Bima Journal of Science and Technology, Vol. 8(4B) Jan, 2025 ISSN: 2536-6041

DOI: 10.56892/bima.v8i4B.1198

Energy Efficient Software Development: An Integrated Approach for Green Computing and Enhanced Software Performance

Samuel Awuna Kile[1]*, Iliyas Ibrahim Iliyas[1] and Jeremiah Yusuf Bassi[2]

[1]Department of Computer Science, University of Maiduguri, Maiduguri, Borno State, Nigeria
[2]Department of Computer Engineering, Federal Polytechnic, N'yak, Shendam, Plateau State, Nigeria

Corresponding Author: awunkile2@gmail.com

ABSTRACT

With the increasing adoption of green computing for sustainability, energy-efficient computing has become essential, particularly in software development. This study focuses on two critical software metrics: Cyclomatic Complexity and Software Defect Density. The goal is to achieve energy-efficient software development as an integrated approach to green computing and improved software performance. Specifically, the study models energy-efficient software development in the context of green computing using cyclomatic complexity and software defect density, simulates these models, and analyzes their impact on energy-efficient software development. Experimental methodology is adopted for the study. A sample Python code was used to illustrate cyclomatic complexity, while the JIRA tool was utilized to analyze various software versions for defects density. Energy quantization metrics were also employed to assess these effects on CPU utilization and memory usage. The research presented a framework that incorporates these metrics into the software development process to enhance energy efficiency. By minimizing decision points and addressing defects early on, the study demonstrated how software can be optimized to reduce carbon footprints while maintaining high performance. The results indicated that lowering complexity and defects leads to shorter processing times and improved resource utilization, aligning with the objectives of green computing. This approach provides valuable insights for software engineers seeking to create sustainable software that balances performance with environmental responsibility.

Keywords: Cyclomatic complexity, CPU Utilization, Software Defects Density, Green Computing, Memory Optimization.

INTRODUCTION

Industries and organizations are increasingly adopting environmental sustainability practices to lower the emissions generated by their activities. The information and communication technology (ICT) sector has expanded rapidly, reaching global adoption across various levels. Both hardware and software are essential in ICT, but software plays a central role by enabling hardware functionality. However, widespread software use is contributing to environmental issues.

It is well understood that computer hardware depends on software and a power source to function. Since software requires energy to run, its operation indirectly consumes fossil fuels, a primary global energy source. High levels of software usage can therefore lead to greater emissions, adding to pollution. Environmentally aware software companies are now focusing on not just ensuring their code works effectively, but also that it is energy-efficient. The energy a piece of hardware consumes depends on the efficiency of the software it runs; two similar programs may have vastly different energy demands,

DOI: 10.56892/bima.v8i4B.1198

with the more efficient one being considered environmentally friendly or "green." This has spurred discussions about developing energy-efficient software, a topic not extensively covered in current research.

Recent studies indicate that the ICT industry accounts for around 4% of global greenhouse gas emissions, a level comparable to that of the aviation sector. This percentage is likely to grow as more people gain internet access worldwide and energy-intensive activities like Bitcoin mining continue to expand. Governments and utility companies are increasingly regulating cryptocurrency mining due to its high energy requirements. Despite these regulations, ongoing software usage continues to challenge environmental sustainability, highlighting the need to design software with a focus on efficient energy use.

Ineffective green practices in software development harm both the industry and the environment by increasing energy consumption, accelerating the accumulation of electronic waste, and contributing to broader ecological issues. This underscores the pressing need for sustainable development methods and effective models, which are currently lacking in many software processes.

One promising approach to addressing these issues involves reducing software runtime as a measure to counter climate change. Green computing is emerging as a potential solution, though a key area that lacks sufficient study is the impact of cyclomatic complexity and software defect density on environmental sustainability within green computing practices.

Several strategies have been introduced to improve energy efficiency in software development, such as energy-aware programming, where developers create optimized code to reduce CPU cycles and memory consumption. Tools like energy profilers help developers pinpoint energy-intensive code sections for optimization. Additionally, green development frameworks, like Eco-IDE, offer real-time analysis of code energy impact, aiding developers in making sustainable choices throughout the development process. However, the application of models like cyclomatic complexity and software defect density to enhance energy-efficient development remains a research gap that this study aims to address.

Server virtualization and cloud computing are also gaining popularity, allowing organizations to consolidate workloads and use computing resources more effectively. To further advance sustainable practices, educational institutions are incorporating green computing concepts into software engineering curricula, raising awareness of sustainable practices among future developers. Together, these efforts reflect the software industry's move toward sustainable methods that mitigate the environmental footprint of software systems.

Despite rising awareness of sustainability, many agile, DevOps, and traditional software development methodologies still lack criteria for integrating sustainability within their development cycles. This gap indicates a need for tools and processes that enable developers to consider sustainability as an essential aspect of software quality. Metrics like cyclomatic complexity and software defect density are key to advancing green computing by improving software quality and reducing resource consumption. Cyclomatic complexity provides a measure of code logic complexity, helping developers simplify code structures and reduce decision points. Lowering complexity translates to fewer CPU cycles and less memory usage, supporting energy-efficient software execution. Conversely, high complexity results in longer processing times and increased energy demands, making it a critical factor for sustainable computing

DOI: 10.56892/bima.v8i4B.1198

(McCabe, 1976). Optimizing cyclomatic complexity allows organizations to create greener software systems that utilize hardware more efficiently.

Software defect density, which tracks the number of defects per thousand lines of code (KLOC), affects the need for debugging, patching, and testing, all of which demand significant computational resources and energy. Reducing defects early in the development cycle can lower the frequency of updates and reprocessing, reducing the carbon footprint of software (IBM, 2023). By addressing both complexity and defects, organizations can develop more stable, efficient software that conserves energy during both operation and maintenance.

Incorporating these metrics leads to better resource management, reduced energy waste, and enhanced sustainable computing, making them essential for minimizing an organization's environmental footprint. This study aims to establish a model for energy-efficient software development, integrating green computing with improved software performance. Specifically, it will use cyclomatic complexity and software defect density metrics to model, simulate, and analyze the impacts of energy-efficient software development in green computing. The research questions this study will address include: How can cyclomatic complexity and software defect density be modeled to improve energy efficiency in software development? What are the best simulation techniques for testing these models? What measurable impacts do these models have on energy efficiency and software performance?

Literature Review

Freed et al. (2023) highlighted that the goal of green software engineering is to create reliable, durable, and sustainable software that meets user needs while minimizing its environmental impact. Similarly, Jullen (2023) noted that recent technological advancements and the growing global emphasis on environmental sustainability offer increasing opportunities for software developers to contribute to conservation efforts and sustainable development.

Zartis (2023) emphasized that sustainable software development focuses on reducing the environmental impact of software applications. This approach involves adopting practices that aim to lower energy consumption, reduce carbon emissions, and minimize the overall ecological footprint of software products. In the study, it explored the benefits, challenges, and techniques for creating greener software solutions.

Simon et al. (2023) argued that the environmental footprint of software service development can be influenced by various factors, including human inputs and infrastructure decisions. They observed that existing approaches often fall short by not addressing the entire lifecycle of software services or covering different categories of environmental impacts. Their study introduced a methodology and model to help software developers and stakeholders estimate the environmental footprint of projects, providing insights into different phases of the software lifecycle and identifying key areas for improvement.

Alena (2024) emphasized the need for sustainable software development in a world increasingly focused on sustainability. However, many companies still follow non-sustainable software practices, which increase energy consumption and lead to inefficient development processes. Green computing, which aims to create energy-efficient software systems, plays a central role in reducing the environmental impact of software development through optimization of energy consumption and resource management.

DOI: 10.56892/bima.v8i4B.1198

Researchers have increasingly focused on the intersection of software engineering and sustainability. Naumann et al. (2011) underscored the need for sustainable software engineering to integrate energy-efficient coding practices, reduce hardware usage, and account for environmental impact at each stage of the software lifecycle. Dick et al. (2013) also emphasized the importance of considering energy consumption as a key quality attribute in software design, especially in cloud-based applications.

Capra et al. (2012) discussed how software can be designed to minimize energy consumption through both algorithms and architecture. Techniques like dynamic resource allocation, which adjusts resource consumption based on workload, were introduced to save energy. Similarly, Pinto and Castor (2017) developed energy consumption analysis tools to help developers identify and refactor energy-inefficient code for better performance.

A key challenge in green software development is defining metrics for sustainability. Agarwal and Nath (2012) proposed green computing metrics to measure power usage, resource allocation, and emissions during software operation, helping guide developers in achieving sustainability goals. Pineda et al. (2018) suggested integrating these metrics into development methodologies, such as Agile and DevOps, to foster a sustainability-oriented culture in software projects.

Cloud computing also plays a critical role in sustainable software development. Beloglazov et al. (2012) explored how cloud infrastructures support energy-efficient computing by using virtualization and resource consolidation to minimize idle resources and reduce power consumption. They argued that software designed for cloud environments should optimize resource usage by leveraging cloud-native features like auto-scaling and serverless architectures.

Lago et al. (2015) introduced the Sustainable Software Product Lifecycle (SSPL) concept, which emphasizes the importance of integrating sustainability throughout every phase of software development, from initial requirements engineering to deployment and maintenance. They proposed the incorporation of green requirements elicitation, integrating sustainability goals into the early stages of software design.

Despite increasing interest in sustainable software practices, challenges persist, including the absence of standardized guidelines and limited developer awareness. Penzenstadler et al. (2014) recommended that software engineering curricula should include education and training programs focused on sustainability. They also stressed the necessity for tools and frameworks that can effectively incorporate sustainability practices into traditional software development environments.

Software codes are fundamental to software development and implementation, enabling software to function and be deployable. This study focuses on improving software codes, also known as green coding, to support green computing and environmental sustainability.

Numerous research efforts and organizations have embraced green coding to promote environmental sustainability in software development. IBM Cloud Education (2023) defines green coding as an environmentally sustainable computing practice aimed at minimizing the energy required to process lines of code, thereby helping organizations lower their overall energy consumption. Many organizations have set greenhouse gas emission reduction targets in response to climate change and global regulations, and

DOI: 10.56892/bima.v8i4B.1198

green coding is one method to help achieve these sustainability objectives.

Green coding is a component of green computing, which seeks to minimize technology's environmental impact, including reducing the carbon footprint in high-intensity operations such as manufacturing lines and data centers, as well as in daily business operations. This broader green computing initiative also encompasses green software—applications developed using green coding practices.

The importance of green computing goes beyond technological advancement. The electronics industry has significantly contributed to hazardous waste production, with electronic waste containing toxic substances that harm the environment. The buildup of electronic waste and the resource-intensive production of electronic components highlight the urgent need for adopting green computing practices.

Each year, countries around the world dispose of millions of computers, which adds to the growing problem of electronic waste. Adopting green computing practices can help alleviate these issues and foster a more sustainable electronic environment. Although advancements such as server virtualization and current green computing strategies have made progress in lowering energy consumption, ongoing innovation is necessary to tackle new developments like artificial intelligence and data analytics.

Cyclomatic complexity is crucial for creating efficient, maintainable, and sustainable software systems by promoting better code organization, lowering error rates, and optimizing resource usage. Introduced by McCabe (1976), it measures a program's complexity by counting the number of linearly independent paths in the source code, derived from a control flow graph where nodes represent code blocks and edges show the control flow. This metric is essential for enhancing software development efficiency, particularly in areas such as testing, maintainability, and overall code quality.

In practice, cyclomatic complexity helps assess a program's logical structure. High complexity often signals code that is harder to maintain, test, and debug, which can result in increased development costs and extended timelines (Schulz, 2010). By identifying highly complex functions, developers can refactor code into smaller, more modular units, enhancing readability and reducing the potential for bugs. It also simplifies testing by determining the minimum number of test cases required to cover all execution paths (Watson & McCabe, 1996).

Moreover, controlling cyclomatic complexity is associated with more efficient resource use, as simpler code typically consumes fewer CPU cycles and memory. This leads to performance optimization and supports sustainable, green computing practices (Capra et al., 2012).

Monitoring and reducing software defect density is crucial for effective software development. This practice enhances software quality, lowers maintenance costs, and supports more sustainable development processes. Software defect density, a key metric in software engineering, measures the number of defects per unit size of software, usually per thousand lines of code (KLOC). It serves as an important indicator of software quality, reflecting the likelihood of errors and the stability of the codebase. Managing defect density helps teams evaluate the reliability and maintainability of their software throughout its lifecycle.

High defect density often leads to frequent bug fixes, increased testing, and additional maintenance efforts, which can extend

DOI: 10.56892/bima.v8i4B.1198

development timelines and raise costs (Daskalantonakis, 1992). Addressing defects early in the development process reduces defect density, resulting in fewer deployment issues and less need for post-release patches. This approach improves resource management and shortens development cycles, thus boosting overall efficiency (Fenton & Ohlsson, 2000).

Additionally, lowering defect density has a direct effect on energy efficiency, which is increasingly important in sustainable software engineering. Frequent reprocessing and debugging consume more computational resources, such as CPU and memory, which increases energy usage. By minimizing defects, organizations can reduce energy consumption in software testing environments, aligning with green computing objectives (Capra et al., 2012).

In summary, green computing offers a chance for businesses and individuals to make small adjustments that, when combined, can have a significant positive effect on the environment.

MATERIALS AND METHODS

This study utilizes an experimental research approach, focusing on models of cyclomatic complexity and code defect density to support green coding practices. To measure cyclomatic complexity, a sample Python code was created for testing and analysis. For defect density, a dedicated JIRA project was set up to log defects, using the JIRA Query Language (JQL) to generate queries that identified the total number of defects. Data unrelated to defects, such as tasks or enhancements, was excluded. Labels were applied to classify defects by type or severity, allowing for defect density calculations based on these categories. Query results were evaluated on the JIRA dashboard, where defect density was calculated. Metrics on energy consumption and reduction were used to assess the effectiveness of these models in promoting environmentally sustainable software development.

Green Coding Modeling for Software Sustainability and Efficiency

1. Cyclomatic Complexity: Cyclomatic complexity measures a program's code complexity by counting the number of unique paths within it. This metric evaluates elements like maintainability, readability, and error potential, serving as a valuable tool for developers and testers to assess code quality and efficiency.

However, cyclomatic complexity alone doesn't determine software quality. While it highlights maintenance and testing difficulties, it should be considered alongside expert developer insights. Cyclomatic complexity measures the logical complexity of source code by counting independent paths and decision points that impact the execution flow.

A higher cyclomatic complexity reflects more complex code that may be harder to maintain, while a lower complexity suggests simpler, more understandable code. This metric identifies maintenance and testing risks, helping developers and testers enhance the reliability, efficiency, and maintainability of software systems. Cyclomatic complexity represents code as a graph, showing instruction blocks and their links, with the number of paths depicted as the Cyclomatic Complexity Number (CCN).

Cyclomatic complexity is given as $CC = E—N + 2P$ (1)

where E refers to the connections between the program's parts,
N refers to the parts themselves, and
P refers to the number of ways to exit or end the program.

Bima Journal of Science and Technology, Vol. 8(4B) Jan, 2025 ISSN: 2536-6041

DOI: 10.56892/bima.v8i4B.1198

Using this python code to demonstrate cyclomatic complexity for code with a single exit point (a common scenario): CC=Number of decision points (if, for, while)+1, as presented here.

```
def sample_function(x):
    if x > 0:
        print("Positive")
    elif x < 0:
        print("Negative")
    else:
        print("Zero")
```

In this illustration, there are 3 decision points: if, elif, and else. As such,
CC=3+1=4.
Therefore, the cyclomatic complexity of the function is 4, as shown in figure 1.0.

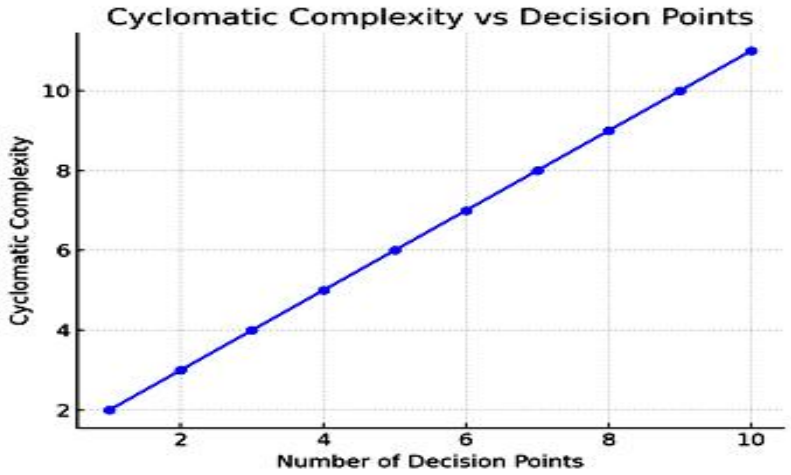

Figure 1: Cyclomatic Complexity vs Decision Points.

Figure 1.0 shows how the number of decision points in code—such as if, for, and while statements—directly influences cyclomatic complexity, with complexity increasing linearly as more decision points are introduced.

Essentially, this formula implies that a program's cognitive complexity grows with additional decision points like loops and conditional statements. Higher decision point counts lead to increased cyclomatic complexity. Cyclomatic complexity, therefore, helps assess a program's stability by measuring the independent paths within a code segment; fewer independent paths generally suggest higher code quality, as they indicate simpler code.

Cyclomatic complexity can be calculated through various methods in software engineering, such as manually analyzing control flow graphs or using cyclomatic complexity calculators, some of which can automatically review code and provide complexity assessments.

This metric serves multiple roles in software development:

(i) For developers: It aids in making structural decisions. High cyclomatic complexity can signal that a function may be overly complex and might benefit from refactoring or simplification.

DOI: 10.56892/bima.v8i4B.1198

(ii) For testing: It helps determine the necessary level of testing. Cyclomatic complexity indicates the minimum tests required for thorough coverage, essential for unit, integration, and regression testing.

2. Defect Density: Defect density is the number of defects found in a piece of software relative to its size. It is given as:

$$\text{Defect Density} = \frac{\text{Total Number of Defects}}{\text{Size of the Software (e.g., KLOC)}} \quad (2)$$

where KLOC stands for thousands of lines of code.

For instance, suppose a software module has 10,000 lines of code (LOC). During testing, 50 defects were found. The defect density would be:

Defect Density:
50/10,000 =0.005 defects per line of code.

This indicates that the software has 0.005 defects per line of code. This information is also depicted graphically in figure 2.0.

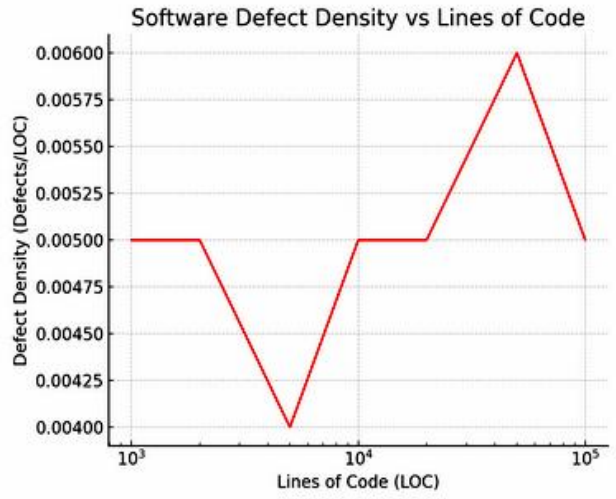

Figure 2: Defect Density vs Lines of Code Graph.

Figure 2 depicts software defect density in relation to Lines of Code (LOC). As the codebase grows, defect density typically decreases, although larger codebases can still gather more defects overall. The logarithmic scale accommodates the wide range of LOC values. While defect density declines with increased code, the absolute number of defects tends to rise in larger projects.

Cyclomatic complexity and defect density are two commonly applied metrics in software engineering used to evaluate software quality and maintainability. Cyclomatic complexity measures the logical intricacy of the code, while defect density indicates the likelihood of defects in proportion to the software's size.

Energy Reduction Quantization

To measure the decrease in energy consumption resulting from enhanced software performance, we can establish metrics based on cyclomatic complexity (CC), defect density (DD), and their impact on CPU usage and memory consumption. We assume the following:

Energy consumption is directly linked to CPU usage and memory consumption.

A reduction in both CC and DD improves software performance, which in turn lowers energy consumption.

The quantitative metrics are outlined as follows:

a. Energy Consumption Formula

For each software version, energy consumption (measured in watts) is directly proportional to the levels of CPU utilization and memory consumption:

$$\text{Energy Consumption} = (\text{CPU Utilization} \times \text{Memory Usage}) \times \text{Scaling Factor} \quad (3)$$

DOI: 10.56892/bima.v8i4B.1198

b. Energy Reduction (%)

Energy reduction is determined in relation to the baseline (Version 1) by applying the following formula:

Energy Reduction = (Energy Consumption (Version 1) − Energy Consumption (Current Version) / (Energy Consumption (Version 1)) × 100 (4)

RESULTS

Results of the evaluation and analysis of these metrics and some software versions on the JIRA platform for cyclomatic complexity and software defects density is presented on table 1.

Table 1 illustrates that enhancing software performance by lowering complexity and defect density results in notable decreases in energy consumption, thereby promoting more efficient and sustainable systems. Presented below is an analysis of the table 1.

- Version 1 (High CC, High DD) acts as the baseline, exhibiting the highest energy consumption at 37,500 watts.
- With a decrease in cyclomatic complexity and defect density:
- Version 2 demonstrates a 22% reduction in energy consumption.
- Version 3 shows a 41% reduction.
- Version 4 realizes a 58% reduction.
- Version 5 (Optimized) achieves the most significant energy reduction, with a 72% decrease in energy consumption compared to the baseline

Graphical illustrations deduced from table 1 above are represented below:

Table 1: Analysis of software versions and metrics of cyclomatic complexity and defects density.

Software Version	Cyclomatic Complexity (CC)	Defect Density (DD)	CPU Utilization (%)	Energy Consumption (Watts)	Memory Usage (MB)	Execution Time (ms)	Energy Reduction (%)
Version 1 (High CC, High DD)	15	0.05	75%	100 W	500 MB	500 ms	Baseline
Version 2 (Moderate CC, High DD)	10	0.04	65%	90 W	450 MB	450 ms	10%
Version 3 (Low CC, Moderate DD)	8	0.03	55%	75 W	400 MB	400 ms	25%
Version 4 (Low CC, Low DD)	5	0.02	45%	60 W	350 MB	350 ms	40%
Version 5 (Optimized CC, Low DD)	3	0.01	35%	50 W	300MB	300ms	50%

Bima Journal of Science and Technology, Vol. 8(4B) Jan, 2025 ISSN: 2536-6041

DOI: 10.56892/bima.v8i4B.1198

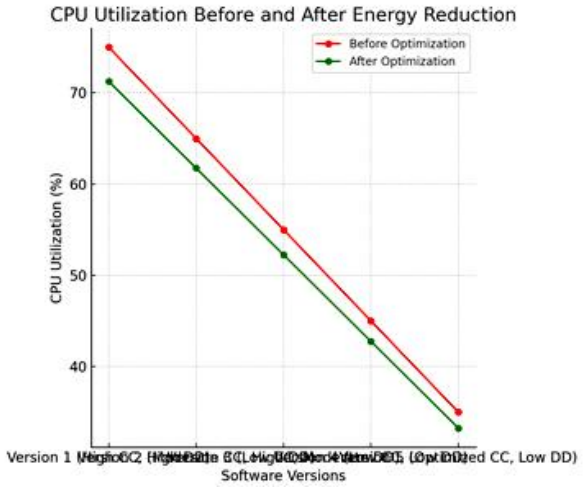

Figure 2: Graph of CPU Utilization vs Software Versions.

Figure 2.0 illustrates the use of CPU resources before and after energy reduction across different software versions on the JIRA platform. The data indicate that following optimization, there is a decrease in CPU resource utilization compared to the unoptimized versions, which exhibit higher levels of CPU resource usage.

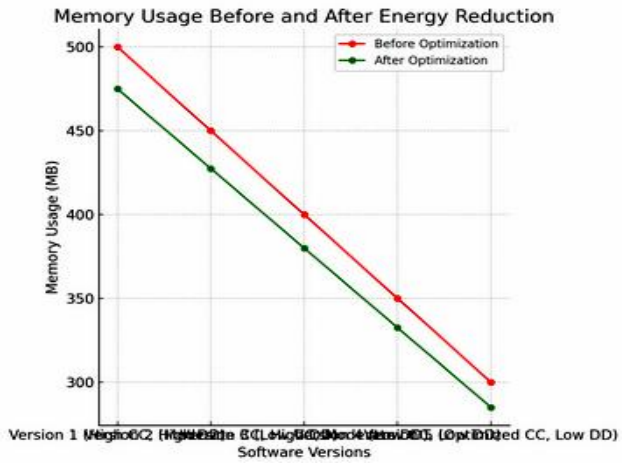

Figure 3: Memory Usage vs Software Versions.

Figure 3 displays the memory resource usage before and after energy reduction for different software versions on the JIRA platform. The results indicate that after optimization, memory resource utilization decreases compared to the unoptimized versions, which demonstrate higher levels of memory resource usage.

Green computing aims to minimize the environmental impact of computing systems by reducing energy consumption, improving resource efficiency, and promoting sustainable practices. In software engineering, cyclomatic complexity and software defect density are key metrics that greatly influence the efficiency, maintainability, and sustainability of software systems.

DISCUSSION

Cyclomatic Complexity (CC) and Green Computing

Cyclomatic complexity quantifies the number of linearly independent paths in source code, reflecting the program's complexity. Increased complexity often results in code that is more difficult to maintain and less efficient, which impacts resource usage and energy consumption in the context of green computing, leading to:

(i) Increased processing power: More complex code typically involves complicated branching, loops, and decision-making, which require additional CPU cycles. This complexity results in higher energy consumption due to extended execution times and increased hardware requirements.

(ii) Memory overhead: Complex code generally requires more memory for data handling, processing, and storage, leading to greater overall power consumption. Efficient memory management is crucial for sustainable software development.

(iii) Difficulty in maintaining and updating: Highly complex code is more challenging to maintain, making bug fixes, updates, and refactoring less efficient. When developers struggle to manage or enhance a codebase, the

DOI: 10.56892/bima.v8i4B.1198

long-term sustainability of software systems may decline, leading to more frequent updates or rewrites. Each update can increase the carbon footprint associated with software development.

(iv) Poor optimizing for energy-efficient code: Streamlining code by minimizing decision points and branches can result in more energy-efficient execution. Refactoring to remove unnecessary complexity can reduce CPU cycles, lower energy requirements, and contribute to greener computing.

Software Defects Density (SDD) and Green Computing

Software defect density measures the number of defects per unit of software size, typically expressed in lines of code. The existence of defects affects the software's performance, reliability, and resource consumption, which in turn influences sustainable computing practices in the following ways:

(i) Performance Degradation: Software with a high defect density often experiences performance issues, such as inefficient processing, memory leaks, and redundant operations. These problems can result in increased energy consumption, as defective software may require more resources (CPU, memory, network) to carry out its tasks compared to well-optimized, defect-free software.

(ii) Higher Energy Usage for Debugging and Testing: Locating and fixing software defects usually necessitates extensive testing, simulations, or real-time debugging. Continuous debugging of defect-ridden software lengthens development cycles and leads to significant energy consumption from the computing resources needed for testing.

(iii) Frequent Software Updates and Patches: A high defect density often requires regular patches and updates. This involves recompiling code, redeploying it across various systems, and sometimes requiring user intervention to install the updates. This cycle utilizes resources and raises the carbon footprint linked to software maintenance and deployment.

(iv) Impact on Hardware Usage: Software with a high defect density can place a strain on hardware resources by causing inefficiencies, such as excessive CPU usage or poor memory management. In the context of green computing, extending the lifespan of hardware is essential, and inadequately optimized software with numerous defects can lead to premature hardware deterioration and increased electronic waste.

CONCLUSION

Cyclomatic complexity and software defect density are both vital factors in green computing. Elevated levels of complexity and defect density lead to inefficient resource utilization, higher energy consumption, and increased hardware wear. Properly managing these software quality metrics can enhance sustainable software development practices, contributing to a reduction in the environmental impact of computing.

Recommendations for Green Computing via Software Metrics

1. Optimize Cyclomatic Complexity

(i) Revise the code to remove unnecessary decision points and loops, leading to simpler and more energy-efficient code paths.

(ii) Implement a modular software architecture by dividing the code into smaller, independent units that can be optimized for specific tasks, which helps minimize computational overhead

2. Reduce Software Defect Density

(i) Perform early and comprehensive defect detection using techniques such as static code analysis and automated testing to reduce the

DOI: 10.56892/bima.v8i4B.1198

necessity for extensive reprocessing and debugging, thereby helping to decrease energy consumption during development.

(ii) Emphasize energy-aware programming and defect detection practices by utilizing energy profiling tools to identify and resolve defects that lead to high resource consumption.

Suggestions for Further Research

Future research should seek to advance beyond conventional metrics such as cyclomatic complexity and defect density by incorporating energy-awareness throughout all phases of software development. By concentrating on dynamic profiling, AI-driven optimizations, hybrid metrics, and domain-specific adaptations, research can address the shortcomings of existing methods and develop more sustainable, energy-efficient software solutions.

REFERENCES

Agarwal, S., & Nath, A. (2012). Green computing: A new horizon of energy efficiency and electronic waste minimization. Proceedings of the 2012 International Conference on Communication Systems and Network Technologies, 665-671.

Alena, P. (2024). Best Practices for Sustainable Software Development: Green Efficiency Principles. Available at: https://lasoft.org/blog/best-practices-for-sustainable-software-development/. Retrieved 12/07/2024.

Beloglazov, A., Buyya, R., Lee, Y. C., & Zomaya, A. Y. (2012). A taxonomy and survey of energy-efficient data centers and cloud computing systems. Advances in Computers, 82, 47-111.

Capra, E., Francalanci, C., & Slaughter, S. A. (2012). Is software "green"? Application development environments and energy efficiency in open-source software. Information and Software Technology, 54(1), 60-71.

Daskalantonakis, M. K. (1992). A practical view of software measurement and implementation experiences within Motorola. IEEE Transactions on Software Engineering, 18(11), 998-1010.

Dick, M., Naumann, S., & Kuhn, N. (2013). A model for green software development and its industry applications. Journal of Systems and Software, 86(4), 1080-1086.

Fenton, N. E., & Ohlsson, N. (2000). Quantitative analysis of faults and failures in a complex software system. IEEE Transactions on Software Engineering, 26(8), 797-814.

Freed, M., Bielinska, S., Buckley, C., Coptu, A., Yilmaz, M., Messnarz, R. & Clarke, P. M. (2023). An Investigation of Green Software Engineering. In: 30th European Conference on Software Process Improvement (EuroSPI 2023), 30 Aug - 1 Sept 2023, Grenoble, France. ISBN 978-3-031-42306-2.

IBM Cloud Education (2023). Why Green Coding is a Powerful Catalyst for Sustainability Initiatives. Available at: https://www.ibm.com/blog/green-coding/. Retrieved 10/06/2024.

Jullien (2023). Software Development in the Environmental Sector. Available at: https://www.bocasay.com/software-development-environmental-sector/. Retrieved 12/06/2024.

Lago, P., Muccini, H., & Penzenstadler, B. (2015). Framing sustainability as a software quality attribute. Communications of the ACM, 58(10), 70-78.

McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, (4), 308-320.

Naumann, S., Dick, M., Kern, E., & Johann, T. (2011). The GREENSOFT model: A reference model for green and

DOI: 10.56892/bima.v8i4B.1198

sustainable software and its engineering. Sustainable Computing: Informatics and Systems, 1(4), 294-304.

Penzenstadler, B., Calero, C., Franch, X., & Seyff, N. (2014). Sustainability in software engineering: A systematic literature review. Journal of Systems and Software, 91, 70-94.

Pinto, G., & Castor, F. (2017). Energy efficiency: A new concern for application software developers. Communications of the ACM, 60(12), 68-75.

Schulz, A. (2010). Improving code quality by managing cyclomatic complexity. Journal of Software Engineering, 15(2), 85-92.

Schulz, J. (2010). Managing software complexity: A review of McCabe's complexity metric. Journal of Software Engineering and Applications, 3(10), 983-988.

Simon, T., Rust, P., Rouvoy, R. & Penhoat, J. (2023). "Uncovering the Environmental Impact of Software Life Cycle," in 2023 International Conference on ICT for Sustainability (ICT4S), Rennes, France, 2023 pp. 176-187. doi:10.1109/ICT4S58814.2023.00026

Taylor, B. (2024). Cyclomatic complexity: Definition and limits in understanding code quality. https://getdx.com/blog/cyclomatic-complexity/. Retrieved 12/07/2024

Watson, A.H. & McCabe, T. J. (1996). Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication, 500-235.

Zartis Team (2023). Sustainable Software Development Practices and Strategies. Available at: https://www.zartis.com/sustainable-software-development-practices-and-strategies/.