Influence of Static Code Analysis on Energy Consumption of Software

Christoph Brosch

Abstract: In recent years, the rise of mobile devices, such as smartphones, smartwatches, or tablets, has led to an increased demand for energy-efficient software. In order to achieve this, developers can use static code analysis tools, such as Pylint, to detect potential issues in their code. This paper investigates how the usage of static code analysis influences the energy consumption of software. More specifically, we used the programming language Python and the general-purpose static code analysis tool Pylint [Py22]. For this purpose, we measured the energy consumption for algorithms implemented in the Benchmarks Game [Go22] before and after implementing the annotations and compared the results. Our findings suggest that resolving the annotations can have a negative impact on energy consumption. This was the case in 3 out of 8 algorithms. The remaining cases showed no significant difference. We assume that the increased energy consumption is due to the multitude of possibilities to implement annotations, leading to a possibility for worsening performance. Further research and experimentation are needed to objectively evaluate the impact of Pylint and static code analysis by extension, on energy consumption.

Keywords: Static code analysis; Linter; Programming; Energy consumption; Efficiency; Python

1 Introduction

Code quality is an important aspect of modern software development, considered essential by all stakeholders [GT22]. To increase code quality, a wide range of procedures are commonly used [GT22]. Some are manual processes that involve at least two people, like code reviews. The author and the reviewer. Other procedures, like static code analysis, can be automated and handled by one developer only. Furthermore, procedures can be divided by their time-consumption, where code reviews and dynamic code analysis both take longer than static code analysis, which can be quickly run after each file save [GS15].

To evaluate software quality, the authors of [Ka10] presented a set of criteria on which software could be tested on. The high-level view of these criteria consists of following components: Functionality, Efficiency, Maintainability and Portability. They are based on ISO/IEC 9126 [IS01]. In this paper, we examine the effect of static code analysis on efficiency, more specific, energy consumption. According to a published survey [CB16], in which developers were asked what they expect from resolving the annotations generated by static code analysis tool, energy consumption took the last place in terms of importance. The 4 key areas, where developers expect improvements are (ordered by importance):

1 Hochschule Trier, Standort Birkenfeld, Institut für Softwaresysteme, Campusallee, 55768 Hoppstädten-Weiersbach, Deutschland, c.brosch@umwelt-campus.de
In this paper we investigate the influence of static code analysis on the energy consumption of software. Differences in the source code of algorithms can lead to an in- or decrease in energy consumption [Ke18]. Therefore, an interest in optimizing algorithms for energy consumption is justified. In addition, the hardware, which the software is run on, also influences the energy consumption [Ke18]. Both variants of the algorithms were run on the same hardware, to ensure that the results are comparable.

This paper is structured as follows. Section 2 highlights related work. Section 3 introduces static code analysis, the examined source code and python package used to measure energy consumption. Afterwards, in section 4 we describe the experiments and the hardware environment. In section 5 we present the results and look in detail at certain algorithms. We end the paper with a discussion of the results and potential future work in section 6.

2 Related work

Energy consumption of software has become an increasingly important topic in recent years. Especially regarding embedded or mobile devices. Therefore, the scientific community published a variety of papers on this topic. One tool to minimize the energy consumption of mobile applications during development is static code analysis [Ba23; Fe22; KKK16; Li17]. One reoccurring topic for both mobile devices [Pa19] and general software [Go12] are code smells [SGS14], which relate to energy consumption. In this work we focus on algorithms written in Python, which is primarily used in server-side or desktop applications. In contrast to other work [Gr15; Li18] we did not implement our own static code analysis tool. Instead, we utilized the Python package Pylint [Py22], which, according to [GP19], is the most popular static code analysis tool in the python community.

3 Methods

3.1 Static code analysis

Static code analysis is a tool to improve code quality during software development [GS15]. It can be seen as a program, which accepts a source file as input and annotates it with suggestions on how to improve the code quality. The user can then choose to act on these annotations. The process of annotating the source file is commonly referred to as linting and the software tools are called linters. Tools for static code analysis exist for most of the common programming languages. Pylint distinguishes between a variety of different so-called checkers. According to their documentation[Py22], Pylint provides 29 different checkers. For example: Basic Checker, Classes Checkers and Design Checkers. Associated with each checker are different messages, which are categorized into: Fatal (F), Error (E), Warning (W), Convention (C), Refactor (R) and Information (I). Across all checkers, Pylint
provides 433 different messages. This information has been extracted from the documentation for version 2.15.10 of Pylint [Py22]. We did not differentiate between the checkers nor their associated messages for their respective influence on energy consumption. This aspect is discussed in more detail for future work. When resolving said annotations from linter tools, the developer must decide how to implement the solution. This allows for varying amounts of effort. In this paper, we decided to resolve the annotations with minimal effort while preserving the program flow if possible. An example of this can be seen in listing 1, which shows an example of the original source code on the left, the annotations provided by Pylint on the bottom and the manually linted version on the right. Pylint awards a score for the submitted source code. Points are deducted from the maximum score of 10 depending on the weighting of the respective checker, whereby the score does not fall below 0.

3.2 Algorithms

The source code examined was taken from the *Benchmarks game* [Go22], a community-driven project, which compares 25 different programming languages regarding their execution time. For this purpose, they provide different implementations of 10 algorithms for a multitude of programming languages (26). The implementations are contributed by the community. We had to select one python implementation from multiple candidates and based our decision on the selection made by Couto et al. in [Gr22]. The only exception being the implementation of the *Pinigits* algorithm since it relies on an additional third-party Python package. Therefore, we chose a different implementation from [Go22]. In the following, the term algorithm refers to the specific implementation we chose for that algorithm.

For 2 out of 10 algorithms the full Pylint score of 10 could not be achieved without changes in the program flow, therefore they are not part of the evaluation. In addition, it must be noted that the same author was involved in 3 of the 8 remaining algorithms. The relation between author and energy consumption is not considered.

3.3 Running Average Power Limit

To measure the energy consumption, we used the Python package pyRAPL [22a]. The package provides an interface to the *Running Average Power Limit* (RAPL) interface integrated in Intel CPUs from generation Sandy Bridge and onwards [22b]. In the following, energy consumption over all processor cores is measured in microjoule and converted to joule to improve readability.

4 Measurements

We measured two versions of each implementation for 8 algorithms. The initial versions, called the *default* algorithms, were compared to revised versions, called the *linted* algorithms.
The revision process for the lintered versions is described in section 3. The updated source code was saved in separate files named lintered.py. All the lintered versions received a perfect score of 10 from Pylint, indicating high code quality from the perspective of the python package.

To evaluate the performance of both the default and lintered versions, we executed each algorithm 10 times, measuring the energy consumption and recording the results. The results were then averaged. It should be noted that, since the measurement system is used as a daily work computer, the presence of system daemons might have had a slight impact on the measurements.

In addition to the measurements conducted on the aforementioned system, we also performed the same experiment on a separate system. The results of these measurements can be found in the repository associated with this work2, specifically in the branch named run_07_11_22_8_10. For further details on this specific run, including the experimental setup and methodology, please refer to the README file in the repository.

---

2 The repository is hosted on GitLab and can be accessed under following link: https://gitlab.rlp.net/cbrosch/nasowat-ws22-messungen-brosch-christoph.
the second system differ significantly from those on the system described in Table 1. Still, despite these differences, the overall trends remain consistent.

![Energy consumption comparison](image1.png)

![Energy consumption comparison](image2.png)

**Fig. 1:** Comparison of the energy consumption of the two different versions of the algorithms. Two plots were generated because of differences in the scale of the y-axis.

## 5 Results

The results presented indicate that the *linted* versions of the algorithms consume, on average, 15.265% more energy compared to the *default* versions. Figure 1 provides a more detailed breakdown of the measurements, showing an average deviation in the range of −2% to 2% for 5 of the 8 algorithms. These small deviations can be attributed to measurement inaccuracies and are considered negligible. Repeated measurements on the same hardware, have shown that the averaged results for each algorithm can vary by a few percentage points, as recorded in the git branch `run_10_11_22_8_10`.

The following sections focus on examining the specific annotations from Pylint, which led to a significant difference in energy consumption in the *linted* versions of the algorithms. Minor changes were made to all algorithms based on the annotations provided by Pylint. Each of these changes could potentially contribute to the increase in energy consumption. To provide further evidence, additional research has been recorded in the associated repository,
demonstrating that the highlighted changes in the following sections have a notable impact on energy consumption.

5.1 n-body

In the given implementation of the n-body algorithm, Pylint highlighted that there were too many local variables used within the function in two instances. The code changes in Listing 2 aimed to address this issue by combining certain local variables into tuples.

### Listing 2: Excerpt of both variants of the n-body algorithm and the annotations provided by Pylint, which led to higher energy consumption.

63:0: R0914: Too many local variables (26/15) (too-many-locals)
86:0: R0914: Too many local variables (21/15) (too-many-locals)

However, it is important to note that accessing elements within a tuple incurs a higher access time compared to accessing a normal variable. This is because tuples hold references that require an additional step to be resolved. Due to the increased access time and the frequency at which the function is called during runtime, the implementation with tuples resulted in higher energy consumption (1).

5.2 spectral-norm

In the case of the spectral-norm algorithm, Pylint identifies that the function eval_A() is called with out-of-order arguments. This is apparent when looking at the entire source code. The code changes in Listing 3 address this issue by introducing the use of the reversed() function to reverse the order of elements in array-like objects.
List. 3: Excerpt of both variants of the spectral-norm-algorithm and the annotations provided by Pylint, which led to a higher energy consumption.

While reversing the arguments resolves the out-of-order issue, the addition of the reversed() function incurs some computational overhead. Consequently, each call to eval_A() within the part_at_times_u() function becomes slightly more time-consuming, resulting in an increase in energy consumption.

5.3 fannkuch-redux

For the fannkuch-redux algorithm, Pylint recommends substituting the if block with a call to the max() function (as shown in Listing 4). This suggestion differs from previous annotations we looked at, where alternative approaches could potentially mitigate significant energy increases. In this case, Pylint’s annotation incorporates a proposed solution, which increases energy consumption. This increase is because of the computational overhead when executing a function, compared to an if statement.

6 Discussion and future work

The measurements conducted for this work indicate that resolving annotations provided by Pylint may lead to an increase in energy consumption. In our showcased experiment the linted source code required 15.265% more energy, for 8 algorithms in total. In detail, only 3 of the 8 algorithms showed a significant increase in energy consumption of, on average, 25%. These algorithms were highlighted in the previous section and required significant changes in their source code to receive the full rating from Pylint. By adding additional function calls or replacing primitive variables with array-like objects inside these algorithms, the energy consumption increased. We chose these changes due to annotations, that either lack clear solutions or require significant code restructuring to avoid negative energy impact. In a production environment, developers would likely make better decisions regarding the
implementation of Pylint’s annotations. Since they are more familiar with the code base. As a result, the findings presented in this work do not provide a conclusive assessment of the impact of Pylint as a representative of static code analysis on energy consumption. To draw more definitive conclusions, several issues have emerged within the scope of this research, which can be explored in future work.

- The documentation of Pylint contains a range of different checker classes [Py22]. In this work, we did not differentiate between them. In future work, we would like to investigate their individual impact on energy consumption.
- By leveraging Pylint’s capabilities which allow users to enhance existing, or define completely new, checker classes, it could be possible to investigate the detection of Energy Code Smells outlined in [Go12] by Gottschalk et al., using static code analysis.
- An evaluation of various implementation options for specific Pylint annotations could be conducted to compare their impact on energy consumption. Based on the results, concrete recommendations can be provided to the user for implementing various individual Pylint annotations.

In conclusion, this paper presented an initial investigation into the impact of static code analysis on energy consumption. The results indicate that resolving annotations provided by Pylint may lead to an increase in energy consumption. However, further research is needed to provide a more definitive assessment of the impact of Pylint on energy consumption.
References


