On the Effect of Feature Reduction on Energy Consumption: An Exploratory Study

Anonymous Author(s)*

Abstract

Energy consumption has become a growing concern in the pursuit of a more sustainable software industry, particularly in advocating for its reduction. In the last decade, this issue has been increasingly investigated by the software engineering community. However, few studies have investigated it in configurable systems that can embed thousands of implemented features. As configurable systems become more complex, feature reduction can help focus on essential features while eliminating bloated and unnecessary ones. Although several studies have explored how feature interactions and runtime performance affect energy consumption, none, to our knowledge, have studied the impact of feature reduction. This paper fills this gap. In particular, we investigate how both *on-demand* and *built-in* feature reduction affect the energy consumption of configurable systems. On-demand reduction allows users to retain only the features necessary for their specific usage. In contrast, built-in reduction provides a predefined set of features tailored to address a fixed set of usage. We conducted a first exploratory study using 28 programs from three systems that offer built-in feature reduction, namely `ToyBox`, `BusyBox`, and `GNU`, along with 6 `GNU` programs debloated on-demand with `Chisel`, `Debop`, and `Cov` tools. In our results, built-in feature reduction led to statistically significant energy decreases in 7% of the cases, while on-demand reduction, despite achieving energy decreases in 67% of cases, showed no statistical significance. However, when energy consumption increased, it was often more substantial than the reductions observed (occurring in 25% of built-in cases and 11% of on-demand cases) showing the complex and sometimes counterintuitive interplay between feature reduction and energy. Additionally, the observed strong correlation between energy consumption and execution time motivates a shift from traditional debloating goals, centered on binary size/attack surface, to energy-aware strategies that prioritize performance concerns. Finally, we provide an in depth analysis and discuss the perspective.

CCS Concepts

• **Software and its engineering** → **Software performance**; • **General and reference** → *Empirical studies.*

Keywords

energy consumption, configurable systems, feature reduction

SPLC'25, September 01–September 05, 2025, A Coruña, Spain

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Anonymous Author(s). 2018. On the Effect of Feature Reduction on Energy Consumption: An Exploratory Study. In *Proceedings of 29th International Systems and Software Product Line Conference (SPLC'25).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Energy consumption is nowadays a growing concern in the pursuit of a more sustainable software industry [5, 13, 14, 18, 26, 43, 45, 50]. A report from the ShiftProject [1] and others has raised environmental concerns regarding Information and Communication Technology (ICT) and called for reducing energy consumption. During the past decade, this issue has been increasingly investigated by the software engineering community (*e.g.*, [9, 20, 34, 35, 37, 38, 41, 45]).

As software becomes more essential to our daily lives, reducing its energy consumption has become essential for sustainable software development. This paper focuses on configurable systems that allow developers to customize variants and use a subset of implemented features by activating or deactivating *configuration options*. However, as more features are added over time, configurable systems can become increasingly complex. For example, Linux kernel has around 15 thousand options in version 5.8 [1, 28].

One way to manage the increasing complexity of a configurable system is through feature reduction, which has been studied for several purposes, such as for defect and bug prediction [27, 47, 48]. Several studies have explored the topic of energy consumption for configurable systems, including the impact of feature interactions on energy consumption [17], the use of static analysis to evaluate energy consumption in configurable systems [10, 11], and the correlation between execution time and energy consumption [52]. However, to our knowledge, no work has studied the impact of feature reduction of configurable systems on energy consumption.

This paper addresses this gap by investigating how feature reduction can impact the energy footprint of configurable systems. We distinguish two types of feature reduction in configurable systems, as shown in Figure 1. First, *built-in feature reduction*, where developers create and use alternative systems with fewer features. These systems are made to be simpler or smaller. For example, `ToyBox` and `BusyBox` are re-implementations of `GNU` software with a reduced set of features (shown on the right side of Figure 1). Second, *on-demand feature reduction* is when developers debloat software to remove unnecessary features. Several research tool prototypes, such as `Chisel` [19], `Debop` [53], and `Cov` [54] can assist in this process. For example, `GNU` programs can hopefully be debloated by such tools to reduce their features (see the left part of Figure 1).

Whether developers deal with built-in or on-demand feature reduction, they end up using configurable software with reduced binary sizes and fewer features. The goal of this paper is to investigate and reveal the effect of both types of feature reduction on

[1] https://theshiftproject.org/wp-content/uploads/2019/03/Lean-ICT-Report_The-Shift-Project_2019.pdf

energy consumption in configurable software. *We hypothesize that feature reduction impacts energy consumption* (H_1), as the reduction of features affects the code that is executed and can impact energy use. To explore this, we conducted an exploratory study to examine how both built-in and on-demand feature reduction impact energy consumption. For built-in feature reduction, we first selected three configurable systems, namely GNU [16], ToyBox [29] and BusyBox [51], along with 28 programs, each included in all three systems. For example, the program `mkdir` is available in GNU, ToyBox, and BusyBox, with different implementations and a reduced set of features in the last two systems. For on-demand feature reduction, we selected 6 GNU programs from Xin et al. [54] that have been debloated with three debloating tools, namely Chisel [19], Debop [53], and Cov [54]. For example, the program `mkdir` has been debloated by the three aforementioned tools, resulting in three variants of `mkdir` with a reduced set of features. Finally, we measured energy consumption and analyzed its correlation with the configuration options, binary size, and execution time of the programs. Programs were selected based on our ability to successfully compile and execute all three debloated variants, ensuring valid outputs for the common set of features used during debloating. The findings suggest that built-in feature reduction led to statistically significant energy decreases in ≈ 7% of cases, while on-demand reduction achieved energy decreases in ≈ 67% of cases, but without statistical significance. However, both approaches showed significant energy increases in some cases: 25% for built-in and ≈ 11% for on-demand. These results highlight the complex relationship between feature reduction and energy consumption. Additionally, we found a weak correlation between binary size, configuration options, and energy consumption, suggesting that smaller binaries and fewer options do not necessarily result in lower energy consumption. However, execution time showed a strong correlation, making it a good proxy for energy consumption. These findings are consistent with prior observations at the scale of entire configuration spaces [52]. The contributions of the paper are as follows:

- We distinguish two types of feature reduction and propose a terminology for them: *built-in feature reduction* for alternative program implementations with fewer features, and *on-demand feature reduction* for debloated programs.
- A novel study exploring the impact of both built-in and on-demand feature reduction on energy consumption in configurable software systems, which resulted to be nuanced with potential reductions (≈ 64%) but also notable increases (≈ 36%) in some cases. The amount of energy consumed depends on the approach used and the program.
- We identify key factors influencing energy consumption, showing that execution time is a stronger predictor than binary size or the number of configuration options, providing valuable insights for future debloating strategies.
- A replication package is also made available online[2] to facilitate replication and further experimentation.

The remainder of this paper is structured as follows. Section 2 introduces background on feature reduction and energy consumption techniques. Section 3 outlines our experimental approach. Section 4 presents the methodology used to study the impact of built-in feature reduction on energy consumption, with corresponding results in section 5. Section 6 describes the on-demand feature reduction approach, and section 7 discusses its results. Section 8, provides an in-depth analysis and discusses key insights. Section 9 covers threats to validity and section 10 reviews related work. Finally, section 11 concludes the paper with perspectives for future work.

[2]https://anonymous.4open.science/r/bloat-energy-consumption-836C

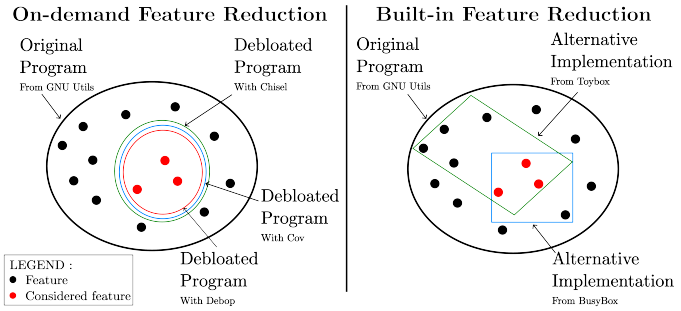

Figure 1: On-demand *vs.* Built-in feature reduction.

2 Background

This section discusses the methods used to measure energy consumption in configurable systems with reduced features.

2.1 Feature Reduction

Configurable software systems offer several features that activate specific functionalities or impact non-functional properties (*e.g.*, execution time, accuracy, etc.). The goal of *feature reduction* is to prioritize core features that provide the most value to users while eliminating code that does not significantly contribute to the overall software system or its qualities. For instance, feature reduction can simplify the system and ensure it adheres to a given usage by removing unnecessary source code, as discussed in [21].

We distinguish two approaches for achieving feature reduction, as shown in Figure 1, namely *built-in* feature reduction and *on-demand* feature reduction. The first approach, *built-in* feature reduction, requires developers to create alternative systems with fewer features. Typically, this includes alternative configurable systems with different implementations of a reduced set of features. The right part of Figure 1 shows how ToyBox and BusyBox are re-implementations of a reduced set of GNU features. The second approach, *on-demand* feature reduction, is when developers intentionally debloat a configurable system to remove unnecessary features from it for different purposes. This results in multiple debloated variants with a reduced set of features, compared to the original system that includes all features. The left part of Figure 1 illustrates how GNU can be debloated to reduce its features using debloating tools such as Chisel [19], Debop [53], and Cov [54]. For example, the `mkdir` program is in the GNU configurable system, which has a binary size of 420.60 KiB and 7 runtime configuration options. A built-in feature reduction of `mkdir` exists in both ToyBox and BusyBox as alternative implementations. The two alternative implementations have binary sizes of 18.60 KiB and 18.70 KiB, respectively, along with 3 and 2 configuration options. Moreover,

with debloating tools, one may debloat `mkdir` in GNU. For example, two different debloated variants of `mkdir` can be produced using the `Chisel` tool [19], based on usage profiles describing the use of 2 and 3 configuration options, respectively. Its debloated programs (used in this study) have binary sizes of 22.90 KiB and 16.50 KiB.

Whether built-in or on-demand, the goal of developers and users is to reduce binary size, configuration options, and attack surface of programs [19]. However, it is unclear how feature reduction impacts energy consumption and whether it correlates with other non-functional properties (*e.g.*, binary size). In this paper, we explore the impact of both types of feature reduction on energy consumption.

2.2 Energy Consumption Methods

The methods for measuring energy consumption in IT have evolved over time, shifting from hardware to software. Indeed, software energy consumption monitoring through hardware introduces certain challenges, including the requirement of physical instrumentation and the high level of granularity (i.e., measure at the level of the entire hardware system). This section provides an overview of software energy measurement.

Since 2011, Intel has implemented the Running Average Power Limit (RAPL) in their Central Processing Units (CPUs) to provide a more reliable measurement interface than hardware power meters [12]. RAPL reports energy consumption values in joules (J), more precisely in micro-joules (μJ), for different CPU elements (cores, integrated GPU, socket, and RAM). Intel CPUs have a Model-Specific Register (MSR) that updates at a high frequency (1 000 Hz) where the RAPL value is stored [23]. In this way, RAPL value is accessible through the operating system (OS) using the MSR, making the measurement easier and more reliable since it is directly linked to the CPU.

Several tools and prototypes are available [3] to capture the RAPL value to measure application energy consumption, such as Intel Power Gadget, Intel PowerLog, Perf, PowerTOP, PowerStat, Likwid, and *Jouleit* [4]. They mostly rely on power monitors or energy profilers. However, Intel Power Gadget, PowerTOP, PowerStat, and Likwid are unsuitable for our needs as they cannot measure process consumption during execution. Although Intel PowerLog supports this functionality, it is only compatible with Windows. Perf, on the other hand, does not report RAPL values in a usable format like CSV or JSON. Among the available tools, *Jouleit* is the most suitable option in our context. It has the ability to monitor a process, automatically execute and monitor it multiple times, and generate reports in CSV format. Furthermore, *Jouleit* is compatible with Linux, making it the ideal choice for our needs.

3 Experimental Approach

In this section, we outline the research questions and the experimental approach of our study.

3.1 Research Questions

To answer the objective of feature reduction's impact on energy consumption, we pose the following research questions.

Built-in feature reduction impact. We pose the following three research questions to examine how built-in feature reduction affects energy consumption in alternative software implementations.

$RQ_{1.1}$: **How does the binary size of programs with built-in feature reduction impact energy consumption, and what is the correlation between the two?**

$RQ_{1.2}$: **How does the number of configuration options of programs with built-in feature reduction impact energy consumption, and what is the correlation?**

$RQ_{1.3}$: **How does the execution time of programs with built-in feature reduction impact energy consumption, and what is the correlation?**

On-demand feature reduction impact. Next, we explore how program debloating (*i.e.*, on-demand reduction) impacts energy consumption and whether findings on alternative implementations extend to debloated programs. From this, we pose two research questions.

$RQ_{2.1}$: **How does the binary size of programs with on-demand feature reduction impact energy consumption, and what is the correlation?**

$RQ_{2.2}$: **How does the execution time of programs with on-demand feature reduction impact energy consumption, and what is the correlation?**

We could not investigate the correlation with configuration options, as this information was unavailable for the debloated programs. Although, in theory, a debloated program should only handle the desired feature, there is no guarantee that the entire input space for a feature is covered.

3.2 Dependent and Independent Variables

Our experiment focuses on quantifying the impact of feature reduction on energy consumption in configurable software systems. In this context, the independent variable under our control is the *feature reduction* itself, specifically the *executable binary size* and the *number of run-time configuration options* in software programs, both before and after feature reduction. To address our research questions, we observe and measure two dependent variables, namely the *execution time* and *energy consumption* of software programs.

3.3 Measurement Setup

We measured software power consumption using the RAPL [25] value, which specifically relates to the Package-Level Power (PSYS), representing the entire CPU, including cores, cache memory controller, and iGPU. The RAPL values are captured using a tool from the PowerAPI [5] toolkit [7] called *Jouleit* [6]. All energy consumption measurements are expressed in micro-joules (μJ).

However, there is still the issue of effectively communicating the meaning of the measured energy consumption. While RAPL values are provided in micro-joules through the CPU interface, this unit may not be easily understandable for developers and end-users. To make energy consumption values more meaningful, we propose converting them into the time of use of various devices commonly used on a daily basis, thereby associating them with real-world

[3] https://luiscruz.github.io/2021/07/20/measuring-energy.html
[4] https://github.com/powerapi-ng/jouleit
[5] https://powerapi.org/
[6] https://github.com/powerapi-ng/jouleit

devices familiar to developers and end-users. Specifically, we chose to convert them to the *time of use* for three devices: an HDD Seagate BarraCuda 3.5" 1 TB 7 200 RPM (5 W) [7], an NVIDIA GPU RTX4080 (320 W) [8], and a Phillips LED light bulb (10.5 W) [9].

To ensure reproducibility and minimize biases, we followed the best practices outlined by Ournani et al. [36]. In particular, to ensure accurate measurements, we considered the following factors:

- We chose a CPU with a low TDP (15 W). CPUs with lower TDP values have less variability in power measurements [36].
- CPU performance options (C-States, Hyper-Threading and TurboBoost) significantly impact the standard deviation during energy consumption measurement [2, 36]. Therefore, we disabled these options in both the BIOS and OS.
- The installed OS (Lubuntu) is a lightweight Ubuntu distribution, chosen to minimize its impact on the measurements. Additionally, we disabled the network service to avoid interference from network traffic [36].

Moreover, *Jouleit* also allows us to collect execution time data. This is done by capturing a timestamp (using the `date` command) immediately before executing the program and another timestamp right after the execution completes. The difference between these two timestamps gives the precise execution time of the program.

Finally, we used a Dell Latitude 7490 machine with an Intel Core i7-8650U processor, which has a Thermal Design Power (TDP) of 15 W. The machine is equipped with 32 GB of DDR4 RAM and runs Linux Lubuntu 22.04.2 LTS with kernel version 5.19.0-45-generic.

4 Methodology for Built-in Feature Reduction experiment

This section details our initial experiment conducted on software systems with built-in feature reduction.

4.1 Subject Systems

To obtain a representative set of software systems with built-in feature reduction, we selected three configurable software systems: `GNU` [16], `ToyBox` [29], and `BusyBox` [51]. The `GNU` core programs include essential file, shell, and text manipulation utilities that are typically available on every operating system, which we refer to as *programs*. Examples of these programs include `mkdir`, which concatenates and write files, `ls`, which lists directory contents, and `mv`, which moves or renames files and directories. We chose these programs because they are well-known and used daily by millions of users. Additionally, they are often used to evaluate different debloating approaches in software engineering [4, 8, 19, 53]. Importantly, the programs in both `ToyBox` and `BusyBox` represent a deliberate application of feature reduction (*i.e.*, built-in) of `GNU` programs. These alternative programs were specifically developed to reduce binary size, configuration space, and to simplify implementation, while improving execution time.

4.2 Pre-experiment Settings

First, we built all programs included by default in each set, specifically in the recent versions: `GNU` version 9.3, `ToyBox` version 0.8.9, and `BusyBox` version 1.36.0. Then, we extracted only the programs common to all three systems, resulting in a set of 75 shared programs. From this set, we selected 28 programs based on diversity in binary size and the number of configuration options.

Let $\mathcal{S}$ represent the set of 28 selected subject programs. Each program $p \in \mathcal{S}$ has its own executable in `GNU`, `ToyBox`, and `BusyBox`, differing in binary size and the number of run-time configuration options available. Next, we enumerated all configuration options for each implementation of p. For example, the `cat` executable is 185 KiB with 12 options in `GNU`, while in `ToyBox`, it is 18 KiB with 4 options, and in `BusyBox`, it is 18 KiB with 6 options.

4.3 Experiment Settings

Using the 28 selected subject programs, we measured both the energy consumption and execution time of each program. To measure energy consumption, we used *Jouleit*.

For each $p \in \mathcal{S}$, we selected two valid configurations $c_1, c_2 \in C$ along with a valid input $i \in \mathcal{I}$. Here, C represents the configuration space of program p, encompassing all feasible configurations or potential combinations of configuration options, and $\mathcal{I}$ denotes the set of possible inputs for the program. Configurations c_1 and c_2 were selected based on the availability of their configuration options in each of the respective implementations of p within `GNU`, `ToyBox`, and `BusyBox`. Specifically, we used purposive sampling [32] to select two common configurations for each program, ranging from none to multiple options depending on the program. This selection was based on the documentation, our expertise, and a carefully chosen input i, when required. For instance, Listing 1 shows the `ls` program in `GNU` with configurations c_1 = *−R* and c_2 = *−Ral*, and input = */path/to/directory* used in our experiment. The same configurations and input(s) were then applied to the respective implementations of p (*e.g.*, of `ls`) in `ToyBox` and `BusyBox`.

```
$ ./GNU/ls -R /path/to/directory // First configuration
$ ./GNU/ls -Ral /path/to/directory // Second configuration
```

Listing 1: `./GNU/ls` with 2 configurations and input used

Next, we compared the binary size and the number of run-time configuration options of the selected programs from `GNU`, `ToyBox`, and `BusyBox`. Using the `GNU` implementation as the baseline, Figure 2 shows that each counterpart in `ToyBox` and `BusyBox` has a smaller binary size and fewer run-time options. On average, the 28 selected programs in `ToyBox` exhibit a 92% reduction in binary size and 64% fewer options, while in `BusyBox`, they show a 93% reduction in binary size and 66% fewer options compared to `GNU`. These data confirm that each selected program for the experiment undergoes a genuine *built-in feature reduction*.

To prevent side effects, all experiments were run 10 times sequentially as the only active processes on the workstation (see Section 3.3). Additionally, all subjects, configurations, and data are available online [10] for reproducibility.

5 Results of Built-in Feature Reduction

This section presents the results and discusses the findings related to our first set of research questions, namely $RQ_{1.1}$, $RQ_{1.2}$, and

[7] Seagate BarraCuda 3.5" datasheet
[8] NVIDIA RTX4080 datasheet
[9] Phillips LED light bulb datasheet

[10] https://anonymous.4open.science/r/bloat-energy-consumption-836C

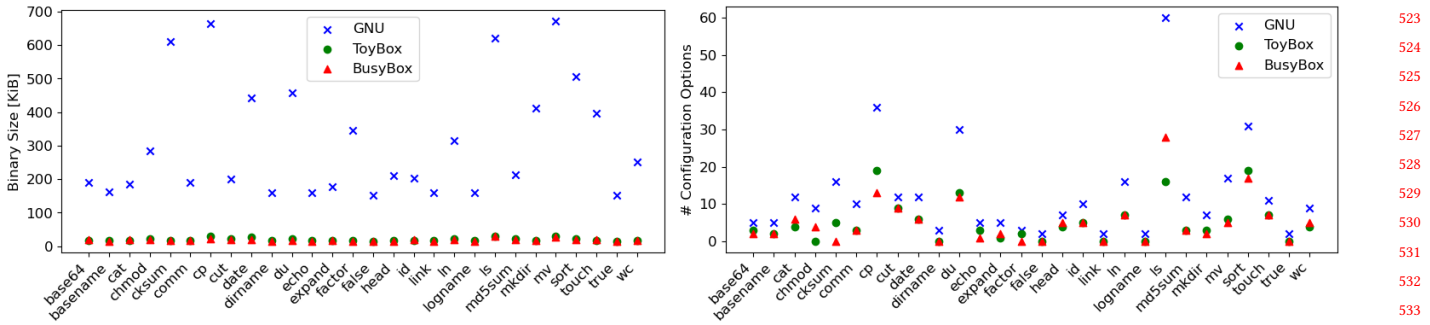

Figure 2: 28 selected programs along with their respective binary size (left) and # options (right) in GNU, ToyBox, and BusyBox.

$RQ_{1.3}$. These questions examine the effects of built-in feature reduction on energy consumption, by exploring its correlation with other non-functional properties, such as the program's binary size, configuration options, and execution time.

5.1 Results on Energy Consumption

General insights. Table 1 shows the average energy consumption for each program in GNU, ToyBox, and BusyBox, along with the standard deviation (StD) for the 10 repetitions. To conserve space, the values in the table are presented in *joules* (J), even though the measurements were conducted in *micro-joules* (μJ).

Looking at Table 1, we observed that the majority of programs (uncolored) consume less than ≈ 7.5 J: specifically, 22 in GNU, 20 in ToyBox, and 20 in BusyBox. An energy consumption of 7.5 J is equivalent to having a 10.5 W LED light bulb turned on for 0.72 seconds.These program are considered as energy efficient according to our own categorization.

On the other hand, only 1 to 3 programs fall into the 10 - 100 J consumption range, making them *less energy-efficient* ( ). 10 J of energy consumption is equivalent to turning on a 10.5 W LED light bulb for 0.95 seconds, while 100 J is equivalent to 9.52 seconds of use. Additionally, there are 3 *energy-inefficient* programs in GNU, 7 in ToyBox, and 5 in BusyBox, as they consume over 100 J ( ). Although the values differ across various realizations of the same program, only cp and md5sum programs consistently consume over 100 J in all three implementations. Although consuming less than 100 J in its original version, sort in ToyBox is the most energy-consuming program, consuming ≈ 469.33 J. This is equivalent to the energy consumed by a 10.5 W LED light bulb running for 44.7 seconds. It is noteworthy that the programs classifies as energy-inefficient exhibit large standard deviations, indicating that their energy consumption values are more spread out from the average. Among these programs, cksum, md5sum, and comm have over 100% StD (only in ToyBox), suggesting that the energy consumed in each run varies significantly from the average. Despite that we conducted multiple repetitions, identifying the precise factors influencing these deviations in energy consumption for these programs proved challenging.

Comparative analysis. As shown in Figure 2, the binary size and number of run-time options in the programs from GNU are higher than those in ToyBox and BusyBox. Therefore, using the programs in GNU as a baseline, we conducted a comparative analysis to determine whether the programs in ToyBox and BusyBox, with their reduced features, are more energy-efficient than their counterparts in GNU. In the *Diff. %* columns of Table 1, we present the comparative analysis results. For each case, we use the Mann-Whitney U

Table 1: Energy Consumption ± Standard Deviation (StD) in joules [J] for 28 programs with imposed feature reduction

Program	GNU ± StD	ToyBox ± StD	Diff.%	BusyBox ± StD	Diff.%
base64	0.35 ± 0.04	0.91 ± 0.04	160	0.43 ± 0.08	23.13
basename	0.29 ± 0.02	0.29 ± 0.01	-0.48	0.29 ± 0.02	-0.11
cat	2.13 ± 0.11	1517.43 ± 81	71284	37.56 ± 1.93	1667
chmod	1.54 ± 1.28	1.52 ± 1.26	-1.06	1.95 ± 1.71	26.78
cksum	11.39 ± 11.42	206.25 ± 212	1710	245.80 ± 253	2057
comm	55.07 ± 5.56	338.32 ± 286	514	120.87 ± 8.63	120
cp	108.85 ± 7.20	147.44 ± 56.97	35.46	111.25 ± 18.99	2.21
cut	2.21 ± 0.25	11.40 ± 0.72	416	4.93 ± 1.20	123
date	0.32 ± 0.02	0.32 ± 0.02	-0.02	0.31 ± 0.02	-1.00
dirname	0.32 ± 0.02	0.31 ± 0.02	-2.89	0.31 ± 0.02	-3.56
du	0.52 ± 0.23	0.52 ± 0.23	-0.90	0.49 ± 0.21	-6.24
echo	0.32 ± 0.02	0.31 ± 0.02	-3.29	0.31 ± 0.02	-5.02
expand	7.27 ± 0.33	0.50 ± 0.03	-93.09	14.57 ± 0.85	100
factor	0.32 ± 0.02	0.32 ± 0.02	-1.80	0.32 ± 0.02	-1.77
false	0.32 ± 0.02	0.32 ± 0.02	-1.68	0.31 ± 0.02	-4.63
head	0.32 ± 0.02	0.31 ± 0.02	-1.62	0.31 ± 0.02	-3.77
id	0.33 ± 0.02	0.31 ± 0.02	-4.49	0.31 ± 0.02	-6.01
link	0.32 ± 0.02	0.31 ± 0.02	-1.55	0.31 ± 0.02	-3.45
ln	0.32 ± 0.02	0.34 ± 0.02	4.10	0.31 ± 0.02	-5.38
logname	0.32 ± 0.02	0.31 ± 0.02	-3.58	0.31 ± 0.02	-3.82
ls	1.05 ± 0.47	3.49 ± 1.03	233	0.90 ± 0.22	-14.34
md5sum	118.60 ± 122	264.03 ± 271	123	143.15 ± 147	20.70
mkdir	0.31 ± 0.02	0.30 ± 0.02	-2.05	0.30 ± 0.01	-2.92
mv	0.31 ± 0.02	0.30 ± 0.02	-3.27	0.30 ± 0.02	-3.80
sort	95.54 ± 4.75	469.33 ± 25.61	391	318.30 ± 16.59	233
touch	0.30 ± 0.02	0.30 ± 0.01	-0.01	0.29 ± 0.02	-0.23
true	0.29 ± 0.02	0.29 ± 0.02	-1.33	0.29 ± 0.02	0.02
wc	454.69 ± 24	151.48 ± 32.48	-66.68	87.56 ± 6.07	-80.74

Energy-inefficient programs
Less Energy-efficient programs
Lower energy consumption
Higher energy consumption
Non-significant Difference

Table 2: Spearman correlation of execution time, binary size, and number of options with energy consumption.

	Energy/Size		Energy/Options		Energy/Exec Time	
	Spearman	p-value	Spearman	p-value	Spearman	p-value
GNU	0.251	0.197	0.461	0.014	0.891	2.07e-10
ToyBox	0.350	0.068	0.392	0.039	0.966	1.02e-16
BusyBox	0.305	0.115	0.271	0.164	0.959	7.98e-16

No significant correlation (p-value> 0.05).
Interpretation of the coefficient: ±.9 - ±1.0 (very high); ±.70 - ±.90 (high); ±.50 - ±.70 (moderate); ±.30 - ±.50 (low); .00 - ±.30 (negligible)

test [31] to assess the statistical significance of the observed differences. A program's difference is marked with the red line pattern (▨) if it lacks statistical significance in either of the two cases (p-value above the 0.05 significance level). The computed p-values are available in the reproduction package.

The results show that 18 of the 28 programs (≈ 64%) in ToyBox consume less energy than their GNU counterparts for the same configurations and inputs. Similarly, 17 of the 28 programs (≈ 61%) in BusyBox use less energy. However, only 2 programs in each system demonstrate statistically significant energy savings (■). This indicates that most energy reductions, while present, are not strong enough to be considered statistically reliable (*cf.* negative values with ▨). Conversely, 10 programs (≈ 36%) in ToyBox and 11 programs (≈ 39%) in BusyBox consume more energy than their GNU counterparts. Among these, 7 programs in each system show statistically significant energy increases (■), meaning that these differences are more robust and likely not to random variation. However, in 3 ToyBox programs and 4 BusyBox programs, the observed energy increases are not statistically significant (*cf.* positive values with ▨). These results show that while a majority of programs in ToyBox and BusyBox consume less energy compared to their counterparts in GNU, significant energy savings occur in only 7.14% of cases. In contrast, 25% of the cases exhibit statistically significant increases in energy consumption.

These results provide initial insights suggesting that reducing the binary size of an executable and the number of configuration options in a program do not necessarily lead to lower energy consumption.

5.2 Binary Size Impact

To explore the relationship between binary size and energy consumption, we calculated the Spearman correlation, which, unlike the Pearson correlation, does not require any normality assumption. Spearman correlation, which captures monotonic relationships, with results interpreted according to the Evans rule [24]. This analyses encompassed all 28 programs in GNU, ToyBox, and BusyBox. Basically, we focused on determining the correlations between the binary size of the programs and their corresponding average energy consumption, as outlined in Figure 2 (left) and Table 1 (columns with StD), respectively.

The *Energy/Size* column in Table 2 presents the calculated correlations. Spearman coefficients range from 0.251 to 0.350. The highest correlations are in ToyBox and BusyBox, indicating a weak correlation between binary size and energy consumption in these programs. In contrast, the programs in GNU show a negligible correlation, suggesting that the relationship between binary size and energy consumption is insignificant. These findings, showing *negligible* to *weak* correlations, reinforce the initial insights and conclusions drawn from the comparative analysis in Section 5.1.

To determine whether the correlations are statistically significant, we also calculated the *p-value* with the significance level (α) of 5% for each set of programs. The second column of *Energy/Size* in Table 2 shows the obtained values. All p-values are greater than 0.05 (with ▨), indicating that the negligible correlations are not statistically significant. This suggests insufficient evidence to confidently claim a relationship between the program's binary sizes and their energy consumption. In contrast, since the Spearman p-value for ToyBox is close to the α significance level, we can conclude that the weak correlation between program size and energy consumption in ToyBox is almost statistically significant.

> ***RQ*$_{1.1}$ insights:** The findings of our study, which involved 28 programs, suggest a very weak correlation between a program's binary size and its energy consumption, indicating that other factors play a more significant role in determining energy consumption.

5.3 Number of Configuration Options Impact

Similarly, to assess the impact of configuration options on a program's energy consumption, we measured the correlation between the number of run-time configuration options (Figure 2, right) and its energy consumption given (Table 1, columns with StD).

The *Energy/Options* column in Table 2 present the Spearman calculated correlations for all programs in GNU, ToyBox, and BusyBox. These coefficients, representing the relationship between the number of run-time options and energy consumption, range from 0.271 to 0.461. Except in BusyBox, the coefficients for GNU and ToyBox show meaningful, though weak, relationships. These findings provide a contrast to the insights and conclusions drawn in Section 5.1.

Regarding statistical significance, we calculated the p-values with a significance level of $\alpha = 5\%$. The fifth column in Table 2 shows the obtained values. The p-values for GNU and ToyBox are below the significance level, indicating that the weak correlation in GNU and ToyBox are statistically significant. For BusyBox, the p-value exceeds α (with ▨), suggesting that while there is a correlation between the run-time options and energy consumption, the evidence is not strong enough to confirm a meaningful relationship.

> ***RQ*$_{1.2}$ insights:** Based on our findings, reducing the number of run-time configuration options in a program does not always result in lower energy consumption. However, in some cases, a weak but statistically significant correlation exists between the number of options and energy consumption. This suggests that other factors may play a stronger role in determining a program's energy consumption.

5.4 Impact on Execution Time

Due to the weak to negligible correlations observed between a program's binary size and energy consumption, as well as between the

Table 3: Six selected programs and their Bloated and Debloated binary sizes (B. size, D. size) and Energy Consumption ± Standard Deviation (StD) in joules [J], for `Chisel`, `Cov`, and `Debop`.

Program	B. size (KiB)	D. size (KiB)	`Bloated` ± StD	`Chisel` ± StD	Diff.%	`Cov` ± StD	Diff.%	`Debop` ± StD	Diff.%
date-8.21	92.03	33.54	0.288 ± 0.014	0.284 ± 0.010	-1.30	0.284 ± 0.009	-1.24	0.284 ± 0.010	-1.24
grep-2.4.2	158.83	93.42	0.317 ± 0.020	0.457 ± 0.021	44.22	0.315 ± 0.018	-0.58	0.315 ± 0.018	-0.53
gzip-1.3	101.71	66.72	0.335 ± 0.013	0.549 ± 0.022	64.03	0.332 ± 0.016	-0.87	0.332 ± 0.008	-0.91
mkdir-5.2.1	48.20	21.64	0.289 ± 0.016	0.288 ± 0.014	-0.50	0.288 ± 0.012	-0.63	0.288 ± 0.011	-0.67
printtokens2	20.68	20.68	0.282 ± 0.009	0.281 ± 0.012	-0.25	0.285 ± 0.021	1.11	0.282 ± 0.013	-0.08
sed-4.1.5	170.38	107.05	0.286 ± 0.010	0.289 ± 0.015	1.12	0.290 ± 0.022	1.69	0.287 ± 0.014	0.63

Lower energy consumption Higher energy consumption Non-significant difference on the consumed energy

number of run-time configuration options and energy consumption, we conducted another investigation, specifically exploring the correlation between a program's *execution time* and its *energy consumption*. Our aim was to determine whether execution time influences energy consumption during a program's execution and to evaluate its consistency under built-in feature reduction.

The *Energy/Exec Time* column in Table 2 shows the computed Spearman correlations for all three implementations of our 28 subject programs. They range from 0.891 to 0.966, indicating a consistently high or nearly perfect correlation between a program's execution time and its energy consumption. The last column in Table 2 also presents the associated p-values. All of them are between $1.02 \cdot 10^{-16}$ and $2.07 \cdot 10^{-10}$, significantly below the significance level or $\alpha = 5\%$. This indicates that the strong correlations between execution time and energy consumption are statistically significant, suggesting similar relationships may exist in other programs as well.

> **$RQ_{1.3}$ insights:** The relationship between energy consumption and execution time remains consistent despite built-in feature reduction. Furthermore, alternative implementations, such as `BusyBox` and `ToyBox`, exhibit a stronger correlation between energy consumption and execution time compared to their `GNU` counterparts.

6 Methodology of the Experiment: On-demand Feature Reduction and Energy Consumption

This section outlines our second experiment, conducted with software programs subjected to on-demand feature reduction.

6.1 Subject Systems

To establish a representative set of software with on-demand feature reduction, we utilized open-access artifacts from Xin et al. [54], which include debloated programs produced by three state-of-the-art debloating techniques [8]. These used techniques and programs are widely recognized as benchmarks in debloating studies (*e.g.*, [8, 30, 54]). Specifically, we selected 6 `GNU` programs debloated by `Chisel`, `Debop`, and `Cov` tools, focusing on those that we could successfully compile and execute with the provided training input and configuration. Notably, some of the compilation challenges we encountered are also highlighted in [54]. These tools employ debloating techniques that operate at the source code level. Each of them requires both a bloated program and a specification of desired or undesired features (*i.e.*, a usage profile with configuration options and an input) to produce debloated artifacts. This approach enables controlled and on-demand feature reduction.

To strengthen our experiment, not only we applied three debloating techniques, but we also selected the 6 programs similar to those in our initial experiment. This enables us to provide comparable results. Table 3 presents their original binary size (*i.e.*, before debloat) and reduced binary size (*i.e.*, after debloat) in Kibibyte (KiB).

6.2 Experiment Settings

For each of the 6 programs, we measured energy consumption and execution time in both bloated and debloated states.

We first extracted and compiled the source code of both the bloated and debloated variants of the 6 subject programs. Each program had three debloated variants, one produced by each of the debloating tools. In addition, multiple usage profiles (*i.e.*, configurations with inputs) were available. For each debloated variant we considered two different configurations, as in our first experiment (*i.e.*, same program, same debloating technique, but two different input profiles per program). Each profile (configuration and input) used in our study was selected based on two criteria. First, the profile that produced correct output for all debloated variants of the program. Secondly, similar to the first experiment, we used purposive sampling [32] to select two profiles that represent the program's typical usage based on its documentation.

Thus, we measured the energy consumption of the 6 bloated programs and their 18 debloated counterparts ($6 * 3$), following the same procedure and measurement setup described in Section 3.3. To prevent side effects, all experiments were sequentially run 10 times as the only processes on the workstation.

7 Results of On-demand Feature Reduction

In this section, we present the results and examine the findings related to built-in feature reduction research questions, $RQ_{2.1}$ and $RQ_{2.2}$, focusing on how binary size and execution time affect energy consumption.

7.1 Results on Energy Consumption

The columns labeled ... ± StD in Table 3 show the average energy consumption of software programs both before and after debloating. For example, the original `grep` (bloated) consumes 0.317 J, while its

debloated variant using the Cov tool consumes slightly less, about 0.315 J, equivalent to 0.03 seconds of LED light usage (10.5 W).

General insights. Table 3 shows that both bloated and debloated programs consume less than 1 J, which is consistent with the trend observed in our initial experiment with programs featuring built-in feature reduction. The small standard deviation across our 10 repetitions suggests that the energy consumption values are consistent around the average for all bloated and debloated programs.

Comparative analysis. Similarly, we performed a comparative analysis to determine if the initial experiment's findings hold for programs with on-demand feature reduction. Specifically, we compared the binary size and energy consumption of programs before and after debloating. The columns labeled *Diff .%* in Table 3 show the percentage change in energy consumption for debloated programs, compared to their bloated variants, which serve as the baseline. This shows whether energy consumption increased or decreased after debloating. The statistical significance of the difference is also computed. A program's difference is highlighted with a red diagonal line pattern (▨) if it lacks statistical significance in either of the two scenarios, as determined by the Mann-Whitney U test [31].

All debloated programs have smaller binary size. However, as in the initial experiment, the majority of differences in energy consumption are not statistically significant. Specifically, 4 out of 6 programs debloated with Chisel, and all programs debloated with Cov and Debop, show no significant differences (▨). Only 2 out of 6 programs debloated with Chisel demonstrate significant differences (■), with energy consumption increased by ≈ 44% and ≈ 64%, respectively. Moreover, no debloated program consumes *significantly* less energy (■) than its bloated counterparts. This reinforces our conclusion from the initial experiment that reducing a program's binary size through debloating does not necessarily lead to lower energy consumption. The differences observed are largely non-significant, and when significant, they tend towards higher energy consumption, as are the two cases with Chisel.

Similar to the first experiment, we calculated the correlation using Spearman coefficients [24]. This analysis included all 6 programs, both bloated and debloated variants of them, with a focus on the correlations between binary size and average energy consumption. We found that the correlation between binary size and energy consumption in bloated programs was 0.314, indicating a weak relationship. In contrast, the debloated programs showed a moderate correlation of 0.60, as seen in the *Energy/Binary Size* column in Table 4. However, the p-values for all correlations exceed the significance threshold of $\alpha = 0.05$, suggesting that the observed correlations are not statistically significant. This indicates that there is not enough evidence to show a meaningful relationship between binary size and energy consumption in the debloated programs.

> ***RQ*$_{2.1}$ insights:** Debloating techniques, aimed at reducing a program's binary size, do not consistently lower energy consumption. In most cases, the differences in energy usage between debloated and bloated variants are either statistically insignificant or, occasionally, higher in the debloated variants.

Table 4: Spearman correlation of execution time and binary size with energy consumption. ▨ No significant correlation

	Energy/Exec Time		Energy/Binary Size	
	Spearman	p-value	Spearman	p-value
bloated	1.000	0.000	0.314	0.544
Chisel	1.000	0.000	0.600	0.208
Cov	0.886	0.019	0.600	0.208
Debop	0.943	0.005	0.600	0.208

7.2 Execution Time Impact

Due to the weak correlation between binary size and energy consumption in both bloated and debloated programs, we extended our investigation to explore whether execution time exhibits a stronger correlation, as observed in our initial experiment. The *Energy/Exec Time* column in Table 4 shows that the correlations computed for the 6 debloated subjects by three tools fall between 0.886 and 1.00. In bloated programs, the correlation between execution time and energy consumption was 1.00, signifying a perfect relationship. For debloated programs, the results demonstrate a high or nearly perfect correlation between execution time and energy consumption. Additionally, the p-values listed in the same column in Table 4 are significantly below the $\alpha = 5\%$ threshold. This indicates that the strong observed correlation between energy consumption and execution time in debloated programs is statistically significant, suggesting that it may also hold true for other software programs.

> ***RQ*$_{2.2}$ insights:** Similar to the first experiment, there is a strong and consistent correlation between execution time and energy consumption. This indicates that energy consumption in software is closely linked to the program's execution time. Regardless of debloating, optimizing execution time emerges as a critical factor in achieving energy-efficient software.

8 In-Depth Analysis, Discussion, and Insights

8.1 Summary of Key Findings

Our study reveals that feature reduction does not consistently lead to lower energy consumption. In most cases, the impact of feature reduction on energy consumption were statistically insignificant. Specifically, in built-in alternatives, 4 out of 56 cases showed a significant reduction, 14 a significant increase, and 38 had no significant difference. In debloated programs, 2 out of 18 cases showed a significant increase in energy consumption, while 16 had no significant change. Basically, when differences were significant, they tended to result in increased energy consumption.

We found that there is a consistent and strong correlation between energy consumption and execution time across both types of reduced programs. This suggests that reducing execution time is an effective strategy for improving energy efficiency, regardless of feature reduction. Moreover, our findings align with a recent comparative evaluation of software debloating tools [8], which reports that debloating techniques often have no significant impact on execution time, and when they do, the effect tends to be negative, consistent with our observations regarding energy consumption.

```
grep-2.4.2 - prtext()
// ...
if (! out_quiet) {
        if (pending > 0) {
                prpending(beg);
  }}  // ...
```

```
gzip-1.3 - ct_init()
// ...
if ((int) static_dtree[0].dl.len != 0) {
        return;
}
// ...
```

Listing 2: Optimizing code removal by `Chisel`: LOC deleted but executed in bloated, LOC newly executed in debloated.

Debloating approaches leverage feature reduction to optimize non-functional properties, generally binary size or attack surface. However, this mono-objective optimization can negatively affect other non-functional properties. For instance, debloating with a focus on reducing binary size may increase the attack surface [53].

8.2 Unintended Energy Impacts of Debloating

In light of these findings, it is essential to investigate why some debloating techniques inadvertently lead to increased energy consumption. One plausible explanation is that when debloating focuses solely on reducing binary size or attack surface, it might also eliminate code segments that contribute to energy efficiency. To explore this, we analyzed in depth the executed code of the two debloated programs with significant increases: `grep-2.4.2` and `gzip-1.3` (*cf.* marked with in Table 3). Both were debloated using `Chisel`, a mono-objective debloating tool focusing on reducing binary size. To understand this difference, we analyzed the execution coverage using `gcov` [11] for both bloated and debloated variants of these programs. We observed that there is a reduction in executed lines of code for `grep` (−10.5%) and `gzip` (−11.4%) in the debloated variants compared to their bloated counterparts. This reduction seems counterintuitive given the increased energy consumption. But, aligning code coverage reports with code *diffs* revealed that `Chisel` removed certain optimizing code segments.

Specifically, in `grep-2.4.2` (Listing 2), we observed that removing a conditional linked to the intentionally debloated `--quiet` option inadvertently caused additional code execution. For instance, line 4, which previously prevented some unnecessary function calls, was correctly removed with the removed option, but this resulted in line 5 being executed 76 times in the debloated program variant. Whereas, in `gzip-1.3` (Listing 2), an early return removal caused also some unnecessary computation. While the method was invoked fewer times in the debloated variant (once versus seven), we think that the lack of early return logic could lead to inefficiency in different contexts, negatively affecting energy consumption.

These observations show the need for an in-depth investigation into the impact of removing code snippets which may play a role in optimizing energy consumption. Additional cases that we found are provided in the reproduction package. An interesting hypothesis to explore is how debloating techniques, which aim to reduce binary size, may unintentionally remove optimizing code, leading to increased energy consumption despite reduced code execution.

8.3 ecv: Energy Consumption Visualizer

Both built-in and on-demand feature reduction are generally not conducted with the aim to optimize energy consumption. Thus,

[11] Code coverage analysis tool - gcov: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

```
==> Command ls -la == Max (µJ): 35950 | Min (µJ): 17150 |
 Mean (x5) config (µJ): 22290.0
(Less) ■■■■■■■■□□□□□□□□□□□□□□□□□□□□□□□□ (More)
0.004206 seconds of HDD use (5.3W)
0.002123 seconds of LED light bulb use (10.5W)
7e-05 seconds of GPU use (320W)
```

Figure 3: Output of our tool for the command "`ls -la`" showing energy consumption measured over 5 repetitions.

providing information on energy consumption to developers is crucial when selecting an alternative implementation or using a debloated tool. To address this, we developed a tool called **ecv** [12]. It uses progress bars to visually represent a software program's energy consumption. Additionally, below the progress bar, **ecv** provides information about how long the program's energy consumption could power familiar devices like HDDs, GPUs, or LED light bulbs. This helps developers easily understand the program's energy usage and how different configurations affect it. Listing 3 demonstrates how to measure energy consumption of the `"ls -la"` command using the `--measure` option of **ecv**, with five repetitions specified by the `--repeat 5` option. Figure 3 displays the corresponding output, including details on how long the measured energy could power three common electronics.

```
$ ./ecv --command "ls -la" --measure --repeat 5
```

Listing 3: Measures the energy consumption of `ls`, 5 rept.

ecv offers two usage scenarios: one for developers and one for end-users. For developers working on configurable systems, the tool helps measure and track their programs' energy consumption as the software evolves. On the other hand, end-users can use **ecv** to compare alternative programs based on their energy consumption, as demonstrated in our initial experiment. We believe that this enables users to make more informed decisions, particularly when choosing between alternative program implementations. Furthermore, **ecv** is open source and available for evaluation. Its repository includes additional usage examples to explore its capabilities.

8.4 Towards Energy-Aware Debloating

Our exploratory study shows the promising potential of debloating techniques to reduce energy consumption, while also revealing the challenges to fully realize this goal. While it may seem intuitive that removing code, potentially executed but associated with non-necessary feature for a given profile would reduce energy consumption, our observations suggest the contrary. We identify snippets where debloating tools removes potential optimizing code in term of energy efficiency, opening the way for approach preserving these codes. Multi-objective approaches, such as the one proposed by `Debop`, which simultaneously optimizes for binary size and attack surface, represent a promising direction for further exploration. Researchers might explore and investigate the configuration space and its relationship with energy consumption, either to predict execution time or to identify the most promising configuration options to debloat when considering energy.

Finally, researchers can also leverage **ecv** to identify among the alternative programs those with the lowest energy consumption

[12] Available with a video demonstration: https://anonymous.4open.science/r/ecv-5E6E

and further investigate the reasons behind it, beyond binary size and configuration options. For example, energy efficiency could be influenced by the libraries used, or the algorithms employed.

9 Threats to Validity

Internal threats. In the initial experiment, energy consumption measurements were based on two configurations for each program. One potential threat is the *selection bias*, meaning the results may have been influenced by the specific configurations chosen. However, we opted for the most commonly used configurations to ensure that the measured energy consumption reflected typical usage for these programs. Another potential threat is choice of input data for each program, as input can affect energy consumption. We mitigated this by selecting representative inputs and keeping them constant across similar programs. In future work, we plan to expand our investigation by varying inputs and configurations to explore their impact on energy consumption. Finally, we had to compile both the bloated and particularly the debloated program variants provided by Xin et al. [54]. Many of the debloated programs failed to compile without segmentation faults (segfaults), so we included only those that worked to ensure the results were unbiased.

External threats. Our experiments focus on a specific set of programs, namely `GNU`, `ToyBox`, and `BusyBox` programs written in C. Although these programs are widely used in evaluations of debloating approaches and other studies, recent research has shown that energy consumption can vary significantly between programming languages [15, 26, 40]. Thus, we cannot generalize our findings to feature reduction in configurable software systems implemented in other languages or to systems with medium-to-large binary sizes and configuration spaces, such as Linux or Chromium [6, 42].

10 Related Work

This work intersects three research fields, namely configurable software systems, software bloat, and green computing.

Configurable software systems and debloating. Several studies have investigated debloating techniques on configurable systems. Ahmad et al. [3] proposed an approach to debloat code based on user-provided command line arguments and application-specific configuration files. Sharif et al. [46] developed an approach that leverages user-provided configuration data to specialize an application to its deployment context. Heo et al. [19] improved on previous work by using a novel reinforcement learning based approach to accelerate the search for debloated program and scale to large applications. Xin et al. [53] proposed a general multi-objective optimization approach for debloating. Xin et al. [54] studied the trade-offs between generality and reduction in software debloating. Tërnava et al. [49] examined how the run-time configuration space affects binary size, attack surface, and execution time. Moreover, two recent surveys by Alhanahnah et al. [4] and Brown et al. [8] provide an overview of existing debloating approaches and tools developed over the past two decades. However, none of these studies has explored the impact of feature reduction on energy consumption.

Software product lines and green computing. Some studies measure the correlation between system execution time and energy consumption, focusing on adjusting rather than removing runtime configuration options [52]. In our case, we aim to reduce binary size by debloating and observe the system's footprint and its correlation with energy consumption. Sahin et al. [44] have investigated how different types of refactoring impact energy consumption, using hardware instrumented measurements. They demonstrate that refactoring significantly affects energy consumption, though this effect can either increase or decrease energy usage depending on the application and execution platform. Guégain et al. [17] have explored energy consumption in software product lines, proposing a method to measure energy usage in such contexts. Using a T-wise algorithm, they produce 602 configurations of BerkeleyDB-SPL and measured the consumption for each, identifying the most energy-efficient ones. With this method, Guégain et al. [17] identified some configurations with a reduction in consumption of at least 40%. While their approach is based on SPL, it does not take into consideration feature reduction. Islam et al. [22] measured the energy consumption of individual features by slicing the source code associated with each feature, then monitoring energy consumption. This approach works well for individual features but does not consider interactions between them. Other works are focused on power consumption of specific code patterns or "energy hotspots" (*e.g.*, [33, 39]). In 2015, Noureddine et al. [34] proposed a method for identifying and evaluating high energy consuming spots in Java source code.Pereira et al. [39] developed a method to locate inefficient source code fragments, demonstrating that developers using their technique improved energy efficiency by 43% on average. Like Islam et al. [22], these two approaches do not take the interactions between features into account. To the best of our knowledge, our study is the first to investigate the effect of feature reduction on energy consumption of configurable systems.

11 Conclusion and Perspectives

This paper presented a novel exploratory study on the effect of built-in and on-demand feature reduction of configurable systems on energy consumption. We found weak and mostly no significant correlations between energy consumption and factors such as binary size or the number of configuration options in software systems. This suggests that smaller binaries and fewer options do not necessarily lead to lower energy consumption. Interestingly, a given feature in a built-in reduced software system, similarly in an on-demand debloated software system, is more likely consuming more than in the original configurable software system. However, execution time showed a strong and significant link to energy consumption, indicating that its optimization is the key to making debloated software more efficient in terms of energy.

While traditional debloating techniques aim to remove unnecessary features to minimize binary size to reduce the attack surface, our study provides evidence for a new research direction: developing debloating strategies that prioritize energy efficiency and execution time, shifting the focus from size and security to sustainability and performance. In addition, analyzing long-term energy consumption in real-world usage, such as widely deployed tools like `coreutils`, can identify patterns and usage contexts where energy costs are highest. These insights can directly inform and motivate energy-aware debloating techniques by focusing on features with most energy-intensive in practice and therefore promising candidates for removal or optimization.

References

[1] Mathieu Acher, Hugo Martin, Luc Lesoil, Arnaud Blouin, Jean-Marc Jézéquel, Djamel Eddine Khelladi, Olivier Barais, and Juliana Alves Pereira. 2022. Feature Subset Selection for Learning Huge Configuration Spaces: The Case of Linux Kernel Size. In *Proceedings of the 26th ACM International Systems and Software Product Line Conference - Volume A* (Graz, Austria) *(SPLC '22)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/3546932.3546997

[2] Bilge Acun, Phil Miller, and Laxmikant V. Kale. 2016. Variation Among Processors Under Turbo Boost in HPC Systems. In *Proceedings of the 2016 International Conference on Supercomputing* (Istanbul, Turkey) *(ICS '16)*. Association for Computing Machinery, New York, NY, USA, Article 6, 12 pages. https://doi.org/10.1145/2925426.2926289

[3] Aatira Anum Ahmad, Abdul Rafae Noor, Hashim Sharif, Usama Hameed, Shoaib Asif, Mubashir Anwar, Ashish Gehani, Fareed Zaffar, and Junaid Haroon Siddiqui. 2022. Trimmer: An Automated System for Configuration-Based Software Debloating. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3485–3505. https://doi.org/10.1109/TSE.2021.3095716

[4] Mohannad Alhanahnah, Yazan Boshmaf, and Ashish Gehani. 2024. SoK: Software Debloating Landscape and Future Directions. In *Proceedings of the 2024 Workshop on Forming an Ecosystem Around Software Transformation* (Salt Lake City, UT, USA) *(FEAST '24)*. Association for Computing Machinery, New York, NY, USA, 11–18. https://doi.org/10.1145/3689937.3695792

[5] Peter Bambazek, Iris Groher, and Norbert Seyff. 2022. Sustainability in Agile Software Development: A Survey Study among Practitioners. In *2022 International Conference on ICT for Sustainability (ICT4S)*. 13–23. https://doi.org/10.1109/ICT4S55073.2022.00013

[6] Peter Beverloo. 2020-08-12. List of Chromium Command Line Switches. https://peter.sh/experiments/chromium-command-line-switches/.

[7] Aurélien Bourdon, Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. 2013. PowerAPI: A Software Library to Monitor the Energy Consumed at the Process-Level. *ERCIM News* 92 (Jan. 2013), 43–44. https://inria.hal.science/hal-00772454

[8] Michael D. Brown, Adam Meily, Brian Fairservice, Akshay Sood, Jonathan Dorn, Eric Kilmer, and Ronald Eytchison. 2024. A Broad Comparative Evaluation of Software Debloating Tools. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3927–3943. https://www.usenix.org/conference/usenixsecurity24/presentation/brown

[9] Shaiful Alam Chowdhury and Abram Hindle. 2016. GreenOracle: Estimating Software Energy Consumption With Energy Measurement Corpora. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2901739.2901763

[10] Marco Couto, Paulo Borba, Jácome Cunha, João Paulo Fernandes, Rui Pereira, and João Saraiva. 2017. Products go Green: Worst-Case Energy Consumption in Software Product Lines. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A* (Sevilla, Spain) *(SPLC '17)*. Association for Computing Machinery, New York, NY, USA, 84–93. https://doi.org/10.1145/3106195.3106214

[11] Marco Couto, João Paulo Fernandes, and João Saraiva. 2021. Statically analyzing the energy efficiency of software product lines. *Journal of Low Power Electronics and Applications* 11, 1 (2021), 13. https://doi.org/10.3390/jlpea11010013

[12] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design* (Austin, Texas, USA) *(ISLPED '10)*. Association for Computing Machinery, New York, NY, USA, 189–194. https://doi.org/10.1145/1840845.1840883

[13] João De Macedo, Rui Abreu, Rui Pereira, and João Saraiva. 2022. WebAssembly versus JavaScript: Energy and Runtime Performance. In *2022 International Conference on ICT for Sustainability (ICT4S)*. 24–34. https://doi.org/10.1109/ICT4S55073.2022.00014

[14] Gerhard Fettweis and Ernesto Zimmermann. 2008. ICT: Energy Consumption-Trends and Challenges. In *Proceedings of the 11th International Symposium on Wireless Personal Multimedia Communications*, Vol. 2. Lapland, Finland, 6. https://api.semanticscholar.org/CorpusID:9129930

[15] Stefanos Georgiou, Maria Kechagia, and Diomidis Spinellis. 2017. Analyzing Programming Languages' Energy Consumption: An Empirical Study. In *Proceedings of the 21st Pan-Hellenic Conference on Informatics* (Larissa, Greece) *(PCI '17)*. Association for Computing Machinery, New York, NY, USA, Article 42, 6 pages. https://doi.org/10.1145/3139367.3139418

[16] GNU. [n. d.]. GNU Core Utilities. https://www.gnu.org/software/coreutils/

[17] Édouard Guégain, Clément Quinton, and Romain Rouvoy. 2021. On Reducing the Energy Consumption of Software Product Lines. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference - Volume A* (Leicester, United Kingdom) *(SPLC '21)*. Association for Computing Machinery, New York, NY, USA, 89–99. https://doi.org/10.1145/3461001.3471142

[18] María Gutiérrez, Ma Ángeles Moraga, and Félix García. 2022. Analysing the Energy Impact of Different Optimisations for Machine Learning Models. In *2022 International Conference on ICT for Sustainability (ICT4S)*. 46–52. https://doi.org/10.1109/ICT4S55073.2022.00016

[19] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

[20] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. 2014. GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 12–21. https://doi.org/10.1145/2597073.2597097

[21] Gerard J. Holzmann. 2015. Code Inflation. http://spinroot.com/gerard/pdf/Code_Inflation.pdf. Accessed: 2025-04-20.

[22] Syed Islam, Adel Noureddine, and Rabih Bashroush. 2016. Measuring Energy Footprint of Software Features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. 1–4. https://doi.org/10.1109/ICPC.2016.7503726

[23] Mathilde Jay, Vladimir Ostapenco, Laurent Lefevre, Denis Trystram, Anne-Cécile Orgerie, and Benjamin Fichel. 2023. An Experimental Comparison of Software-Based Power Meters: Focus on CPU and GPU. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. 106–118. https://doi.org/10.1109/CCGrid57682.2023.00020

[24] Maurice George Kendall. 1948. Rank Correlation Methods. (1948).

[25] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3, 2, Article 9 (March 2018), 26 pages. https://doi.org/10.1145/3177754

[26] Lukas Koedijk and Ana Oprescu. 2022. Finding Significant Differences in the Energy Consumption when Comparing Programming Languages and Programs. In *2022 International Conference on ICT for Sustainability (ICT4S)*. 1–12. https://doi.org/10.1109/ICT4S55073.2022.00012

[27] Masanari Kondo, Cor-Paul Bezemer, Yasutaka Kamei, Ahmed E Hassan, and Osamu Mizuno. 2019. The Impact of Feature Reduction Techniques on Defect Prediction Models. *Empirical Software Engineering* 24 (2019), 1925–1963. https://doi.org/10.1007/s10664-018-9679-5

[28] Elias Kuiter, Chico Sundermann, Thomas Thüm, Tobias Hess, Sebastian Krieter, and Gunter Saake. 2025. How Configurable is the Linux Kernel? Analyzing Two Decades of Feature-Model History. *ACM Trans. Softw. Eng. Methodol.* (April 2025). https://doi.org/10.1145/3729423 Just Accepted.

[29] Robert Landley and et. al. [n. d.]. ToyBox. http://landley.net/toybox/about.html

[30] Bo Lin, Shangwen Wang, Yihao Qin, Liqian Chen, and Xiaoguang Mao. 2025. Large Language Models-Aided Program Debloating. *arXiv preprint arXiv:2503.08969* (2025). https://doi.org/10.48550/arXiv.2503.08969

[31] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. http://www.jstor.org/stable/2236101

[32] B Mattew Miles and Michael A Huberman. 1994. *An Expanded Sourcebook: Qualitative Data Analysis*. Sage publications.

[33] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. 2013. A Review of Energy Measurement Approaches. *SIGOPS Oper. Syst. Rev.* 47, 3 (Nov. 2013), 42–49. https://doi.org/10.1145/2553070.2553077

[34] Adel Noureddine, Romain Rouvoy, and Lionel Seinturier. 2015. Monitoring Energy Hotspots in Software: Energy Profiling of Software Code. *Automated Software Engineering* 22 (2015), 291–332. https://doi.org/10.1007/s10515-014-0171-1

[35] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, and Joël Penhoat. 2021. Evaluating the Impact of Java Virtual Machines on Energy Consumption. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '21)*. Association for Computing Machinery, New York, NY, USA, Article 15, 11 pages. https://doi.org/10.1145/3475716.3475774

[36] Zakaria Ournani, Mohammed Chakib Belgaid, Romain Rouvoy, Pierre Rust, Joel Penhoat, and Lionel Seinturier. 2020. Taming Energy Consumption Variations In Systems Benchmarking. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 36–47. https://doi.org/10.1145/3358960.3379142

[37] Zakaria Ournani, Romain Rouvoy, Pierre Rust, and Joel Penhoat. 2020. On Reducing the Energy Consumption of Software: From Hurdles to Requirements. In *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)* (Bari, Italy) *(ESEM '20)*. Association for Computing Machinery, New York, NY, USA, Article 14, 12 pages. https://doi.org/10.1145/3382494.3410678

[38] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. 2016. What Do Programmers Know about Software Energy Consumption? *IEEE Software* 33, 3 (2016), 83–89. https://doi.org/10.1109/MS.2015.83

[39] Rui Pereira, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2020. SPELLing Out Energy Leaks: Aiding Developers Locate

Energy Inefficient Code. *Journal of Systems and Software* 161 (2020), 110463. https://doi.org/10.1016/j.jss.2019.110463

[40] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. 2021. Ranking Programming Languages by Energy Efficiency. *Science of Computer Programming* 205 (2021), 102609. https://doi.org/10.1016/j.scico.2021.102609

[41] Gustavo Pinto, Fernando Castor, and Yu David Liu. 2014. Mining Questions About Software Energy Consumption *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 22–31. https://doi.org/10.1145/2597073.2597110

[42] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: Debloating the Chromium Browser with Feature Subsetting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 461–476. https://doi.org/10.1145/3372297.3417866

[43] Stephen Romansky, Neil C. Borle, Shaiful Chowdhury, Abram Hindle, and Russ Greiner. 2017. Deep Green: Modelling Time-Series of Software Energy Consumption. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 273–283. https://doi.org/10.1109/ICSME.2017.79

[44] Cagri Sahin, Lori Pollock, and James Clause. 2014. How do Code Refactorings Affect Energy Usage?. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (Torino, Italy) *(ESEM '14)*. Association for Computing Machinery, New York, NY, USA, Article 36, 10 pages. https://doi.org/10.1145/2652524.2652538

[45] Simon Schubert, Dejan Kostic, Willy Zwaenepoel, and Kang G. Shin. 2012. Profiling Software for Energy Consumption. In *2012 IEEE International Conference on Green Computing and Communications*. 515–522. https://doi.org/10.1109/GreenCom.2012.86

[46] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 329–339. https://doi.org/10.1145/3238147.3238160

[47] Shivkumar Shivaji, E. James Whitehead, Ram Akella, and Sunghun Kim. 2013. Reducing Features to Improve Code Change-Based Bug Prediction. *IEEE Transactions on Software Engineering* 39, 4 (2013), 552–569. https://doi.org/10.1109/TSE.2012.43

[48] Shivkumar Shivaji, E. James Whitehead, Ram Akella, and Sunghun Kim. 2009. Reducing Features to Improve Bug Prediction. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 600–604. https://doi.org/10.1109/ASE.2009.76

[49] Xhevahire Tërnava, Mathieu Acher, and Benoit Combemale. 2023. Specialization of Run-time Configuration Space at Compile-time: An Exploratory Study. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing* (Tallinn, Estonia) *(SAC '23)*. Association for Computing Machinery, New York, NY, USA, 1459–1468. https://doi.org/10.1145/3555776.3578613

[50] Roberto Verdecchia, Luís Cruz, June Sallou, Michelle Lin, James Wickenden, and Estelle Hotellier. 2022. Data-Centric Green AI An Exploratory Empirical Study. In *2022 International Conference on ICT for Sustainability (ICT4S)*. 35–45. https://doi.org/10.1109/ICT4S55073.2022.00015

[51] Denys Vlasenko and et. al. [n. d.]. BusyBox. https://busybox.net/

[52] Max Weber, Christian Kaltenecker, Florian Sattler, Sven Apel, and Norbert Siegmund. 2023. Twins or False Friends? A Study on Energy Consumption and Performance of Configurable Software. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2098–2110. https://doi.org/10.1109/ICSE48619.2023.00177

[53] Qi Xin, Myeongsoo Kim, Qirun Zhang, and Alessandro Orso. 2020. Program Debloating via Stochastic Optimization. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (Seoul, South Korea) *(ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 65–68. https://doi.org/10.1145/3377816.3381739

[54] Qi Xin, Qirun Zhang, and Alessandro Orso. 2023. Studying and Understanding the Tradeoffs Between Generality and Reduction in Software Debloating. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 99, 13 pages. https://doi.org/10.1145/3551349.3556970