Energy Codesumption, Leveraging Test Execution for Source Code Energy Consumption Analysis

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

Maxime Cauz
NADI, University of Namur
Namur, Belgium
maxime.cauz@unamur.be

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

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

Abstract

The software engineering community has increasingly recognized sustainability as a key research area. However, developers often have limited knowledge of effective strategies to reduce software energy consumption. To address this, we analyze energy consumption in software execution, aiming to raise developer awareness by linking energy consumption with each line of code. We rely on unit test executions to identify energy-intensive executions and manually analyze five hot and five cold spots to identify potentially energy-intensive source code constructs. Our findings suggest a link between the energy consumption of the source code and the number of objects' attributes created within that code. These results lay the groundwork for further analysis of the relationship between object instantiation and energy consumption in Java.

CCS Concepts

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

Keywords

energy consumption, source code analysis, test execution, java

ACM Reference Format:
Jérôme Maquoi, Maxime Cauz, Benoît Vanderose, and Xavier Devroey. 2025. Energy Codesumption, Leveraging Test Execution for Source Code Energy Consumption Analysis. In *33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), June 23–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3696630.3728707

1 Introduction

Reducing the environmental impact of all aspects of IT, including energy consumption of running software, is becoming crucial to achieving a more sustainable society. In recent years, various aspects of software sustainability and green software engineering have been investigated by the research community [4, 20, 22–24, 27]. In particular, [17] has shown that developers recognize, to a certain extent, the challenges associated with software energy consumption. However, they often lack knowledge of effective strategies to reduce their software's energy footprint. To bridge this gap, we need (i) methods to measure and report source code energy consumption and (ii) identify code constructs that increase energy usage to develop appropriate tooling.

This short paper lays the foundations for our research objective: gaining a deeper understanding of the underlying causes of energy consumption from the source code. More specifically, we answer the following research question (**RQ**): *how does the source code of a Java project impact its energy consumption?* For that, we rely on JoularJX [16], a state-of-the-art tool, to measure energy consumption at the source code level of Java projects (i.e., each execution branch is associated with its energy consumption). Based on the measurements, we identified high- and low-energy-intensive parts of the code and performed a manual analysis to identify recurring code constructs.

2 Background and related work

A recent survey [13] highlights the need for improved skills among developers in energy-aware development. Developing such skills should be paired with raising developer awareness about the energy consumption of their code, which requires tools that can assess the energy consumption associated with the source code.

Energy consumption assessment. The first method relies on theoretical models to estimate consumption without direct measurements. For example, the TEEC model estimates CPU, memory, and disk power usage during application execution through established mathematical expressions [1]. Alternatively, physical measurement utilizes a power meter connected to the hardware. This approach provides the most accurate representation of energy consumption during software execution. For instance, a framework that detects energy hotspots in Android applications employs a physical power meter for accurate detection of these energy-intensive areas [3]. More recently, hardware sensors with software interfaces have been able to measure components like CPUs, GPUs, RAMs, and disks. Manufacturers provide interfaces for power data, such as Intel's Running Average Power Limit (RAPL) for CPUs and NVIDIA Management Library (NVML) for NVIDIA GPUs. RAPL provides CPU electricity consumption data to the operating system [25], making it popular in research for CPU power measurement [11, 14, 26].

FSE Companion '25, Trondheim, Norway

It can be used directly, for example, to compare the energy consumption of various Java I/O libraries and methods [18], or through specialized tools: such as PowerAPI, a tool that provides real-time insights into the power consumption of software applications [10], and JoularJX, used in this work, which is designed for monitoring power usage of Java projects at the source code level [16]. It works as a Java-based agent that seamlessly integrates with the Java Virtual Machine (JVM) at the program's startup.

Development tools. [8] highlights the challenge of notifying developers about increased energy consumption in applications during Continuous Integration (CI). It indicates that developers' tests can identify energy regressions due to code changes, though effectiveness varies by configuration. In contrast, we focus on a more detailed analysis at the source code level.

[24] explored the relationship between source code patterns and energy consumption. Specifically, they examined how design patterns, code smells, and refactoring techniques impact energy consumption. Their findings suggest that, in general, implementing design patterns may decrease energy efficiency, while removing code smells tends to enhance it. They found no conclusive link between refactoring techniques and energy consumption. In this study, we aim to investigate the relationship between source code and energy consumption by analyzing the effects of Java constructors on energy usage.

Several studies have investigated the energy consumption associated with mobile applications. This area of research is critical due to the concerns developers have regarding battery performance [7]. For example, [7] evaluated eight mobile UI automation frameworks for mobile app energy efficiency, finding that some can increase energy consumption by up to 2200%. They also provided a decision tree to help developers choose the best framework for their needs. [12] researched energy-related code issues in Android, which can be identified through code inspection, leading to the creation of ecoCode, a SonarQube plugin that identifies energy-inefficient code structures. They compiled a catalog of 40 such issues for the Android platform. Unlike these two studies, our focus is on projects beyond mobile applications.

Spectrum-based Energy Leak Localization (SPELL) [21] identifies energy hotspots in programs using fault localization methods. Experiments on five projects demonstrated SPELL's effectiveness in helping developers improve energy consumption. Unlike SPELL, targeting energy hotspots, our approach investigates potential connections between hotspots and the associated source code.

The Power Measurement Toolkit (PMT) [5] is a high-level library for collecting power consumption data from Python and C++ code across CPU and GPU architectures. Evaluated against a benchmark, PMT effectively assesses application energy efficiency through an intuitive interface on various hardware configurations. A key distinction between this study and our approach lies in the requirement for PMT to modify the source code to include instructions for initiating and terminating measurements. In contrast, our approach aims to measure energy consumption by executing project tests without modifying the source code.

Table 1: Projects specifications used for the evaluation

	Spring Boot	Spoon
Version	v3.1.4	v10.4.2
Commit SHA	3ed1f1a064a10e53adc2ad8c0b4-6a4b2c148ee21	066f4cf207359e06d30911a553d-edd054aef595c
JDK version	19	17
Total / Failed / Ignored tests	4217 / 0 / 12	4276 / 0 / 12
Lines of Code (LOC)	23,358	28,739
Class Coverage	76% (795 / 1037)	97% (922 / 943)
Method Coverage	70% (4662 / 6630)	88% (6691 / 7546)
Line Coverage	68% (16031 / 23,358)	87% (25,045 / 28,739)
Branch Coverage	65% (5902 / 9062)	77% (10,822 / 14,020)

3 Evaluation Setup

To answer our RQ, we analyze two Java projects, Spoon [19] and Spring Boot[1], using JoularJX [16] to measure their code energy consumption by executing their test suites. We manually analyzed the highest and lowest consuming code locations to identify code constructs. Our results are available in our replication package [15].

Energy consumption measurement. Energy measurements, even when conducted on the same computer, can be influenced by various factors. We follow the best practices [6] to account for this. First, we freeze the system's configurations and ensure that only our software runs so other background system processes operate uniformly across all experiments. All the experiments are executed on a dedicated Ubuntu 22.04.5 server with 64 Intel(R) Xeon(R) Gold 6326 (2.90GHz) and 256GB of RAM. We also warm up the system by conducting a five-minute preliminary test before the actual measurements to stabilize the temperature.

As pointed out by [8], executing a project's test suite provides an effective method for approximating the total energy consumption of the project. In our case, we execute the projects' test suite 30 times to account for randomness [2, 6] using a modified JoularJX based on v2.8.2 to add line numbers to the collected data (as illustrated in Listing 1). To prevent temperature increases between executions leading to artificially elevated energy consumption, we implement a one-minute cooling down period between each test suite execution. Lastly, we maintain a stable room temperature for the testing equipment to mitigate any environmental factors that might affect the energy measurements.

Bash scripts automate cloning project repositories at specific commits, preparing the JoularJX agent, and executing the JUnit test suite 30 times per project, including a warm-up phase and rest intervals between each execution. The resulting data is stored in a MongoDB database for analysis.

Selected projects. This study analyzes Spoon and Spring-Boot. Spoon is an open-source library for analyzing, transforming, and transpilating Java source code. It is well-tested and has high code coverage, ensuring energy consumption can be measured across a large part of the source code. Spring-Boot is a widely adopted framework, making it relevant for our evaluation, for creating standalone Java applications such as microservices and large-scale enterprise systems. It streamlines the Java development process by providing pre-configured templates and dependencies. This study focuses on

[1] https://github.com/spring-projects/spring-boot

```
spoon.[...].jdt.JDTBasedSpoonCompilerTest.
      testOrderCompilationUnits 35
spoon.[...].jdt.JDTBasedSpoonCompiler.buildUnits 418
spoon.[...].jdt.JDTBatchCompiler.getUnits 282
spoon.[...].jdt.TreeBuilderCompiler.buildUnits 82
```

Listing 1: Example of Java stack trace from the spoon project with, for each frame, its corresponding line number

the *spring-boot-project/spring-boot* sub-project. The characteristics of those two projects are presented in Table 1.

Data analysis. The data collected by JoularJX during the experiment comprises energy consumption linked to *Call Traces* (CTs). Each CT indicates a nesting of method calls with the corresponding energy consumed along that specific execution point. For example, Listing 1 shows an example of a stack trace with four lines (called *frames*) from the Spoon project indicating that method `testOrderCompilationUnits` calls method `buildUnits` at line 35, which calls method `getUnits` at line 418, etc. Linked to an energy consumption value, for instance 318 joules, this stack trace is called a call trace. The energy consumption represents the energy consumed by the CPU for the entire stack trace for one execution of the Spoon test suite. We store each collected call trace and its corresponding energy consumption value, generated by executing the corresponding test suite for a project, in a MongoDB database to ease the analysis.

Following Cruz's recommendations [6], we execute a series of steps to process and analyze the data collected from the database. (i) We retain CTs only for those instances where JoularJX collected energy consumption data at least 25 times, as this threshold ensures the suitability of the data for our intended analysis. This results in 50 CTs for Spring Boot and 43 CTs for Spoon; (ii) We filter outliers from our dataset using the standard deviation method consisting in removing all the data points that deviate from the mean more than three times the standard deviation [6], i.e., $|\overline{x} - x| > 3s$ where $\overline{x}$ is the sample mean, x is the value of the measurement and s is standard deviation of the sample. This allows us to exclude extreme values that may distort our analysis, thereby providing a more accurate representation of typical energy consumption patterns. After this filtering, 48 CTs remain for Spring Boot and 40 fo Spoon; (iii) We perform a Shapiro-Wilk test to evaluate the normality of the remaining data. This statistical test enables us to detect any non-normal data distributions that could impact the reliability of our subsequent analyses. We exclude call traces with non-normal distributions from the dataset as they might be a symptom of a problem during the energy consumption measurement. Following this step, 27 CT's remain for Spring Boot and 31 for Spoon.

For this short paper, we limit our manual analysis to the five CTs that exhibit the highest and lowest energy consumption for each project ($5 \times 2 \times 2 = 20$ CTs analyzed in total) to identify potential patterns in the source code. For each frame in the CT pointing to a specific line of code, we categorize the frame's method to determine its specific role (e.g., getters, setters, finders, constructors, delegators, etc.). This categorization primarily relies on methods' names. The list of method roles identified during our analysis is presented in Table 2. This analysis enables us to formulate hypotheses regarding whether particular coding practices or structures are linked to increased energy usage, thereby informing further exploration of energy-efficient programming techniques. Future work includes analyzing more call traces to confirm our emerging results. Due to space limitation, an example of CT analysis is present in the replication package.

Table 2: Method roles discovered during the manual analysis

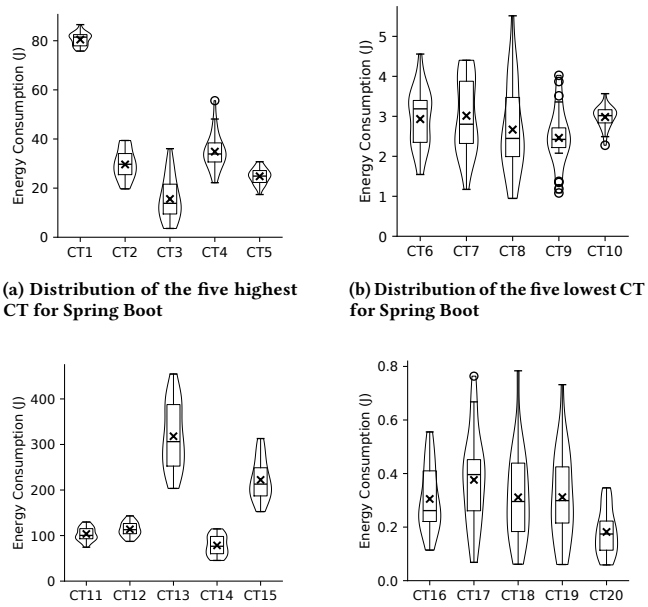
Method role	Definition
Builder (*bui.*)	Modifies the internal state of an object and typically returns the object itself for method chaining
Constructor (*con.*)	Initialize a new object
Delegator (*del.*)	Delegates something to another method
Factory (*fac.*)	Creates something, a new object or something else
Finder (*fin.*)	Searches something in a data structure
Formatter (*for.*)	Converts an object into a string representation to make it human-readable or structured in a specific way
Getter (*get.*)	Gets the parameters of an object
Lifecycle manager (*lif.*)	Orchestrates the various phases or states of a process, such as initialization, refresh or shutdown
Listener (*lis.*)	Responds to events or changes in states within a system
Serializer (*ser.*)	Handles the process of converting objects into a format that can be stored or transmitted, often using serialization and sometimes applying compression techniques
Setter (*set.*)	Sets the values of the parameters of an object
Test (*tes.*)	Each method with a test annotation, each method whose job is to make assertions
Utility (*uti.*)	Provides common operations or tasks, often related to system or file manipulation
Visitor (*vis.*)	Separates algorithms from the objects on which they operate

4 Evaluation results

Energy Consumption. Figure 1 illustrates the energy consumption distributions for each project's five most energy-intensive and the five least energy-intensive call traces. Call traces CT1 to CT5 are the highest energy consumers in the Spring Boot project, while CT6 to CT10 are the lowests. For the Spoon project, CT11 to CT15 are the highest energy consumers, and CT16 to CT20 are the lowest. Table 3 details each call trace's mean and standard deviation. Spoon's highest energy consumption ranges from 78.42 to 318.07 joules, which is significantly higher than Spring Boot's 15.58 to 80.47 joules. However, Spoon's minimum energy usage averages 0.18 to 0.38 joules, compared to Spring Boot's 2.46 to 3.02 joules, suggesting that, overall, Spoon is less energy-intensive in terms of unit test execution.

Code Patterns. During our manual analysis, we collected data presented in Table 3, which revealed a notable pattern. Specifically, we found that 7 out of the 10 most energy-intensive call traces, respectively CT1, CT2, CT4, CT5, CT11, and CT14, involved the use of constructors. This observation prompts an inquiry into whether creating Java objects through constructors significantly influences energy consumption. However, further examination indicated that 5 out of 10 least energy-intensive call traces, respectively CT6, CT7, CT8, CT9 and CT10, also utilized constructors.

We expanded our analysis to investigate *hidden non-static attributes* generated during constructor execution in each call trace. Hidden attributes are not directly defined in a class but are created during object initialization, often due to associations with other classes (inheritance, composition, reference, etc.). Using a debugger, we assessed variable states at the end of each call trace, placing breakpoints after the last execution line to examine the program state and constructor-created attributes. Table 3 lists the number of attributes associated with the constructors identified in each CT



(a) Distribution of the five highest CT for Spring Boot

(b) Distribution of the five lowest CT for Spring Boot

(c) Distribution of the five highest CT for Spoon

(d) Distribution of the five lowest CT for Spoon

Figure 1: Distribution of the energy consumption of the five highest and five lowest CT of the Spring Boot and Spoon

Table 3: Gathered data for each call trace of the Spoon and Spring Boot projects

CT	Mean	σ	# frames	Method roles	# attr.	# tot. attr.
Highest spring-boot CT						
CT1	80.47	2.65	3	2 con., 1 fac.	5	203
CT2	29.64	5.93	7	1 con., 3 fac., 1 fin., 1 get.	14	250
CT3	15.58	8.63	8	1 con., 1 del., 2 get., 2 lis.	9	26
CT4	34.85	6.97	4	1 con., 1 del., 1 fac., 2 get.	2	181
CT5	24.82	3.08	10	1 con., 5 del., 3 fac., 1 get.	1	153
Lowest spring-boot CT						
CT6	2.93	0.72	6	1 con., 4 del., 1 fin.	0	0
CT7	3.02	0.93	6	1 con., 4 del., 3 fac.	30	41
CT8	2.67	1.11	7	1 con., 1 del., 5 fac.	1	25
CT9	2.46	0.73	11	1 con., 4 del., 2 fac., 2 get., 1 lif., 1 other	0	0
CT10	2.98	0.26	3	1 con., 1 del., 1 fac.	0	0
Highest spoon CT						
CT11	103.54	14.33	2	1 con., 1 ser.	1	500+
CT12	113.77	14.93	65	2 fac., 50+ vis.	0	0
CT13	318.07	70.24	4	2 bui., 1 fin., 1 lif.	0	0
CT14	78.42	21.31	3	1 con., 2 fac.,	27	446
CT15	222.32	44.21	2	1 lis., 1 uti.	0	0
Lowest spoon CT						
CT16	0.31	0.12	4	3 for., 1 other	0	0
CT17	0.38	0.17	5	1 bui., 1 fac., 1 fin., 1 get., 1 set.	0	0
CT18	0.31	0.16	3	3 bui.	0	0
CT19	0.31	0.16	5	3 fin., 1 vis., 1 other	0	0
CT20	0.18	0.08	12	1 del., 2 fac., 5 fin., 1 get., 3 vis.	0	0

(# attr.) with the overall total of hidden attributes defined within those constructors (# tot. attr.).

Impact of source code on energy consumption. Answering our **RQ**, our findings suggest a link between energy-intensive call trace constructors and the generation of numerous hidden attributes. Specifically, 6 out of the 7 most energy-consuming constructors produced between 181 and over 500 attributes, while the least energy-intensive ones generated only 0 to 41 attributes. To support this observation, we conducted both Spearman's and Kendall's correlation tests on our dataset. Spearman's test demonstrated a moderate correlation that is not statistically significant ($\rho = 0.439$, $p - value = 0.052$). In contrast, Kendall's test showed a moderate correlation that is statistically significant ($\tau = 0.4308$, $p - value = 0.0138$). This suggests that energy costs may stem from the quantity and complexity of generated attributes rather than just the constructors themselves, highlighting the impact of object creation on energy consumption in Java applications.

5 Conclusion and Future Work

Our findings suggest significant hidden complexity in constructors' attributes among the highest CTs across both projects. Specifically, these constructors explicitly declare between 1 and 27 attributes, while the total number of hidden attributes ranges from 26 to over 500, as shown in Table 3. There appears to be a potential correlation between the energy consumption of the CTs and the difference between the total number of hidden attributes and the number of attributes declared in these constructors.

This complexity highlights the need for a tool to automate the identification and counting of attributes within classes, enabling a comprehensive assessment of attribute creation across all CTs. Automating this process will enhance our understanding of how attribute creation influences energy consumption while ensuring consistency and accuracy in attribute counting. Additionally, this tool could categorize method roles across CTs, facilitating a systematic analysis of energy usage beyond constructors and revealing the relationship between code structure and energy consumption.

A limitation of this study lies in the energy measurements from JoularJX. Although RAPL reliably measures energy consumption, its accuracy is lower than direct power meters [9]. However, the uniform bias across all call traces supports consistent comparative analysis within the dataset.

Future work should consider: (i) Categorizing method roles using objective and systematic criteria rather than relying solely on their names. (ii) Automating the process of identifying and counting attributes within classes, which is currently performed manually. (iii) Expanding the analysis beyond the *spring-boot-project/spring-boot* folder to include additional subprojects and other types of projects, allowing us to evaluate the applicability of our findings and provide a comprehensive view of energy consumption trends across the ecosystem. (iv) Integrating static analysis data to correlate energy consumption with relevant metrics, revealing patterns related to energy efficiency and insights into factors affecting energy consumption. (v) Analyzing multiple commits to track the impact of code changes on energy consumption, providing valuable information for sustainable software development practices.

Acknowledgments

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

References

[1] Hayri Acar, Gülfem I Alptekin, Jean-Patrick Gelas, and Parisa Ghodous. 2016. The Impact of Source Code in Software on Power Consumption. *IJEBM* 14 (2016), 42–52. http://ijebm-ojs.ie.nthu.edu.tw/IJEBM_OJS/index.php/IJEBM/article/view/693

[2] Andrea Arcuri and Lionel Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *STVR* 24, 3 (2014), 219–250. doi:10.1002/stvr.1486

[3] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting Energy Bugs and Hotspots in Mobile Apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Hong Kong China, 588–598. doi:10.1145/2635868.2635871

[4] Alexandre Bonvoisin, Clément Quinton, and Romain Rouvoy. 2024. Understanding the Performance-Energy Tradeoffs of Object-Relational Mapping Frameworks. In *SANER '24'*. IEEE, Rovaniemi, Finland, 626–636. doi:10.1109/SANER60148.2024.00069

[5] Stefano Corda, Bram Veenboer, and Emma Tolley. 2022. PMT: Power Measurement Toolkit. In *2022 IEEE/ACM International Workshop on HPC User Support Tools (HUST)*. IEEE, Dallas, TX, USA, 44–47. doi:10.1109/HUST56722.2022.00011

[6] Luís Cruz. 2021. Green Software Engineering Done Right: a Scientific Guide to Set Up Energy Efficiency Experiments. http://luiscruz.github.io/2021/10/10/scientific-guide.html. doi:10.6084/m9.figshare.22067846.v1 Blog post..

[7] Luis Cruz and Rui Abreu. 2021. On the Energy Footprint of Mobile Testing Frameworks. *IEEE TSE* 47, 10 (Oct. 2021), 2260–2271. doi:10.1109/tse.2019.2946163

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

[9] 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

[10] Guillaume Fieni, Daniel Romero Acero, Pierre Rust, and Romain Rouvoy. 2024. PowerAPI: A Python Framework for Building Software-Defined Power Meters. *JOSS* 9, 98 (June 2024), 6670. doi:10.21105/joss.06670

[11] 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 *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, Bangalore, India, 106–118. doi:10.1109/CCGrid57682.2023.00020

[12] Olivier Le Goaer and Julien Hertout. 2023. ecoCode: A SonarQube Plugin to Remove Energy Smells from Android Projects. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, 1–4. doi:10.1145/3551349.3559518

[13] Sung Une Lee, Niroshinie Fernando, Kevin Lee, and Jean-Guy Schneider. 2024. A Survey of Energy Concerns for Software Engineering. *JSS* 210 (April 2024), 111944. doi:10.1016/j.jss.2023.111944

[14] Kenan Liu, Gustavo Pinto, and Yu David Liu. 2015. Data-Oriented Characterization of Application-Level Energy Optimization. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, Alexander Egyed and Ina Schaefer (Eds.). Springer, Berlin, Heidelberg, 316–331. doi:10.1007/978-3-662-46675-9_21

[15] Jérôme Maquoi. 2025. *Replication package of Energy Codesumption, Leveraging Test Execution for Source Code Energy Consumption Analysis*. University of Namur. doi:10.5281/zenodo.15276280

[16] Adel Noureddine. 2022. PowerJoular and JoularJX: Multi-Platform Software Power Monitoring Tools. In *18th International Conference on Intelligent Environments*. IEEE, Biarritz, France, 1–4. doi:10.1109/IE54923.2022.9826760

[17] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joël Penhoat. 2020. On Reducing the Energy Consumption of Software: From Hurdles to Requirements. In *ESEM 2020 - ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3382494.3410678

[18] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. 2021. Evaluating The Energy Consumption of Java I/O APIs. In *ICSME 2021 - 37th International Conference on Software Maintenance and Evolution (Proceedings of the 37th International Conference on Software Maintenance and Evolution (ICSME))*. IEEE, Luxembourg / Virtual, Luxembourg, 1–11. doi:10.1109/ICSME52107.2021.00007

[19] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2015. Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *SPE* 46 (2015), 1155–1179. doi:10.1002/spe.2346

[20] B. Penzenstadler, V. Bauer, C. Calero, and X. Franch. 2012. Sustainability in Software Engineering: A Systematic Literature Review. In *16th International Conference on Evaluation & Assessment in Software Engineering (EASE 2012)*. Institution of Engineering and Technology, Ciudad Real, 32–41. doi:10.1049/ic.2012.0004

[21] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2020. SPELLing out Energy Leaks: Aiding Developers Locate Energy Inefficient Code. *JSS* 161 (March 2020), 110463. doi:10.1016/j.jss.2019.110463

[22] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2017. 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, 256–267. doi:10.1145/3136014.3136031

[23] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2021. Ranking programming languages by energy efficiency. *Sci. Comput. Program.* 205 (May 2021), 102609. doi:10.1016/j.scico.2021.102609

[24] 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. *Journal of Systems and Software* 222 (April 2025), 112303. doi:10.1016/j.jss.2024.112303

[25] Guillaume Raffin and Denis Trystram. 2025. Dissecting the Software-Based Measurement of CPU Energy Consumption: A Comparative Analysis. *IEEE Transactions on Parallel and Distributed Systems* 36 (2025), 96–107. doi:10.1109/TPDS.2024.3492336 arXiv:2401.15985 [cs]

[26] Andreas Schuler and Gabriele Kotsis. 2022. MANAi – An IntelliJ Plugin for Software Energy Consumption Profiling. arXiv:2205.03120 [cs]

[27] Thibault Simon, David Ekchajzer, Adrien Berthelot, Eric Fourboul, Samuel Rince, and Romain Rouvoy. 2025. BoaviztAPI: A Bottom-Up Model to Assess the Environmental Impacts of Cloud Services. *SIGENERGY Energy Inform. Rev.* 4, 5 (April 2025), 84–90. doi:10.1145/3727200.3727213