servers, and single-board computers (such as Raspberry Pi). It uses Intel RAPL interface in Linux for servers, and our own empirical regression power model for Raspberry Pi devices. We also presented `JoularJX`, a Java agent capable of monitoring, in real time, the power consumption of every method in a program.

Our tools are aimed towards software developers in helping them understand and analyze the power footprint of their software and source code, across multiple platforms and devices. It can also be used by system administrators and automated tools to monitor, in real time, a large number of devices (such as through a dashboard), and use the power data to take energy-aware decisions. Currently, an active probe displays energy information on the screen or in a file, which may incur a small overhead on the system resources. Although negligible in most situations, we plan to extend our tools to support on-demand monitoring, and providing data through OSs' interfaces such as D-Bus. We plan to extend `PowerJoular` to support additional devices, operating systems, architectures and hardware components. And we plan to expand `JoularJX` to support software written in other programming languages.

References

[1] Aurélien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Powerapi: A software library to monitor the energy consumed at the process-level. *ERCIM News*, 2013(92), 2013.

[2] Thanh Do, Suhib Rawshdeh, and Weisong Shi. ptop: A process-level power profiling tool. In *in Proceedings of the 2nd Workshop on Power Aware Computing and Systems (HotPower'09*, 2009.

[3] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), March 2018.

[4] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *IEEE Symposium on Security and Privacy (SP)*, 2021.

[5] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 237–248, New York, NY, USA, 2016. Association for Computing Machinery.

[6] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. A review of energy measurement approaches. *ACM SIGOPS Operating Systems Review*, 47(3):42–49, 2013.

[7] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. Monitoring energy hotspots in software. *Automated Software Engineering*, 22(3):291–332, 2015.

[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] Gustavo Pinto and Fernando Castor. Energy efficiency: A new concern for application software developers. *Commun. ACM*, 60(12):68–75, November 2017.