Evaluating Power Consumption of Python Test Generation using Pynguin

Victor Horneț, Elena Mihalache, Andreea Mocanu, Alexandru Postu, Kian Sie

March 2025

1 Introduction

According to the United Nations Agenda of Sustainable Development [1], a prosperous future requires energy efficiency, sustainable consumption, and, implicitly, the reduction of pollution.

Among the factors that exacerbate pollution is the process of software development [2]. Given the increased use of software in society, we intuit that software development practices risk using increasingly more energy. This fact sparked the appearance of a Green Software movement, calling for software development practices that minimize carbon emissions [3, 4]. Therefore, software engineers must become more mindful of the energy consumed by development pipelines. These pipelines include a large variety of processes, one of them being testing.

Writing test cases is a vital part of the development process, as it ensures software reliability and quality by minimizing program bugs. Nonetheless, software engineers find writing tests tedious, particularly for simple test assertions [5, 6]. Consequently, automated test generation tools have been created to alleviate this issue [7]. By extension, minimizing the energy requirements of our development processes implies that we must minimize the power usage of test generation tools as well.

Whilst there is significant research related to Java test generators [7], increasingly more codebases use Python – in fact, it has become the most popular programming language on GitHub as of 2024 [8]. Despite this, there are significantly fewer tools that automate the test case generation process in Python, particularly due to issues arising from type inference [7]. A test generation tool that aims to fill this gap is Pynguin [7].

However, we currently identify no research investigating Pynguin's power usage. Moreover, Python itself is less energy efficient due to being an interpreted language [9], a fact which leads us to assume that Pynguin might also be less mindful of energy resources than its Java counterparts. However, Python's popularity makes it all the more possible that test generation tools such as Pynguin will gain more commercial use, thus motivating us to investigate how well it currently fares when it comes to power usage.

In this report, we investigate Pynguin's power usage, providing usage guidelines to users who aim to leverage this tool in a more energy-conscious way. To do so, we run three of the tool's test generation algorithms, MIO[10], DynaMOSA[11], and MOSA[12], on two different seeds, namely 42 and 1337. We do this in multiple iterations, providing power and test generation time averages for each algorithm-seed combination.

To achieve our goal, we structure the report as follows. First, we contextualize the tools that we use and evaluate in section 2. In section 3, we highlight other existing work in this domain, further explaining the added value that our work presents. Following this, we detail our experimental setup in section 4. Having appropriately set the foundation, we then showcase and interpret our results in 5. Next, we discuss the implications of said results in section 6. We acknowledge the limitations that we faced due to

time constraints under section 8. With this being said, section 9 points to future research that can be done on this topic. Finally, we conclude with some guidelines in section 10.

2 Background

2.1 Pynguin

The focus of this paper, Pynguin [7], is an automated unit test generation tool designed to address the lack of such tools for dynamically typed languages, as there have been several tools such as Randoop [13] and EvoSuite [14], however, their focus lies solely with Java, a statically typed language. Pynguin, however, focuses on Python whilst employing a similar search-based approach for test generation as EvoSuite.

2.2 EnergiBridge & PyEnergiBridge

EnergiBridge [15] is a tool designed to collect resource usage data, allowing us to easily monitor system power consumption for the duration of Pynguin's test generation. However, EnergiBridge does not natively integrate with Python programs. PyEnergiBridge [16] provides a Python wrapper for EnergiBridge, and allows us to address this issue. As the goal of our research is to evaluate the energy consumption of Python test generation configurations, this enables us to utilize EnergyBridge with improved ease of integration.

2.3 Docker

Docker[1] is a virtualization tool that allows us to run our tests in a container, ensuring that the tests are generated in an isolated environment. This prevents any unwanted access to the broader filesystem that could affect generation or power metrics. These unwanted accesses could result in invalid energy metrics. Utilizing Docker ensures reproducibility and minimizes the aforementioned issues that could arise.

[1] https://www.docker.com/

3 Relevant Literature

Much research has been done on the functional aspects of automated test generation (e.g., coverage and performance), such as the original Pynguin [17] and EvoSuite [14] papers. However, there has been a significant lack of research regarding the power consumption of these automated test generation tools.

The only paper that we were able to find, at the time of writing, was "On the energy consumption of test generation" by Kifetew et al. [18], which evaluates the energy consumption of Evosuite by considering factors such as Cyclomatic Complexity (CC), test coverage, and the algorithm used in EvoSuite's search-based approach. They determined that complex programs with $<100\%$ coverage consumed the most power, while programs with 100% coverage typically consumed considerably less power, regardless of the complexity. Interestingly, they also determined that in simple programs with low CC, the search algorithm used (e.g., Random, DynaMOSA, etc.) had relatively little impact on the overall power consumption.

Their findings highlight the need for further research into power consumption for test generation tools, due to a strong focus on a single testing tool for a single language. Our research aims to build on this gap by investigating the power consumption of a Pynguin for the current most popular programming language, Python. As Python is a dynamically typed language, this adds insight to our findings.

4 Experimental Setup

To evaluate the power usage of automated test generation, we used Pynguin across three different configurations. These differ by the tool's generation algorithm, namely DynaMOSA, MOSA, or MIO. We chose to change these configurations as we want to see if different generation methods consume more power.

The projects we chose for our experiments are taken from [17], which were selected from

the PyPI package index of Python projects, using the 'typed' category such that projects use type annotations. These were also selected such that Pynguin can run on them, so for example, projects with native-code library dependencies were not selected. Out of all 10 different projects, we chose 2 of them, namely `codetiming` and `docstring_parser`. As these were the fastest running programs for us, we selected them due to time constraints.

While the dataset is not representative enough for all types of Python software due to our limitations, the 2 projects have different use cases and sizes. `Codetiming` is a customizable timer for Python code, with 2 modules and 145 lines of code, which is quite a small project. However, `Docstring_parser`, which is used to parse Python code documentation, has 6 modules with 924 lines of code in total.

For each project and configuration, we executed Pynguin to generate unit tests and recorded the power consumption using EnergiBridge. This tool was used to measure the consumption across the entire test generation process, for each configuration on all the files. We chose to focus on average power consumption (Watts) rather than total energy consumption (Joules) as we want to consider how much is consumed per unit of time. For example, a configuration might use less total energy because it finishes quickly, even if it uses a lot of power while running. By looking at average power instead, we can compare how demanding each configuration is over time. This helps us compare the efficiency of different configurations in a more balanced way.

For validity, each experiment was repeated 14 times, and the average was taken as the final result. Between executions, we used `time.sleep(10)` to prevent previous measurements from affecting the current one. To account for external conditions that change over time, such as room temperature, we shuffle the order of configurations, the order of projects per configuration, and the order of modules per project. With this, we aim to reduce the risk of biased results. We also ran the evaluation on a single device, with the following specifications: Apple M1 Pro chip, 10-core CPU with 8 performance cores and 2 efficiency cores, 16-core GPU, 16-core Neural Engine, 200GB/s memory bandwidth. To reduce confounding variables, all experiments were executed under the following zen conditions:

- all other applications closed
- adaptive screen brightness disabled
- screen saver disabled
- consistent screen brightness set
- Wi-Fi interface turned off
- Bluetooth and Airdrop disabled
- display sleep disabled.

5 Evaluation

Our results showcase slight differences between algorithms in terms of power consumption. To account for the different values per iteration, figure 1 illustrates the power consumption of algorithms as box plots.

Herein, we identify that MIO requires the most median power, and that this value is only marginally affected by the random seed choice. However, we note that MIO-42 has a bigger upper quartile, which suggests that this seed value might lead to more variance in terms of power consumption. In addition, both MIO-42 and -1337 have outliers on the higher end of values, a fact which, given the limited number of 14 iterations, may suggest that the actual median power consumption could increase after more iterations.

MOSA and DynaMOSA perform similarly in terms of required power, with the mention that DynaMOSA has slightly lower median power consumption. Moreover, we notice the absence of outliers, suggesting that the statistics might be more reliable for these algorithms than for MIO. Here, the seed choices do not appear to meaningfully impact overall power consumption, neither in terms of variance nor median values.

The ANOVA results show that the choice of algorithm and seed produces statistically significant differences ($p < 0.001$). Specifically, the

ANOVA on power (Figure 5) shows $F = 84.581$ and a very small p-value, indicating that at least one algorithm-seed combination differs meaningfully from the others in power used. Post-hoc tests reveal consistent groupings in which MIO-1337 and MIO-42 generally form a distinct cluster with higher average power but lower execution times (thus lower total energy), while DynaMOSA and MOSA variants cluster separately, drawing less power at any moment but requiring more time overall. These findings statistically validate that the observed differences in energy usage and performance across algorithms are unlikely to be due to chance.

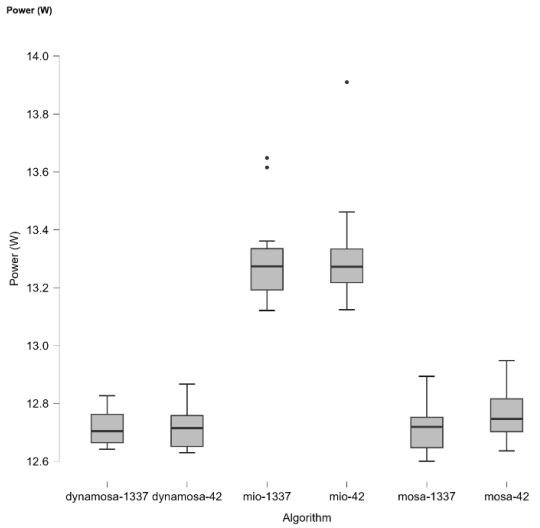

Figure 1: Power per algorithm

In addition to power-related statistics, one should also consider the test generation duration (Figure 2). Despite having the highest median power, MIO had the lowest execution time, requiring around 713–716 seconds. In terms of actually minimizing the total energy consumption, this may mean that MIO might be a superior choice to MOSA or DynaMOSA.

On the topic of time, we recall the experimental setup that Pynguin's authors use [7]. They limited the test generation time to 600 seconds, which is above the time required for any of the test generations to finish. Thus, considering a time execution cap of 600 seconds, MOSA and DynaMOSA might be better choices, as they require less power and, implicitly, less energy over the same number of seconds.

Furthermore, it is important to consider the coverage performance of each algorithm. To do so, we point to the original experiments which are run across multiple projects for 600 seconds each, including those used for our experiment [7]. DynaMOSA performed best in terms of branch coverage, followed by MIO and MOSA. However, the differences themselves are negligible, with the highest average branch coverage being 68.0%, and the lowest, 67.0% [17]. Thus, there is no clear best algorithm in terms of branch coverage.

We sum up the results section with three main points. First, there are no significant differences in branch coverage performance across algorithms. Secondly, MIO maximizes power consumption. This contrasts with the third main observation, which is that MIO may take less time to finish execution (around 2 minutes, in our case). However, this may not be relevant if a developer chose to cap the test generation time.

This means that, sustainability-wise, MIO may be the best option for larger projects and no test generation time cap, as the algorithm has the quickest execution time and implicitly consumes the least energy. Assuming a time cap, however, DynaMOSA or MOSA might be better alternatives.

However, we elaborate in the discussion section (see 6) that picking MIO may come at the cost of decreased test quality, particularly because we identified unexpected behaviour when it came to its test writing capabilities.

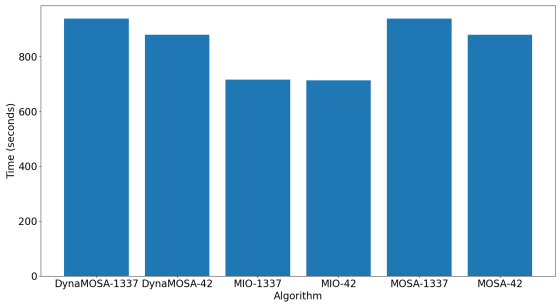

Figure 2: Test generation time per algorithm

6 Discussion

In this section, we reflect on the results presented above and interpret their implications. We primarily focus on the power consumption results (Figures 5, 8, 11, and 15), though we also try to contextualize them using total energy consumption (Figures 4, 7, 10, 14) and execution time (Figures 6, 9, 12, 16).

A main observation is that MIO shows the highest average power draw of about 13.3W (Figures 8 and 11) but also the shortest execution times of around 713–716s (Figures 9 and 12). In contrast, DynaMOSA and MOSA consume less power on average of about 12.7–12.8W but run substantially longer for about 879–940s. MIO's mutation-based search likely causes more intense CPU utilization in shorter bursts, increasing power draw. In other words, it is working harder at any given moment, thus driving up the wattage. It can be that MIO converges faster than DynaMOSA and MOSA on our relatively small or less complex modules we tested. Additionally, in some modules, MIO attempted mutation testing but all mutants survived, and no tests were written. The potential convergence and no killed mutants can, as a result, shorten overall runtime.

All three algorithms were run using two different seeds (42 and 1337). As seen in Figures 8, 7, and the ANOVA tables, seed changes do cause slight shifts in the distributions, but the overall ranking of the algorithms in terms of power and energy does not fundamentally change. MIO's mutation-driven search may be more deterministic in how it explores mutations, so changing the seed does not drastically alter its short search phases. DynaMOSA and MOSA are both coverage-driven algorithms that rely more heavily on random selection of test cases during their search, so different seeds can more noticeably shift the coverage search path. However, these shifts did not overshadow the observed pattern that MIO completes faster with higher power, whereas DynaMOSA and MOSA run longer at lower power.

Certain files triggered Pynguin messages that there was no testable code. This typically arises when a file only contains import statements, constants, or empty class definitions or when the file's methods do not meet Pynguin's criteria (e.g., no public functions or typed signatures for Pynguin to analyze). As a result, Pynguin (and thus MIO, DynaMOSA, or MOSA) simply cannot generate any tests. This lowers the total number of testable units but does not necessarily reflect a shortcoming of the algorithms, it is rather a quirk of the code's structure or the tool's heuristics for identifying testable functionality.

Therefore, we introduce the following suggestions for developers who aim to use Pynguin in a sustainable way. Firstly, they should cap the test generation time similarly to the initial study, which is 600 seconds. Assuming a reasonable time cap, developers should then use either MOSA or DynaMOSA, as they have a lower power draw than MIO. However, if test generation time cannot be reduced, we suggest picking MIO instead, as it might take less time to generate tests.

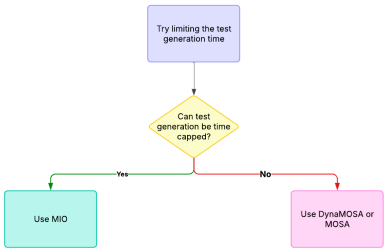

Figure 3: Guidelines for developers

7 Limitations

Our study on the power consumption of test generation is subject to several limitations that may affect the accuracy and generalizability of the results. The first limitation was hardware constraints, as the experiments were conducted on a personal computer. Additionally, we faced technical difficulties in the form of errors such as Pynguin saying there was nothing to test, or modules not being recognized, which further delayed data collection.

The execution of the experiment took significantly longer than anticipated due to the complexity of the test generation process and the need for repeated measurements. This limited the number of configurations and projects we could explore to only two, affecting the comprehensiveness of our findings.

Another effect of the long test generation duration was our limitation of 14 iterations per configuration. Although more iterations would have been preferred, and would potentially have increased accuracy, we unfortunately were unable to do so.

The final effect of the time constraints was our lack of consideration for branch coverage and type annotations. Although this would allow us to understand the trade-offs between the power consumption and performance for the various algorithms, we did not manage to include them in our findings.

Our study relied on software-based energy measurement tools, which can be inaccurate. These tools are sensitive to noise from concurrent processes, making the results susceptible to measurement errors.

To ensure isolation during test execution, we ran the experiments inside Docker containers. While this approach helped minimize interference from other processes, it also introduced additional CPU and memory overhead. The energy consumed by the container management itself could have affected the overall readings, making it difficult to isolate the true energy costs of test generation alone.

8 Threats to validity

8.1 Internal threats

As mentioned previously, due to time and technical constraints we were limited to only running *docstring_parser* and *codetiming_local*, from the 8 total projects. This creates an internal threat to the validity of our findings, as we were unable to account for the broad variety of use cases which Pynguin discussed in their original paper. As the projects were chosen due to their speed, and not based on specific insights they provide, our results may not be an adequate reflection of the real-world power consumption of Pynguin.

To account for tail bias, we utilize `time.sleep(10)`. We aimed to ensure that there was enough time between configuration runs and avoid picking up any tail energy measurements. As we simply did this on a time basis, however, we did not verify if the processes fully concluded in this time frame. As a result, we were unable to ensure that our duration was long enough to avoid tail bias.

Another crucial aspect that we were unable to account for was the charging state of the device while the tests were running. As the tests were running overnight (as a result of the long generation durations), we were unable to monitor the battery status throughout the experiment. Fluctuations in the charging state could have had unknown effects on our readings, which further threatens the validity of our results.

8.2 External threats

A key external threat is the lack of results on Windows, which may limit the completeness of our results. Pynguin is not limited to a specific operating system, which likely means the size of the Windows user base is not insignificant. Our lack of Windows results limits the insight into the broader range of power consumption of Pynguin. As Pynguin is not limited to a specific operating system, understanding the power consumption across different operating systems, as well as the relative power consumption between platforms, would improve our understanding.

9 Future Work

Expanding on the previously mentioned threats to the project validity, verifying whether results remain consistent across operating systems, and running likewise tests on Windows is another aspect that could benefit from further research and provide more context. Furthermore, controlling conditions to a greater extent would likely also ensure higher consistency, by reducing the number of outside variables that could affect the results. Similarly, a future study that monitors background processes to more accurately track and prevent tail bias affecting the results would contribute to a much more accurate set of results, and thus to a clearer picture of the problem at hand.

As there were several aspects we were unable to cover in our experiment, the first area of future work could be to evaluate the power consumption of the other projects discussed in the original Pynguin paper [17], such as *api_mid*, *flutes_local*, *flutils_local*, and *pypara_local*. As mentioned, due to time constraints, we only ran two projects, which is not indicative of the complete picture.

Furthermore, there were additional algorithms that could similarly be evaluated as well. Alongside DynaMOSA, MIO, and MOSA, Pynguin also offers RANDOM, RANDOM_TEST_CASE_SEARCH, RANDOM_TEST_SUITE_SEARCH, and WHOLE_SUITE as algorithm options. Assessing the remaining algorithm options would provide additional context to the problem.

The paper regarding EvoSuite's power consumption [18] took Cyclomatic Complexity(CC) into account as well. Their findings concluded a strong correlation between the CC of a program and its power consumption. Expanding on our research to investigate if Python programs are affected similarly would provide much merit.

Finally a future study considering branch coverage and the evaluation of the generated tests themselves would provide merit to anyone intending to follow the proposed guidelines, and allow them to be more informed of the benefits and tradeoffs of the various algorithms.

10 Conclusion

This study investigated the power consumption of Pynguin's test generation algorithms, focusing on DynaMOSA, MOSA, and MIO. Our experiments revealed that MIO consumed the least total energy, despite its higher instantaneous power draw, due to its shorter execution time. Our findings reveal an important trade-off between instantaneous power consumption and execution time, underscoring that energy efficiency in test generation cannot be assessed through power metrics alone but must account for the interplay between power, time, and algorithmic efficiency. Our results emphasize the importance of considering power consumption as a metric in automated test generation. While functional metrics like coverage remain important, energy efficiency emerges as a complementary criterion that could influence tool design and configuration choices in the future. However, limitations such as small sample sizes and hardware variability warrant caution in generalizing these findings. Future work should expand the evaluation to more projects, additional algorithms, and cross-platform environments, while integrating coverage and mutation score analyses. Ultimately, as the demand for sustainable software practices grows, understanding the energy footprint of tools like Pynguin becomes essential.

11 Reproducibility

The reproduction package is available at `https://github.com/victorhornet/sse-testgen`, along with the setup instructions. We recommend using `uv` for the environment setup and running the experiment.

References

[1] U. Nations, "THE 17 GOALS | Sustainable Development — sdgs.un.org," https://sdgs.un.org/goals, [Accessed 03-04-2025].

[2] J. Manner, "Black software — the energy unsustainability of software systems in the 21st century," *Oxford Open Energy*, vol. 2,

p. oiac011, 12 2022. [Online]. Available: https://doi.org/10.1093/ooenergy/oiac011

[3] L. Ardito, G. Procaccianti, M. Torchiano, and A. Vetro, "Understanding green software development: A conceptual framework," *IT professional*, vol. 17, no. 1, pp. 44–50, 2015.

[4] E. Kern, M. Dick, S. Naumann, A. Guldner, and T. Johann, "Green software and green software engineering–definitions, measurements, and quality aspects," in *First International Conference on Information and Communication Technologies for Sustainability (ICT4S2013), 2013b ETH Zurich*, 2013, pp. 87–91.

[5] I. Ciupa, "Test studio: An environment for automatic test generation based on design by contracttm," *Master's thesis, Chair of Software Engineering, Eidgenössische Technische Hochschule Zürich, Switzerland*, 2004.

[6] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Software developers' perceptions of productivity," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 19–29. [Online]. Available: https://doi.org/10.1145/2635868.2635892

[7] S. Lukasczyk and G. Fraser, "Pynguin: automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. ACM, May 2022. [Online]. Available: http://dx.doi.org/10.1145/3510454.3516829

[8] GitHub, "Octoverse: AI leads Python to top language as the number of global developers surges — github.blog," https://github.blog/news-insights/octoverse/octoverse-2024/, [Accessed 20-03-2025].

[9] R.-H. Pfeiffer, "On the energy consumption of cpython," in *Quality of Information and Communications Technology*, A. Bertolino, J. Pascoal Faria, P. Lago, and L. Semini, Eds. Cham: Springer Nature Switzerland, 2024, pp. 194–209.

[10] A. Arcuri, *Many Independent Objective (MIO) Algorithm for Test Suite Generation.* Springer International Publishing, 2017, p. 3–17. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-66299-2_1

[11] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018.

[12] K. Amine, "Multiobjective simulated annealing: Principles and algorithm variants," *Advances in Operations Research*, vol. 2019, p. 8134674, 2019. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1155/2019/8134674

[13] C. Pacheco and M. Ernst, "Randoop: Feedback-directed random testing for java," in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, 10 2007, pp. 815–816.

[14] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 416–419. [Online]. Available: https://doi.org/10.1145/2025113.2025179

[15] J. Sallou, L. Cruz, and T. Durieux, "Energibridge: Empowering software sustainability through cross-platform energy measurement," 2023. [Online]. Available: https://arxiv.org/abs/2312.13897

[16] L. Cruz, "pyenergibridge," 2023, accessed: 2025-03-25. [Online]. Available: https://github.com/luiscruz/pyEnergiBridge

[17] S. Lukasczyk and G. Fraser, "Pynguin: automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '22. ACM, May 2022. [Online]. Available: http://dx.doi.org/10.1145/3510454.3516829

[18] F. Kifetew, D. Prandi, and A. Susi, "On the energy consumption of test generation," 2025. [Online]. Available: https://arxiv.org/abs/2501.09657

A Appendix: Additional Figures

ANOVA - Energy Consumption (J)

Cases	Sum of Squares	df	Mean Square	F	p
Algorithm	$8.648\times10^{+7}$	5	$1.730\times10^{+7}$	1798.326	< .001
Residuals	750195.658	78	9617.893		

Note. Type III Sum of Squares

Post Hoc Tests

Standard (HSD)

Post Hoc Comparisons - Algorithm

		Mean Difference	SE	df	t	p_{tukey}
(dynamosa-1337)	(dynamosa-42)	750.694	37.067	78	20.252	< .001***
	(mio-1337)	2415.055	37.067	78	65.153	< .001***
	(mio-42)	2435.254	37.067	78	65.698	< .001***
	(mosa-1337)	9.495	37.067	78	0.256	1.000
	(mosa-42)	703.336	37.067	78	18.975	< .001***
(dynamosa-42)	(mio-1337)	1664.361	37.067	78	44.901	< .001***
	(mio-42)	1684.560	37.067	78	45.446	< .001***
	(mosa-1337)	−741.199	37.067	78	−19.996	< .001***
	(mosa-42)	−47.358	37.067	78	−1.278	0.796
(mio-1337)	(mio-42)	20.199	37.067	78	0.545	0.994
	(mosa-1337)	−2405.560	37.067	78	−64.897	< .001***
	(mosa-42)	−1711.719	37.067	78	−46.179	< .001***
(mio-42)	(mosa-1337)	−2425.759	37.067	78	−65.442	< .001***
	(mosa-42)	−1731.918	37.067	78	−46.724	< .001***
(mosa-1337)	(mosa-42)	693.841	37.067	78	18.718	< .001***

*** p < .001

Note. P-value adjusted for comparing a family of 6 estimates.

Figure 4: ANOVA Joules

ANOVA - Power (W)

Cases	Sum of Squares	df	Mean Square	F	p
Algorithm	6.327	5	1.265	84.581	< .001
Residuals	1.167	78	0.015		

Note. Type III Sum of Squares

Post Hoc Tests

Standard (HSD)

Post Hoc Comparisons - Algorithm

		Mean Difference	SE	df	t	p_{tukey}
(dynamosa-1337)	(dynamosa-42)	−0.004	0.046	78	−0.095	1.000
	(mio-1337)	−0.585	0.046	78	−12.654	< .001***
	(mio-42)	−0.599	0.046	78	−12.955	< .001***
	(mosa-1337)	0.011	0.046	78	0.229	1.000
	(mosa-42)	−0.051	0.046	78	−1.112	0.875
(dynamosa-42)	(mio-1337)	−0.581	0.046	78	−12.559	< .001***
	(mio-42)	−0.595	0.046	78	−12.861	< .001***
	(mosa-1337)	0.015	0.046	78	0.324	1.000
	(mosa-42)	−0.047	0.046	78	−1.018	0.911
(mio-1337)	(mio-42)	−0.014	0.046	78	−0.301	1.000
	(mosa-1337)	0.596	0.046	78	12.883	< .001***
	(mosa-42)	0.534	0.046	78	11.542	< .001***
(mio-42)	(mosa-1337)	0.610	0.046	78	13.185	< .001***
	(mosa-42)	0.547	0.046	78	11.843	< .001***
(mosa-1337)	(mosa-42)	−0.062	0.046	78	−1.342	0.761

*** p < .001

Note. P-value adjusted for comparing a family of 6 estimates.

Figure 5: ANOVA Power

ANOVA - Execution Time (s)

Cases	Sum of Squares	df	Mean Square	F	p
Algorithm	754626.841	5	150925.368	101945.825	< .001
Residuals	115.475	78	1.480		

Note. Type III Sum of Squares

Post Hoc Tests

Standard (HSD)

Post Hoc Comparisons - Algorithm

		Mean Difference	SE	df	t	P_{tukey}
(dynamosa-1337)	(dynamosa-42)	59.333	0.460	78	129.018	< .001***
	(mio-1337)	222.836	0.460	78	484.550	< .001***
	(mio-42)	225.106	0.460	78	489.486	< .001***
	(mosa-1337)	−0.036	0.460	78	−0.078	1.000
	(mosa-42)	58.865	0.460	78	128.000	< .001***
(dynamosa-42)	(mio-1337)	163.503	0.460	78	355.532	< .001***
	(mio-42)	165.773	0.460	78	360.468	< .001***
	(mosa-1337)	−59.369	0.460	78	−129.096	< .001***
	(mosa-42)	−0.468	0.460	78	−1.018	0.911
(mio-1337)	(mio-42)	2.270	0.460	78	4.936	< .001***
	(mosa-1337)	−222.872	0.460	78	−484.628	< .001***
	(mosa-42)	−163.971	0.460	78	−356.550	< .001***
(mio-42)	(mosa-1337)	−225.142	0.460	78	−489.564	< .001***
	(mosa-42)	−166.241	0.460	78	−361.486	< .001***
(mosa-1337)	(mosa-42)	58.901	0.460	78	128.078	< .001***

*** p < .001

Note. P-value adjusted for comparing a family of 6 estimates.

Figure 6: ANOVA Time

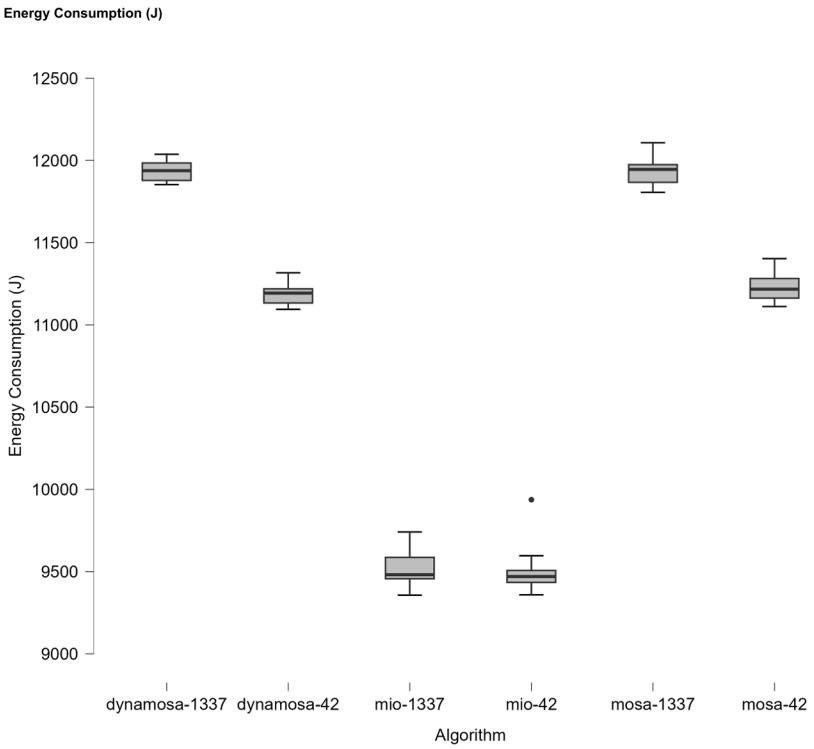

Figure 7: Boxplot Joules

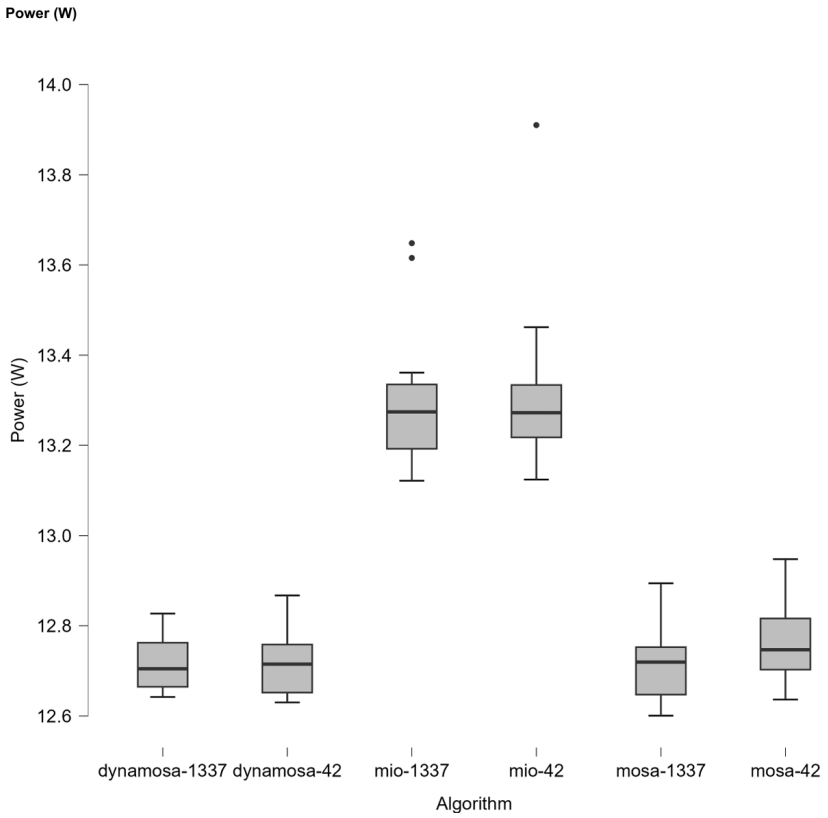

Figure 8: Boxplot Power

z5z5

Figure 9: Boxplot Time

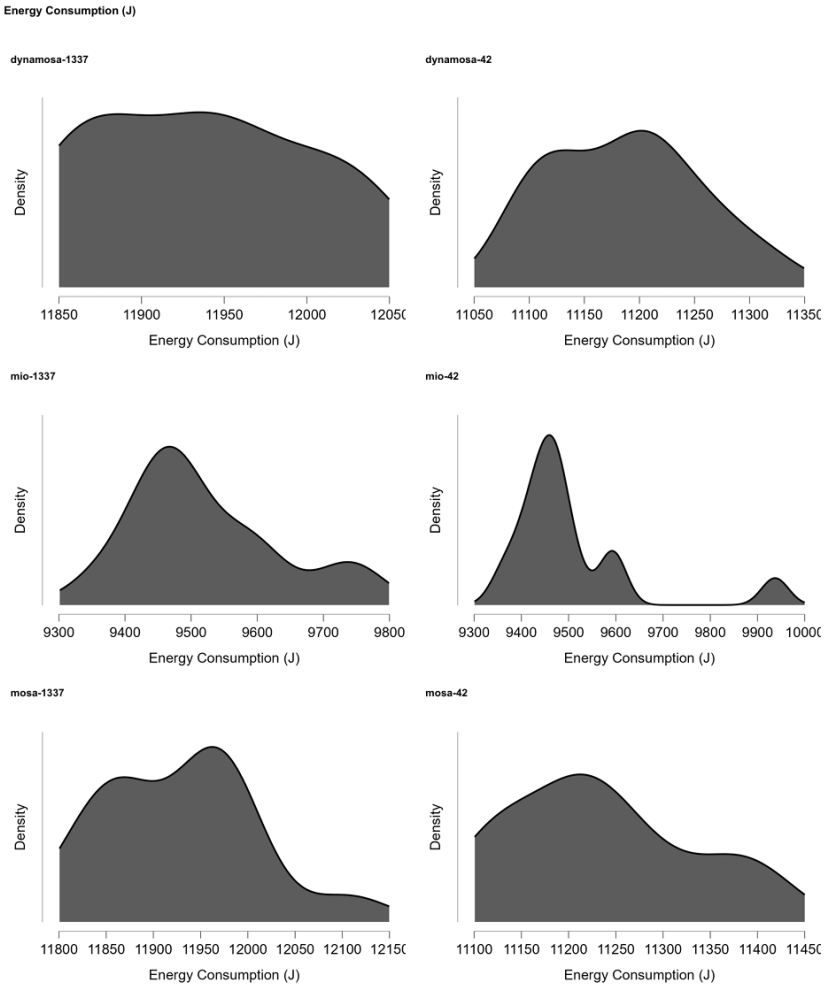

Figure 10: Density Plot - Joules

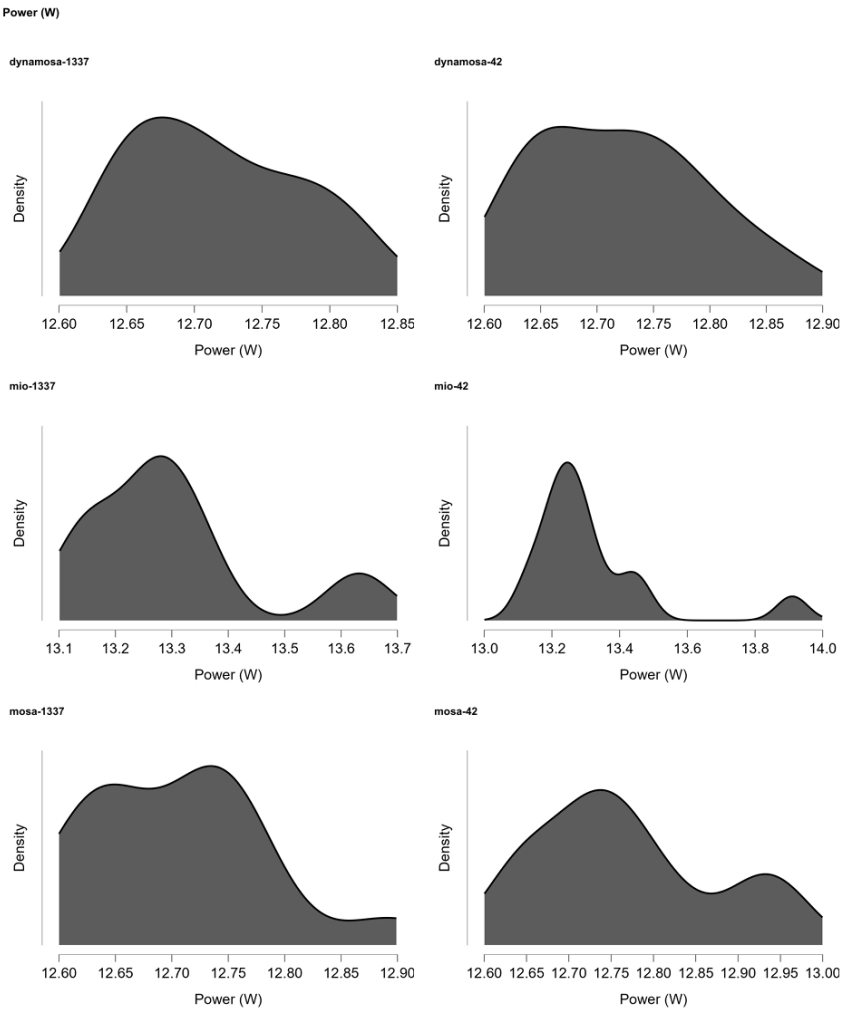

Figure 11: Density Plot - Power

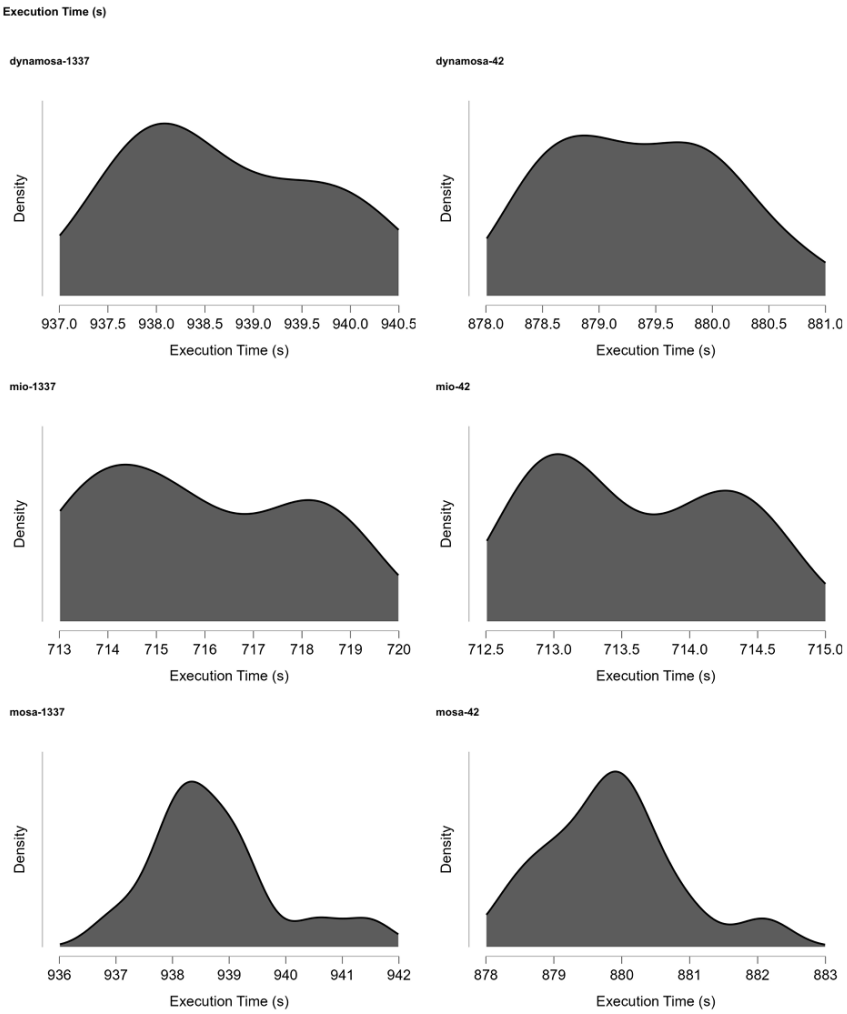

Figure 12: Density Plot - Time

Descriptive Statistics

		Valid	Mean	Std. Deviation	Variance	Range	Minimum	Maximum
Energy Consumption (J)	dynamosa-1337	14	11937.350	65.771	4325.836	184.614	11853.011	12037.625
Energy Consumption (J)	dynamosa-42	14	11186.656	67.801	4596.932	222.104	11094.595	11316.699
Energy Consumption (J)	mio-1337	14	9522.295	113.908	12975.085	384.453	9356.721	9741.174
Energy Consumption (J)	mio-42	14	9502.096	141.545	20035.043	578.165	9359.007	9937.172
Energy Consumption (J)	mosa-1337	14	11927.855	79.762	6362.023	301.540	11805.906	12107.446
Energy Consumption (J)	mosa-42	14	11234.014	97.018	9412.439	291.154	11111.766	11402.919
Execution Time (s)	dynamosa-1337	14	938.705	1.013	1.027	3.333	937.154	940.487
Execution Time (s)	dynamosa-42	14	879.371	0.757	0.573	2.458	878.336	880.794
Execution Time (s)	mio-1337	14	715.868	2.115	4.473	5.943	713.084	719.027
Execution Time (s)	mio-42	14	713.598	0.689	0.475	1.835	712.785	714.619
Execution Time (s)	mosa-1337	14	938.741	1.171	1.372	4.548	936.927	941.475
Execution Time (s)	mosa-42	14	879.840	0.982	0.963	3.781	878.313	882.095
Power (W)	dynamosa-1337	14	12.717	0.062	0.004	0.185	12.642	12.827
Power (W)	dynamosa-42	14	12.721	0.074	0.005	0.237	12.630	12.867
Power (W)	mio-1337	14	13.302	0.158	0.025	0.527	13.121	13.648
Power (W)	mio-42	14	13.316	0.195	0.038	0.786	13.124	13.910
Power (W)	mosa-1337	14	12.706	0.079	0.006	0.294	12.601	12.894
Power (W)	mosa-42	14	12.768	0.105	0.011	0.311	12.636	12.948

Figure 13: Descriptive Statistics

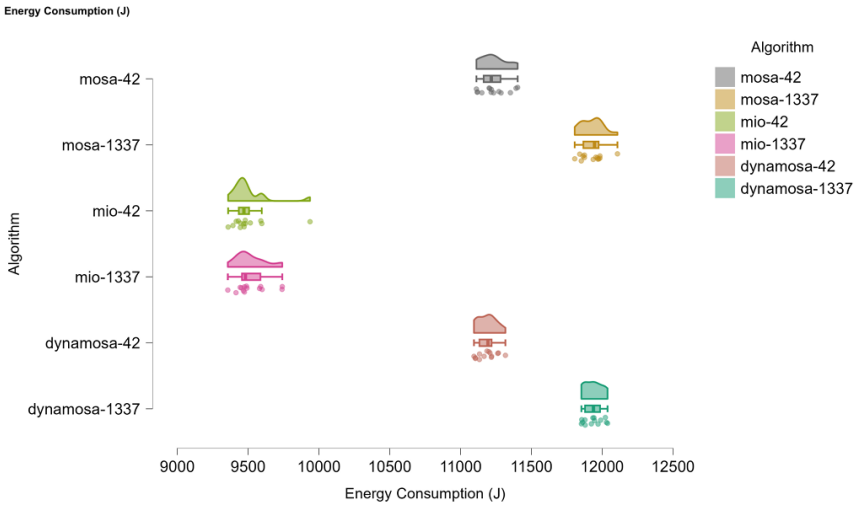

Figure 14: Raincloud Plot - Joules

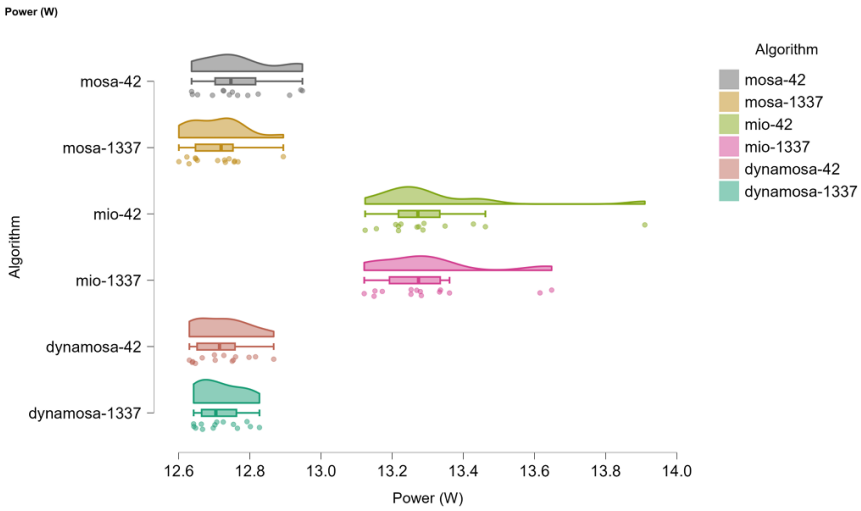

Figure 15: Raincloud Plot - Power

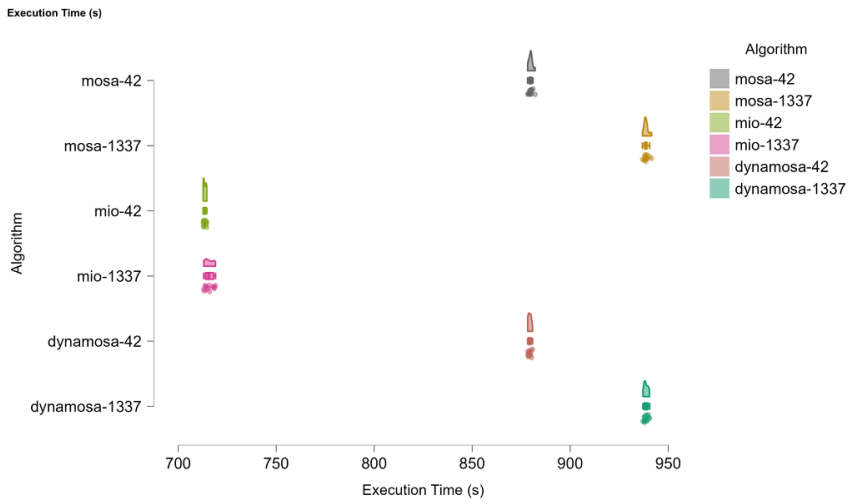

Figure 16: Raincloud Plot - Time

Energy Consumption (J)

Primary Factor	*N*	Lower Whisker	25th Percentile	Median	75th Percentile	Upper Whisker
dynamosa-1337	14	11853.011	11878.407	11937.967	11984.607	12037.625
dynamosa-42	14	11094.595	11133.218	11193.146	11218.956	11316.699
mio-1337	14	9356.721	9456.718	9481.232	9586.662	9741.174
mio-42	14	9359.007	9434.372	9470.236	9506.837	9596.608
mosa-1337	14	11805.906	11866.688	11945.622	11974.348	12107.446
mosa-42	14	11111.766	11163.045	11216.925	11282.010	11402.919

Note. $N_{\text{Total}} = 84$.

Execution Time (s)

Primary Factor	*N*	Lower Whisker	25th Percentile	Median	75th Percentile	Upper Whisker
dynamosa-1337	14	937.154	937.867	938.447	939.567	940.487
dynamosa-42	14	878.336	878.757	879.298	879.942	880.794
mio-1337	14	713.084	714.049	715.627	717.978	719.027
mio-42	14	712.785	712.947	713.432	714.171	714.619
mosa-1337	14	936.927	938.057	938.475	939.132	940.532
mosa-42	14	878.313	879.220	879.803	880.231	880.986

Note. $N_{\text{Total}} = 84$.

Power (W)

Primary Factor	*N*	Lower Whisker	25th Percentile	Median	75th Percentile	Upper Whisker
dynamosa-1337	14	12.642	12.665	12.705	12.762	12.827
dynamosa-42	14	12.630	12.652	12.715	12.758	12.867
mio-1337	14	13.121	13.192	13.274	13.335	13.361
mio-42	14	13.124	13.218	13.272	13.334	13.462
mosa-1337	14	12.601	12.647	12.719	12.753	12.894
mosa-42	14	12.636	12.703	12.747	12.816	12.948

Note. $N_{\text{Total}} = 84$.

Figure 17: General Statistics