Coppers - A Rust Test Framework That Measures Energy Consumption Over Time

Katja Schmahl
Delft University of Technology
Delft, The Netherlands
K.G.Schmahl@student.tudelft.nl

Thijs Raymakers
Delft University of Technology
Delft, The Netherlands
T.Raijmakers@student.tudelft.nl

Jeffrey Bouman
Delft University of Technology
Delft, The Netherlands
J.Q.Bouman@student.tudelft.nl

***Abstract*—By 2030, IT is expected to consume about a third of the global energy demand [14]. However, case studies have shown that developers do not take energy efficiency into account when programming, mostly due to not having access to good measurement tools [6, 7, 8]. We introduce Coppers [12], a custom test harness for Rust that enables developers to measure the energy usage of their programs and visualize the results over time to see potential energy efficiency regressions. We show that it is easy to integrate with existing Rust projects and do a case study using Coppers with a popular library.**

***Index Terms*—sustainable software engineering, energy efficiency, Rust, software testing**

I. Introduction

IT was expected to consume more than 14% of all energy produced world wide in the year 2020 [9]. In 2030, it is estimated to already be a third of global demand [14]. This shows the importance of more energy efficient hardware, as well as software. In the last decade, green software design has become more of a core value in software development [7]. All software projects, small and large, can be run many times on many different machines worldwide, consuming huge amounts of energy. Every piece of code written can positively or negatively impact the footprint of an application.

To take away the hurdles towards more energy-aware software development, better tools are required. In this report we present Coppers, a possible solution for this problem when developing software using the Rust language on Linux machines. Coppers is a simple tool, designed for developers to see the energy usage of their project in a user-friendly way. It utilizes their previously written tests, and integrates with the existing test workflow of Rust. When adding Coppers to an existing Rust project, it will extend the regular testing process with an additional feature. The Coppers testing harness calculates the power consumed during the tests. It will compare these results to old executions of the tests done with the Coppers testing harness. It utilizes this data to show how the changes made to the code, just now or further in the past, have impacted the energy usage. These energy results will be displayed graphically to give programmers more insight in the energy impact of their code.

II. Problem

More than 80% of the developers in an online survey done by Pang et al. did not take energy consumption into account when developing. Most of them did however consider the energy consumption an important factor in decisions [8]. A different case study showed that many developers would feel more comfortable incorporating energy consumption into their project, if they had dedicated tools and would be able to set measurable objectives [7]. Similar results were found in an empirical study by Manotas et al., showing the importance of better intuition and targets, also during maintenance. The field of software engineering has great tools that can measure or estimate energy consumption. However, developers are generally not aware of the existence of these tools or don't have a way to easily integrate it into their own systems [7].

To be able to develop more energy efficient, software companies and developers need be more aware of their energy footprint and have more guidelines on how to reduce this. This will have direct benefits such as prolonged battery life and less hardware or cooling required, but also have long-term climate consequences by reducing carbon emissions.

III. Implementation

A. Using the Rust programming language

The Coppers framework is written in and for Rust [12]. One of the main goals of this project was to make it is as easy as possible to integrate the energy consumption measurement tools with existing systems, because then developers are more likely to experiment with these kinds of tools. This is one of the reasons why this project focuses on the Rust ecosystem. Almost all Rust projects can be built, tested and executed in the same manner. Extending this familiar workflow reduces the barrier of entry and allows them to test the energy consumption measurement tool on their own code base within minutes. Besides that, the Rust community is also quite serious about the run-time performance of their projects [2]. Solving the same problem in a shorter amount of time can have an positive impact on the energy consumption of a program. This makes the Rust community a good target demographic for our project.

B. Technical details

The project uses a project's existing test suite in order to calculate its power consumption. Normally, Rust uses a default testing harness. This is a program that is responsible for executing all the tests and verifying their results. This testing harness can be replaced by a custom one. Coppers

is such a custom test harness. It reuses the existing test cases that developers have written to measure the energy consumption of specific parts of the code, without requiring major modifications to an existing code base. The energy consumption is measured with the help of Intel's Running Average Power Limit (RAPL). RAPL is a tool that is used to measure the energy consumption, which was shown to closely match the measurements from the power plug [5]. In the Copper test harness, the values reported by RAPL are read right before and after each test. These values can be used to calculate the energy usage of the system during the execution of a test. Additionally, the values reported by RAPL are measured before and after *all* tests are executed. This is used to determine the overhead of the testing framework itself. All results are reported on a test-by-test basis, both in human readable and machine readable formats. The results contain information about which test was executed, the amount of energy that test used during execution and the timestamp. It also contains the version of the software that was tested, in the form of a *git hash*. The latter can be used to compare the results of different software versions with each other, to see whether the energy consumption has improved or degraded over time.

A human readable report is generated with the help of a Python script using the Plotly [10] library for interactive data visualization and using Jinja2 [13] for HTML templates. In order to minimize the amount of friction in the user experience, this script is run within Rust as well using the bindings for Rust provided by PyO3 [11].

C. Usage details

Our goal was to minimize the amount of changes a developer would need to do in order to run Coppers within an existing project. Developers now have to do three minor things. First, they need to add Coppers as a development dependency.

```
1 [dev-dependencies]
2 coppers = "0.1"
```

Second, they have to use the Rust Nightly tool chain.

```
1 rustup install nightly
2 rustup override set nightly
```

And finally, they have to add two lines of code to their *crate root*, the file that contains the entry point of their program.

```
1 #![feature(custom_test_frameworks)]
2 #![test_runner(coppers::runner)]
```

After these one-time changes, the energy consumption of all unit tests is automatically measured when running 'cargo test'. Part of the resulting terminal output of running can be seen in Listing 1, the output is cropped to show only the last 15 tests and the following output. Behind each test name, the indication of a successful or unsuccessful result is indicated as well as the power and time consumption of the test. After the tests are done the total amount of energy spent on all tests in combination with the overhead is printed to the screen. If the report feature is enabled, the location of the generated report is shown as well.

```
1 test seq::test::test_slice_choose ... ok - [773922
      μJ in 37954 μs]
2 test seq::test::value_stability_slice ... ok -
      [658690 μJ in 32818 μs]
3 test seq::test::test_iterator_choose ... ok -
      [4676873 μJ in 232748 μs]
4 test seq::test::test_iterator_choose_stable ... ok
      - [8381996 μJ in 429556 μs]
5 test seq::test::
      test_iterator_choose_stable_stability ... ok -
      [8632544 μJ in 432291 μs]
6 test seq::test::test_shuffle ... ok - [9076820 μJ
      in 451666 μs]
7 test seq::test::test_partial_shuffle ... ok -
      [666502 μJ in 31860 μs]
8 test seq::test::test_sample_iter ... ok - [639891
      μJ in 33517 μs]
9 test seq::test::test_weighted ... ok - [5805040 μJ
      in 292627 μs]
10 test seq::test::value_stability_choose ... ok -
      [656003 μJ in 33261 μs]
11 test seq::test::value_stability_choose_stable ...
      ok - [683349 μJ in 34847 μs]
12 test seq::test::value_stability_choose_multiple
      ... ok - [685425 μJ in 33109 μs]
13 test seq::test::test_multiple_weighted_edge_cases
      ... ok - [973204 μJ in 47325 μs]
14 test seq::test::
      test_multiple_weighted_distributions ... ok -
      [15714924 μJ in 778563 μs]
15 test test::test_random ... ok - [667845 μJ in
      32693 μs]
16 test result: ok.
17   75 passed;
18   0 failed;
19   0 ignored;
20   finished in 8669843 μs consuming 172476243 μJ
21   spend 6202331 μs and 123255450 μJ on tests
22   spend 2467512 μs and 49220793 μJ on overhead
23 > Generated report of energy consumption results
      in "target/coppers_report"
```

Listing 1. Output of cargo test to the terminal.

After all test cases have finished, a report is generated for the developer to gain easier insight into the results. This report is designed with two goals. The first goal is to establish which tests consume a lot of energy, and potentially more energy than the developer would expect from their functionality. This would then allow a developer to focus on these parts of the code when trying to optimize for energy efficiency. The second goal is to gain insight into the regression over time. Programmers can use the report to see which tests have an increase or decrease in their energy consumption, which can help them to understand the impact of their code changes.

To obtain these goals, the report contains four different visualizations. The figures used here as an example are generated from an execution of the Coppers harness on the Rust `rand` crate [3]. More explanation on these results can be found in the case study in Section IV. On the top of the report, it shows the user which three tests consumed the most amount energy, see Fig. 1, and which three tests consumed the least amount energy in the last execution. This helps in obtaining the first goal, to know which piece of code needs to be focused on when trying to optimize for energy consumption.

Following this is an interactive line graph of the energy consumption of the executions over time, see Fig. 4. All

Most energy consuming tests

1. seq::test::test_multiple_weighted_distributions 25254 μJ
2. seq::test::test_iterator_choose_stable_stability 13852 μJ
3. distributions::uniform::tests::test_integers 13144 μJ

Least energy consuming tests

1. seq::test::test_partial_shuffle 846 μJ
2. rngs::std::test::test_stdrng_construction 860 μJ
3. distributions::weighted_index::test::test_accepting_nan 868 μJ

Fig. 1. Most and least power consuming tests from report.

executions are ordered based on the commit. Within multiple executions of the same commit, the time of executing the tests is used. We chose this ordering to allow the user to look back on older versions of the code, and at the same time see changes between their last few measurements. This visualization can show which tests are flaky in usage, and which tests have a significant change in energy efficiency in comparison with previous versions. This is more suited for the low-level inspection of the code, and helps to obtain the second goal of the report.

The next visualisation is also focused towards the second goal, to gain an insight in the change over time. It is a large interactive table, see Fig. 2, allowing search and sorting. It shows all test cases that were executed both in the current and previous run. The changes in energy usage and execution time are shown. During development, this allows users to see which tests have been impacted the most by their most recent code changes, and whether this improves or degrades the energy efficiency.

Lastly, the report shows a bar plot of all the tests in Fig. 3, sorted by their energy consumption. It gives a more visual representation of the distribution of energy usage of the test cases in the test suite by vertically going over each test and horizontally showing the power consumption. In turn giving insight in how the tests are proportionally consuming more or less data. Thereby guiding the developer on where to put in effort to decrease the most power consumption.

IV. Case Study

A. Motivation

To showcase the usage of Coppers, we have done a case study on the most downloaded Rust crate: `rand` [3]. This project has more than 116 million downloads and over 7 thousand dependents. To analyze their energy consumption, we have run the tests of all patch releases of 0.8 (version 0.8.0 until version 0.8.5) of ‘rand’ with our custom test harness. This resulted in a report, which we analysed to highlight potentially beneficial information for the developers of the rand crate.

B. Results

In Fig. 4, the energy consumption of the individual tests over time can be found. The vertical axis shows the power consumption in micro Joules, on the horizontal axis are the distinct versions of rand shown. The different colors indicate particular tests. Each line represents the energy usage of each test over time. There is one test that stands out from the rest, which is the red line for the `distribution::uniform::tests::test_integers` test, which has an increase in energy consumption between versions 0.8.1 and 0.8.2+. The other tests show only small fluctuations in energy usage.

In Fig. 1, the top 3 of most and least energy consuming tests are displayed with their average energy consumption and in Fig. 3 there is a bar plot that shows the distribution over all the tests. These figures show that the tests `seq::test::test_multiple_weighted_distributions`, `seq::test::test_iterator_choose_stable_stability` and `distributions::uniform::tests::test_integers` are quite energy consuming in comparison to the rest of the code. It also shows that most of the tests are actually really energy efficient. The majority of the energy is used by only the 10 most consuming tests.

In Fig. 2, a comparison with the previous release is made for every test. By interacting with the table, it is possible to sort based on absolute or percentile change. It shows that all tests were within a [-1200, +1000] micro Joules range.

C. Conclusion

The `rand` crate adheres to semantic versioning rules, which only allows for internal backwards compatible bug fixes in patch releases. This explains why the energy consumption of most tests stays roughly constant. The increase in usage by the `distribution::uniform::tests::test_integers` test can be explained when the difference between version 0.8.1 and 0.8.2 is examined. This difference shows the addition of a for loop to the test in question [1]. While the underlying implementation has not changed, the amount of work that the test has to do has increased. This might not be a very suitable example of a part of the code that could be optimized. However, it shows that if significant changes to the production codes would be made, this would be easy to extract from the graph. Besides that, we can see that when no large changes were made, the energy efficiency results were rather consistent, making it more reliable.

Moreover, a programmer could know, based on this, that it is worthwhile to look into the functionality behind the top 10 tests in the table. These make up almost all of the test suite energy consumption, which could point to some less energy efficient production code. It is of course also possible that these are really large test cases, which might need a test refactor to split them up in smaller unit tests.

Furthermore, from the table it can be seen that over the entire test suite, 5404 micro Joules energy increase was measured. This shows that overall, the developers have neither

The total energy consumption changed with 5404 μJ. The change per test can be found in the table below.

Show 10 entries Search:

	Name	Usage (μJ) before	Usage (μJ) new	Change usage (μJ)	Change usage (%)	Time (μs) new	Time (μs) before	Change time (μs)	Change time (%)
0	distributions::bernoulli::test::test_trivial	972	1076	104	10.7	76	75	-1	-1.6
1	distributions::bernoulli::test::test_average	7839	7865	26	0.3	726	727	1	0.1
2	distributions::bernoulli::test::value_stability	929	1056	127	13.6	73	74	1	1.4
3	distributions::distribution::tests::test_distributions_iter	1066	962	-103	-9.7	73	72	-1	-1.4
4	distributions::distribution::tests::test_distributions_map	882	1074	192	21.8	72	72	0	0.2
5	distributions::distribution::tests::test_make_an_iter	984	977	-7	-0.7	71	72	1	1.4
6	distributions::distribution::tests::test_dist_string	1039	998	-41	-3.9	74	73	-2	-2.5
7	distributions::float::tests::f32_edge_cases	1072	928	-143	-13.4	72	73	1	1.4
8	distributions::float::tests::f64_edge_cases	851	1009	158	18.6	71	71	0	-0.3
9	distributions::float::tests::value_stability	1082	1115	33	3.0	72	74	1	1.8

Showing 1 to 10 of 72 entries Previous 1 2 3 4 5 … 8 Next

Fig. 2. Comparison table between last and previous test run.

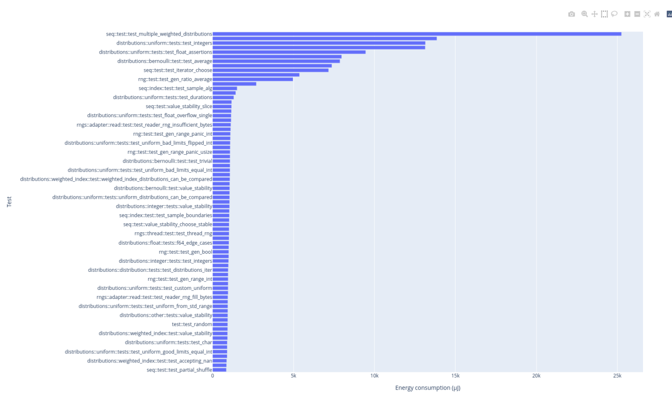

Fig. 3. Horizontal bar graph of all tests in last run.

improved nor decayed the energy efficiency of the project since the last release. Depending on their current target, this could be beneficial information to stakeholders.

V. Discussion

A. Expected Impact

Multiple case studies have shown that most developers do not currently take energy efficiency strongly into account when programming [7, 6, 8]. This is mostly due to not having the tools to gain the necessary insight in the current energy efficiency or intuition in what impacts the consumption of their software [7, 6]. Another factor that was often named for not incorporating it, was a lack of time or priority given to this aspect [7]. Being able to see how changes to a code base affect the energy efficiency of a program could be a great way to take energy consumption into consideration during development. Besides that, it generates metrics that can be used to set goals, which makes it easier to prioritize energy efficiency. The importance of this was also stressed by the study of Manotas et al.. Developers have expressed their interest and motivation to focus more on the energy usage aspect of development and we believe that this tool can help with their aspirations.

In the case study by Ournani et al., four sustainability guidelines were established for developers. We believe that we are able to partly fulfill three of these guidelines.

The first guideline is the availability of a global score to approximate the energetic footprint of the source code [7]. If the tests of a code base change too much, the total energy usage is not a reliable global score when it is compared to different iterations of itself. If stakeholders want a reliable global score, then this could be obtained by executing a fixed set of tests. This global score would however only be useful for comparison between different versions of the same project. Comparing different projects with each other is a much larger problem, which will require a new tool or metric still. Coppers does not provide such an inter-tool comparison metric.

The second guideline is that a tool should allow for low-level diagnosis, while also being user-friendly and have interactive graphics [7]. The report that can be generated by Coppers has been designed exactly for this purpose. By being able to see results on a test-by-test basis, it is easy to diagnose which test, and by extension, which part of the code is using most energy.

The third guideline is that it should integrate seamlessly with the tools already used [7]. This was accomplished by integrating it with `cargo`, which is the de facto way of running test suites for Rust. This makes it possible to use it with every existing Rust project that uses this standard testing harness. Besides, only a few lines of code need to be added to a project to use Coppers. This makes it easy and quick to deploy for a large number of projects. The design of Coppers meets a lot of the guidelines defined by Ournani et al.. Therefore, we believe that Coppers has the potential for widespread use, making it easier for many developers to measure energy consumption of their projects.

B. Limitations

Measuring the energy consumption of software comes with certain assumptions. These assumptions lead to some limitations of our tool.

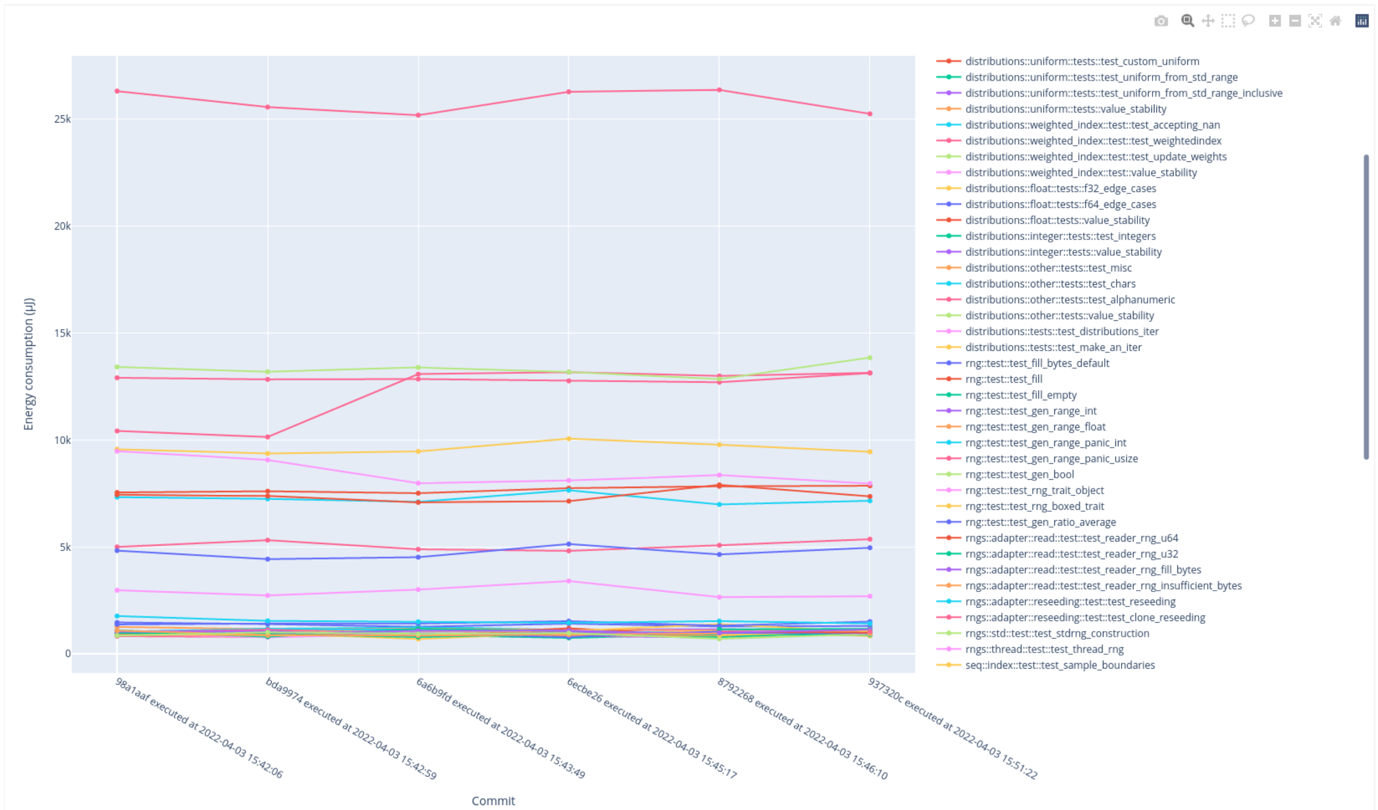

Fig. 4. The energy consumption per test of the rand library over the latest minor releases (0.8.0 to 0.8.5).

The first and most important limitation is the accuracy. The energy measurements are done on the system as a whole. This means that all other activity on the machine, as well as other outside elements, will influence the results. Tests may also be non-deterministic and vary over executions. We tried to limit this by measuring the consumption over multiple executions, but this cannot fully remove the influence of other components.

The second limitation we came across originates from the fact that there is no standard way to distinguish set-up, execution and tear-down stages of tests in Rust, as is often the case with test frameworks in other programming languages. This has the effect that a test with a very complicated set-up procedure might bias the energy consumption of the test, as it would mostly measure the consumption of the set-up procedure itself and not of the actual test. It could still be used for comparison over time, but changes in this stage of the test are of course less relevant for developers than energy efficiency changes in production code.

Another limitation of our current approach is the reliance on Intel RAPL, a technology that is only available on Intel platforms. This makes the current approach unsuitable for embedded systems that can greatly benefit from easy to use energy consumption measurement tools. To reduce this limitation, the implementation of Coppers is done in such a way that use case dependent extensions can be added without a complete redesign of the system. This allows embedded systems developers to extend Coppers with platform-dependent energy measurement tools, as long as the embedded system does not rely on the `#![no_std]` attribute.

Besides, the tool currently executes only on unit tests and does not work on documentation, system or integration tests. Coppers relies on an unstable feature of the Rust compiler that does not provide friction-less custom test harness support for non-unit test types [4]. Especially system and integration tests could prove very useful in approximating the energy impact more completely.

The last limitation, is the lack of reliable global score. This tool only allows comparing versions of the code that have the same tests. Comparing versions with different test suites or comparing to a different project requires a more global assessment. We believe it can still be used to set targets for a project, but as the test suite develops, these targets need to be continuously adapted.

C. Future Steps

The project as is does have a lot to offer to Rust developers, however the project does have room for improvement. In this section a few possible extensions will be discussed, ranging from simple additions to massive project expansion.

1) More Sensors: Rust is used for a range of different tasks. The current implementation is not suited for solutions that offload their computations to devices like GPUs, DSPs or FGPAs. Intel RAPL does not measure the energy consumption of other devices. Reading the energy consumption sensors of

graphic cards and other hardware inside a computers is a logical step to increase the usability of the Coppers. External sensors, like power supplies, could be supported in order to test more specific hardware configurations such as embedded systems or smart phones.

2) Including more tests: One of the current limitations as mentioned in section V-B, is the inability to use all types tests. Currently it is not easily possible to use Coppers with the integration and documentation tests of a project. In case the project is a library, integration tests are a more realistic scenario of regular usage that that library. If it is possible to utilize documentation or integration tests, then it might be possible to report more accurate information about the real world usage of a program.

3) Usage in Continuous Integration environment: Building the report for every version or even every pull request is a lot of tedious work when this is done manually. By automatically reporting the power consumption changes in the continuous integration pipeline, the maintainers of projects are able to get insight in whether a specific addition or edit to a projects code base changes the energy efficiency of their overall project. The integration with CI systems would allow maintainers to act on energy usage regressions before they land the changes in a release. It could also make the power consumption of online code repositories search-able, which could give users another way of comparing different projects. Using software that takes sustainability into account might be more in line with the goals and policies of companies that use the software. Integrating it with CI pipelines offers the possibility to create a ”CI status badge”, an image that projects often include in their `README`. Such an image can act as a form of promotion towards other developers and motivate them to include energy consumption measurement tools into their own CI pipeline as well.

4) Combine usage with code coverage: Testing is an integral part of writing hardened and reliable software. However, not all parts of the software are tested equally. Some parts of a project might not be tested at all, giving no information about energy consumption of the overall library. These parts of the code might be consuming the most energy, which will in turn skew the results for end users. It might also also be true in the other direction. A few lines of code can be tested over and over again, even when they are not used as much in real world scenarios. A combination of both the code coverage *and* the power consumption of specific tests might increase the accuracy of the results.

5) More extensive insight: Reporting is only useful if the programmer also understands the full implication of every line of code that is written. This is especially difficult when other libraries are used that are not part of the main project. Developers might not fully understand which functions are more suitable for their specific use case and which combination of functions can achieve similar results while reducing the overall energy usage of a project. Connecting energy measurement data from other projects could steer programmers to write more efficient code at the moment of creation. A possible implementation of this could take the form of rating function calls of libraries by their energy efficiency.

References

[1] Comparing 0.8.1...0.8.2 - rust-random/rand, 2022. URL https://github.com/rust-random/rand/compare/0.8.1...0.8.2.

[2] Rust survey 2021 results: Rust blog, Feb 2022. URL https://blog.rust-lang.org/2022/02/15/Rust-Survey-2021.html.

[3] crates.io. rand, 2022. https://crates.io/crates/rand/ [Accessed: 2022-03-31].

[4] Mazdak Farrokhzad. custom_test_frameworks, 2022. URL https://github.com/rust-lang/rfcs/blob/master/text/2318-custom-test-frameworks.md#integration-with-doctests.

[5] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018.

[6] 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 *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 237–248. IEEE, 2016.

[7] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. On reducing the energy consumption of software: From hurdles to requirements. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2020.

[8] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2015.

[9] Mario Pickavet, Willem Vereecken, Sofie Demeyer, Pieter Audenaert, Brecht Vermeulen, Chris Develder, Didier Colle, Bart Dhoedt, and Piet Demeester. Worldwide energy needs for ict: The rise of power-aware networking. In *2008 2nd international symposium on advanced networks and telecommunication systems*, pages 1–3. IEEE, 2008.

[10] Inc. Plotly, 2019. https://plotly.com/python/ [Accessed: 2022-03-31].

[11] PyO3. Pyo3 user guide, 2022. https://pyo3.rs/v0.16.2/ [Accessed: 2022-03-31].

[12] Thijs Raymakers, Jeffrey Bouman, and Katja Schmahl. Coppers: a custom test harnass for Rust that measures the energy usage of your test suite. URL https://github.com/ThijsRay/coppers.

[13] Armin Ronacher, 2022. https://palletsprojects.com/p/jinja/ [Accessed: 2022-03-31].

[14] Roberto Verdecchia, Patricia Lago, Christof Ebert, and Carol De Vries. Green it and green software. *IEEE Software*, 38(6):7–15, 2021.