EnergyTrackr: A Modular Energy Regressions Detection Tool

François Bechet
University of Namur
Namur, Belgium
francois.bechet@tutanota.com

Jérôme Maquoi✉
University of Namur
NADI
Namur, Belgium
jerome.maquoi@unamur.be

Luís Cruz
TU Delft
Delft, Netherlands
L.Cruz@tudelft.nl

Benoît Vanderose
University of Namur
NADI
Namur, Belgium
benoit.vanderose@unamur.be

Xavier Devroey
University of Namur
NADI
Namur, Belgium
xavier.devroey@unamur.be

Abstract

Energy efficiency is increasingly recognized as an important dimension of software quality. As systems evolve rapidly, tracking how energy consumption changes across versions can provide valuable insights for developers and researchers. Although detecting energy regressions (i.e., unintended increases in energy consumption) remains challenging due to measurement variability and complex execution environments, recent advances now make systematic analysis practical. Still, dedicated support for monitoring energy behavior over a project's history is missing. This paper presents EnergyTrackr, a modular, open-source tool for automatically detecting, classifying, and visualizing energy regressions across a commit history. EnergyTrackr traverses a specified set of commits, builds each revision, and measures application-level energy consumption by executing the test suite using the RAPL-based Linux perf utility. It applies established best practices for measurements such as warm-up, randomized execution, repetition, and thermal control. The resulting data are analyzed through a statistical pipeline that filters outliers and classifies energy changes using multiple criteria (e.g., significance, practical impact, ...). EnergyTrackr generates interactive reports that provide a comprehensive view of energy evolution and is designed for easy integration into existing Linux workflows. A preliminary evaluation on large Java projects shows that EnergyTrackr can produce stable measurements at scale and effectively identify energy regressions. Source: snail-unamur/energytrackr. Screencast: doi:10.5281/zenodo.21507425.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; • **Hardware → Power estimation and optimization**.

Keywords

energy regression, software testing, green software engineering

ASE '26, Munich, Germany

ACM Reference Format:
François Bechet, Jérôme Maquoi, Luís Cruz, Benoît Vanderose, and Xavier Devroey. 2026. EnergyTrackr: A Modular Energy Regressions Detection Tool. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3832783.3834640

1 Introduction

Recently, the software engineering research community has dedicated significant effort to exploring various facets of software sustainability and green software engineering [7, 10, 13]. Despite these advancements, the field remains in its early development, with numerous challenges yet to be addressed. One such challenge involves identifying commits that may lead to *energy regressions*, an increase in energy consumption between two software versions. Detecting these regressions in software projects is complex due to numerous factors, including measurement, regression detection, and comprehensive overview of the collected data.

There are three main methods to assess the energy consumption of software [6]: (i) *System-level physical measurements* using physical power meters, requiring a complex setup, and only providing insights at the hardware level. (ii) *Energy predictive models* offer the benefit of eliminating the need for additional infrastructure or time-consuming measurement processes, while enabling analysis of energy consumption during execution. However, existing research indicates that their overall accuracy may be limited [12]. (iii) *On-chip power sensors* with software interfaces can measure power consumption of components such as CPUs, GPUs, disks, and RAM. In particular, Intel's Running Average Power Limit (RAPL) is one of the most accurate tools for measuring processor energy consumption [9]. We focus on energy measurement at the application level, as it is more portable and introduces less overhead (and therefore potential biases) in the measurement process. Our implementation relies on the standard Linux `perf` utility to measure CPU energy consumption via RAPL. We can then leverage commit changes to identify possible code patterns that cause those changes.

Danglot et al. [5] introduced the concept of *energy regression testing*, extending the scope of regression testing (*i.e.,* ensuring that recent changes have not introduced new defects in pre-existing functionality) to monitor changes in software energy consumption resulting, *e.g.,* from source code modifications. Building on this, we

seek to provide an extended view of energy consumption throughout the project's history, including potential energy regressions.

We propose EnergyTrackr, an approach to help minimize the energy consumption associated with software development. In a nutshell, it considers the entire commit history (or an arbitrary portion of it), builds each commit individually, and measures its energy consumption based on test case executions. EnergyTrackr provides statistical analysis support with configurable heuristics to identify energy regressions and perform triage, as well as graphical representations of the history to help developers spot problematic code patterns that may be causing an energy regression. A difference between two commits is marked as a regression if the difference in their energy consumption is statistically significant.

Other approaches to identifying energy regressions exist. Poy et al. [13] conducted a systematic mapping examining how design patterns, code smells, and refactoring techniques influence software energy consumption. Regarding code smells, Poy et al. [13] suggests that removing nearly all code smells in their primary studies enhances software energy efficiency. However, Connolly Bree and Ó Cinnéide [3] conducted a systematic literature review examining the impact of various design elements, including code smells, on software energy consumption and highlighted that removing code smells does not always reduce energy usage. Further research is needed to better understand instances in which refactoring efforts may cause energy regressions due to code smells. EnergyTrackr goes beyond a single code change, as it is specifically tailored to identify energy regressions by tracking the evolution of code commits over time and can serve as a basis for future research.

2 Challenges for energy assessment

Energy assessment faces several challenges, including choosing the right *level of granularity* (overall versus code-block energy consumption). EnergyTrackr operates at a mid-grained granularity, as our objective is to identify energy regressions across commits using tests execution. The measurement also has to account for *inherent variability arising from external factors* such as component temperatures (CPU, GPU, memory), ambient room temperature, background system processes, or thread scheduling [4]. EnergyTrackr relies on Cruz's [4] widely adopted principles [7]: (i) **Zen mode** seeks to minimize the computer environment impact, for example, by closing as many background processes as possible. (ii) **Freeze your settings** saves the current state of the machine and reuses that same setting during the entire testing process. (iii) a **Warm-up** phase accounts for the fact that a CPU performs more efficiently when its temperature stabilizes after being idle. (iv) **Repetition** addresses measurement instability by performing multiple runs and using the median as the final value. Cruz [4] recommends 30 measurements. (v) **Resting** between measurements accounts for temperature influencing CPU energy efficiency, causing a tail effect where the end of one measurement affects the next (i.e., increasing temperature). (vi) **Shuffling** measurements by creating batches of tasks executed in a random order, e.g., measuring n commits k times results in $n \times k$ tasks, can help mitigate variability due to background processes. (vii) **Keep it cool** mitigates the risk that the CPU gets too hot and starts throttling, which can impact the measurements. (viii) **Automated executions** enable the reproducibility of the measurement process.

3 Approach and implementation

In a nutshell, the core steps of EnergyTrackr are as follows: (i) running over each commit of the project to measure its energy consumption based on test cases execution; (ii) sorting the result, as tasks are run in a random order to counter environment effects on energy measurements; and (iii) building a dynamic report containing the analysis of the results and flagging potential energy regression commits.

Preparation phase. This phase stabilizes the Linux machine for repeatable, stable energy measurements by disabling power-saving features and pinning frequencies (**Zen mode**) via a configuration file. All modifications are reversible, as the device parameters are copied rather than directly altered, allowing users to switch between configurations of EnergyTrackr (**Freeze your settings**).

Stability verification. A stability verification step, which can also be performed in subsequent stages if needed, is performed before conducting the measurements (**Warm-up**). We rely on the modified z-score, robust to non-normal distributions and reliable for small datasets, to detect outliers. The verification collects five initial warm-up samples to establish a baseline. Then, measurements are taken from 30 additional samples, each of which is individually compared to the established baseline. If any measurement's modified z-score exceeds the predefined threshold of 3.5, the system is flagged as potentially unstable, and the process is halted.

Measurement phase. We choose to wrap the entire test suite execution (with `perf` in our implementation) rather than instrumenting individual tests for greater precision. This method offers benefits such as reduced overhead and improved measurement accuracy. Jay et al. [8] have established that `perf` is reliable when complemented by additional techniques, such as repeated measurements.

First, EnergyTrackr clones the target repository and collects the relevant commit history for analysis. Commits are then organized into batches, which are randomized to counter environmental effects on measurement quality (**Shuffle**). After that, all commits within each batch are built. The energy consumption measurement is performed during test suite execution. The test execution command can be configured to accommodate different build systems and test environments (*e.g.,* `mvn test` for Java projects). The pipeline can be configured to run the same commit a specified number of times (**Repeat**), with a default of 30 repetitions. Additionally, we provide a custom bash script that sets up the machine on which EnergyTrackr will run (**Automate executions**).

The current CPU temperature is monitored to ensure it remains within safe operational limits (**Keep it cool**). If temperatures are too high, the process pauses until cooling conditions are achieved. This ensures that the temperature remains stable without requiring pausing execution between measurements, which slows down the pipeline (**Rest**). Energy consumption measurements are then collected for each commit in the batch and compiled for analysis.

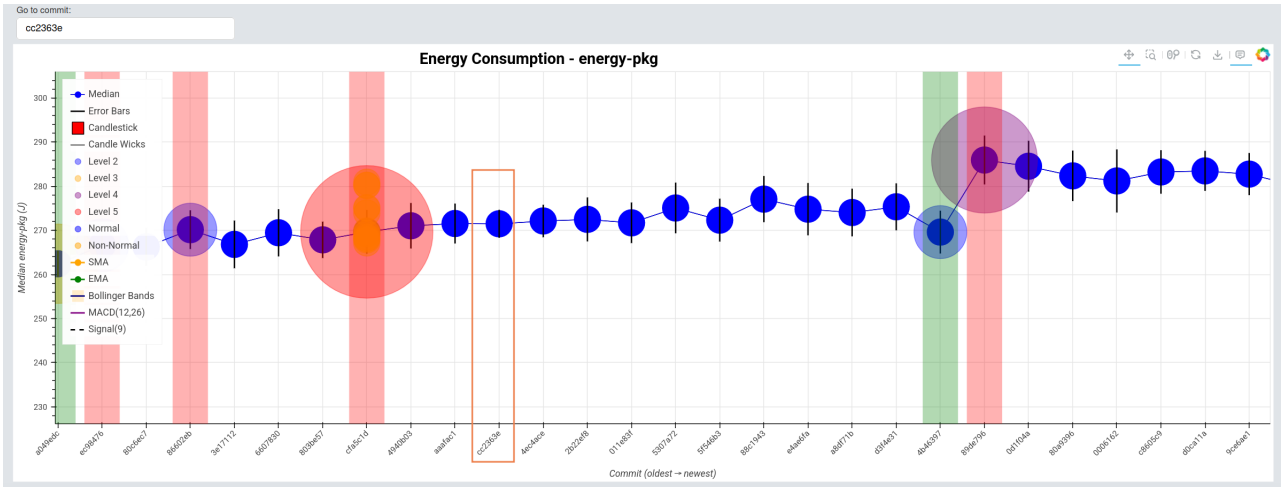

Figure 1: Evolution plot showing the commit `cc2363e` and nearby commits from JSoup. Each blue dot is the median energy consumption, with corresponding black error bars. The red (resp. green) vertical bars represent commits with a level of at least 2 with an energy regression (resp. improvement). Commits with an additional colored circle denote regressions with their certainty level: blue for level 2, orange for level 3, purple for level 4, and red for level 5. The levels can be cumulated and are all visually represented.

Regression detection and classification phase. After randomized execution, the data are sorted for analysis. In our implementation, this step produces CSV files. The data are grouped by commit and sorted in descending order by date before being preprocessed. *Data preprocessing.* The first stage checks, for each commit c_i, whether *the data follows a normal distribution* (H_0) using the Shapiro-Wilk test for the different energy measures for c_i. It is important to note that commits with non-normal distributions can still be classified as significant changes, since distributional normality is not part of the significance classification. It serves as an additional indicator of measurement quality. Second, following best practices [4], we filter out outliers: transient outlier commits, *i.e.,* drops in sustained energy that indicate potential outliers rather than real regressions, are detected by using the interquartile range (IQR) statistic, with a Tukey multiplier of 1.5. The following can be configured in EnergyTrackr: the rolling window range of commits used to detect outliers, the Tukey multiplier, the maximum number of transient outlier commits, and the aggregation method for combining multiple measurements.

Change detection. Significant changes or regressions (level 1) in the energy data are sorted into *five certainty levels.* A commit is tested for each level above 1 only if it passes level 1. The highest level the change satisfies determines its classification, meaning a change can satisfy multiple levels or fail some intermediate ones: *e.g.,,* a change not passing statistical tests, but with a commit message containing the word "*performance*" will be marked as *level 5* as information from the developers is considered predominant. Those settings can be configured, depending on the project: **Level 1**: Performs a parametric Welch's unequal variances t-test to assess statistically significant difference with a threshold $\alpha = 0.05$. If the commit passes this test, then it is classified as *level 1* and considered a regression (or improvement). **Level 2**: Calculates Cohen's d to measure effect size. It uses the following thresholds to classify the change: small for $0.2 < |d| \leq 0.5$; medium for $0.5 < |d| \leq 0.8$; large for $|d| > 0.8$. If the commit is at least in the small category ($0.2 < |d|$), then it is classified as *level 2.* **Level 3**: Computes the relative change $\Delta\%$ on the median energy values. Let m_b, resp. m_t, be the median of the baseline sample, resp., test sample, $\Delta\% = \left|\frac{m_t - m_b}{m_b}\right|$. The change is minor for $\Delta\% < 5.0\%$; moderate for $5.0\% \leq \Delta\% < 10.0\%$; and major for $\Delta\% \geq 10.0\%$. If the commit is at least moderate, then it is classified as *level 3.* **Level 4**: Computes the practical significance ΔJ, *i.e.,* how strong the regression is relative to the baseline. Let m_b, resp. m_t, be the median of the baseline sample, resp., test sample, $\Delta J = m_t - m_b$. The change is info for $\Delta J < 5.0\% \times m_b$; warning for $5.0\% \times m_b \leq \Delta J < 10.0\% \times m_b$; critical for $\Delta J \geq 10.0\% \times m_b$. If the commit is at least in the warning category, then it is classified as *level 4.* Improvements will always exhibit a negative ΔJ and be categorized under the "*info*". **Level 5**: When given, configurable, context tags are present in the commit message (*e.g.,* words like '*performance*' or '*regression*'), then the commit is flagged to *level 5.* Although it could raise false positives, the combination of different levels can provide information about the presence of a regression (or its absence despite the commit message).

Cruz [4] and Danglot et al. [5] inspired the design of levels 1 to 3. The additional levels are designed to provide exhaustive and comprehensive classification and are specific to EnergyTrackr. This combined approach ensures that only meaningful changes are reported, filtering out random fluctuations while highlighting regressions that developers should address in order of priority.

Data visualization phase. To enable developers to analyze results and identify energy regressions based on our classification, EnergyTrackr provides a report with various interactive plots, additional information, such as the level's configuration with the various thresholds, and a table listing the different commits. EnergyTrackr supports the following plots: **Evolution** plot shows

the history of the commit range. For instance, Figure 1 presents the evolution of energy consumption for a portion of JSoup's commit history. To the right, a level 4 regression is highlighted in purple, indicating that this commit has been classified as "*warning*" with a negative impact estimated at 5.0%-10.0% relative to the previous commit. **Boxplots and violin** plots allow for comparing two commits. **Quantile-Quantile** plot compares the distributions of two commits. **Bootstrap plot** is an interactive histogram that helps confirm if the difference between two commits is statistically significant. **Cusum plot** shows the cumulative sum of improvements and regressions for the commit history. **Change point detection** plot highlights change points in the median energy data.

Open source implementation. EnergyTrackr is implemented as an open-source tool available on GitHub. It follows a *pipes-and-filters architecture* to offer great modularity: each phase is implemented as a separate module, allowing for easier experimenting and extensibility. This choice of modular design with proper separation between configuration, energy measurement, reporting, and shared utilities was deliberate to ensure: (i) cross-language capabilities in both the analysis of different languages projects (currently, only Java projects are supported);(ii) a better reproducibility, thanks to two centralized configuration files; and (iii) an easier integration into existing developers' workflow and pipelines.

4 Preliminary evaluation

We validate our approach for energy regression detection by running EnergyTrackr on a selection of real-world GitHub projects with strong test suites. In our preliminary work [1, 2], we selected three Java projects, based on the following inclusion criteria: (i) a *long commit history* with at least 500 commits for rich comparison; (ii) a *good test suite* executable over the commit history; (iii) *open source* popular projects recommended in the Awesome-Java list of open source projects; (iv) relying on a *dependency management system* like Maven; and (v) the project should use the *Git version control system*. The selected projects are `Jsoup`, `univocity-parsers`, and `fastexcel`, listed in Table 1 with their characteristics. We used two machines to provide the energy results, each equipped with high-end CPUs, proper cooling systems, and certified Power Supply Units (PSUs) to maximize stability. The exact specifications are detailed in Bechet et al. [2].

Results. Table 1 contains the results summary for each project. The `fastexcel` project, although the smallest, consumes the most energy (812.75 joules on average). It is also the one with the fewest level 5 changes. One is an improvement: commit 0cfb264 (message: "*e2e maven profile for running test with **memory** constraints*") decreases energy consumption from 1,338.64J to 588.52J, and was marked as level 5 because of the presence of the word '*memory*'. The other level 5 commit is commit 9b226e2 (message: "*[...] update poi read streaming **benchmark***"), increasing energy from 644.05J to 651.97J. This small increase is confirmed by the *minor* relative change (Δ%) and practical significance Δ*J* set to *info*. However, the commit message contains the word '*benchmark*', one of the default words leading to labeling a commit as level 5. It illustrates that levels are not necessarily cumulative. In this case, a change in the software's benchmarking policy that impacts energy consumption could lead to energy issues, hence, level 5, requiring attention.

Table 1: Results summary

	Jsoup	univocity-parsers	fastexcel
JDK version	24.0.1	1.6	1.8
Lines of Code	9450	9700	2222
Tot./Failed/Ignored	1490 / 0 / 46 tests	1106 / 2 / 0 tests	81 / 0 / 0 tests
Line Coverage	88% (8406 / 9450)	77% (7519 / 9700)	87% (1950 / 2222)
Branch Coverage	83% (4170 / 4976)	74% (4220 / 5651)	70% (699 / 997)
Total commits	1944	788	500
Commit range	2bc4205 — 01b3900	e536b75 — ed3d0d3	b978577 — 66a9dcb
Commit dates	(2011-07-02) — (2025-04-03)	(2014-07-12) — (2021-04-19)	(2020-04-24) — (2025-05-18)
Significant changes	216 (124 regressions)	381 (189 regressions)	113 (61 regressions)
Med./Mean/σ energy (Joules)	190.86 / 190.43 / 126.51	131.56 / 127.05/ 9.99	801.75 / 812.75 / 198.54
Normal dist.	1801 / 1944 (92.64%)	691 / 788 (87.69%)	439 / 500 (87.8%)
Outliers removed	42	25	38
Rel. change (Δ%)	190 minor, 12 moderate, 14 major	376 minor, 2 moderate, 3 major	104 minor, 3 moderate, 6 major
Pract. signif. (Δ*J*)	197 info, 10 warning, 9 critical	377 info, 2 warning, 2 critical	108 info, 1 warning, 4 critical
Level 2 / 3 / 4 / 5	155 / 33 / 18 / 10	360 / 0 / 4 / 12	98 / 8 / 5 / 2

Of the 1,944 commits analyzed for the `Jsoup` project, 92.64% of them passed the Shapiro-Wilk normality test: 216 significant changes were detected, and 124 were classified as regressions. From the 788 commits of `univocity-parser`, 87.69% passed the Shapiro-Wilk normality test: 381 were significant, and 189 were classified as regressions. For `fastexcel`, out of the 500 commits analyzed, 87.8% passed the Shapiro-Wilk normality test: 113 were significant, and 61 were classified as regression. In summary, the EnergyTrackr approach enables the production of stable and consistent energy values across repeated test executions, as demonstrated by a high proportion of normal distributions. The data and report can serve as a strong basis for identifying energy regression.

5 Discussion and future work

Variability during measurement. Measurement variability is mitigated through repeated tests, randomized execution and thermal control via a temperature-check stage to prevent thermal throttling. RAPL is limited as it only monitors the whole CPU and cannot pinpoint which components (memory, storage, network) cause energy regressions. Once a regression is detected, a profiler can then help locate the code responsible for the extra energy use. Batch size is another factor. Smaller batches reduce the benefit of randomization, while larger batches increase the time between a commit's first and last measurements, raising variance from environmental noise. This parameter must be tuned in a pilot study for each application.

Measurement limitations. Using `perf` introduces some limitations, including dependence on RAPL support, which may not be available on all environment. Additionally, `perf` is available only on Linux, limiting its applicability to Linux-based systems. Future work includes the support of other measurement tools (*e.g.,* JoularJX [11]). Another improvement to EnergyTrackr would be to assess the non-normality of measurements directly within the measurement pipeline and automatically schedule non-normal commits for re-measurement. This would strengthen the results by ensuring a higher number of normally distributed measurements.

Changes in the test suite. Building on Danglot et al. [5], we assume that running tests can reveal energy regressions, especially

with good coverage. While test-based measurements can be biased by test changes that must be ignored, they avoid writing custom tests by hand and thus scale without requiring deep project-specific knowledge. However, the number of passing tests was not measured, so some commits spotted as regressions may be due to failing tests rather than true energy regressions.

Energy regression detection. The current algorithm is straightforward: it traverses the entire commit history sequentially. While this approach effectively identifies all potential energy fluctuations, it is relatively time-intensive and may include examining extensive commit ranges with minimal or no significant energy variation. In our future work, we plan to devise a more efficient algorithm, based on the binary search, that will take measurements into account to slice the commit history and reduce the number of analysis.

6 Conclusion

We presented and evaluated EnergyTrackr, a modular energy regression detection and reporting tool. Built on `perf` and RAPL, it detects regressions across commit histories using statistical metrics and categories. Our approach identified significant regressions in multiple commits. EnergyTrackr offers a comprehensive view of energy changes at the commit level, with an intuitive reporting interface and a highly configurable system suitable for diverse projects. Besides improvement of the tool itself (*i.e.,* supporting other programming languages, non-normality evaluation directly during the execution, automatic test suite change detection, *etc.*), our future work will extend the evaluation of EnergyTrackr.

Data Availability Statement

Replication package available in Zenodo under MIT License [1]: F. Bechet, J. Maquoi, L. Cruz, B. Vanderose, and X. Devroey. 2026. Replication package of EnergyTrackr. doi:10.5281/zenodo.21506868

Acknowledgments

This research was partially funded by the CyberExcellence by DigitalWallonia project (No. 2110186) funded by the Public Service of Wallonia (SPW Recherche).

References

[1] François Bechet, Jérôme Maquoi, Luis Cruz, Benoît Vanderose, and Xavier Devroey. 2026. *Replication package of EnergyTrackr: A modular energy regressions detection tool.* doi:10.5281/zenodo.21506868

[2] François Bechet, Jérôme Maquoi, Luís Cruz, Benoît Vanderose, and Xavier Devroey. 2026. Systematic Detection of Energy Regression and Corresponding Code Patterns in Java Projects. (2026). arXiv:2604.19373 [cs.SE]

[3] Déaglán Connolly Bree and Mel Ó Cinnéide. 2025. How Software Design Affects Energy Performance: A Systematic Literature Review. *J. Soft.-Evol. Proc.* 37, 4 (2025), e70014. doi:10.1002/smr.70014

[4] Luís Cruz. 2021. Green Software Engineering Done Right: A Scientific Guide to Set Up Energy Efficiency Experiments. doi:10.6084/m9.figshare.22067846.v1

[5] Benjamin Danglot, Jean-Rémy Falleri, and Romain Rouvoy. 2024. Can We Spot Energy Regressions Using Developers Tests? *Empir. Softw. Eng.* 29, 5 (2024), 121. doi:10.1007/s10664-023-10429-1

[6] Muhammad Fahad, Arsalan Shahid, Ravi Reddy Manumachu, and Alexey Lastovetsky. 2019. A Comparative Study of Methods for Measurement of Energy of Computing. *Energies* 12, 11 (Jan. 2019), 2204. doi:10.3390/en12112204

[7] Stefanos Georgiou, Maria Kechagia, Tushar Sharma, Federica Sarro, and Ying Zou. 2022. Green AI: Do Deep Learning Frameworks Have Different Costs?. In *ICSE '22*. ACM, 1082–1094. doi:10.1145/3510003.3510221

[8] Mathilde Jay, Vladimir Ostapenco, Laurent Lefevre, Denis Trystram, Anne-Cécile Orgerie, and Benjamin Fichel. 2023. An Experimental Comparison of Software-Based Power Meters: Focus on CPU and GPU. In *CCGrid '23*. IEEE, 106–118. doi:10.1109/CCGrid57682.2023.00020

[9] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ToMPECS* 3, 2 (March 2018), 9:1–9:26. doi:10.1145/3177754

[10] Olivier Le Goaer and Julien Hertout. 2023. ecoCode: A SonarQube Plugin to Remove Energy Smells from Android Projects. In *ASE '22*. ACM, 1–4. doi:10.1145/3551349.3559518

[11] Adel Noureddine. 2022. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *IE '22*. IEEE, 1–4. doi:10.1109/IE54923.2022.9826760

[12] Kenneth O'brien, Ilia Pietri, Ravi Reddy, Alexey Lastovetsky, and Rizos Sakellariou. 2017. A Survey of Power and Energy Predictive Models in HPC Systems and Applications. *ACM Comput. Surv.* 50, 3 (2017), 37:1–37:38. doi:10.1145/3078811

[13] Olivia Poy, M Ángeles Moraga, Félix García, and Coral Calero. 2025. Impact on Energy Consumption of Design Patterns, Code Smells and Refactoring Techniques: A Systematic Mapping Study. *J. Syst. Software* 222 (2025), 112303. doi:10.1016/j.jss.2024.112303

Received 2026-05-11; accepted 2026-06-19