CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Assessing the Energy Impact of Java Software Debloating Tools

Master's Thesis in Computer science and engineering

Martin Engström

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's Thesis 2025

Assessing the Energy Impact of Java Software Debloating Tools

Martin Engström

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

Assessing the Energy Impact of Java Software Debloating Tools
Martin Engström

Supervisor: Gregory Gay, Mohannad Alhanahnah, Department of Computer Science and Engineering
Examiner In Practice: Vladislav Indykov, Department of Computer Science and Engineering
Examiner: Daniel Strüber, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

Martin Engström
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Context: Software debloating tools are commonly used to reduce application size, minimize attack surface, and enhance performance, but may also offer potential for lowering energy consumption in Java applications.
Objectives: This study examines the energy-related impact of four Java debloating tools—ProGuard, DepClean, DepTrim, and JLink—by quantifying their effects on energy consumption and identifying factors that influence the energy use of debloated Java applications.
Methods: An empirical evaluation was conducted on 10 benchmark Java projects and 5 real-world Java applications. For each system, CPU and memory usage, execution time, power draw, and total energy consumption were measured across 30 independent trials, comparing debloated versions produced by each tool against the original baseline.
Results: Across all systems, energy and performance metrics remained largely consistent between debloated and baseline versions. While tool effectiveness varied by project, DepTrim achieved the greatest mean energy reduction, with a decrease of 0.80% in real-world applications.
Conclusion: The evaluated Java debloating tools yielded minimal improvements in energy efficiency and impact on the performance metrics for the tested systems, despite employing varied optimization strategies. These findings highlight the need for more energy-conscious debloating techniques and robust evaluation frameworks to promote sustainable software engineering practices. The methodology presented in this study can serve as a foundation for future work at the intersection of energy consumption measurement and software debloating strategies.

Keywords: Software Bloat, Energy Consumption, Software Debloating

Acknowledgements

I would like to thank my supervisors, Gregory Gay and Mohannad Alhanahnah, for their invaluable guidance, support, and encouragement throughout the course of this project.

I am also thankful to Vladislav Indykov for their considerate and supportive demeanor during the examination process.

Thank you all for your contributions and professionalism.

Martin Engström, Gothenburg, June 2025

Contents

List of Figures

List of Tables

1

Introduction

In 2020, it was estimated that roughly 4% of global use stage electricity consumption—a figure that appears to grow steadily—can be attributed to the information and communication technology (ICT) sector [1]. This suggests that any widely applicable technique that increases the energy efficiency of software can have broad positive effects on the energy consumption of the world. While improvements are continuously being made to reduce the energy consumption of hardware the software is run on, there is less consideration for energy efficiency of the software itself. This results from a lack of awareness, knowledge on how to reduce energy consumption in software, as well as the lack of knowledge of existing tools for the problem [2]. A recent study involving 40 participants from various software domains found that 62.5% had little to no knowledge about energy consumption, and that 48% had never considered energy consumption at any stage of software development [3].

Software bloat is a term that refers to code and features that are not necessary to run an application [4]. Unused code consumes memory, necessitates increased storage capacity, and has the potential for causing unnecessary processing and data overhead [5]. Duplicate code can be introduced when developers are unaware of pre-existing functionality, and develops it again or imports it from somewhere else for the same project. Software bloat can also arise from overly complex algorithms and unnecessary functionalities, which create performance bottlenecks and require more CPU cycles to accomplish tasks that could be executed more efficiently. This decline in performance directly contributes to increased energy consumption. Bloat is sometimes introduced to improve maintainability and readability, particularly in modular programs with added abstractions and dependencies for flexibility and to simplify the development [5]. However, not all of this code is needed during execution.

To enhance software energy efficiency, the most effective approach is to plan ahead and develop sustainable software during the design phase, a strategy employed by at least 40% of participants in [3]. However, when developers lack the necessary skills or are working with an already inefficient solution, automated tools could help to quickly optimize energy efficiency. Automated methods to reduce unnecessary bloat, referred to as software debloating techniques, can enhance performance and efficiency in deployment versions of the software while maintaining clarity in development versions. Historically, limited hardware resources necessitated optimized

code, but advancements in hardware performance and a growing focus on broader functionality have led to significant growth in software bloat [5].

Automated debloating tools aim to remove unnecessary components from software, which could directly impact its energy efficiency. According to [6], the majority of debloating tools have the goal of improving performance and reducing attack surfaces of the software, and have not adequately explored their impact on energy efficiency. The study reveals that only one out of 48 debloating tools focuses primarily on reducing energy consumption. The goal of this thesis is to help fill that gap by evaluating the effects that a set of debloating tools has on energy consumption, and to produce insight on how suitable the chosen tools are for energy efficiency improvements. This insight will offer software engineers and debloating tool developers valuable context on the debloating tool landscape, emphasizing an energy-conscious perspective.

1.1 Problem Description

The energy demand of software is steadily increasing. Between 2015 and 2020, emissions from the ICT sector rose by 5%, with the sector accounting for approximately 4% of global energy consumption in 2020 [1]. This equates to an estimated 916 TWh—enough to power Sweden for over five years based on energy usage statistics between 2001–2023 [7]. Since energy generation often depends on carbon-emitting processes, increased consumption leads to a larger carbon footprint, exacerbating climate change. In 2020, ICT sector energy usage accounted for approximately 492 megatonnes of CO_2 emissions—about six times Sweden's total emissions that year [8].

Beyond environmental concerns, excessive energy consumption also leads to inefficient resource utilization. Batteries drain faster and degrade more quickly, while higher energy demands translate into increased costs for households, businesses, and the public sector.

Existing debloating tools remove excess code to enhance performance, usability, and robustness while minimizing security threats by reducing potential attack surfaces [6]. Additionally, such tools could potentially lower software energy consumption. However, a study by Alhanahnah et al. found that 47 out of 48 mapped debloating tools do not consider energy consumption and were not designed with energy efficiency in mind [6]. As a result, their impact on energy consumption remains largely unknown. This oversight may stem from a lack of awareness among software developers and companies regarding the issues associated with high energy consumption, as suggested by previous studies [2]. The absence of a sustainability perspective limits the ability to factor energy efficiency into debloating tool selection. Given the challenges posed by excessive energy consumption, integrating energy awareness into the development and use of debloating tools could offer significant benefits.

1.2 Purpose of the Study

This study aims to explore the impact of software debloating tools on energy consumption. By examining this relationship, the findings are expected to provide engineers with the knowledge necessary to reduce both energy costs and the carbon footprint associated with software systems. On a broader scale, reducing energy consumption and carbon emissions will contribute to environmental sustainability.

Current software debloating tools and related studies have not fully addressed their impact on energy consumption, creating a gap that this thesis seeks to explore. Specifically, this research investigates how software debloating can reduce energy consumption. To achieve this, the study focuses on three primary objectives:

1. Assess the impact of a selected set of debloating tools on energy consumption.

2. Identify what factors influence the energy consumption of debloated applications.

3. Establish benchmarks to support future research in this area.

To address objective (1), energy consumption was measured through controlled experiments using four debloating tools applied to five real-world programs and ten benchmark programs. For objective (2), the relationships between energy consumption and other performance metrics—power consumption, CPU and memory utilization, and execution time—were analyzed, alongside the tools' direct impact on these metrics. These insights were used to establish benchmarks, fulfilling objective (3).

This study focuses on Java-based debloating tools for three key reasons. First, Java is the fourth most widely used programming language in the world [9], and is widely used across both server-side and mobile platforms, making the findings broadly applicable.

Second, Java benefits from an extensive set of mature open source software. Platforms such as Maven Central provide an extensive repository of Java libraries and frameworks, which helped with data collection and experimentation, and enabled a more comprehensive analysis.

Third, upon reviewing available debloating tools, it was found that Java has the most extensive selection. This proved advantageous when tools did not perform as expected or were unsuitable for the study. Focusing on Java tools kept the research scope manageable and flexible.

1.3 Significance of the Study

This study made two key contributions. First, it established energy consumption benchmarks for tools used in software debloating for Java. This is a noteworthy contribution, as our research reveals a lack of prior studies examining the energy con-

sumption aspect of software debloating tools. These benchmarks will provide valuable guidance to developers interested in reducing energy usage and to researchers who are designing new debloating tools.

Second, this study will propose a methodology for measuring the impact of debloating tools on energy consumption, paving the way for future research in this area. This is crucial because both debloating software while keeping the intended functionality and measuring energy consumption at a fine-grained level are complex and resource-intensive tasks. Although substantial literature exists on these topics individually, there is a notable lack of resources addressing their intersection. This study aims to bridge this divide, providing a foundation for further exploration and refinement.

1.4 Thesis Outline

This thesis is structured as follows:

- **Chapter 1: Introduction** – This chapter provides an overview of the research, including its motivation, objectives, and background. It establishes the context for the study and highlights the key problems addressed.
- **Chapter 2: Background** – This chapter introduces fundamental concepts relevant to this research, including file types, Bayesian statistical analysis, various approaches to debloating, and energy measurement techniques.
- **Chapter 3: Review of the Literature** – A review of existing research on energy savings through debloating is presented. This chapter also examines the landscape of Java debloating tools and explores related optimization techniques, such as genetic algorithms for reducing energy consumption.
- **Chapter 4: Methods** – This chapter outlines the research questions and describes the methodology used to address them. It details the debloating tools analyzed, the selected Systems Under Test (SUTs), and the experimental setup.
- **Chapter 5: Results** – The findings from the conducted experiments are presented, offering insights into the impact of debloating on energy consumption and other performance metrics.
- **Chapter 6: Discussion** – This chapter interprets the results, discusses their implications, and addresses the limitations of the study.
- **Chapter 7: Conclusion** – This chapter summarizes the key findings, reflects on the research questions, and outlines directions for future work in energy-aware software debloating.

2

Background

This chapter provides an overview of key concepts relevant to this study. The following sections introduce Java ARchive (JAR) files and their variations, different strategies of software debloating, and the process of measuring energy consumption in software. Additionally, a discussion on Bayesian statistics is presented, outlining its principles and its significance in statistical modeling and inference.

2.1 JAR Files

JAR (Java ARchive) is a packaging format used to bundle all the components necessary to run a Java application as a single compressed file [10]. A JAR file typically includes compiled class files, associated resources, and any dependent libraries required by the application. Jar files can include an optional manifest file. The manifest file can specify the application's main class through the Main-Class attribute, enabling the JAR to be executed directly with the `java -jar` command. When this attribute is set, the Java Runtime Environment (JRE) can automatically identify the entry point of the application. However, not all JAR files are executable—some serve as libraries or dependencies for other applications. In such cases, a main class is not required, and the JAR functions similarly to traditional software libraries, providing reusable code and resources.

2.1.1 Fat JAR Files

A fat JAR, also known as an "uber" JAR, is a self-contained Java archive that packages an application's code, resources, and all external dependencies into a single file. Unlike standard JAR files, which rely on external classpath configurations or dynamically retrieved dependencies at runtime, fat JARs embed all required components internally. This self-containment ensures that the application can run on any system with a compatible Java Runtime Environment (JRE) without requiring additional libraries or network access to resolve dependencies.

The primary advantage of a fat JAR is its portability and ease of deployment, as it eliminates dependency management issues across different environments. However, embedding all dependencies increases the file size, leading to higher storage requirements. Additionally, updating dependencies becomes more complex, as modifica-

tions require rebuilding the entire JAR. Despite these drawbacks, bundling dependencies can be beneficial in certain cases, such as when developers modify libraries using debloating tools to reduce the overall application size or improve efficiency.

2.2 Software Debloating

Software debloating is the process of removing unnecessary code to streamline development, reduce project size, enhance execution performance, and mitigate security risks [6]. In modular design, many imported libraries contain unused code, as only a subset of their functionality is required. This redundant code can often be safely removed. Additionally, software projects may contain unused or redundant code that can be minimized or optimized for greater efficiency.

Various debloating techniques exist across different programming languages. For Java, several techniques target different aspects of the system, including dependency management, library reduction, and bytecode optimization. While some tools specialize in a single debloating approach, others integrate multiple techniques for broader applicability. The following sections outline the key debloating techniques employed by the Java software debloating tools that were selected for this study.

2.2.1 Source-to-Source (S2S)

In this technique, tools operate directly on the source code to generate a debloated version. This approach provides access to higher-level information, such as documentation and comments, which remain intact at this stage. Additionally, it allows the compiler to detect any errors introduced during the debloating process [11]. This makes it easier for users to verify the changes, as the debloated version closely resembles the original code they were working on. Moreover, users can leverage support from their IDE and the compilation process to identify and address potential issues.

2.2.2 Static Library (SL) Debloating

SL debloating tools typically analyze a project's bytecode to identify and remove unused dependencies, based on the static analysis of the code [11]. This process typically involves scanning import statements, method calls, and object instantiations to construct a usage map of required components. Based on this analysis, the tool optimizes the project's dependency configuration by eliminating unused dependencies, and in some instances promoting essential transitive dependencies—libraries that direct dependencies rely on—to direct dependencies [12, 13, 14]. Some tools further refine dependencies by trimming unused portions of required libraries, reducing the overall application size and improving efficiency [14].

2.2.3 Bytecode-to-Bytecode (B2B)

B2B debloating is a riskier approach where tools operate directly on the bytecode of an application to produce a debloated version. Unlike source code, bytecode lacks

higher-level information, which is stripped away during compilation. As a result, B2B debloating tools must rely on techniques such as coverage-based analysis and pre-existing test suites to verify correctness [6].

While this approach carries a greater risk of producing debloated versions with incorrect functionality [11], it appears to be a more common choice among debloating tools than S2S debloating. As shown in Table 3.1 in Section 3.2, the majority of the tools we identified employ either SL or B2B debloating. This trend may be due to bytecode's and dependency tree's rigid and standardized structure compared to source code [11], which can simplify the development of those types of debloating tools. This structural uniformity likely contributes to the popularity of B2B debloating despite its inherent challenges.

2.3 Measuring Energy Consumption

Measuring energy performance is inherently complex. For example, [5] suggests that debloating a non-bottleneck resource typically reduces energy consumption, primarily by reducing the number of tasks performed, but did not empirically evaluate this statement. They further state that debloating a performance bottleneck may result in improved software efficiency which can result in higher energy peaks during execution, as increased throughput and faster task completion demand more power temporarily. This efficiency could lead to lower overall energy consumption by minimizing unnecessary operations, but it is also theoretically possible that energy efficiency is reduced at energy peaks due to complexities in hardware design [5]. Therefore, to ensure a fair comparison of energy consumption between original and debloated versions of software, it is crucial to analyze them under equivalent workloads. This can be achieved through:

1. **Lifecycle Measurement:** Measuring energy consumption over the complete execution lifecycle for software that terminates naturally.

2. **Task-Based Measurement:** Measuring energy consumption for an equivalent number of tasks for software with execution lifecycles that do not terminate naturally.

These approaches ensure that energy evaluations are consistent and provide meaningful insights into the impact of debloating. Additionally, [15] recommends measuring energy consumption at two levels, depending on how the program operates. If the system terminates naturally within the duration of the measurement, energy consumption is best measured in Joules. Conversely, if the program does not terminate naturally within the measurement period, Watts is the preferred unit. The study also highlights that while these two units can be converted into one another, it is essential to clearly distinguish between the different scenarios in which they are used.

2.3.1 Measurement Techniques

Further complexities, extending beyond the debloating domain, arise from the diverse methods available for measuring software energy consumption, each with its own advantages and limitations. Some of these complexities have been systematically explored in [15], which provides a detailed overview of measurement techniques:

1. **Hardware-Based Approach:** This method measures the energy consumption of the entire device by using physical equipment placed between the power source and the computer. When properly configured, it provides highly reliable energy readings, as highlighted in the guidelines, and is supported by extensive documentation on how to conduct these measurements. However, it remains a complex and potentially costly process. Additionally, it provides coarse-grained measurements that may introduce significant noise, as it captures energy usage from all background applications running alongside the target software. Therefore, many considerations for how this process is carried out are still required.

2. **Software-Based Approach:** This method measures energy consumption at varying levels of granularity, either by combining software-based energy profilers with hardware-based tools in a hybrid approach, or by relying entirely on software-based methods. The levels of granularity range from system-wide consumption (coarse-grained) to application-specific consumption (mid-grained), code block-level measurements (fine-grained), and even down to individual lines of code (very fine-grained).

Purely software-based energy measurement methods are appealing due to their ease of use, low cost, and accessibility. However, a key challenge lies in the lack of a widely adopted methodology. While numerous studies have explored this area and proposed tools for obtaining measurements [16, 17, 18], no consensus has emerged on the best practices for conducting these measurements. Additionally, there is a significant gap in research considering the validity of purely software-based energy measurement methods, as this remains an area of ongoing development.

2.3.2 Energy Measurment Tools

EnergiBridge is a software-based tool for collecting energy measurements [16]. Designed for students, researchers, and practitioners, it supports Linux, Mac, and Windows across Intel, AMD, and ARM chipsets. Notably, it is one of the few tools capable of measuring GPU energy consumption, specifically for Nvidia GPUs. EnergiBridge measures the energy consumption, CPU utilization, and memory utilization of a single process, generating a CSV file with detailed measurement data.

During initial testing, EnergiBridge proved easy to use and reflected increased energy consumption for more resource-intensive tasks. In email correspondence with one of its developers, we were informed that while the relative differences in energy measurements are reliable, the absolute values are not. Therefore, establishing a

baseline for comparison is necessary. Although this limitation exists, it does not significantly impact this study, as the focus is on analyzing relative energy differences between debloating techniques rather than measuring the absolute energy consumption of the executed software.

PowerTOP is a Linux-based energy measurement tool developed by Intel [19]. A key advantage of PowerTOP is its ability to simultaneously monitor all running processes on a system, providing an overview of system-wide power consumption. However, it is more complex to use compared to EnergiBridge and does not natively generate CSV files with individual energy readings. Instead, it reports the average power consumption during the execution period for each process, which could still be useful for a study such as this.

2.4 Bayesian Statistics

To assess the results of our experiment, we performed Bayesian analysis. Unlike frequentist statistical analysis, Bayesian analysis explicitly incorporates prior knowledge into the modeling process through the inclusion of priors. This unique feature distinguishes Bayesian analysis by allowing prior beliefs to directly influence the outcome. Bayesian inference is commonly framed using Bayes' theorem [20], which in its proportional form is expressed as:

$$P(\theta \mid D) \propto P(D \mid \theta) \cdot P(\theta)$$

In this equation, θ represents the parameter or set of parameters that we are trying to estimate or infer, which can vary depending on the context of the analysis. For example, in a regression analysis, θ might represent the coefficients of the model; in a binomial experiment, θ could be the probability of success. Here, the posterior distribution $P(\theta \mid D)$ describes which values of θ are most probable after observing the data D. The likelihood $P(D \mid \theta)$ describes the probability of observing the data given different values of θ, while the prior $P(\theta)$ encodes our initial beliefs about the parameter before seeing the data. This prior can stem from a variety of sources, such as previous knowledge or expert judgment. The inclusion of the prior helps guide the search for plausible parameter values and influences how the model adapts to new data.

For example, consider a parameter that has been normalized; the possible values for this parameter would lie within the range [0,1]. In this case, a prior that reflects this constraint would be logical. By restricting the range of possible values, the prior helps to prevent the posterior distribution from assigning excessive weight to implausible values, especially when limited data is available.

However, Bayesian analysis involves more than just incorporating priors, and Richard McElreath presents a comprehensive workflow for performing Bayesian inference in [20]. This process is structured into a series of steps, each aimed at systematically building and validating statistical models.

The first step is to define the scientific model, which explains the causal relationships between the variables in the study, typically represented using a Directed Acyclic Graph (DAG). This visualization helps in clarifying the structure of the relationships between variables and understanding how different factors influence one another. The next step involves summarizing the data with descriptive statistics to gain an initial understanding of its key characteristics.

Once the scientific model is defined and descriptive statistics are collected, the next phase is to develop a generative model. This model simulates data based on the assumptions made about the data's generation process, providing a framework for testing whether the statistical models can adequately describe the data, given the previous assumptions. At this stage, priors are an essential part of the analysis. As the data is observed, the prior is updated using Bayes' theorem to form a posterior distribution, which reflects our updated beliefs after considering the observed data.

After developing the generative model, statistical models are constructed to measure the target variables of interest, based on the proposed scientific model. These models are then validated using data from the generative model. Model checking is an essential component of the analysis, where prior and posterior predictive checks are used to assess how well the model fits the observed data and how it can be improved. If the target variables can be measured with the proposed models, **and the assumptions are realistic**, then it provides good evidence for the claim that we should be able to measure the target variables from the true data.

Finally, the model is applied to the real-world data, and the results are analysed. Bayesian analysis is inherently iterative, with continuous refinement of the model based on feedback from model checking and new data. This iterative process helps ensure that our assumptions about the data generation process are realistic, and that our models adequately describes the data. As the model is trained on the real data, the results are interpreted and explained in the context of the scientific questions posed, providing insights into the relationships between variables and the validity of the assumptions.

3

Review of the Literature

A review of existing literature on software debloating tools and energy consumption revealed that only one tool explicitly addresses this aspect [21], highlighting the novelty and underexplored nature of this research area. This gap underscores the critical need for further investigation and reinforces the importance of this thesis in addressing a significant void in the field.

To provide context, the following sections will provide a broad overview of prior research on software debloating, notably with the aspect of software energy consumption. This discussion will begin with an overview of the software debloating tool landscape that inspired the research questions for this thesis. It will then explore the debloating techniques and tools available for Java, and related studies in this domain. Finally, other methods that are closely related to software debloating that have been used for reducing energy consumption will be detailed.

3.1 Software Debloating Tools

In the current landscape of software debloating tools, there is a significant gap in the evaluation criteria being considered. Research conducted in [6] highlights that the primary focus of evaluation has been on security and robustness, with 37 and 29 out of 48 tested debloating tools emphasizing these metrics, respectively. In stark contrast, only one tool prioritized energy consumption reduction as its main evaluation criterion. However, this tool is not available as standalone software but instead comprises a collection of debloating techniques and a workflow of tools intended to reduce energy consumption in software [21]. While the proposed techniques proved effective, achieving an average energy savings of 24%, the tool's lack of user-friendliness—due to its reliance on other tools and techniques rather than being a standalone debloating solution—somewhat limits its practical applicability. This limitation in the only tool found to prioritize energy consumption reduction, coupled with the absence of other tools focused on this aspect, highlights a critical gap in the field. It underscores the need to prioritize energy efficiency as an evaluation criterion for debloating tools—an aspect that has been historically overlooked.

3.2 Software Debloating for Java Applications

One of the primary contributors to software bloat in modern applications is the adoption of modular design. While modularity facilitates efficient development and integration of new functionality, it also introduces challenges in selecting only the necessary components from imported modules. This often results in excessive processing overhead and unnecessary data usage [5].

Java, with its long-standing maturity and inherently modular architecture, combined with dependency management tools such as Maven, makes it particularly susceptible to software bloat. A few lines of code can easily introduce multiple dependencies, significantly increasing the application's size, even when most of the added functionality remains unused.

This issue becomes especially pronounced when applications are packaged as fat JARs, where dependencies are bundled directly within the application itself (see Section 2.1.1). However, even when dependencies are not included in the final packaged application, they must still reside on the target system where the application is executed. Consequently, the problem of software bloat remains largely unavoidable when following modern modular design, whether dependencies are embedded or externally managed.

Valero et al. [4] evaluated the effectiveness of coverage-based debloating techniques applied to Java bytecode and library dependencies. Their results show that 68.3% of the bytecode in libraries and 20.3% of the dependencies can be safely removed without impacting functionality. To validate this, the debloated libraries were tested across 988 client projects. Remarkably, 81.5% of these projects successfully compiled and passed their test suites after replacing the original libraries with the debloated versions. This was achieved by leveraging coverage-based tools to analyze which parts of the Java code were necessary for the specific workload. The tools identified unused code, which could then be automatically removed. This demonstrates the substantial potential of coverage-based debloating for reducing unnecessary code, and maintaining compatibility and correctness in Java applications.

3.2.1 Debloating Tools for Java

We conducted an analysis to assess which Java debloating tools might be most relevant for this thesis project. The tools examined included ProGuard [22], JLink [23], DepClean [13], DepTrim [14], JShrink [24], JDBL [4], JReduce [25], BloatLibD [12], Piranha [26], and Slicer4J [27], as detailed in Table 3.1.

Table 3.1: Debloating tool suitability analysis.

Tool Name	Supported Systems	Debloating Technique	Popularity	Availability	Suitability
ProGuard	L/M/W	B2B	High	High	High
JLink	L/M/W	B2B	High	High	High
DepClean	L/M/W	SL	Medium	High	High
DepTrim	L/M/W	SL	Low	High	High
JShrink	W	B2B	Unknown	High	Medium
JDBL	L/M	SL	Low	Medium	Medium
JReduce	L/M	B2B	Low	Low	Medium
BloatLibD	L	SL	Unknown	Low	Medium
Piranha	L/M/W	S2S	High	Low	Low
Slicer4J	L/M/W	S2S	Low	Medium	Low

This analysis involved a comprehensive review of the documentation and relevant research literature for each tool, as well as small-scale testing performed with each tool. The evaluation was conducted based on several critical criteria:

1. **Supported Platforms:** Each tool's compatibility with major operating systems, specifically Linux (L), Mac (M), and Windows (W).

2. **Debloating Technique:** The general approach employed by the tool to reduce software bloat: Bytecode-to-Bytecode (B2B), Static Library (SL), and Source-to-Source (S2S) level debloating (Section 2.2).

3. **Availability:** This criterion assesses the accessibility and comprehensiveness of documentation, along with the overall ease of use and learning curve associated with each tool.

    - Tools with high availability are characterized by well-structured documentation and intuitive workflows. Notable examples include ProGuard, JLink, and JShrink, which provide extensive documentation relative to their complexity. DepClean and DepTrim also exhibit high availability due to their minimal configuration requirements.

    - Tools classified as medium or low availability presented challenges due to insufficient documentation and complex setup procedures.

4. **Popularity:** The popularity of each tool were examined using metrics such as GitHub stars and industry backing.

    - Highly popular tools include ProGuard (3k GitHub stars), Piranha (2.3k stars), and JLink, which is officially maintained by Oracle.

    - Tools with medium popularity include DepClean, which has received 263 GitHub stars, indicating moderate community engagement.

- Tools categorized under low popularity have received fewer than 50 GitHub stars, such as JDBL, JReduce, and Slicer4J.

- Tools for which popularity data was unavailable should be conservatively viewed as having low popularity.

5. **Suitability:** The suitability of each tool was determined based on its ability to meet the core objectives of this study: performing automated debloating with minimal configuration while ensuring that software functionality remains intact.

   - Highly suitable tools identified in this study include ProGuard, JLink, DepClean, and DepTrim. These tools were well-documented in proportion to their complexity and, based on preliminary testing, showed a strong likelihood of preserving software functionality after debloating.

   - Moderately suitable tools were considered viable but proved to require excessive manual configuration or displaying a tendency to break software functionality.

   - Less suitable tools include Slicer4J and Piranha. Slicer4J primarily identifies potential debloating opportunities without executing the actual debloating process, necessitating additional tooling for effective use. Piranha requires developers to manually specify each code section for removal, making it impractical for large-scale debloating and increasing the risk of creating biased results.

3.3 Search-Based and Genetic Optimization Techniques for Energy-Efficient Debloating

In [28], the authors introduced an automated, search-based API optimization approach for Java, where various Java Collections were analyzed to identify more energy-efficient alternatives for use within an application. This method was empirically evaluated in a small-scale experiment, demonstrating up to a 17% reduction in energy consumption. While it is not a part of a traditional debloating tool, this technique presents a strategy that could be integrated into debloating efforts to enhance energy efficiency. Given that debloating tools primarily focus on improving performance by eliminating unnecessary code or modifying code structure, search-based optimizations could be incorporated into such tools to achieve a balance between performance improvements and energy savings. Some debloating tools such as ProGuard [22] makes use of different optimization steps that are aimed at increasing performance by modifying existing code, rather than strictly removing unnecessary code. Instead, this can be viewed as a step to debloat an application through reducing unnecessary processing. The optimization approach employed in [28] would be particularly relevant for S2S-type debloating tools, since it operates on the source code level.

A similar approach, also relevant to S2S debloating, is presented in [29], where genetic improvement techniques are employed to reduce the carbon footprint of web pages through the reduction of energy consumption. In this study, they developed a tool that applies transformations to HTML, CSS, JavaScript, and image resources. The tool aims to minimize data transfer, reduce memory usage, and enhance load times without compromising user experience. Evaluations across ten open-source projects revealed consistent reductions in data transfer volumes and memory consumption, which are strongly correlated to energy consumption, as well as reducing load times.

4

Methods

This study adopts an experimental methodology in the form of an experimental simulation to investigate the effectiveness of debloating tools in reducing energy consumption in Java applications [30]. In this chapter, the methodology used to address the research questions will be detailed.

All code, data, and configuration files necessary to replicate the experiments are available in the replication package published in [31].

Research Questions

- **RQ1:** How do the selected debloating tools affect energy consumption in Java applications?

  This research question examines the impact of selected debloating tools and the debloating techniques they employ on software energy consumption, assessing whether and to what extent they improve energy efficiency. To address this, an experiment is conducted to measure power and energy consumption when debloating tools are applied.

- **RQ2:** What factors influence the energy consumption of a debloated application?

  This question explores how artifact size, CPU utilization, memory usage, and execution time relate to the observed energy consumption of the debloated applications in the experiment.

4.1 Overview

The experiment involves several independent and dependent variables. The independent variables, which are manipulated or controlled during the experiment, include:

- **Debloating Tools (D)**: The set of tools used to remove unnecessary components from the software under test.

- **Systems Under Test (S)**: The software systems being evaluated, which

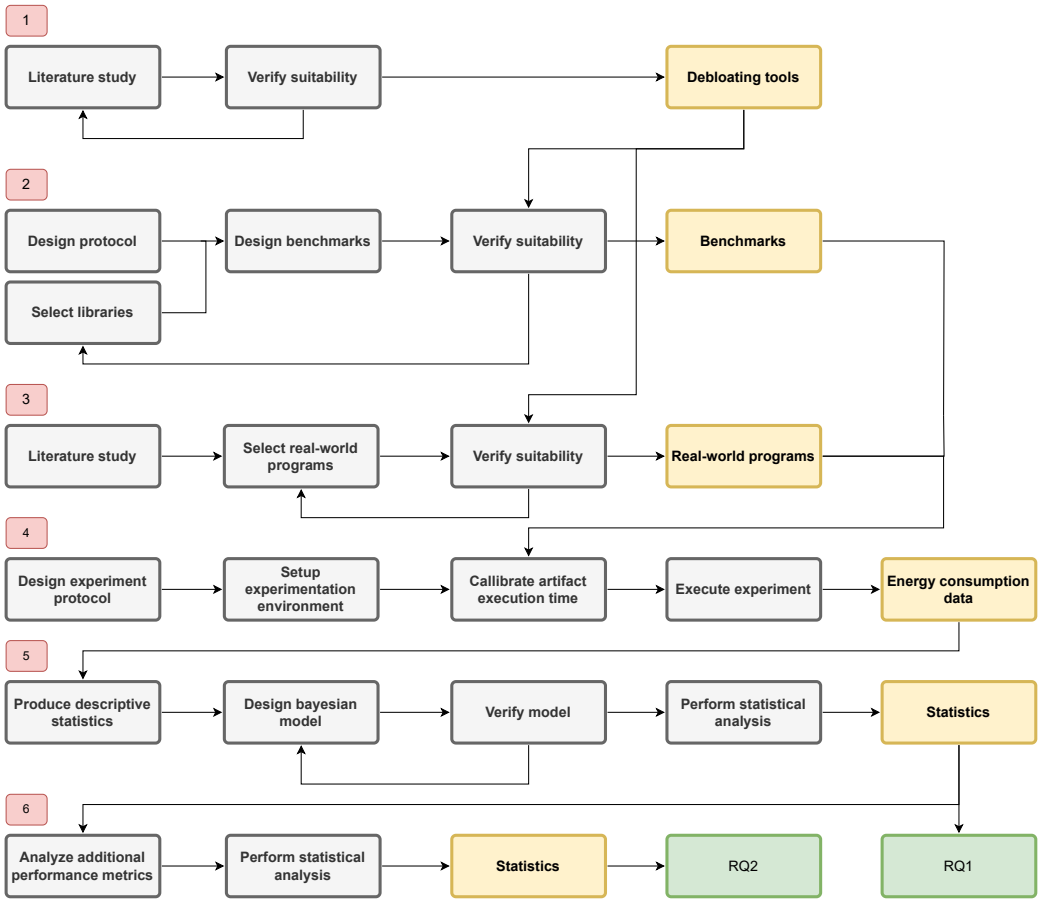


Figure 4.1: Overview of the activities in our research and their connection to each research question. Grey = activity, yellow = artifact, green = research question.

consist of both benchmarks and real-world programs, together with the Java runtime that runs the programs.

- **Experiment Environment (EN)**: The conditions and setup under which the experiment is conducted, such as hardware configurations and operating system settings.

The dependent variables, which are observed and measured outcomes resulting from the manipulation of the independent variables, include:

- **Debloated Software (DS)**: The versions of the SUTs that have undergone the debloating process.
- **Work (W)**: The amount of work that is done during execution by the debloated software.
- **Experiment Duration (T)**: The amount of time over which the experiments are executed, which is affected by the amount of work that is done in the

debloated software itself, as well as the surrounding environment that schedule the work.

- **Power (P)**: The amount of energy that is used per time unit, which is affected by the amount of work that is done by the debloated software, as well as noise introduced by the environment.

- **Energy Consumption (E)**: The amount of energy used during the execution of the debloated software, which is the primary metric being analyzed. This is derived from $\sum_i T_i \times P_i$, where P_i is the power usage measured during time T_i.

The variables in the experiment and the relationship between them will be described in depth in Section 4.6. For an overview of the methodology section, see Figure 4.1.

1. A literature study and initial experimentation was performed to identify Java debloating tools that was most suitable for this experiment (Section 4.2).

2. A protocol for benchmark creation was designed, and libraries to benchmark were selected and verified for suitability (Section 4.3).

3. A literature study was performed to select real-world programs, which were verified for suitability (Section 4.4).

4. The experimentation environment was designed and the experiment was performed to obtain energy measurements by running the debloated and original artifacts (Section 4.5).

5. The energy consumption data was analyzed using Bayesian statistical methods (Section 4.6.1). This addressed **RQ1** and contributed to answering **RQ2**.

6. Additional metrics that may influence energy consumption—and may themselves be influenced by the debloating tools—were analyzed using statistical methods (Section 4.6.2). This analysis addressed **RQ2**.

4.2 Debloating Tool Selection

For this experiment, DepClean, DepTrim, JLink, and ProGuard were selected based on their compatibility and reliability. A key requirement was that all tools could be applied to the same programs without breaking functionality. After extensive testing and reviewing documentation with this goal in mind, these four tools proved to be the most suitable, see Table 4.1.

The selected tools employ different debloating strategies with varying levels of granularity, making them valuable for comparison.

DepClean: At the most basic level, DepClean automatically detects and removes unused dependencies in the dependency tree of Java projects. It analyzes the

Table 4.1: Information on the debloating tools that were selected, including the Java versions they support.

	ProGuard CLI	**JLink**	**DepClean**	**DepTrim**
Version	7.6.1	17.0.9	2.0.5	0.1.2
Debloating focus	Bytecode	Java runtime	Dependencies	Dependencies
Java version	[1.1, 23]	>=9	>=11	>=17
URL	`www.github.com/Guardsquare/proguard`	`https://docs.oracle.com/en/java/javase/11/tools/jlink.html`	`www.github.com/ASSERT-KTH/depclean`	`www.github.com/ASSERT-KTH/deptrim`

project's pom.xml file, statically identifies which types are actually used, and eliminates dependencies that are not referenced in the program.

DepTrim: A step further, DepTrim automatically specializes a project's dependencies in the dependency tree of Java projects. It analyzes the pom.xml file, statically removes unused types within libraries, and stores the trimmed versions in a local folder.

JLink: Moving beyond dependency-level debloating, JLink creates a minimal Java runtime tailored to run a specific program by removing unused modules and unnecessary development tools.

ProGuard: At the most advanced level, Proguard is a tool for removing unused code in Java projects. It analyzes classes and class members marked for preservation, determines which other code is unreachable, and eliminates it. Additionally, it performs name obfuscation, preverification, and various bytecode optimizations. In this project, the CLI version of ProGuard is used for its simplicity and ability to run from the command line.

4.2.1 ProGuard Configuration

For each SUT, a configuration file was created to inform ProGuard about the specific elements to include in the output and the debloating steps to apply. This process was structured into a systematic workflow that proved effective for the purposes of this thesis. The workflow consisted of the following steps:

1. **Define input and output JARs:** Specify the input JAR file to be processed and the name of the output JAR file.

2. **Include required libraries:** Add any external libraries or modules required by the program using the `-libraryjars` option.

3. **Configure obfuscation settings:** Disable code obfuscation with the `-dontobfuscate` option.

4. **Preserve critical classes and members:** Identify and mark essential classes and class members to keep as entry points to the code with the `-keep` option.

5. **Handle missing essentials after execution:** If runtime errors reveal missing classes or members, add them with the `-keep` option to ensure they are preserved in the output JAR.

6. **Verify correctness of program:** Ensure that the output JAR from ProGuard executes correctly.

7. **Apply optimization settings:** Enable aggressive interface merging and code optimization, and set the number of optimization passes to the maximum number. Thereafter, test the output JAR to ensure it runs correctly. If issues arise, revise previous steps and repeat the process.

The process of configuring ProGuard required manual adjustments, particularly because ProGuard's analysis can sometimes fail due to dynamic code features, such as Java reflection. These failures often necessitated the addition of missing classes or class members to the keep-rules. However, for most SUTs, the main class was typically sufficient to serve as the entry point. ProGuard then used this class to determine which other classes and members were required, resulting in minimal adjustments.

For instance, in the case of the Zxing benchmark, the main class was the benchmark runner class. After running the output JAR, error logs indicated that the classes `ResultMetaDataType` and `IIOServiceProvider` could not be recognized. These classes were subsequently added to the configuration file with additional keep rules. The configuration file for Zxing is shown in Listing 4.1. The configuration was performed to enable as many of the optimization strategies as possible for all of the SUTs without breaking the functionality.

ProGuard applies a series of optimization passes that are implemented by ProGuard, executing all enabled optimizations in each iteration. This process continues until either (i) the code is considered fully optimized by the tool, or (ii) the maximum number of passes specified in the configuration is reached. ProGuard fully optimized the benchmarks in 9 out of 10 cases, with one SUT only able to complete a single optimization pass without breaking the functionality of the code. For the real-world programs, ProGuard fully optimized the programs in 4 out of 5 cases, where no optimzation was possible for one of the SUTs. This real-world program was later regarded as an outlier and excluded from the analysis.

Listing 4.1: ProGuard Configuration for Zxing Benchmark

```
-injars target/artifact.jar
-outjars artifacts/artifact_proguard.jar

# Identified with jdeps
-libraryjars <java.home>/jmods/java.base.jmod
-libraryjars <java.home>/jmods/java.desktop.jmod
-libraryjars <java.home>/jmods/java.xml.jmod

-dontobfuscate # Obfuscation Disabled
#-printmapping artifacts/mapping.txt # (Used if obfuscation was enabled)

# Preserve critical classes and members
-keep class org.example.BenchmarkRunner {*;}

# Keep required classes for ZXing (QR Code processing)
-keep public class com.google.zxing.ResultMetadataType {*;}

# Keep ImageIO-related classes to prevent missing service providers
-keep class * extends javax.imageio.spi.IIOServiceProvider {*;}

# ProGuard optimization options
-verbose
-mergeinterfacesaggressively # Aggressively merges interfaces (use with
    caution)
-optimizeaggressively # Optimizes aggressively (use with caution)
-optimizationpasses 50 # Number of optimization passes
```

4.2.2 Other Debloating Tool Configurations

The configuration of the other debloating tools was relatively straightforward compared to ProGuard.

DepClean: It was installed via Maven and executed with the following command:

```
mvn se.kth.castor:depclean-maven-plugin:2.0.6:depclean
    -DcreatePomDebloated=true -DignoreDependencies="regex expression"
```

This command analyzed the project's `pom.xml` file, identified unused dependencies, and produced a debloated version of the configuration file. DepClean had the potiential of marking used dependencies as unused in some cases, which was dealt with by passing misclassified dependencies to the *-DignoreDependencies* flag.

DepTrim: It was included as a plugin in the projects' Maven build configurations, which produces a specialized `pom` file with minimal dependencies and the trimmed libraries. Similarly to DepClean, if a library was misclassified then the dependencies could be marked as ignored.

JLink: A tool included in the Java Development Kit, along with JDeps, that helps identify the necessary modules for creating a custom runtime image. The `jdeps` command generates a list of Java modules that the program depends on. These modules are then added to the *–add-modules* argument of the `jlink` command. The `jlink` command creates a compressed Java runtime image, removes debug information, excludes header files for C/C++ integration, skips generating man pages for UNIX systems, includes only the identified modules, and outputs the minimal Java Runtime Environment (JRE) with the required modules into the `jlink-runtime` folder. The two commands are as follows:

```
$ jdeps --ignore-missing-deps --list-deps path/to/program.jar
$ jlink --compress=2 --strip-debug --no-header-files --no-man-pages
    --add-modules {module list} --output jlink-runtime
```

4.3 Benchmark Creation

In this project, a benchmark is defined as a small program designed to test specific functionalities of a Java library—a collection of reusable code that enables developers to integrate prebuilt features into their applications. By constructing benchmarks, the aim was to overcome three challenges inherent in evaluating debloating techniques and measuring their impact on energy consumption.

The first challenge was the scarcity of suitable executable programs for analysis. Prior research on Java software debloating has predominantly focused on static analysis, measuring code reduction and verifying correctness in debloated libraries and frameworks [24] [14]—with only a few exceptions where executable applications are considered in addition to libraries [13]. This study depended on Maven-built, executable programs running on a consistent Java version to control for the potential differences in energy consumption that could occur between different Java versions. By designing benchmarks, we alleviated this scarcity, ensuring that sufficient data was available for robust statistical analysis, which was essential for addressing **RQ1** and **RQ2**.

The second challenge involved limiting the non-determinism of energy measurements. Each experimental trial must be conducted with a consistent workload to allow for realistic comparisons between debloating techniques. The controlled nature of the benchmarks enabled direct management and verification of the workload. This precision was critical for obtaining reproducible energy measurements and contributed significantly to the evaluation of debloating strategies, thereby informing both **RQ1** and **RQ2**.

The third challenge was to ensure the correctness of the debloated code, which was a task that required a good understanding of the underlying program code. This task becomes even more complex when using debloating tools that operate during post-compilation, like JLink and the CLI version of ProGuard, which bypass conventional test suites. By limiting the benchmarks to a small set of library function-

alities, a comprehensive understanding of the resulting programs could be achieved, and the functionality of the resulting programs could be fully verified before and after debloating. The reduced complexity and improved understanding of the code also facilitated improved traceability of code changes, while helping with debugging during the debloating process.

Although previous research has introduced benchmarks to verify the functionality of debloating tools [27], these benchmarks have been too simplistic for our use case. They were primarily designed to assess the correctness of a debloating process by evaluating the removal and retention of specific features, rather than to measure execution performance, which was the focus of our study. The idea of creating driver programs for debloated libraries has also been considered in previous research but was ultimately rejected since they would have to create hundreds of these benchmarks [4]. However, in this study, this approach was both feasible and necessary, as it resulted in more SUTs to experiment on, and because the smaller number of SUTs could be offset by increasing other controlled variables to gather more data, such as the number of trials and execution time.

4.3.1 Library Selection

For this experiment, we focused on libraries hosted on the Maven Central repository that were imported in the Maven configuration file for the project. The emphasis on Maven-based projects is essential, as both DepClean and DepTrim require Maven configurations to perform debloating and rely on the libraries being accessible through Maven Central to perform their analysis. Additionally, Maven offers the advantage of automatically downloading dependencies and building the code.

Table 4.2: Libraries selected for evaluating the effect of the debloating tools (**RQ1** & **RQ2**), along with their targeted modules in the case of multi-modular libraries.

Library		Category	Commit ID	GitHub Stars
Zxing	[**core, javase**]	Image Processing	2dfb205	33004
JavaParser	[**javaparser-symbol-solver-core**]	Java Parser	42e17b2	5791
Java-Faker		Data Generation	a8b8ff0	4829
OpenPDF		PDF Editor	4821449	3703
TableSaw		Data Manipulation	05823f6	3611
JTS	[**jts-core**]	Geospatial Processing	6e95fe8	2065
FlyingSaucer	[**core**]	HTML Rendering	0564cae	2047
Closure-templates	[**soy**]	Templating Engine	2caffb9	658
Shacl		RDF Data Validation	89205df	225
JGit		Git Implementation	9d0e4cd	195

The libraries selected for this study are listed in Table 4.2. The selection process prioritized compatibility with Java 17, as the debloating tools—particularly DepTrim—required this version to function. Other factors included the libraries' suitability for energy measurement using Energibridge and their simplicity, which

facilitated faster development and more effective analysis. Popularity, measured by GitHub stars, was also considered to ensure the relevance of the libraries. The selection process consisted of the following steps:

1. **Initial retrieval:** A list of 1000 `pom.xml` files was gathered using the GitHub Search API with the query:

   ```
   filename:pom.xml AND "java.version>17<" AND
   "<distributionManagement>" AND NOT "spring"
   ```

   This query targeted Java 17 projects available on Maven Central, ensuring they were libraries compatible with Deptrim, while excluding Spring Boot projects, which often require launching a web browser.

2. **Mapping to repositories:** Each `pom.xml` file was mapped to its parent repository on GitHub. Since some projects contained multiple `pom.xml` files, they mapped to the same repository, resulting in duplicates. These duplicates were removed, reducing the initial set from 1,000 to 773 unique repositories. This list was compiled into a spreadsheet, including details such as each repository's primary programming language and GitHub star count.

3. **Language filtering:** Only repositories where Java was the primary language were retained, further reducing the list to 375 entries.

4. **Popularity ranking:** The final list was sorted by GitHub stars, selecting the most popular libraries that met our selection criteria.

The Github API restricts the number of matches per search pattern to 1000 and does not allow filtering the results by the number of GitHub stars of the parent projects when searching for content in files. Therefore, an exhaustive search with the query could not be performed, potentially omitting relevant libraries. Additional entries that belonged to this population—identified through similar search patterns during the research phase, as well as review of past literature on debloating tools—were manually entered into the table. In total, five suitable libraries were identified using the above search pattern, two additional benchmarks were found using other search patterns following the same method, and three benchmarks were identified through a review of literature, which helped the process in obtaining more popular libraries.

Some of the most popular libraries identified through this algorithm were found to be unsuitable for our study, as the search functionality could not enforce all selection criteria, particularly those related to program behavior. These libraries were skipped in favor of the next most popular option on the list. The search query was unable to filter out libraries that relied on overly complex functionality, such as AI models, or those that spawned new processes—though filtering out Spring Boot projects helped mitigate the latter issue. Ensuring that the selected libraries did not create new processes was crucial, as the energy measurement tool—Energibridge—could only monitor the initial process created at the start of execution.

Table 4.3: Benchmarks designed to use some functionality from the selected libraries to evaluate the effect of debloating tools on energy consumption (RQ1 & RQ2). NOC: Number of classes.

ID	Library	NOC	Used Functionality
B1	**Zxing**	763	Encode string to QR code Decode QR code to string
B2	**JavaParser**	3772	Parse and modify Java code programmatically
B3	**Java-Faker**	599	Generate realistic pseudorandom mock data, such as user profiles, financial details, and contact information
B4	**OpenPDF**	973	Add new page to PDF file Add table to PDF file Add image to PDF file Add row to PDF file
B5	**TableSaw**	19981	Generate table containing numeric values Perform statistical summarization of the table
B6	**JTS**	735	Calculate the points of intersection of generated lines
B7	**FlyingSaucer**	600	Parse XHTML string into a document Render document as an image
B8	**Closure-templates**	8550	Render a dynamic HTML page from structured data
B9	**Shacl**	8673	Verify constraints on an RDF data file
B10	**JGit**	1923	Switch git branch Create git brach Add and commit file to branch with a commit message

4.3.2 Benchmark Design

Table 4.3 presents the benchmarks created for this project, detailing the used functionality that served as the criterion for retaining code during the debloating process. The debloating tools could therefore remove all code that was unrelated to this functionality from the libraries. Each benchmark was tailored to the library it used, focusing exclusively on invoking some of its core functionality and excluding any auxiliary features. If a library supported multiple use cases, a set of commonly used methods was selected.

The benchmarks were designed to ensure deterministic execution and consistent workload in all trials. To achieve deterministic execution, a set of functions from the selected libraries was invoked in a loop for a number of iterations that could be set for each SUT. Consistent workload was achieved by setting the number of iterations so that the original version of the benchmarks was executed for approximately 1 minute

Table 4.4: Impact of debloating tools on file size reduction in benchmark programs, presented in megabytes and percentage reduction from the original size. For DepClean, DepTrim, and ProGuard, the baseline for measurement is the original JAR file size. For JLink, the baseline is the original Java 17.0.9 JDK size.

	DepClean	DepTrim	ProGuard	JLink
Max	+0.002 (+0.0%)	0.000 (+0.0%)	-0.180 (-14.7%)	-252.401 (-83.2%)
Min	-1.333 (-54.1%)	-1.708 (-56.4%)	-30.261 (-93.6%)	-264.301 (-87.1%)
Mean	-0.411 (-10.0%)	-0.623 (-20.1%)	-5.816 (-53.8%)	-255.237 (-84.1%)
Median	-0.362 (-4.6%)	-0.434 (-8.1%)	-2.348 (-57.8%)	-254.209 (-83.8%)
SD	0.453 (16.5%)	0.635 (22.5%)	9.093 (25.3%)	3.360 (1.1%)
Total	-4.109	-6.232	-58.161	-2552.367

for each SUT. This ensured that any substantial changes in execution time were caused by the debloating tools, rather than differences in the workload. Additionally, to minimize variability caused by thread scheduling or environmental interruptions, the benchmarks were designed to run in a single thread.

To reduce functional similarity across benchmarks, a strategy was employed to avoid using core Java library features within the main execution loops. This approach aimed to prevent excessive behavioral overlap between SUTs, which could obscure meaningful differences in energy consumption. Only essential functionality—such as file I/O and basic data types—was retained from the core libraries, ensuring that the necessary features from the selected libraries remained intact.

In addition to the main class containing the program code, a Maven `pom.xml` file was created to manage dependencies and build the program. The Maven Shade plugin was used to package all required dependencies along with the main program into a fat JAR file.

Table 4.4 presents the effect of the debloating tools on the reduction of software package sizes for the benchmark programs. The outputs of DepClean, DepTrim, and JLink are deterministic, while ProGuard's output is generally considered deterministic, though certain configurations may introduce non-determinism due to obfuscation and optimization processes, as no definitive sources were found to confirm this for all cases. It is important to view these statistics in the context that JLink debloats a different part of the software package, namely all the components required for its execution. One value that stands out is the maximum difference of DepClean, where it mistakenly added an unnecessary dependency to the project. Another interesting observation is that in one instance, ProGuard managed to reduce the size of an application by 93.6%, which is a significant reduction.

4.4 Real-world Program Selection

In addition to the benchmarks, five open-source real-world programs were selected, see Table 4.5. The selection process was guided by previous research in the field

Table 4.5: Real-world programs selected to evaluate the effect of the debloating tools (RQ1 & RQ2), along with their targeted modules in the case of multi-modular libraries and number of classes (NOC).

Program	NOC	Category	Version Number	Commit ID	GitHub Stars
CoreNLP	13500	ML	4.5.8	5970639	9791
CheckStyle	10738	Linter	10.21.3	06c1f64	8475
Error-Prone [**core**]	5959	Linter	2.36.0	ab522c7	6937
Apache Tika [**app**]	30400	Text Processing	3.1.0	2561927	2825
ICIJ Extract [**cli**]	40982	Text Processing	2.0.0	2018ffd	242

of debloating tools [13], as well as real-world programs that were found during the search for libraries to use for benchmarks. Only projects that generated an executable JAR file were targeted, since this was necessary to perform the energy measurements. Similarly to the selection of libraries, it was also necessary that the real-world programs did not spawn new processes to perform tasks, since these could not be measured by EnergiBridge.

Applying debloating tools to real-world programs allowed us to work with more complex projects, yielding more realistic and diverse data, which helped us in addressing both **RQ1** and **RQ2**, in addition to increasing the generalizability of the results.

The primary distinction between the debloating process for benchmarks and real-world applications lies in the level of configuration required to ensure proper functionality. Real-world programs necessitated more extensive tool configuration to prevent the removal of essential components. For DepClean and DepTrim, this involved explicitly specifying certain libraries to be ignored due to misclassification. In the case of ProGuard, broader keep rules had to be applied, reducing the granularity of the debloating process.

The workload for each real-world program was chosen based on the most common or representative functionality that could be executed within a single process. This functionality was invoked by passing appropriate command-line arguments to each program. A high-level overview of the commands used, along with the nature of the inputs provided, is presented in Table 4.6.

In Table 4.7, the effect of the debloating tools on the reduction of software package sizes is presented for the real-world programs. Again, it can be observed that Depclean mistakenly added a dependency for at least one program, causing the program-size to slightly increase.

Table 4.6: Overview of real-world programs used in the evaluation, their main functionality, and the input type provided.

Program	Usage Summary	Input Type
Checkstyle	Executed via JAR with Sun checks configuration to analyze code style.	Java source files
CoreNLP	Run with StanfordCoreNLP for tokenization, POS tagging, and dependency parsing.	Plain text document
Error-Prone	Used as a compiler plugin during Java compilation, configured with JDK module exports and memory settings.	Java source files
ICIJ Extract	Executed with metadata and OCR extraction disabled to extract plain text.	PDF files
Apache Tika	Used to extract plain text with the `-text-all` option.	PDF files

Table 4.7: Impact of debloating tools on file size reduction in real-world programs, presented in megabytes and percentage reduction from the original size. For DepClean, DepTrim, and ProGuard, the baseline for measurement is the original JAR file size. For JLink, the baseline is the original Java 17.0.9 JDK size.

	DepClean	DepTrim	ProGuard	JLink
Max	+0.018 (+0.1%)	-1.868 (-11.1%)	-0.145 (-1.3%)	-223.366 (-73.6%)
Min	-22.812 (-29.5%)	-10.153 (-25.9%)	-43.254 (-76.7%)	-253.425 (-83.5%)
Mean	-7.095 (-10.8%)	-6.040 (-17.5%)	-20.256 (-47.2%)	-246.661 (-81.3%)
Median	-0.496 (-1.9%)	-6.869 (-16.9%)	-20.321 (-52.9%)	-251.913 (-83.0%)
SD	10.215 (14.2%)	3.615 (5.5%)	16.746 (27.8%)	13.046 (4.3%)
Total	-35.474	-30.198	-101.282	-1233.305

4.5 Experimental Environment

The experiment was conducted on a PC running a minimal Linux-based operating system. A lightweight version of Xubuntu was installed, along with only the essential tools required for the experiment: Java 17.0.9, Maven 3.9.9, Energibridge, and Git. To ensure consistent and reproducible results, the system's hardware and software configuration remained unchanged across all trials. CPU turbo functionality was disabled, and the processor clock was fixed to maintain a stable performance profile throughout the experiments.

Using a lightweight Linux-based operating system minimized background processes, reducing the risk of external interference in the measurements. This was critical, as unaccounted background activity could introduce noise, potentially affecting energy consumption readings.

Java 17.0.9 was selected as the execution environment to ensure consistency across experiments, as different Java versions may exhibit variations in energy consumption. Additionally, the latest available version of Maven at the time of experimentation was used.

The hardware specifications of the system used for the experiment are as follows:

- **Processor:** AMD Ryzen 7 2700X, 8 cores, 3.7 GHz
- **Graphics Card:** NVIDIA GeForce RTX 3060 Ti
- **Storage:** Samsung V-NAND SSD 990 PRO, 2TB
- **Motherboard:** ASUS ROG STRIX B550-F GAMING
- **Memory:** 2x8GB DDR4, CL11, 2400 MHz
- **Power Supply Unit:** Corsair RM850X, 850W

4.5.1 Energy Measurement and Data Collection

To ensure that the debloated programs retained their original functionality, each artifact's output was verified against the original version. For the benchmark programs developed in this study, this verification process confirmed that all used functionalities remained intact after debloating. In the case of real-world programs, correctness was effectively confirmed for the functionality that produced the output. Identical results indicated that the debloated program was functionally equivalent for the executed workload. This verification step increased the reliability of the measured energy consumption data.

Each artifact executed the same tasks the same number of times for each SUT. Both benchmark and real-world program artifacts were run under identical conditions, ensuring consistency in task counts and execution environments.

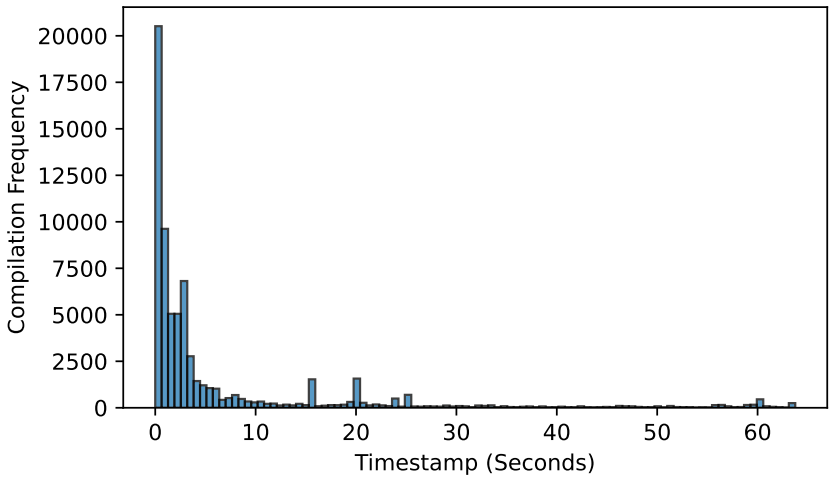


Figure 4.2: Distribution of Java JIT compilation frequency from running all SUTs, produced using the `-XX:+PrintCompilation` flag.

To ensure reliable measurements, execution was structured to account for several key factors. JVM warm-up effects were considered by monitoring the Just-In-Time (JIT)

compilation activity. This allowed identification of the warm-up phase, which, for most programs, stabilized within the first five seconds of execution, see Figure 4.2. While warm-up data was also relevant, the primary focus was on steady-state execution to minimize transient performance effects.

To obtain reliable energy measurements, the methodology followed best practices outlined in [32]. The following procedures were implemented:

- Background processes and unnecessary peripherals were minimized to reduce measurement noise.
- A consistent execution environment was maintained across all measurements.
- Each trial was repeated 30 times to reduce variance and detect anomalies, which was also in accordance with statistical recommendations and guidance from an EnergiBridge developer.
- A one-minute idle period was run where the environmental noise was recorded before each measurement to allow the system to stabilize and to get a profile on the noise that was likely to be present during the experiment.
- The experimental workload was executed for five minutes before data collection to let the system warm up.
- The order of trials was randomized to prevent systematic biases from environmental fluctuations.
- Trial execution was automated to minimize human-induced inconsistencies.
- Ambient temperature was kept stable to prevent external thermal effects from influencing energy consumption.

Energy consumption was measured using EnergiBridge [16], with each artifact tested 30 times per SUT. Since 15 SUTs were evaluated, each with five artifacts, a total of 2,250 trials were conducted—450 for each debloating tool—ensuring statistical robustness. Each trial included a one-minute resting period followed by a one-minute execution phase, leading to a total experiment duration of approximately 75 hours.

The execution time was carefully chosen to balance flexibility and significance. Extending execution time increases the total duration of the experiment, making retries and modifications more costly. Therefore, trial durations had to be long enough for meaningful data collection while allowing adjustments if needed. Given the total data volume, the time constraints of the study, and JIT compilation characteristics, 60 seconds per trial was deemed an appropriate balance.

Following recommendations from the EnergiBridge development team, the tool was configured to collect measurements at one-second intervals. This interval aligns with

the default sampling rate of PowerTop, ensuring compatibility for future comparisons. A trade-off exists between measurement frequency and reliability: shorter intervals capture transient power fluctuations more effectively but may introduce additional variability, while longer intervals provide more stable readings but risk missing power consumption spikes. The selected interval of one second balances these considerations.

The collected data was stored in CSV files, capturing CPU energy consumption, CPU utilization, and memory utilization. For data processing, functions provided by the EnergiBridge team were used to convert raw energy readings into power consumption values. The processed data was then aggregated into a single CSV dataset, where cumulative energy consumption was calculated as the product of the time between measurements and the corresponding power consumption at each timestamp. This final dataset was analyzed using the programming language R, employing Bayesian statistical methods.

4.6 Statistical Analysis and Comparisons

To answer **RQ1**, the aim was to determine whether the use of a debloating tool leads to a measurable difference in energy consumption and, if so, to quantify its significance for each tool. To address **RQ2**, the metrics execution time, CPU usage, and memory usage were analyzed further to attempt to understand to what extent they affect energy consumption, as well as how the debloating tools affected these metrics.

4.6.1 Impact of Debloating Tools on Energy Consumption (RQ1)

The goal of the experiment to answer **RQ1** was to measure the effect of the debloating tool $\boldsymbol{D}$ on the power consumption $\boldsymbol{P}$ and energy consumption $\boldsymbol{E}$, seen in the DAG of Figure 4.3. This is the scientific model that was used for the experiment, detailing all the assumptions regarding the variables in the experiment.

The environment $\boldsymbol{EN}$ is a confounding variable with an unobserved effect on the execution time $\boldsymbol{T}$ and $\boldsymbol{P}$. While the environment is partly controlled, and the effect of $\boldsymbol{EN}$ has been minimized by reducing the number of processes run in the background, it is impossible to completely remove the noise it can induce during this experiment. It is also difficult to measure its effect during the execution of the artifacts in the experiment without causing new confounding variables to emerge.

During the resting periods of the experiment, $\boldsymbol{EN}$ could be measured by monitoring an idle process using EnergiBridge running on the same environment that was used for the experiment. The analysis revealed that the majority of the noise was concentrated around an amplitude of approximately 6W over the duration of the experiment, see Figure 4.4. Although not visible in the figure, sporadic fluctuations in the range of 5.76–33.16W were observed. These lesser fluctuations make up the

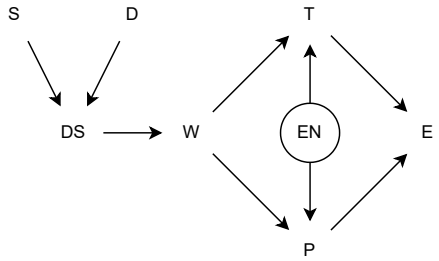


Figure 4.3: DAG of experiment variables with an unobserved variable ***EN***, as denoted by the circle.

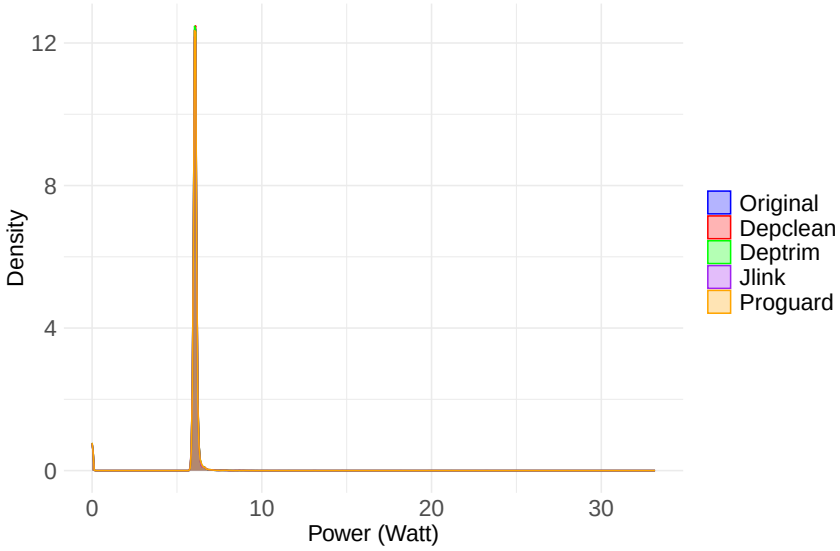


Figure 4.4: Density plot of noise in the experimental environment, recorded over 60 seconds between each trial and labelled after the artifact type that was last run.

bulk of the uncertainty in this experiment.

First, a linear regression model was fit to the normalized data to capture the general trend in power consumption for each debloating technique across all SUTs. This model allowed for easy visualization of the mean and variance of the effect of the debloating tool on the power consumption.

Second, since variations in execution time between trials could not be modelled by a linear model, this meant that if variations in execution time were significant, the linear model would not accurately describe the differences in total energy consumption between debloating tools. To address this, a hierarchical regression model was fit to the total energy consumption data, where the data was normalized for each SUT to make it easier to fit the model.

Since there are no backdoor paths to the target variable ***D***, no adjustment sets were

needed in the hierarchical model. This allowed the effect of $\boldsymbol{D}$ on $\boldsymbol{E}$ to be measured directly using Equation (4.1), which states that the mean of the energy consumption is determined by the debloating tool in use.

$$\mathbb{E}(E_i \mid D_i) = D_i \tag{4.1}$$

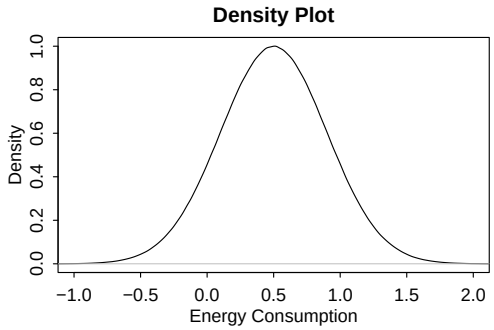
Equation (4.2) extends Equation (4.1) by defining a hierarchical Bayesian model, where energy consumption is normally distributed with mean μ and standard deviation σ. Given that the data is normalized to $[0, 1]$, the prior on the tool effect reflects this scale, with a mean of 0.5 and a standard deviation of 0.4, see Figure 4.5 (a). The assumption of normality arises because the amount of work performed in each trial is largely deterministic, with various independent factors such as background processes, hardware fluctuations, thread scheduling, and temperature variations that introduce noise that together likely approximates a Gaussian distribution, since they are many in number, each with a small added variation to the environment [20].

$$\begin{aligned} E_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= D_i \\ D_i &\sim \text{Normal}(0.5, 0.4) \\ \sigma &\sim \text{Exponential}(1) \end{aligned} \tag{4.2}$$

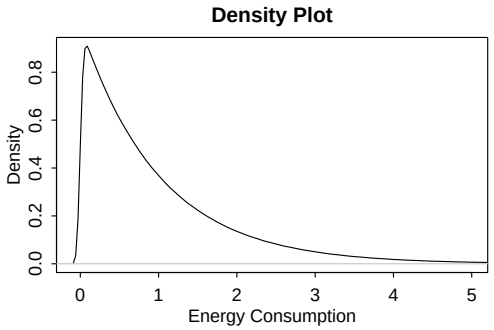
To account for uncertainty in energy variability, the standard deviation σ follows an exponential distribution with rate 1, favouring smaller values and reinforcing the expectation that energy fluctuations remain limited to values in the range of [0,1] and should be much closer to 0 than 1 since with a value of 1 it would mean that we would sometimes see three times the possible amount of energy consumption both in positive and negative directions, see Figure 4.5 (b). Since the environment was the same for each debloating tool, leading to approximately the same variance, the same estimated σ parameter was used for each estimated energy distribution.

To strengthen the credibility of these models, the models were verified using simulated data, based on the assumption that the data can be modelled as approximately linear and with approximately Gaussian noise. The simulated dataset consisted of five simulated SUTs and five artifacts, with energy consumption modelled as a linear function of time. The means of the power consumption for the SUTs ranged from 14W to 34W, while the effect of the artifacts varied as follows: three artifacts increased power consumption by 10%, 4% and 1%, one remained neutral (baseline), and one artifact reduced power consumption by 2%, respectively. The purpose of this simulated experiment was twofold: (1) to verify that the true differences in energy consumption can be detected given our assumptions, and (2) to help estimate the number of samples required to reliably identify these differences.

To introduce realistic variation in the simulated data, two sources of noise were incorporated: temporal noise and power measurement noise. Temporal variation



(a) Mean of debloating tool effect.



(b) Standard deviation of debloating tool effect.

Figure 4.5: Plotted densities of the prior distributions for the estimands of the model in Equation (4.2).

allowed execution time to fluctuate within a normal distribution centred at 60s with a standard deviation of 1s for each individual sample. This reflects the expectation that execution times may vary slightly due to environmental fluctuations that may have caused minor variability between trials for the same setup. Power measurement noise was modelled as a normally distributed perturbation with mean zero and a standard deviation of 20% of the true power consumption for the SUT-technique pair, based on the assumption that moderate fluctuations in power measurements can occur during execution due to variations in the operation of the measurement tool, and that it is amplified by the amplitude of the power that is measured. Additional noise was added that was modelled as a Student-T distribution after a noise profile that was measured on the experiment environment, which was very similar to the noise profile shown in Figure 4.4.

For the simulated data, the model in Equation (4.2) proved to be within acceptable range of the true answers despite the noise, with the increase and decrease in mean power being very well separated between all of the artifacts, see Table 4.8. The noise did not seem to pose any issues for the model to accurately find differences between the techniques with very high confidence, however, due to the offsetting effect by the noise profile which was added to the simulated data, it diminished the true differences between the artifacts by roughly 30%. When the mean of the noise was subtracted from the power in the data and the model was retrained, this issue was corrected. That is why the energy consumption will be shifted by a constant equal to the mean of the noise profile in the results to account for this offset.

The experiment was run and the data was collected and processed. The processing steps included outlier detection for the hierarchical model, where Z-scores were used to identify 11 out of the 2240 samples that deviated more than three standard deviations from the mean of the final energy consumption, which were then discarded by following the recommendations from the guide in [32], which states that if less than six samples from a unique experiment configuration are outliers then they can safely be discarded, since this ensures that 25 or more trials are available for every

Table 4.8: Statistical summary of simulation with statistics relative to D[2] as the baseline, where the values are the difference in percentage from the estimated energy consumption of D[2], after correcting for the offset imposed by background noise.

	Mean	**SD**	**5.5%**	**94.5%**	**True Value**
$D[1]$	-2.026	0.242	-2.415	-1.635	-2
$D[2]$	0.000	0.232	-0.378	0.377	0
$D[3]$	0.907	0.233	0.544	1.268	1
$D[4]$	3.930	0.236	3.555	4.299	4
$D[5]$	10.127	0.240	9.748	10.520	10

combination of SUT and debloating tool.

After verifying the models and processing the data, the models were then fit to the real data and the results were used to infer any differences in energy consumption between each debloating tool the same way it was done for the simulated data.

4.6.2 Analysis of Performance Metrics (RQ2)

To address **RQ2**, the additional metrics—execution time, CPU utilization, and memory utilization—were analyzed using data from the experiment.

First, the impact of each debloating tool on these metrics was assessed in detail to better understand how variations in these factors may have contributed to changes in energy consumption. This analysis used the same hierarchical model framework applied to energy consumption, but retrained to focus on each of the individual metrics. The way this was done was by exchanging the training data E_i of the model to the data for CPU utilization, execution time, and memory utilization, and then run the model in the same manner.

As with energy consumption, each metric was normalized per SUT to express values as a percentage of that SUT's maximum observed value. This normalization allowed for consistent comparisons across different programs and debloating configurations.

Next, the relationships between these metrics and both power and energy consumption were examined using correlation matrices based on Pearson correlation coefficients. This method assumes that the data is normally distributed and that the relationships are linear. Separate analyzes were conducted for both benchmark applications and real-world programs, considering both final and average values per trial, along with instantaneous values recorded throughout execution.

5

Results

This chapter presents the experimental results, starting with descriptive statistics in Section 5.1. Section 5.2 covers the outcomes of linear and hierarchical models on power and energy consumption, addressing **RQ1**. Section 5.3 examines performance metrics that contextualize these results, addressing **RQ2**. The chapter balances descriptive analysis with model-based evaluation of debloating's impact. Two outlier SUTs—*error-prone* and *java-faker*—are discussed separately in Section 5.4.

5.1 Descriptive Statistics

Figures 5.1 and 5.2 show the mean and median final energy consumption, respectively, for each debloating tool from the maximum observed value of each SUT. Real-world programs are shown left of the separator; benchmarks are on the right.

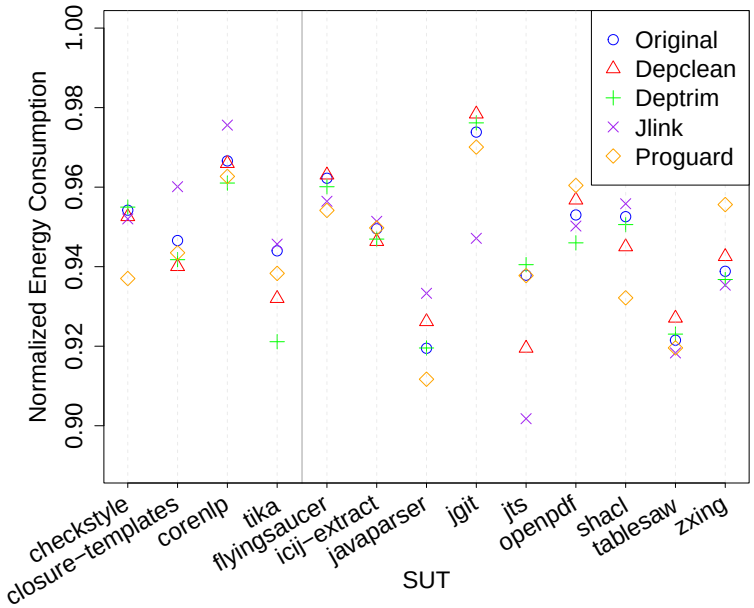


Figure 5.1: Normalized mean final energy consumption per debloating tool/SUT.

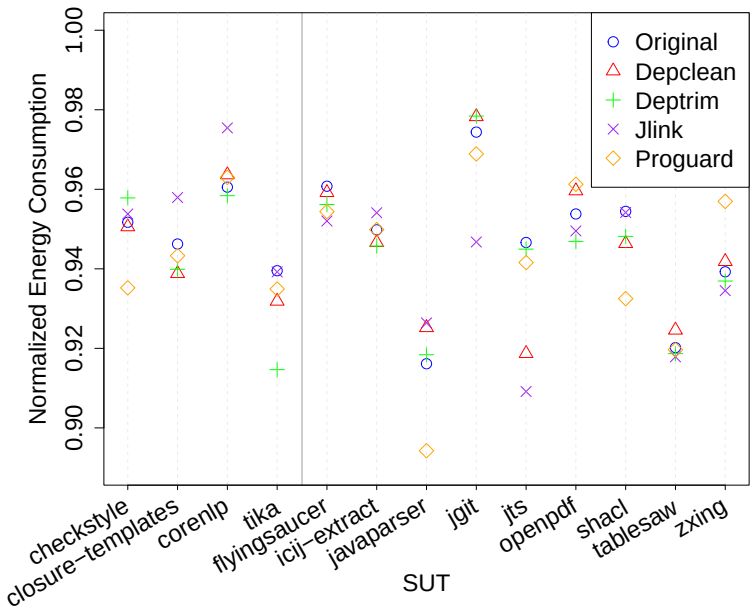
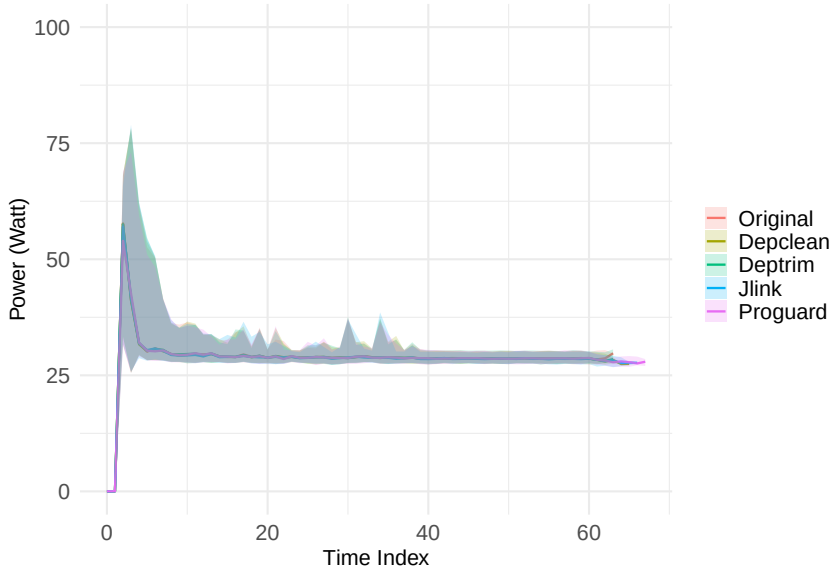


Figure 5.2: Normalized median final energy consumption per debloating tool/SUT.

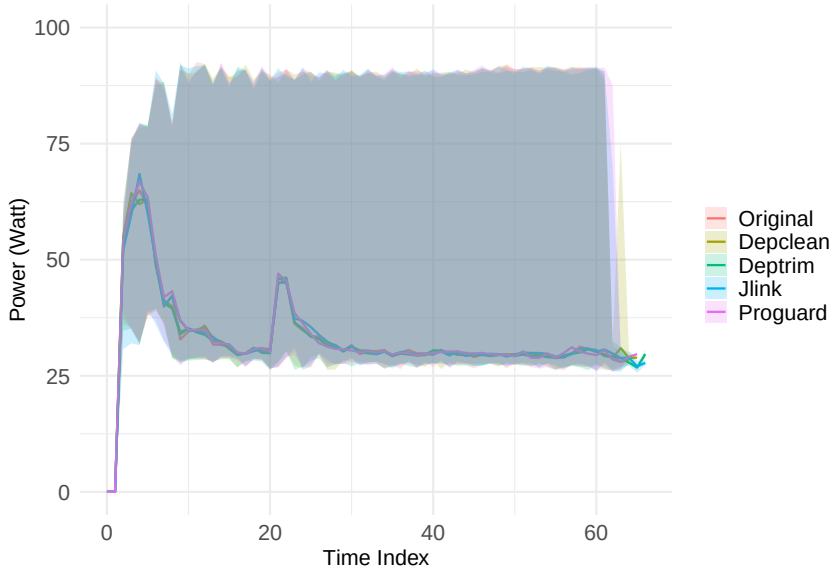
These figures indicate that the impact of debloating tools on energy consumption varies across systems. No tool consistently increased or decreased energy consumption across all SUTs. This is further supported by Figure 5.3, which compares the median power consumption over time for each trial. The power profiles for all debloating tools are nearly identical, with more noticeable yet minor deviations near the end of the time series. This behavior is likely due to the presence of fewer data points caused by small variations in execution time and are unlikely to significantly affect the results of the analysis.

Additionally, these figures show that for some individual SUTs, the debloating tools performed better in reducing energy consumption. For example, ProGuard reduced the median energy consumption by roughly 2% for the benchmark *Javaparser*, and DepTrim reduced it by roughly 2.5% for the real-world program *Tika*.

Although the power consumption data—particularly for the real-world programs—exhibits considerable dispersion, the median power curves in Figure 5.3 show that overall consumption patterns remain largely consistent across tools for both (a) benchmarks and (b) real-world programs, with a common peak early in execution. This shows that the number of trials were likely significant enough to capture the energy consumption characteristics for each tool. The initial peak is likely due to JIT compilation and the loading of program data into memory, which together trigger a spike in CPU and memory utilization. For the interested reader, Appendix A.1 provides additional plots illustrating CPU and memory usage trends.



(a) Benchmarks.



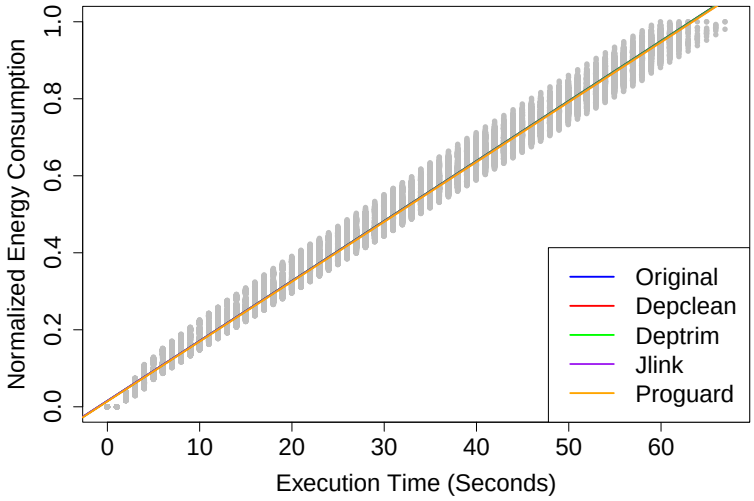
(b) Real-world programs.

Figure 5.3: Median power consumption over time with the shaded area covering 89% of the data distribution at each time index for each debloating tool. Each time index step represents approximately one second.

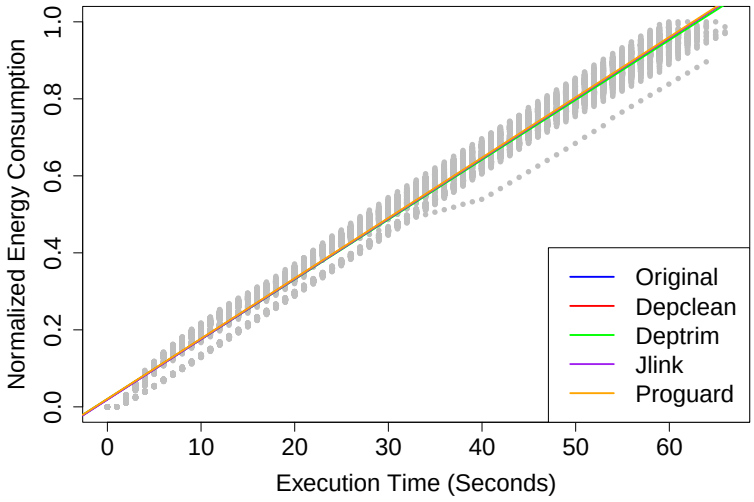
5.2 Impact of Debloating Tools on Energy Consumption (RQ1)

To provide a comprehensive perspective on the impact of debloating tools on energy consumption, results from both linear and hierarchical models fitted separately to

benchmark data and real-world program data are presented. As illustrated in Figure 5.4, the linear models indicate an approximately linear relationship between energy consumption and execution time across all tools.



(a) Benchmarks.

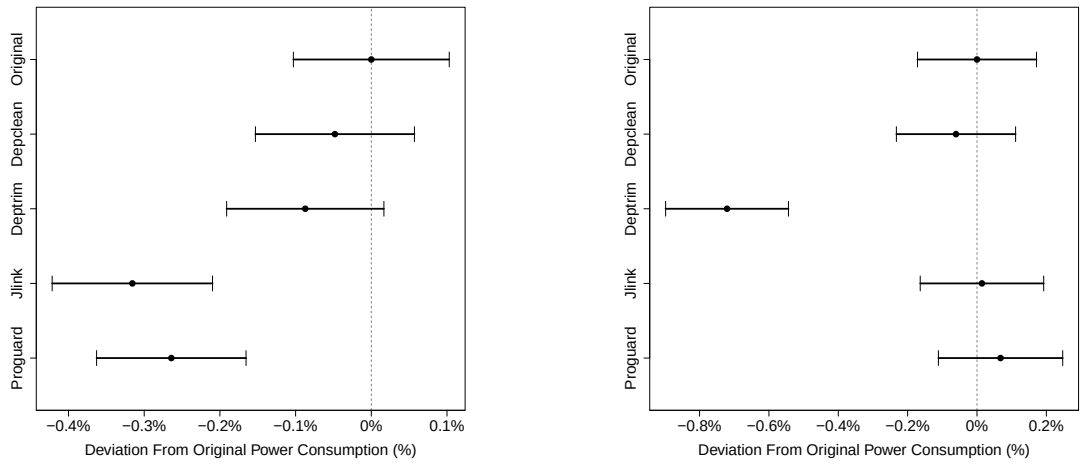


(b) Real-world programs.

Figure 5.4: Linear models for normalized energy consumption over time, plotted against the observed data.

As shown in the linear plots, ProGuard and DepTrim lead to slight reductions in power consumption for benchmarks and real-world programs, respectively, as reflected in differences in the model slopes. These differences are more clearly illustrated in Figure 5.5, which presents a forest plot of posterior means with 89% credible intervals. Specifically, ProGuard reduces power usage by approximately

0.26% for benchmarks, while JLink shows a similar reduction of 0.32%. For real-world programs, DepTrim achieves a modest reduction of 0.72%. Unless otherwise stated, all forest plots in this chapter report posterior means alongside their 89% credible intervals.



(a) Benchmarks. **(b)** Real-world programs.

Figure 5.5: Deviation from original power consumption (linear model).

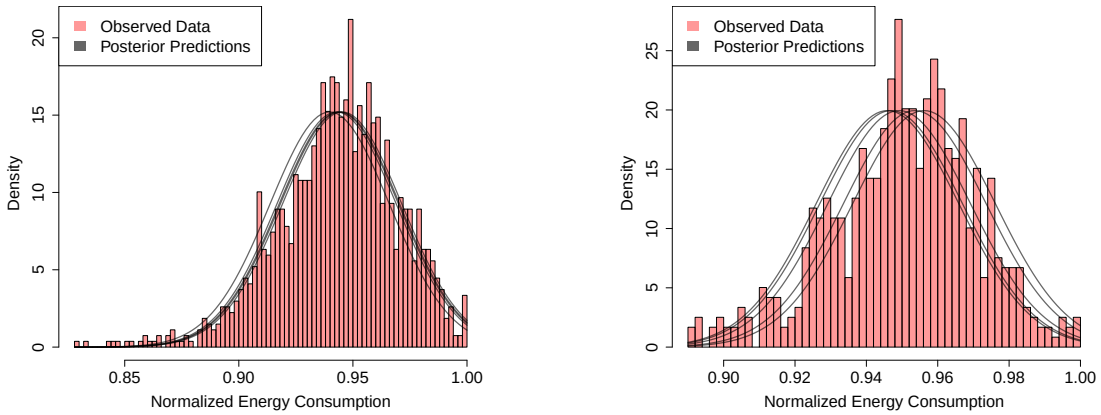
It is important to note that while reductions in power consumption are observed for these tools, execution time plays an equally critical role. A debloating tool that reduces power consumption but increases execution time may still result in higher overall energy consumption.

The analysis of final energy consumption is presented in Figures 5.6 and 5.7. Figure 5.6 displays the fit of the estimand $\boldsymbol{D_i}$ from the hierarchical model, which represents the mean of the assumed normal distribution of energy consumption for each debloating tool, along with the corresponding σ estimand that captures the variability of the data. The estimated densities for each tool are overlaid on histograms of observed data for both benchmarks and real-world programs. These plots confirm that the hierarchical model captures the underlying data structure well, with normalized observed energy consumption closely aligning with the model's posterior distributions.

Figure 5.7 provides a more detailed view of the differences in final energy consumption attributable to each debloating tool. In Figure 5.7a, ProGuard does not exhibit a statistically significant reduction in energy consumption, as its 89% credible interval overlaps zero. This indicates that the observed power savings may have been offset by increases in execution time in some cases, although a slight net decrease in energy use remains plausible. In contrast, JLink shows a statistically significant reduction in energy consumption of approximately -0.58% for the benchmarks. It is important to note, however, that this conclusion is based on the 89% credible interval. Different thresholds (e.g., 95%) could lead to different interpretations of

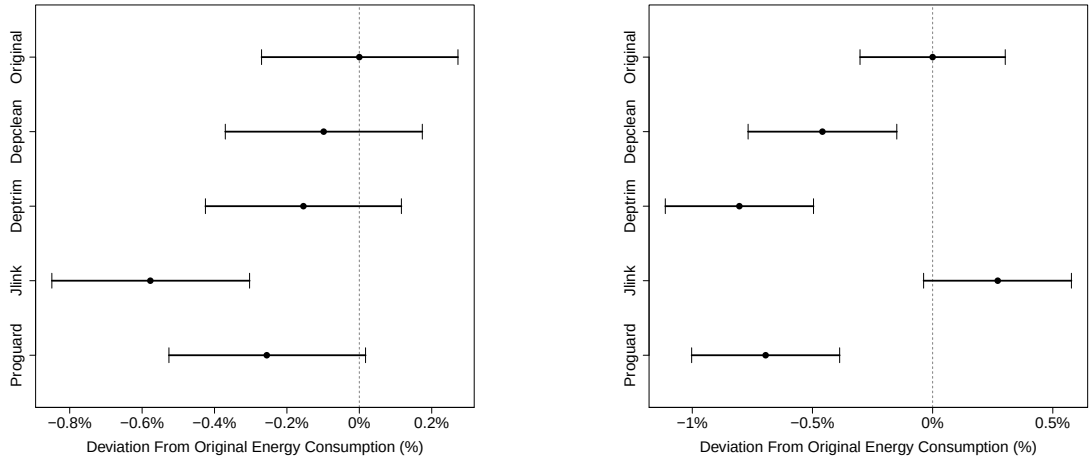
statistical significance and practical relevance.

In Figure 5.7b, DepTrim's reduction in power consumption appears to have contributed to a decrease in energy consumption of approximately 0.80%. Additionally, ProGuard and DepClean, which showed little to no change in power consumption, appear to have achieved modest energy reductions, most likely through reduced execution time in the real-world programs.



(a) Benchmarks. **(b)** Real-world programs.

Figure 5.6: Posterior densities from the hierarchical model overlaid on normalized observed energy consumption data.



(a) Benchmarks. **(b)** Real-world programs.

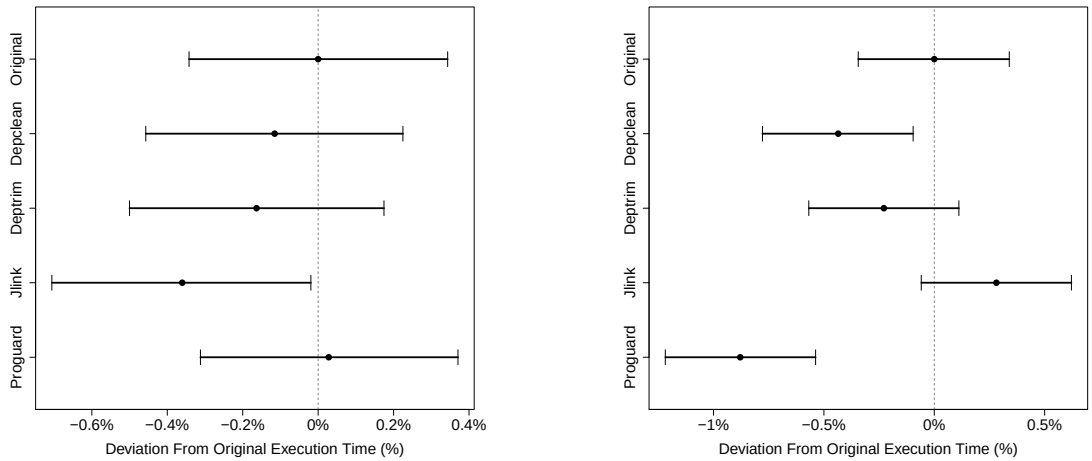
Figure 5.7: Deviation from original energy consumption (hierarchical model).

Impact of Debloating Tools on Energy Consumption (RQ1)

The selected debloating tools had only a marginal effect on energy consumption for the evaluated systems under test. For benchmark applications, JLink achieved a mean reduction of 0.58%. For real-world programs, DepClean, DepTrim, and ProGuard yielded mean reductions of 0.46%, 0.80%, and 0.69%, respectively.

5.3 Analysis of Performance Metrics (RQ2)

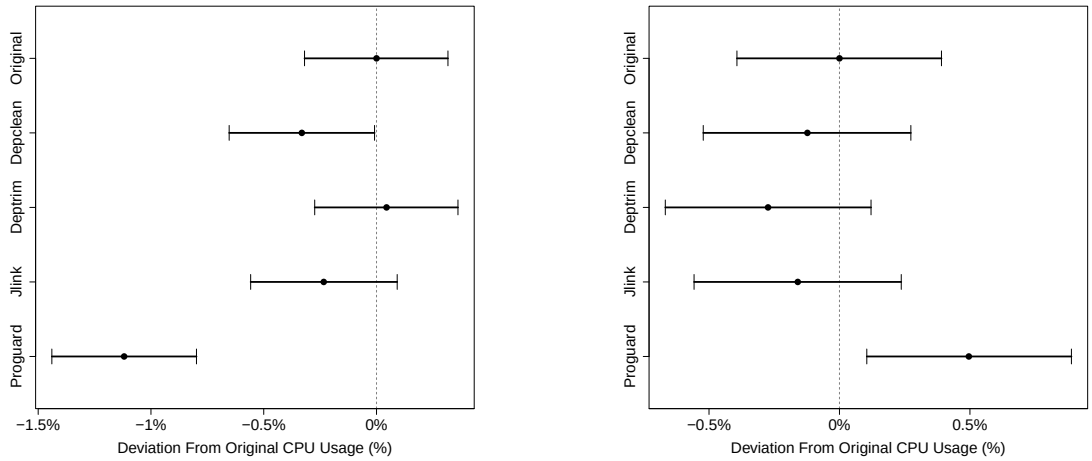
During the experiment, execution times for both benchmarks and real-world programs were approximately normally distributed, with a mean of around 60 seconds and a standard deviation of about one second. Posterior summaries in Figure 5.8 show that JLink led to a slight reduction in execution time for benchmarks, while ProGuard and DepClean yielded improvements for real-world programs. These results, together with the results of power usage, suggest that the observed decrease in energy consumption for ProGuard and DepClean was primarily driven by reduced execution time rather than lower power usage. Conversely, the slight increase in execution time observed with JLink aligns with the corresponding rise in energy consumption, although the increase is very small.



(a) Benchmarks. **(b)** Real-world programs.

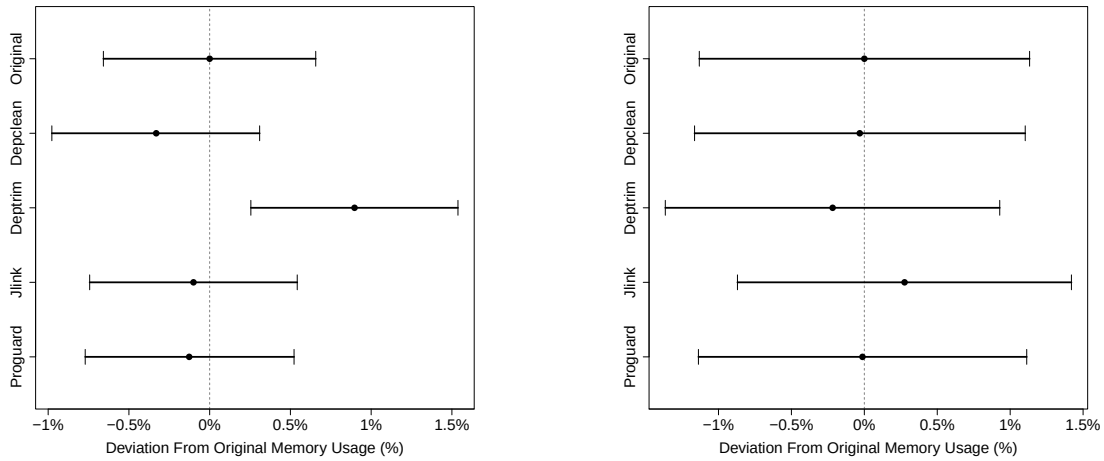
Figure 5.8: Deviation from original execution time (hierarchical model).

To better understand the differences in power consumption and execution time—which together determine overall energy consumption—CPU and memory utilization metrics, which may influence these factors, are presented in Figures 5.9 and 5.10, respectively. ProGuard reduces CPU usage for benchmarks by approximately 1.12%, while for real-world programs, it slightly increases CPU usage by 0.5%. In terms of memory usage, DepTrim leads to a modest increase of 0.9% for benchmarks, whereas differences across all tools for real-world programs remain negligible.



(a) Benchmarks. **(b)** Real-world programs.

Figure 5.9: Deviation from original CPU usage (hierarchical model).



(a) Benchmarks. **(b)** Real-world programs.

Figure 5.10: Deviation from original memory usage (hierarchical model).

The exact values corresponding to the data in the forest plots are available in Tables 5.1 and 5.2. Values in bold indicate statistically significant effects, where the 89% credible intervals of the posterior means exclude zero, reflecting the uncertainty level adopted in this analysis.

Table 5.1: Estimated mean and standard deviation (%) of each metric for benchmarks.

Tool	Execution Time	CPU	Memory	Power	Energy
Original	0.00 ± 0.21	0.00 ± 0.20	0.00 ± 0.41	0.00 ± 0.06	0.00 ± 0.17
DepClean	-0.12 ± 0.21	**-0.33** ± 0.20	-0.33 ± 0.40	-0.05 ± 0.07	-0.10 ± 0.17
DepTrim	-0.16 ± 0.21	0.04 ± 0.20	**0.90** ± 0.40	-0.09 ± 0.06	-0.15 ± 0.17
JLink	**-0.36** ± 0.21	-0.23 ± 0.20	-0.10 ± 0.40	**-0.32** ± 0.07	**-0.58** ± 0.17
ProGuard	0.03 ± 0.21	**-1.12** ± 0.20	-0.13 ± 0.41	**-0.26** ± 0.06	-0.26 ± 0.17

Table 5.2: Estimated mean and standard deviation (%) of each metric for real-world programs.

Tool	Execution Time	CPU	Memory	Power	Energy
Original	0.00 ± 0.21	0.00 ± 0.25	0.00 ± 0.71	0.00 ± 0.11	0.00 ± 0.19
DepClean	**-0.44** ± 0.21	-0.12 ± 0.25	-0.03 ± 0.71	-0.06 ± 0.11	**-0.46** ± 0.19
DepTrim	-0.23 ± 0.21	-0.27 ± 0.25	-0.22 ± 0.72	**-0.72** ± 0.11	**-0.80** ± 0.19
JLink	0.28 ± 0.21	-0.16 ± 0.25	0.28 ± 0.72	0.01 ± 0.11	0.27 ± 0.19
ProGuard	**-0.88** ± 0.21	**0.50** ± 0.25	-0.01 ± 0.70	0.07 ± 0.11	**-0.69** ± 0.19

Tables 5.3 and 5.4 present the correlation matrices for various metrics, calculated from raw data values at each time step. The results indicate a strong positive correlation between CPU utilization, memory usage, and power consumption for the real-world programs. In contrast, these correlations are nearly negligible in the benchmarks.

Table 5.3: Pearson correlation matrix for benchmarks, computed using raw metric values at each timestamp.

	Time	CPU	Memory	Power
Time	1.00	-0.29	0.10	-0.04
CPU	-0.29	1.00	0.00	0.13
Memory	0.10	0.00	1.00	0.10
Power	-0.04	0.13	0.10	1.00

Table 5.4: Pearson correlation matrix for real-world programs, computed using raw metric values at each timestamp.

	Time	CPU	Memory	Power
Time	1.00	-0.04	0.19	0.00
CPU	-0.04	1.00	0.93	0.93
Memory	0.19	0.93	1.00	0.90
Power	0.00	0.93	0.90	1.00

This suggests that: (i) for benchmarks, power consumption remained relatively stable during execution, while memory usage slightly increased and CPU utilization slightly declined over time; and (ii) for real-world programs, memory and power usage increased nearly linearly with CPU utilization, while the time step had no effect on power consumption.

These differences can likely be attributed to the distinct characteristics of the two program types. Most real-world programs processed substantial volumes of data and were often multithreaded, which naturally led to a stronger link between memory and CPU usage, and allowed CPU usage to scale more significantly. Conversely, the benchmarks were predominantly single-threaded and tended to load all required data during program initialization, making memory and CPU usage a less critical factor during execution.

Tables 5.5 and 5.6 present the correlation matrices computed from the final time step of each trial. These matrices capture the relationships between final energy consumption and execution time, along with mean CPU and memory utilization per trial. As shown in the tables, there is a strong positive correlation between energy consumption and execution time for benchmarks, and a moderate positive correlation for real-world programs. A moderate positive correlation between energy consumption and CPU usage is observed only in the benchmarks, while it is negligible for the real-world programs. Additionally, both datasets exhibit a weak negative correlation between memory usage and energy consumption, suggesting either that more memory-intensive systems under test consumed less energy, or that increased memory usage coincided with reduced energy consumption. However, due to the limited sample size, these weaker correlations should be interpreted with caution.

Table 5.5: Pearson correlation matrix for benchmarks, based on final energy and execution time values, along with mean memory and CPU utilization for each trial.

	Time	**CPU**	**Memory**	**Energy**
Time	1.00	0.11	-0.20	0.64
CPU	0.11	1.00	-0.04	0.40
Memory	-0.20	-0.04	1.00	-0.24
Energy	0.64	0.40	-0.24	1.00

Table 5.6: Pearson correlation matrix for real-world programs, based on final energy and execution time values, along with mean memory and CPU utilization for each trial.

	Time	**CPU**	**Memory**	**Energy**
Time	1.00	-0.01	0.27	0.32
CPU	-0.01	1.00	0.35	0.09
Memory	0.27	0.35	1.00	-0.18
Energy	0.32	0.09	-0.18	1.00

These results suggest that execution time is the most influential factor affecting energy consumption for both benchmark and real-world programs, with CPU utilization playing a secondary role in benchmarks. Notably, execution time exhibits only weak correlations with mean CPU and memory usage, indicating that its strong association with energy consumption may stem from two factors: (i) system noise introducing variability in execution time, thereby amplifying its apparent influence on energy consumption beyond what is accounted for by other metrics, and (ii) certain

debloating tools reducing execution time through mechanisms that have minimal effect on mean CPU or memory usage.

This interpretation is supported by the observation that JLink, DepClean, and ProGuard all produced statistically significant reductions in both execution time and energy consumption without significantly reducing mean CPU and memory usage. Interestingly, ProGuard also increased mean CPU usage, suggesting a more complex relationship between performance metrics and energy outcomes. The only exception is DepTrim, which achieved the most substantial reductions in energy and power consumption through mechanisms not directly reflected in the available metrics.

Overall, these findings indicate that the relationships between energy consumption, power usage, and CPU and memory usage may be more nuanced than captured by mean usage statistics alone. This complexity likely explains why mean CPU and memory usage did not consistently align with the observed changes in energy and power consumption across debloating tools.

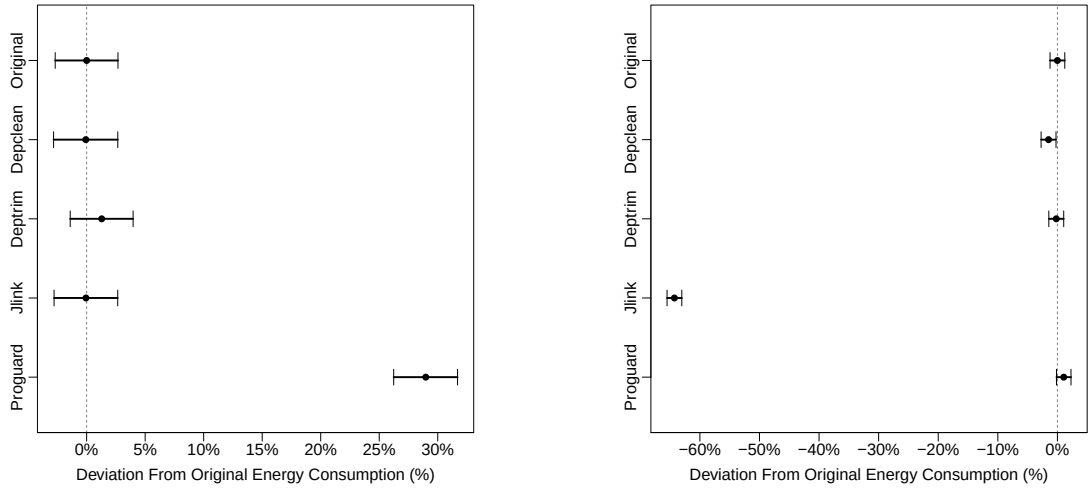
Influential Factors on Energy Consumption in Debloated Applications (RQ2)

Execution time was identified as the most influential factor affecting energy consumption, exhibiting a strong positive correlation for benchmarks (0.64) and a moderate positive correlation for real-world programs (0.32). In real-world applications, power consumption was strongly correlated with both CPU and memory usage (0.93 and 0.90, respectively). Mean CPU usage showed a moderate positive correlation with energy consumption in benchmarks (0.40), while mean memory usage displayed a weak negative correlation in both benchmarks (-0.24) and real-world programs (-0.18).

5.4 Outliers

During the analysis of the results, *error-prone* and *java-faker* were found to behave significantly differently from the other SUTs. In the case of java-faker, an unexplained issue with ProGuard led to highly inconsistent execution times. Approximately 50% of runs experienced a delay of around 20 seconds beyond the typical execution time. This irregularity caused the energy consumption measurements for *java-faker* to be misleading, as illustrated in Figure 5.11a. Due to the severity and unpredictability of this behavior, *java-faker* was excluded from the final analysis.

For error-prone, the discrepancy stemmed from the requirement to use the compiler command *javac* instead of the more common execution command *java*. This distinction significantly affected the performance of JLink since it used a different part of the Java environment, reducing the execution time from approximately 60 seconds to just 17 seconds. As a result, the observed energy consumption dropped by about 65%, as shown in Figure 5.11b. Although this outcome is noteworthy, it indicates that *error-prone* does not share the same execution characteristics as the other SUTs and therefore was also excluded from the analysis.



(a) Java-faker. **(b)** Error-prone.

Figure 5.11: Deviation from original energy consumption (hierarchical model).

6

Discussion

6.1 Discussion of RQ1 Results

The descriptive statistics in Section 5.1 indicated that, although some individual SUTs exhibited small differences in energy consumption—ranging from approximately +2% to -2%—the median power consumption across all SUTs remained close to the baseline for all tools. This suggests that, when considering all SUTs collectively, only minor differences in energy consumption should be expected. This was corroborated by the modelling results in Section 5.2, where the largest reductions observed were 0.80% for real-world programs using DepTrim and 0.58% for benchmarks using JLink.

These differences are relatively modest, especially when contrasted with the energy savings of up to 17% reported for the search-based optimisation method discussed in Section 3.3 [28]. However, a direct comparison is not appropriate, as the results are based on different case examples and the methods differ substantially in both approach and evaluation context.

Although different systems or a larger set of programs may have yielded different outcomes, it appears that, under the conditions of this study and when using the *java* execution command, the techniques implemented by ProGuard, DepClean, DepTrim, and JLink do not substantially reduce energy consumption.

These findings suggest that the specific debloating strategies employed—such as the removal of unused code and modules by all tools, and additional optimisations like inlining, class and method merging, and peephole optimisations by ProGuard—may have limited impact on energy efficiency in practice. Future work could explore alternative optimisation techniques more directly aimed at reducing energy consumption, particularly those that more substantially influence runtime behavior.

6.2 Discussion of RQ2 Results

All of the debloating tools primarily focused on removing unused code, often resulting in substantial reductions in application size. This likely contributed to reduced amounts of code being loaded and compiled by the JIT compiler during startup,

which may partly explain the limited energy savings observed.

In addition, optimisations implemented by JLink and ProGuard may have improved execution efficiency, leading to minor reductions in energy consumption through shorter execution times or marginally lower CPU utilization—particularly for benchmarks in the case of JLink, and for real-world programs in the case of ProGuard.

For the benchmarks, ProGuard had minimal impact on execution time but led to a modest reduction in CPU usage. However, the corresponding decrease in energy consumption was negligible. In contrast, JLink achieved a modest reduction in execution time without a significant effect on CPU usage, which was accompanied by a slight reduction in energy consumption. This suggests that, for the benchmarks, reductions in execution time may have had a greater influence on energy consumption than reductions in CPU usage—an interpretation supported by the stronger positive correlation observed between execution time and energy consumption than between CPU usage and energy consumption. That said, the absolute differences between these metrics were small, making it difficult to draw firm conclusions.

The differences observed in the results between benchmarks and real-world programs might be due to the difference in complexity of the programs. For example, some of the real-world programs were multithreaded, while all of the benchmarks were single-threaded, which could affect the importance of some of the metrics in regards to the energy and power consumption.

It is also important to note that the effectiveness of debloating tools may depend on the specific characteristics of the SUT. For instance, reducing CPU usage on a low-power application may yield less noticeable energy savings than on a more power-intensive one. Thus, the impact of debloating on energy efficiency appears to be influenced by both the nature of the debloating technique and the energy profile of the software under test.

Furthermore, the correlation analysis revealed strong interdependencies between CPU usage, memory usage, and power consumption for real-world programs. This indicates that reducing CPU or memory usage could be beneficial for improving energy efficiency, but not necessarily with the cost of longer execution times.

6.3 Future Research

Future research could expand on this work by including a wider range of systems and debloating tools, both within the Java ecosystem and in other programming languages. A broader dataset would support more general conclusions and help determine whether the patterns observed here remain consistent across different software types and execution environments. It would also be valuable to explore improved Bayesian methods for modeling the relationship between CPU and memory utilization and total energy consumption.

Another potential direction could involve comparing different techniques for reducing

energy consumption in software, such as evaluating debloating techniques alongside search-based code optimization strategies.

Additionally, future studies could explore alternative outputs and configurations of existing tools, such as JLink. In this study, one excluded outlier exhibited a large reduction in energy consumption—approximately 60%—under a specific JLink configuration. While this result was not representative of the overall findings, it suggests that certain features of JLink might offer greater benefits in specific contexts and could be worth exploring further.

6.4 Threats to Validity

This section outlines the limitations and delimitations of this thesis, discussing how they affect the internal, external, and construct validity of the study and how the delimitations shaped its scope.

6.4.1 Threats to Internal Validity

While steps were taken to verify the correctness of the used functionality of the software before and after debloating, without a comprehensive understanding of the systems being tested—primarily due to time constraints—full system functionality could not be guaranteed.

Additionally, while there is a gap in the validation of energy measurements obtained from purely software-based approaches to assess software energy consumption, due to time constraints and limited expertise in this area, the accuracy of the measurements provided by the energy measurement tool could not be verified in this project.

Energy consumption can be measured at various levels of granularity. The simplest method involves coarse-grained measurements, which assess the energy consumed by the entire system. However, this thesis focuses on more fine-grained energy measurements, which isolate the energy usage of specific processes. This approach is significantly more challenging, requiring specialized tools and techniques to accurately capture the energy footprint of individual components or tasks [15, 33]. The difficulty of performing fine-grained energy measurement could have impacted the validity of the collected energy consumption data.

To mitigate this, several steps were taken: background processes were minimized by running the experiments on a lightweight Linux system, each trial was repeated 30 times to allow for statistical smoothing and outlier detection, and the same workload was ensured across all versions of each SUT.

Debloating tools may silently alter functionality of the software without causing visible errors or crashes. At the same time, debloating tools often invalidates test suits as it removes necessary dependencies and functionality for testing, making automated test validation infeasible. This could affect the energy measurements

if the software is not working to its full capacity. To reduce this risk, observable functionality was manually verified before and after debloating, ensuring consistent output across versions.

Tool configuration also affects the fairness and effectiveness of comparisons. Misconfiguration could result in suboptimal debloating outcomes. To mitigate this, only tools with sufficient documentation and community support were selected, as noted in Section 3.2. Care was taken to follow recommended usage instructions to ensure that each tool performed as intended.

6.4.2 Threats to External Validity

The experimental setup involved running software in a controlled environment with minimal background activity, which differs from real-world scenarios where applications often run alongside numerous concurrent processes. This discrepancy limits the external validity, as the results may not fully generalize to practical settings.

All experiments were conducted on a single hardware configuration running Linux. This uniformity limits generalizability, as energy behavior and performance characteristics can differ significantly across operating systems and hardware platforms.

Moreover, the workloads used in testing may not precisely reflect real-world usage. For example, server applications are typically designed to handle multiple concurrent clients—a behavior not easily replicated in a controlled experiment. To partially address this, applications were chosen based on whether realistic usage patterns could be reasonably simulated within the study's constraints.

The scope of the analysis was also limited to a set of ten benchmark programs and five real-world Java applications. While this sample size restricts the breadth of generalization, diversity was prioritized in selecting programs with different characteristics and behaviors to enhance representativeness.

The study's focus on Java constitutes another limitation. While this language is widely used and highly relevant in software engineering, findings may not directly transfer to other languages. Nevertheless, Java's popularity and maturity make it a meaningful starting point for studying the intersection of debloating and energy efficiency.

Finally, although a variety of debloating tools exist for many programming languages and platforms, this study focused on Java and evaluated tools on a single system. This decision was made due to time constraints but could be addressed in future work by expanding the scope to include debloating techniques for other programming languages and platforms.

6.4.3 Threats to Construct Validity

The debloating tools were configured with specific parameters during the experiment, which may not fully represent the range of possible configurations. In par-

ticular, `ProGuard` provides numerous tunable options, and different configurations could result in varying levels of code removal and optimization, potentially affecting energy consumption. Additional configurations and greater familiarity with each tool could improve how well the results reflect the intended constructs related to debloating impact.

The benchmarks and real-world programs may have been constructed or configured in ways that influenced how the key variables—energy, power, CPU utilization, memory usage, and execution time—were manifested and measured. If these configurations imposed artificial constraints or unrealistic behavior, they could affect the validity of the measurements used to represent the constructs of interest.

All measurements were collected using a single tool, `Energibridge`. While this ensured consistency, relying on one measurement source introduces the risk of systematic bias or tool-specific inaccuracies. The validity of the constructs related to energy and power consumption could be strengthened by using additional tools, such as `PowerTOP` or hardware-based approaches.

The statistical models used in this study were based on a scientific model formulated as part of the thesis. While this model offers a useful approximation of the impact on total energy consumption across debloating tools, more advanced or empirically refined models could improve how accurately the constructs are captured.

Finally, the use of mean CPU and memory utilization over an entire trial may not accurately reflect their actual influence on total energy consumption. As noted in the results, these relationships may not be linear, and mean values might obscure important temporal, SUT-specific, and hardware-specific effects on power consumption. Alternative metrics or modeling techniques could better represent how resource usage contributes to energy consumption as a construct.

7

Conclusion

This study investigated the energy-related impact of four Java debloating tools—ProGuard, DepClean, DepTrim, and JLink. The primary objectives were to evaluate their effects on software energy consumption, identify factors influencing their effectiveness, and establish benchmarks along with a replicable methodology to support future research at the intersection of software debloating and energy efficiency.

To this end, we developed a suite of 10 benchmark programs and selected 5 real-world Java applications for evaluation. Each system was executed in 30 independent trials for both the original and the debloated versions produced by each tool. Key performance metrics—including CPU usage, memory usage, power draw, execution time, and total energy consumption—were collected using the energy measurement tool Energibridge. All metrics were analyzed using both linear statistical models and Bayesian hierarchical models to rigorously assess the performance and energy impact of each debloating tool.

To evaluate the impact on energy consumption, we formulated two research questions: **RQ1**: How do the selected debloating tools affect energy consumption in Java applications? and **RQ2**: What factors influence the energy consumption of a debloated application?

Our results for **RQ1** reveal that the selected debloating tools reduced energy consumption only marginally, with a maximum observed mean reduction of 0.80% relative to the baseline. Regarding **RQ2**, the analysis suggests that CPU and memory usage exhibit a strong positive correlation with power consumption for the real-world programs. However, the relationship appears to not be strictly linear, as the mean values of these metrics showed weak to negligible correlation with overall energy consumption. This implies that more sophisticated models may be needed to capture the true influence of CPU and memory usage on total energy consumption. Among all metrics considered, execution time emerged as the most consistent and influential factor that influenced energy consumption.

Across all evaluated systems, energy and power consumption, execution time, and CPU and memory utilization remained largely stable between debloated and baseline versions. Although tool effectiveness varied depending on the system, DepTrim yielded the highest average energy savings, with a mean reduction of 0.80% in real-

world applications. Overall, the tools demonstrated minimal impact on energy efficiency despite employing diverse debloating strategies. These findings underscore the need for more energy-conscious debloating techniques and robust evaluation frameworks to support sustainable software engineering practices.

Bibliography

[1] J. Malmodin, N. Lövehagen, P. Bergmark, and D. Lundén, "ICT sector electricity consumption and greenhouse gas emissions – 2020 outcome," *Telecommunications Policy*, vol. 48, no. 3, p. 102701, Apr. 2024, doi: 10.1016/j.telpol.2023.102701.

[2] W. Wysocki, "Why Don't Software Companies Care About Software Energy Efficiency? A Survey of Software Industry Developers." *Procedia Computer Science*, vol. 246, pp. 5054–5063, 2024, doi: 10.1016/j.procs.2024.09.589.

[3] H. Lyu, G. Gay, and M. Sakamoto, "Developer Views on Software Carbon Footprint and Its Potential for Automated Reduction," in *Search-Based Software Engineering*, P. Arcaini, T. Yue, and E. M. Fredericks, Eds. Cham: Springer Nature Switzerland, Dec. 2023, pp. 35–51, doi: 10.1007/978-3-031-48796-5_3.

[4] C. Soto-Valero, T. Durieux, N. Harrand, and B. Baudry, "Coverage-Based Debloating for Java Bytecode," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, Jun. 2023, doi: 10.1145/3546948.

[5] S. Bhattacharya, K. Gopinath, K. Rajamani, and M. Gupta, "Software Bloat and Wasted Joules: Is Modularity a Hurdle to Green Software?" *Computer*, vol. 44, no. 9, pp. 97–101, Sep. 2011, doi: 10.1109/MC.2011.293.

[6] M. Alhanahnah, Y. Boshmaf, and A. Gehani, "SoK: Software Debloating Landscape and Future Directions," in *Proceedings of the 2024 Workshop on Forming an Ecosystem Around Software Transformation*, ser. FEAST '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 11–18, doi: 10.1145/3689937.3695792.

[7] Statistics Sweden, "Electricity Supply and Use 2001–2023 (GWh)," 2024. [Online]. Available: https://www.scb.se/en/finding-statistics/statistics-by-subject-area/energy/energy-supply-and-use/annual-energy-statistics-electricity-gas-and-district-heating/pong/tables-and-graphs/electricity-supply-and-use-20012023-gwh Accessed: 2025-03-11.

[8] ——, "Environmental Accounts - Environmental Pressure from Consumption 2021," 2021. [Online]. Available: https://www.scb.se/en/finding-

statistics/statistics-by-subject-area/environment/environmental-accounts-and-sustainable-development/system-of-environmental-and-economic-accounts/pong/statistical-news/environmental-accounts--environmental-pressure-from-consumption-2021 Accessed: 2025-03-11.

[9] GitHub, "GitHub Blog: Octoverse: AI Leads Python to Top Language as the Number of Global Developers Surges," 2024. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-2024/#the-most-popular-programming-languages Accessed: 2024-11-18.

[10] Oracle, "JAR File Specification," 2015. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html Accessed: 2025-03-17.

[11] M. D. Brown, A. Meily, B. Fairservice, A. Sood, J. Dorn, E. Kilmer, and R. Eytchison, "A Broad Comparative Evaluation of Software Debloating Tools," Dec. 2023, doi: 10.48550/arXiv.2312.13274.

[12] A. Dewan, P. U. Rao, B. Sodhi, and R. Kapur, "BloatLibD: Detecting Bloat Libraries in Java Applications," in *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, INSTICC. SciTePress, 2021, pp. 126–137, doi: 10.5220/0010459401260137.

[13] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem," *Empirical Software Engineering*, vol. 26, no. 45, Mar. 2021, doi: 10.1007/s10664-020-09914-8.

[14] C. Soto-Valero, D. Tiwari, T. Toady, and B. Baudry, "Automatic Specialization of Third-Party Java Dependencies," Feb. 2023, doi: 10.48550/arXiv.2302.08370.

[15] L. Ardito, R. Coppola, M. Morisio, and M. Torchiano, "Methodological Guidelines for Measuring Energy Consumption of Software Applications," *Scientific Programming*, vol. 2019, no. 1, p. 5284645, Nov. 2019, doi: 10.1155/2019/5284645.

[16] J. Sallou, L. Cruz, and T. Durieux, "EnergiBridge: Empowering Software Sustainability through Cross-Platform Energy Measurement," Dec. 2023, doi: 10.48550/arXiv.2312.13897.

[17] A. Noureddine, S. Islam, and R. Bashroush, "Jolinar: analysing the energy footprint of software applications (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 445–448, doi: 10.1145/2931037.2948706.

[18] A. Noureddine, "PowerJoular and JolarJX: Multi-Platform Software Power Monitoring Tools," in *2022 18th International Conference on Intelligent Environments (IE)*, 2022, pp. 1–4, doi: 10.1109/IE54923.2022.9826760.

[19] Red Hat, *PowerTOP User Guide*, 2014. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/7/html/power_management_guide/PowerTOP Accessed: 2025-03-12.

[20] R. McElreath, *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*, 2nd ed. Boca Raton, FL: Chapman and Hall/CRC, 2020.

[21] E.-Y. Chung, L. Benini, and G. De Micheli, "Automatic source code specialization for energy reduction," in *ISLPED'01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design (IEEE Cat. No.01TH8581)*, Aug. 2001, pp. 80–83, doi: 10.1109/LPE.2001.945378.

[22] Guardsquare NV, "ProGuard," 2023. [Online]. Available: https://github.com/Guardsquare/proguard Accessed: 2025-03-11.

[23] Oracle, "Tools Reference - JLink," 2024. [Online]. Available: https://docs.oracle.com/en/java/javase/11/tools/jlink.html Accessed: 2024-11-18.

[24] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, "JShrink: in-depth investigation into debloating modern Java applications," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 135–146, doi: 10.1145/3368089.3409738.

[25] C. G. Kalhauge and J. Palsberg, "Binary Reduction of Dependency Graphs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 556–566, doi: 10.1145/3338906.3338956.

[26] M. K. Ramanathan, L. Clapp, R. Barik, and M. Sridharan, "Piranha: Reducing Feature Flag Debt at Uber," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 221–230.

[27] K. Ahmed, M. Lis, and J. Rubin, "Slicer4J: a dynamic slicer for Java," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1570–1574, doi: 10.1145/3468264.3473123.

[28] I. Manotas, L. L. Pollock, and J. Clause, "SEEDS: a software engineer's energy-optimization decision support framework," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, pp. 503–514, doi: 10.1145/2568225.2568297.

[29] H. Lyu, G. Gay, and M. Sakamoto, “Exploring Genetic Improvement of the Carbon Footprint of Web Pages,” in *Search-Based Software Engineering*, P. Arcaini, T. Yue, and E. M. Fredericks, Eds. Cham: Springer Nature Switzerland, 2023, pp. 67–83, doi: 10.1007/978-3-031-48796-5_5.

[30] K.-J. Stol and B. Fitzgerald, “The ABC of Software Engineering Research,” *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, Sep. 2018, doi: 10.1145/3241743.

[31] M. Engström, “Replication Package for "Assessing the Energy Impact of Java Software Debloating Tools",” May 2025, doi: 10.5281/zenodo.15540089.

[32] L. Cruz, “Green Software Engineering Done Right: A Scientific Guide to Set Up Energy Efficiency Experiments,” 2021. [Online]. Available: https://luiscruz.github.io/2021/10/10/scientific-guide.html Accessed: 2025-03-12.

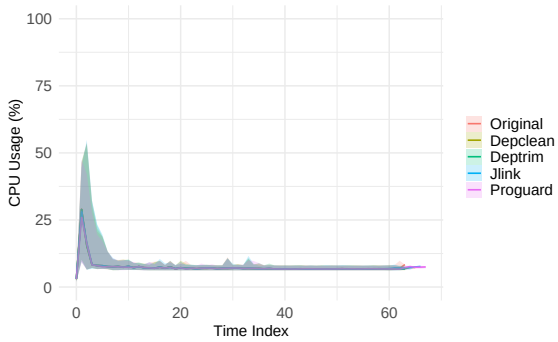
[33] S. Rajput, T. Widmayer, Z. Shang, M. Kechagia, F. Sarro, and T. Sharma, “Enhancing Energy-Awareness in Deep Learning through Fine-Grained Energy Measurement,” *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, Dec. 2024, doi: 10.1145/3680470.

A

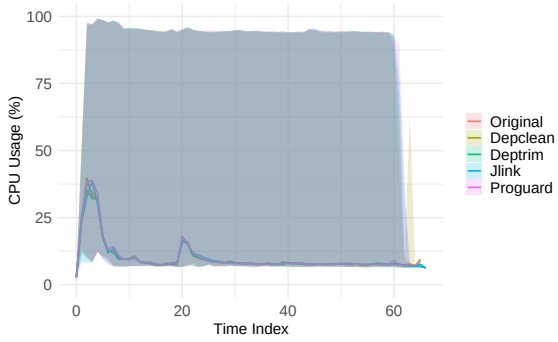
Appendix 1

A.1 CPU and Memory Usage

Figures A.1 and A.2 present the median curves and 89% percentile ranges of CPU and memory utilization, respectively, across all SUT instances excluding outliers. In these figures, the data was separated by the debloating techniques that was used for the measurements.

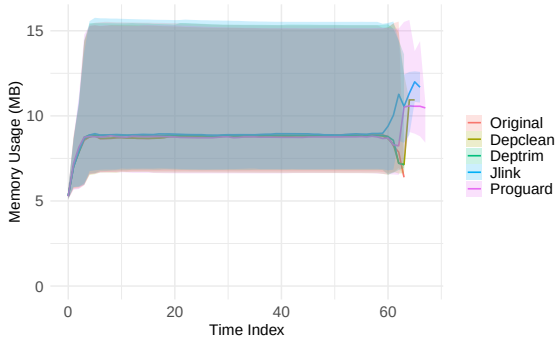


(a) Benchmarks.

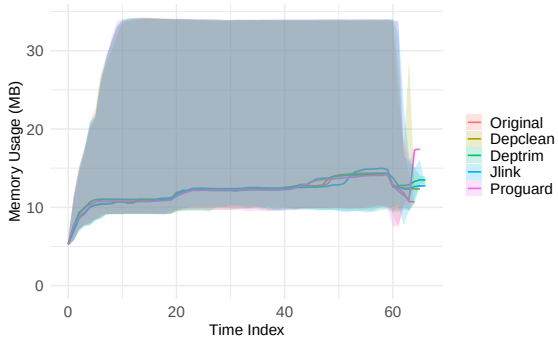


(b) Real-world programs.

Figure A.1: Median CPU utilization over time, with the shaded area representing 89% of the data distribution at each time index for each debloating tool. Each time index step represents approximately one second.



(a) Benchmarks.



(b) Real-world programs.

Figure A.2: Median memory utilization over time, with the shaded area representing 89% of the data distribution at each time index for each debloating tool. Each time index step represents approximately one second.