Sustainable Computing: Informatics and Systems 47 (2025) 101166

Contents lists available at ScienceDirect

Sustainable Computing: Informatics and Systems

journal homepage: www.elsevier.com/locate/suscom

Does faster mean greener? Runtime and energy trade-offs in iOS applications with compiler optimizations

José Miguel Aragón-Jurado [a,*], Abdul Ali Bangash [b], Bernabé Dorronsoro [a], Karim Ali [c], Abram Hindle [d], Patricia Ruiz [a]

[a] School of Engineering, University of Cadiz, Puerto Real, Spain
[b] School of Computing, Queen's University, Kingston, Canada
[c] NYU Abu Dhabi, Abu Dhabi, United Arab Emirates
[d] Department of Computing Science, University of Alberta, Edmonton, Canada

ARTICLE INFO

Dataset link: https://doi.org/10.5281/zenodo.14888155

Keywords:
Green software
Energy consumption
Software performance
iOS development
Compiler optimization
Smartphone applications

ABSTRACT

Smartphones outnumber people nowadays, requiring efficient energy management. High application energy use leads to faster battery drain and frequent recharging, negatively impacting both battery life and the environment. This cycle also contributes to rising electronic and chemical waste due to discarded mobile phone batteries. Compiler optimization flags may play a crucial role in mitigating these issues by optimizing software performance. However, there has been little research on examining how compiler optimization flags impact the energy consumption of smartphone applications. This work presents an empirical study on the effect of the most aggressive iOS compiler optimizations on runtime, power consumption, and energy consumption across six different iOS applications. For each application, we developed a benchmark focused on the specified category we aimed to study. Our results show that reducing application runtime does not always directly correlate with improved energy consumption. In fact, we observed that optimizations aimed at enhancing runtime performance often come at an energy cost in the applications studied, highlighting a trade-off between runtime and energy consumption. For example, we found that using `-Ounchecked` in Swift, combined with `-Oz` from LLVM in the GhostRun video game, increases energy consumption by 34%, despite improving runtime performance by 9%.

1. Introduction

The United Nations' 2030 Agenda for Sustainable Development outlines a set of goals aimed at creating a sustainable future by reducing the carbon footprint [1]. A vital component in improving the sustainability of IT systems is green software. Green software is defined as software that has been optimized to minimize natural resource consumption throughout its entire life cycle [2]. This includes efforts to reduce both direct and indirect use of natural resources during its development, deployment, and usage, with continuous monitoring of these aspects.

Today, smartphones are ubiquitous, highly accessible, and essential to the daily lives of most people. For these battery-powered devices, managing the energy consumption of various applications is crucial. High energy consumption by applications reduces battery autonomy and increases the frequency of recharging, which accelerates battery degradation, leading users to replace batteries and devices more often, thereby contributing to electronic and chemical waste. The disposal of batteries and devices poses serious environmental issues, as hazardous materials can contaminate soil and water. Moreover, both manufacturing and disposal processes increase carbon emissions and deplete valuable resources.

To automatically enhance software performance, primarily focusing on runtime, compilers have historically applied a predefined sequence of code transformations. In C compilers, default optimization flags like `-O1`, `-O2`, or `-O3` are designed to reduce runtime but often lead to increased power consumption in embedded devices [3]. However, the improvement in runtime can be significant enough that it ultimately reduces overall energy consumption, despite the increase in power usage.

The impact of compiler optimizations is highly dependent on the specific software to be optimized and the hardware device where it will run [4]. Therefore, given the wide availability of different programming languages and compilers, it is necessary to study the

* Corresponding author.
E-mail address: josemiguel.aragon@uca.es (J.M. Aragón-Jurado).

https://doi.org/10.1016/j.suscom.2025.101166
Received 11 March 2025; Received in revised form 14 June 2025; Accepted 8 July 2025
Available online 21 July 2025
2210-5379/© 2025 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

impact of various generic optimizations on both runtime and energy consumption.

Apple [5], a leading smartphone company, equips all its models with the same operating system, iOS [6]. Applications for iOS are typically developed using the Swift programming language [7], which provides various generic optimization flags for different stages of the compilation process [8]. While several recent studies have explored how different services impact the energy consumption of iOS applications [9,10], none have analyzed the effect of these optimizations on application sustainability.

In this work, we empirically analyze the combined impact of Swift and LLVM [11] compilation optimization flags on the runtime and energy consumption of iOS applications. Both of these compiler optimizations can be easily applied during the application development process for iOS in Xcode [12]. This study aims to address the following research questions:

RQ1. Do the most aggressive optimization flags improve the runtime of the iOS applications?

RQ2. Do the most aggressive optimization flags improve the energy consumption of the iOS applications?

RQ3. Does an improvement in runtime correlate with an improvement in energy consumption for the iOS applications?

For this purpose, we have selected six applications across three different categories: *(1)* Disk operations, *(2)* Deep learning models, and *(3)* Video games. Moreover, we analyzed the impact of nine different compiler flag combinations for both Swift and LLVM on each of the applications examined.

This work presents several contributions. First of all, we develop various benchmarks for the different types of applications considered. Next, we empirically analyze the energy consumption, runtime, and power consumption of the selected applications, both with and without the selected optimization flags. Finally, we conduct a statistical analysis to evaluate the impact of these flags on energy consumption, runtime, and power consumption.

The document is organized as follows: Section 2 describes the most relevant techniques for measuring and optimizing energy consumption in software from the existing literature. Next, Section 3 provides an analysis and review of relevant studies focused on the empirical investigation of energy consumption in mobile applications. Section 4 presents the research methodology employed, while Section 5 details the different applications examined. Sections 6 and 7 present and discuss the results obtained. Finally, Section 8 addresses potential threats to the validity of the work, and Section 9 offers concluding remarks along with suggestions for future research directions.

2. Background

First, the most relevant concepts and methods for measuring software energy consumption are described in Section 2.1, with their respective classification. Following this, Section 2.2 details various metrics from the literature to analyze software greenness performance.

2.1. Measuring software energy consumption

Accurate measurements are crucial to minimize the residual energy consumption of software, which is influenced by various factors and exhibits significant variability. This inherent complexity makes it challenging to obtain reliable measurements. In consequence, the literature presents diverse methods for measuring the energy consumption of software programs, which are generally categorized into two primary approaches: measurement-based and model-based methods [13].

Energy profiling tools based on measurements, which are hardware-based approaches, employ current meters to capture the energy used by a computing device when executing the software. These tools typically consist of three fundamental components: the Device Under Test (DUT), the target device being measured; a measuring device that captures current readings from the DUT; and a workstation that manages data collection and processing [13]. Several studies in the literature utilize commercial power monitors, with the Monsoon Power Monitor standing out for its high sampling rate and ability to power devices directly [14].

Another approach to measuring energy consumption involves using a shunt resistor, often combined with sensors or microcontrollers, such as the INA219 sensor in GreenMiner [15] or the analog-to-digital converter in Arduino [16]. This method can be particularly useful for portable devices, where some researchers recommend removing the battery to connect an external power source in order to minimize external factors [15]. However, other researchers argue that keeping the battery connected ensures real-world accuracy in measurements [17].

Energy profiling techniques based on models estimate software energy usage on hardware, reducing the need for extensive hardware resources and time but sometimes sacrificing accuracy [18]. These models require hardware-specific calibration [19], done either online [20] (using energy data from the operating system) or offline [18] (using external measurement systems). There are three main categories of energy profiling models:

- *Utilization-based models*. These models establish a relationship between hardware usage and energy consumption, often assuming a linear correlation [21]. Data is collected from hardware components and used to train predictive models [22,23].
- *Event-based models*. These models track asynchronous hardware behaviors, like system calls, to capture complex energy patterns, especially for components that exhibit tail power states, maintaining high consumption after activity [24–26].
- *Code-analysis-based models*. These models analyze software code to predict energy use. Static models analyze code without running it [27,28], while dynamic models run the software on actual hardware, providing more accurate estimates by considering real-time behaviors [29,30].

Our work utilizes an updated version of the iGreenMiner [9,10] hardware measurement system for iOS devices. This updated version facilitates the optimization and automatic deployment of applications on iOS devices, while synchronizing measurements through CPU patterns. Therefore, it addresses the synchronization issues commonly found in several existing measurement tools.

2.2. Green up, power up, and speed up (GPS-UP) metrics and categories

To accurately assess the sustainability improvements of the optimized software, Abdulsalam et al. proposed the Green up, Power up, and Speed up (GPS-UP) metrics [31]. Before introducing these metrics, it is useful to recall the fundamental distinction between energy E (measured in joules, J) and power P (measured in watts, W). The relationship between them is given by:

$$P = \frac{E}{T} \ , \tag{1}$$

where, T represents the time over which the energy E is consumed or delivered.

Given two versions of the software, one optimized and one baseline (non-optimized), the following metrics are defined:

- *Speed up*. This metric measures the performance improvement of the optimized version and is calculated as:

$$Speedup(S_{up}) = \frac{T_{\phi}}{T_o} \ , \tag{2}$$

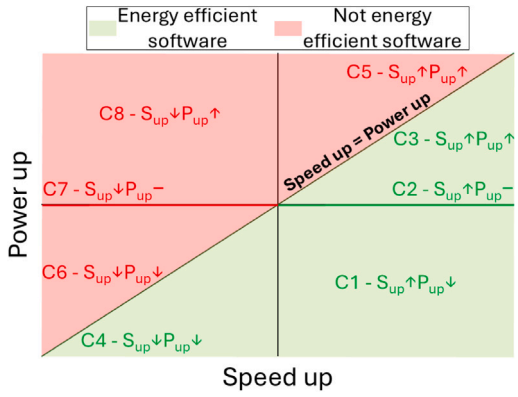


Fig. 1. GPS-UP software energy efficiency quadrant chart [31]. Green sections indicate energy savings; red sections indicate energy loss relative to the baseline. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

where T_ϕ represents the runtime of the non-optimized version, and T_o denotes the runtime of the optimized version. For example, a speed up value of 2 indicates that the optimized version runs twice as fast as the baseline.

- *Green up*. This metric evaluates the energy efficiency of the optimized version and is given by:

$$Greenup(G_{up}) = \frac{E_\phi}{E_o} \quad , \tag{3}$$

where E_ϕ is the energy consumption of the non-optimized version, and E_o is the energy consumption of the optimized version. For example, a green up value of 0.5 indicates that the optimized version consumes twice as much energy as the baseline.

- *Power up*. This metric assesses the impact of the optimized version on overall power consumption and is defined as:

$$Powerup(P_{up}) = \frac{P_o}{P_\phi} = \frac{E_o/T_o}{E_\phi/T_\phi} = \frac{T_\phi/T_o}{E_\phi/E_o} = \frac{Speedup}{Greenup} \quad , \tag{4}$$

where P_o is the power consumption of the optimized version, and P_ϕ is the power consumption of the non-optimized version. A power up value less than 1 indicates that the optimized version leads to reduced power consumption.

By applying the GPS-UP metrics, researchers can evaluate and compare two programs to identify which one excels in performance and energy efficiency. This comparison is facilitated using the GPS-UP Software Energy Efficiency Quadrant Graph, illustrated in Fig. 1.

This graph categorizes software into eight distinct groups based on the metrics of speed up and power up:

- *Category 1* (S_{up} ↑, P_{up} ↓). The optimization makes the program run faster and use less power, so it saves energy in two ways.
- *Category 2* (S_{up} ↑, P_{up} = 1). Runtime improves while the power draw remains unchanged, giving an energy saving derived solely from the increase in speed.
- *Category 3* (S_{up} ↑↑, P_{up} ↑). Power rises, but the gain in speed is even larger, therefore total energy still decreases.
- *Category 4* (S_{up} ↓, P_{up} ↓↓). The program takes longer to run, but the power savings are larger than the increase in time, so total energy use still decreases.
- *Category 5* (S_{up} ↑, P_{up} ↑↑). The code runs faster, but the power increase is bigger than the speed gain, so energy use still rises.
- *Category 6* (S_{up} ↓↓, P_{up} ↓). Power use drops, but performance drops even more, so the program ends up using more energy.
- *Category 7* (S_{up} ↓, P_{up} = 1). The program runs slower, and power stays the same, so energy use increases in proportion to the slowdown.
- *Category 8* (S_{up} ↓, P_{up} ↑). The worst case: the program runs slower and uses more power, so energy use increases the most.

Our work uses GPS-UP metrics to evaluate the impact of compiler optimizations on runtime, energy consumption, and power consumption for the various applications considered.

3. Related work

This section presents the most important papers on the topic of this work. First, Section 3.1 discusses different studies on the impact of code optimizations on software and their influence on its consumption. Then, Section 3.2 explores the energy impact of programming languages and compiler optimizations, highlighting how toolchain, implementation, and optimization choices affect software energy use.

3.1. Code optimizations for a greener software

Automatic software improvement involves modifying code without manual intervention to enhance performance, energy efficiency, and other aspects such as error repair [32] and program specification generation [33]. Most of the research in this area focuses on reducing energy consumption, particularly in smartphone applications [34]. This optimization is typically achieved by improving non-functional properties like runtime and memory usage through two types of code transformations: front-end and back-end [35].

Front-end transformations are applied directly to the source code, making them inherently dependent on the programming language of the software being optimized. These transformations can be classified into two main categories:

- *Approaches that preserve semantic behavior*. These transformations include code refactoring, which can significantly enhance energy efficiency and runtime, obtaining energy savings of up to 50% [36]. Additionally, memory optimization techniques, particularly in cache management and garbage collection, contribute to reduced energy consumption [37]. Advanced methods, such as deep learning and parameter tuning, can further optimize code but typically require a high level of programming expertise [38, 39].
- *Approaches that alter semantic behavior*. These transformations involve techniques such as Genetic Programming (GP), which evolves abstract syntax trees to improve runtime [40] and energy efficiency [41]. While GP is effective for small codebases, it is resource-intensive, and any modifications require extensive testing to ensure that the original functionality is preserved.

Back-end transformations operate independently of programming languages as they work on intermediate code or assembly code representations. Recent research focuses on automatic software performance optimization, offering advantages over front-end solutions by preserving the functionality of the original program and avoiding unintended compiler modifications. These transformations help optimize performance and reduce energy consumption, without compromising software quality aspects like maintainability and design [42]. Techniques such as loop unrolling and tiling may reduce energy consumption by up to 40% [43], though predicting their effects is complex due to conflicting energy-saving strategies [44]. Recent developments also highlight machine learning-based autotuning approaches to identify the best optimization sequences, particularly in GPU applications [45].

Choosing the optimal transformation sequence is crucial for maximizing performance, as no single sequence is universally applicable to all software. The phase ordering problem, which involves finding the ideal order in which a set of code transformations must be applied, remains critical since different sequences can significantly impact performance [46]. Early research on this issue focused on reducing

memory usage [47], while more recent approaches utilize techniques like deep reinforcement learning to improve LLVM optimizations [48].

Furthermore, the flag selection problem involves identifying the best compilation flags for performance optimization, with metaheuristics [49] and Bayesian methods proving effective [50]. A pioneering method for software optimization models the task as a combinatorial optimization problem known as SCOP [4]. This method addresses both the identification of transformation sequences, including repetitions, and the phase ordering problem using a cellular genetic algorithm. In addition, this approach has been extended to optimize not only software performance but also to minimize energy consumption [51] and to obfuscate source code [52].

In this work, we apply both front-end and back-end transformation sequences included in the generic optimizations provided by the Swift compiler and the LLVM infrastructure. These transformations aim to improve the performance of any program; however, their impact on runtime and energy consumption in iOS applications has not been studied.

3.2. Energy impact of programming languages and compiler optimizations

The relationship between programming languages, compiler optimizations, and energy consumption has attracted significant attention in recent years, particularly as software sustainability becomes an increasingly critical concern. Empirical studies have shown that the energy footprint of a program is influenced not only by its algorithmic design but also by the choice of programming language and the compiler optimizations applied during code generation.

An influential study on GCC compiler optimizations using fractional factorial design revealed that no single compiler pass is universally optimal for reducing energy consumption [53]. Instead, program-specific tuning is essential, as the most effective compiler settings for energy efficiency depend on both the application and the hardware platform.

Comparative studies have also highlighted the impact of different compiler toolchains. For example, research comparing GCC, Clang, and Intel ICC compilers found measurable differences in energy consumption [54]. Notably, binaries generated by ICC were approximately 7% more energy-efficient than those produced by GCC, while Clang tended to result in slightly higher energy usage. Other studies have focused on the energy impact of generic LLVM optimization flags. For example, research analyzing LLVM flags on a video game engine demonstrated limited and inconsistent improvements in energy efficiency, with the `-O3` flag achieving only a 0.03% reduction compared to the non-optimized version [55]. Similarly, experiments on embedded benchmarks found that no single LLVM optimization flag consistently reduces energy consumption [51].

Beyond compiler optimizations, programming languages themselves have also been scrutinized for their impact on energy usage. Studies comparing languages such as Java, Kotlin, and C have shown that, when controlling for factors like execution time, memory activity, and core usage, differences in energy consumption attributable solely to the programming language largely disappear [56]. Further investigations into compiler and interpreter versions for languages like C, Java, and Python have revealed that even within the same language, the choice of compiler or interpreter can significantly influence energy profiles [57]. Moreover, a detailed analysis across 27 programming languages demonstrates that, while compiled languages generally achieve higher energy efficiency, substantial variations remain across tasks and languages [58].

Furthermore, recent research has explored the impact of programming language choices on the environmental sustainability of Artificial Intelligence (AI). For example, one study analyzed how different languages affect energy consumption during training and inference of AI algorithms, finding that compiled and semi-compiled languages consistently consume less energy than interpreted languages, which can require up to 54 times more energy in some cases [59].

This study explores how different compiler optimizations used during the development of Swift-written iOS applications influence energy consumption. While such optimizations are typically intended to enhance program performance or reduce application size, their specific effects on runtime and energy efficiency within the context of iOS applications remain underexplored.

4. Methodology

In this study, our objective is to empirically assess the impact of the most aggressive code transformations in LLVM and Swift on the runtime and energy consumption of diverse iOS applications. To achieve this, we must accurately measure the energy consumption of both optimized and unoptimized versions of the subject applications. Section 4.1 introduces the system employed for measuring energy consumption, including the methodology for synchronizing these application measurements. Next, Section 4.2 provides a detailed description of the process for compiling and deploying the applications from the measurement system to the DUT. Subsequently, Section 4.3 presents the methodology for run and evaluation of the different applications studied.

4.1. Measurement system

iGreenMiner [9,10] is a reliable and accurate measurement system for measuring the current of iOS devices. We utilize an updated version of iGreenMiner, which can run multiple instances of a specific iOS application on an iOS device. It utilizes an Apple Developer ID to authenticate the application under test. During each run, iGreenMiner measures the energy consumption of the entire device using a power monitor, and this updated version synchronizes the measurements with the application run based on CPU patterns.

The measurement system includes a macOS device that handles the compilation and deployment of applications into an iOS device using an Apple Developer ID. It also synchronizes the measurement process for each application run on the iOS device and collects the corresponding energy consumption data. Moreover, iGreenMiner measures energy consumption using a Monsoon power monitor, which is a current draw measurement instrument. The Monsoon power monitor supplies a direct voltage of 4.2 V to the device and measures its current draw in milliAmps (mA) at a sampling rate of 5 kHz. The current values, along with timestamps, are then sent to the controller (i.e., the macOS device). The controller records these measurements and converts them into energy based on the time elapsed. Fig. 2 presents a schematic describing the interactions of the primary components of the measurement system.

To ensure accurate and reliable measurements, several precautions are taken during the process. First, we eliminate the impact of battery conditions and aging effects [15] by supplying the measurement device with a constant voltage of 4.2 V. Second, the controller is connected to the measurement device via Wi-Fi, avoiding the charging state that would typically be triggered when the device is connected via a USB cable. Third, the screen brightness is consistently set to maximum to ensure uniformity in the measurements. Fourth, the rest of the iOS device settings remain at their default values, with developer mode enabled. Fifth, before taking measurements, the iOS device undergoes CPU-intensive tasks to warm it up, preventing potential inconsistencies caused by thermal throttling during the run of the applications to be measured and ensuring the governor's state is reliable between tests. Lastly, to synchronize the Monsoon monitor measurements with each application run, we employ a synchronization approach based on CPU loops at the start and end of the run, as explained in the following section.

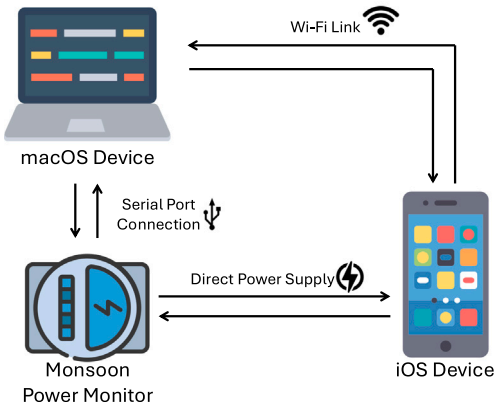


Fig. 2. Diagram illustrating the different components and their connections within the iGreenMiner measurement system.

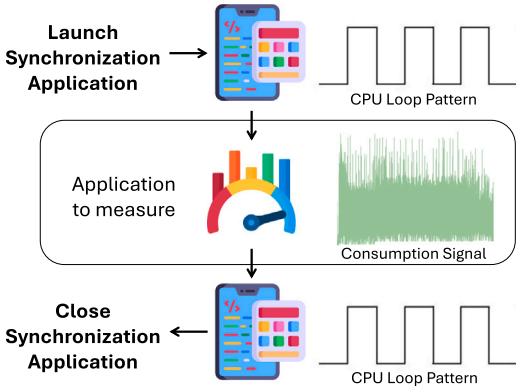


Fig. 3. Schematic illustrating the new synchronization methodology for the iGreenMiner measurement system.

4.2. Compilation and deployment process

This section describes how the considered iOS applications are compiled during our experimentation, as well as the methodology followed to synchronize the executions for exact consumption measurements.

4.2.1. Compilation step

To ensure we apply each optimization in a controlled way, we do not employ a development IDE during the compilation process. As a consequence, we maintain complete control throughout the compilation of the iOS application project regarding the optimizations we employ and the code representation we use. Moreover, the order in which optimizations are applied can impact energy consumption and runtime [60]. To address this, optimizations are applied in a deterministic manner: first, the fixed generic sequence of Swift optimizations, which operates on the Swift Intermediate Language (SIL), and then the fixed generic sequence of LLVM optimizations, which acts on the LLVM Intermediate Representation (IR).

The compilation process begins with the Swift compiler, `swiftc`, which generates the LLVM IR for the entire Swift application in a single file. At this stage, any desired Swift code optimizations are applied at the SIL level. Next, the LLVM optimizer, `opt`, is employed to apply the required LLVM optimizations to this IR code. Finally, the binary of the application is compiled and linked using `swiftc` again. It is important to note that throughout the entire compilation process, the toolchain corresponding to the targeted iOS version must be specified.

Once the application is compiled, we must create a bundle with the content, shared libraries, and metadata of the application. It is important to note that the shared libraries are not optimized. To facilitate this process, we initially generate a bundle with the official Apple IDE, Xcode, and replace the generated binary with our own properly optimized one. Finally, we sign this new bundle with our Apple Developer ID, resulting in a properly optimized application packaged and ready to deploy on the device.

4.2.2. Deployment and synchronization step

For the deployment process of our newly packaged and signed application, we use the `ios-deploy` project [61]. Thanks to this utility, iOS applications can be installed and deployed without relying on any IDE. It can also be used for debugging and testing applications.

For synchronization, we use an additional application on the iOS device that manages the alignment between the current measurements from the meter and the runs of the application being measured. This application generates a pattern in the current signal consisting of rises and falls lasting 0.5 s using CPU loops on the iOS device. These patterns are created just before launching the application to be measured and immediately after it finishes, allowing an accurate identification of the signal portion corresponding to the application run. Furthermore, as an independent application, it remains unaffected by any optimizations applied. Fig. 3 illustrates the behavior of the synchronization application and the CPU loop patterns.

From the macOS device controlling the measurement system, we compute the correlation between the measured signal and a synthetic signal to detect the CPU loop patterns in the raw signal, extracting the segment corresponding to the signal of the target application. This approach enables us to isolate the consumption signal corresponding to the application being measured, thereby calculating its energy consumption and duration.

4.3. Running and evaluation methodology

To evaluate the impact of the different compiler optimization flags on both runtime and energy consumption of the software, we employ the following methodology:

1. We apply the desired combination of LLVM and `swiftc` optimizations to the application being evaluated, following the previously described compilation process.
2. We launch the application and synchronize the measurement of the iOS application with the macOS device using the CPU pattern-based synchronization algorithm described earlier.
3. Finally, we collect the consumption signal of the application and calculate its runtime and energy consumption

All of this process is repeated a total of 30 times, resulting in 30 independent measurements for each application. This approach allows us to accurately identify substantial differences between the compiler flags combinations. Moreover, to rank the energy consumption and runtime associated with the compiler flags combinations, we applied the nonparametric Scott-Knott ESD (NPSK) test at a 95% confidence level. The NPSK test is a multiple comparison method that employs hierarchical clustering to group statistically distinct performance medians into categories that are not significantly different from each other or have minimal effect sizes [62]. This test is particularly useful because it does not assume normality, homogeneous distributions, or a minimum sample size [63]. In our experiments, a lower rank corresponds to higher energy consumption or runtime, meaning that the best results have the highest rank.

Additionally, we have utilized GPS-UP metrics to evaluate the performance and energy efficiency of each optimized version compared to the baseline application (non-optimized). This allows us to categorize each of the optimized versions within the GPS-UP Software Energy Efficiency Quadrant Graph, allowing a detailed analysis of the impact of each compiler optimization.

5. Apps selection

To assess the impact of the most aggressive optimization flags for iOS applications developed in Swift, we employ an iPhone 11 equipped with iOS 13.4.1. Additionally, the macOS device utilized for controlling the measurement system is a MacBook Air M1 with macOS Sonoma 14.5. Furthermore, the version of LLVM employed is 15.0.0, while the version of `swiftc` is 5.10. As optimizations, a total of nine combinations between the generic optimization flags available in LLVM and `swiftc` have been studied. We have selected the optimization flags that focus on runtime performance for both compilers. For LLVM, we chose `-O3` and `-Oz`, the latter focused on reducing code size and energy consumption. For `swiftc`, we selected `-O` and `-Ounchecked`, which aim for even more aggressive performance optimizations. To disable generic optimizations, we use the flags `-Onone` for `swiftc` and `-O0` for LLVM.

For the experimentation, two applications in each of the following three categories are selected:

- *Disk operations*: Represent I/O-bound tasks that are fundamental for data access and storage [64].
- *Deep learning models*: Reflect the growing integration of AI capabilities in mobile applications, requiring intensive CPU and GPU computations [65].
- *Video games*: Constitute a dominant and performance-critical category on mobile platforms, where real-time responsiveness and resource efficiency are essential [66].

These categories were chosen based on their prevalence within the iOS ecosystem, their relevance to real-world mobile workloads, and their distinct computational profiles. Together, they allow the study of compiler optimizations across a representative range of performance-critical use cases in iOS apps.

To choose these applications, we conducted a snowball search on GitHub, using keywords related to each specific category and focusing only on apps developed in Swift for iOS. The application selection process was as follows:

1. A search was performed in GitHub using the keywords related to the category, extracting all applications written in Swift.
2. Applications that were incomplete or not entirely written in Swift were discarded. Additionally, applications with multiple shared libraries or those unrelated to the specific app category were excluded.
3. The available documentation for the applications was reviewed, and they were selected based on their simplicity and clarity.
4. The applications' source code was analyzed, and the cleanest and simplest ones in each category were selected.

For each selected application, we developed a benchmark program that makes the application performing some repetitive task, and this benchmark is then used to analyze its performance with the different compiler optimizations. Selecting applications written in Swift allowed us to easily develop and integrate benchmarking mechanisms to measure the impact of different compiler optimizations on iOS devices. However, this restriction inherently limited the number of available open-source applications which is itself considerably reduced, compared to those for Android devices [67]. After this process, two applications were selected per category, resulting in a total of six applications. It will definitely be interesting to extend this study with more applications and categories, and this is left for future work, as described in Section 9.

The considered categories were chosen because they represent diverse and computationally demanding workloads commonly found in mobile applications. Disk operations are fundamental to data storage and retrieval, impacting app performance and energy efficiency. Deep learning models are increasingly integrated into mobile applications for tasks like image recognition and natural language processing, making them a relevant category for evaluating computational optimizations. Video games are among the most performance-intensive applications, requiring efficient CPU and GPU utilization. Studying these categories helps us understand how compiler optimizations impact different common smartphone uses. Within each category, selecting two different applications allows us to determine whether the effects of various optimizations are consistent or differ across different workloads.

5.1. Disk operations

The first category corresponds to applications where numerous disk operations are intensively performed, a very common operation in the daily usage of a smartphone [68]. Within this category, the following applications have been selected for research:

- *SQLitePolybench (SQLPoly)*, that combines the SQLite benchmark presented in [10] with the Polybench benchmark version from [4]. In this version, it features an loop where a new object is inserted into an SQLite database, the database is updated, and then a iteration of the Polybench benchmark is run. This loop is repeated a total of 10,000 times.
- *FileManagement (FileMng)* [69], where a total of 1000 write and delete operations are performed using the `FileManager` available in the standard iOS library.

5.2. Deep learning models

The second category includes the use of pre-trained deep learning models for inference on iOS devices, as the integration of these models is becoming more and more common in our daily lives [70]. Under this category, two specific applications have been considered:

- *ClassificationInference (ClassInf)* [71], that performs a total of 10,000 image classifications on objects such as batteries, Arduinos, servomotors, resistors, and other components related to electronic circuits. A pre-trained model that utilizes the `ImageClassifier` from the iOS Core Machine Learning library is employed for classifying these images into 11 different categories: (1) *LED*, (2) *battery*, (3) *Arduino*, (4) *Arduino cable*, (5) *jumper*, (6) *breadboard*, (7) *motor*, (8) *resistor*, (9) *servomotor*, (10) *ultrasonic sensor*, and (11) *USB cable*.
- *SegmentationInference (SegInf)* [72], that performs a total of 1000 image segmentations on cat images using a pre-trained model based on DeepLab v3 + MobileNet v2. This model utilizes the `DeepLab` class from the iOS Core Machine Learning library to represent the AI model.

5.3. Video games

The third category focuses on one of the most dominant markets within mobile applications: video games [73]. Mobile games have experienced exponential growth in recent years, becoming one of the main drivers of application development and the video game industry [74]. For this reason, the following two video games have been selected for benchmarking:

- *FlappyFlyBird (FlyBird)* [75], a clone of the popular Flappy Bird game using the SpriteKit framework. To accurately assess the impact of the different compiler optimizations within the game, an identical game session is generated and replayed each time the app is launched. Additionally, 5000 player objects are created and updated simultaneously, which significantly increases the computational load of the application. The game session ends after the first 3000 frames have been generated, ensuring that the session is sufficiently long to reflect a real gameplay.

Table 1
Pre-compiled libraries linked by each benchmark application.

Application	Pre-compiled libraries
SQLitePolybench	`Foundation`, `UIKit`, `libsqlite3`
FileManagement	`Foundation`, `UIKit`
ClassificationInference	`Foundation`, `UIKit`, `CoreML`, `Vision`, `Accelerate`
SegmentationInference	`Foundation`, `UIKit`, `CoreML`, `Vision`, `Accelerate`
FlappyFlyBird	`Foundation`, `UIKit`, `SpriteKit`, `GameplayKit`, `CoreGraphics`
GhostRun	`Foundation`, `UIKit`, `SpriteKit`, `GameplayKit`, `CoreGraphics`

- *GhostRun (GhostR)* [76], a running game featuring a ghost, inspired by the popular browser game, Dinosaur Game. To evaluate the impact of the different compiler optimizations, an identical game session is generated each time the application is launched. In this case, 25,000 player instances are created to significantly increase the computational load. Again, the game session concludes once the first 3000 frames have been generated.

5.4. Library footprint and reuse

Understanding the role of external libraries is crucial for assessing our findings. Modern iOS applications often rely on third-party frameworks or pre-compiled system libraries, and these components can significantly influence both performance and energy usage of programming languages [58]. To isolate the impact of compiler optimizations on developer-controlled code, we analyze the external dependencies of the six benchmark applications.

Table 1 lists every framework or static library that each benchmark application links against.[1] All six apps rely exclusively on Apple-supplied system frameworks, with no third-party dependencies or custom static libraries. In consequence, the corpus contains neither source code nor pre-compiled binaries from external vendors.

Framework reuse is moderate rather than universal. While some core libraries like `Foundation` and `UIKit` are shared across all benchmark applications, more specialized frameworks are reused only within specific domains. For instance, the machine learning applications (ClassificationInference and SegmentationInference) share `CoreML`, `Vision`, and `Accelerate`. Similarly, the games (FlappyFlyBird and GhostRun) both use `SpriteKit`, `GameplayKit`, and `CoreGraphics`. In contrast, SQLitePolybench employs `libsqlite3`, and FileManagement relies only on the core frameworks. This variation in dependencies shows that, while some core frameworks are commonly reused, each benchmark targets a distinct subset of the iOS API surface.

The exact size or complexity of the pre-compiled Apple frameworks is not available, as their source code is not publicly released. Therefore, we cannot quantify how much code these libraries contribute. However, typically in iOS applications, these frameworks account for a significant portion of the final binary size.

These frameworks, shipped as part of the iOS SDK, are compiled by Apple using production optimization pipelines. These binaries are not recompiled during a standard build. As a consequence, the optimization flags we vary affect only the application modules. This design isolates the impact of compiler flags on developer-controlled code and avoids the confounding effects that would arise if third-party libraries were rebuilt under different settings. While a large part of the binary size comes from pre-compiled libraries, our study shows that tuning generic compiler flags for developer-written code can still have a measurable impact on both performance and energy efficiency.

[1] The list was produced with `otool -L` on the source code available in the Zenodo repository [77].

6. Results

This section presents the results of applying the considered compiler optimization combinations to the iOS applications under study. To reduce inconsistencies in energy consumption and runtime measurements caused by uncertainty, we performed 30 independent measurements for each combination of optimizations and applications.

First, Section 6.1 provides a statistical analysis of how the different compiler optimization flags impact runtime (**RQ1**) and energy consumption (**RQ2**) for each of the considered application categories. Next, Section 6.2 examines the placement within the GPS-UP Energy Efficiency Quadrant Chart, classifying the different optimized versions obtained for each selected application and examining the relationship between runtime and energy consumption (**RQ3**).

6.1. Influence of compiler optimization flags on software runtime and energy consumption

We evaluate in this section the influence of the different compiler optimization flags on both runtime (or execution time) and energy consumption of the studied applications. The section is structured into three subsections, one dedicated to each application kind.

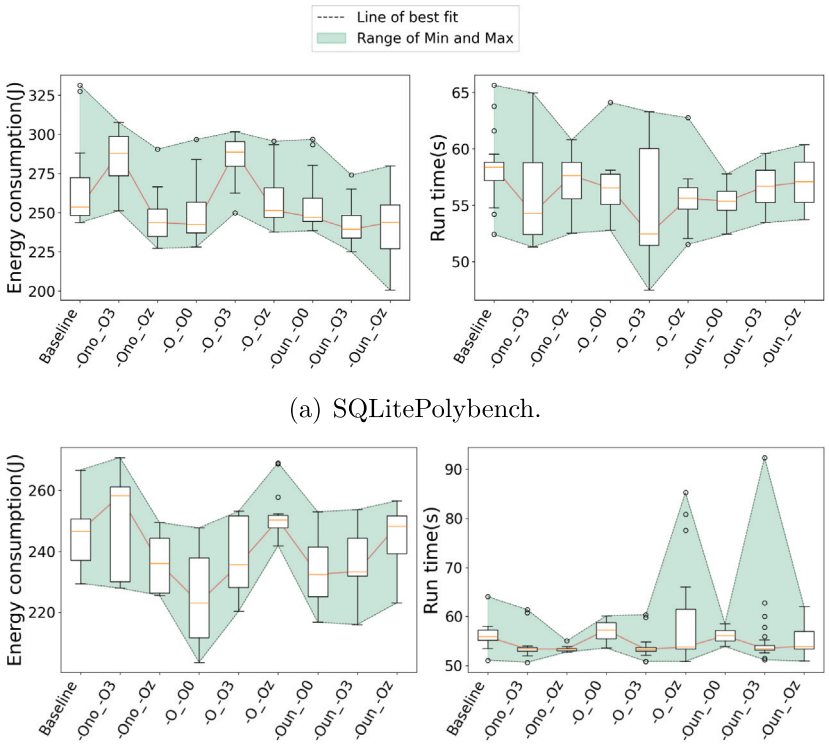
Note that, in the NPSK rankings, Rank 1 corresponds to the best result for each metric (i.e., the shortest runtime or the lowest energy consumption), and combinations appearing in the same rank are not statistically distinguishable from each other.

6.1.1. Disk operations

Starting with the first category of applications, Fig. 4 displays the boxplots for energy consumption, measured in Joules (J), and runtime, measured in seconds (s), for both SQLitePolybench and FileManagement. The shaded green area represents the range between the maximum and minimum values for each box, considering the outliers in the sample.

For SQLitePolybench, Fig. 4(a) shows that all studied optimizations succeed in better runtime performance compared to the non-optimized application, some of them accomplishing significantly worse consumption records. The optimization combination that appears to achieve the shortest runtime is `-O` with `-O3`, showing an improvement of 10.1215% compared to its non-optimized version. However, this combination results in one of the highest values in energy consumption, with a worsening of 13.8323% relative to the non-optimized version. On the other hand, the optimization combination that seems to offer the lowest energy consumption is `-Ounchecked` with `-O3`, which demonstrates an improvement of 5.6344% in energy consumption and also a 2.9130% improvement in runtime, both compared to the non-optimized version.

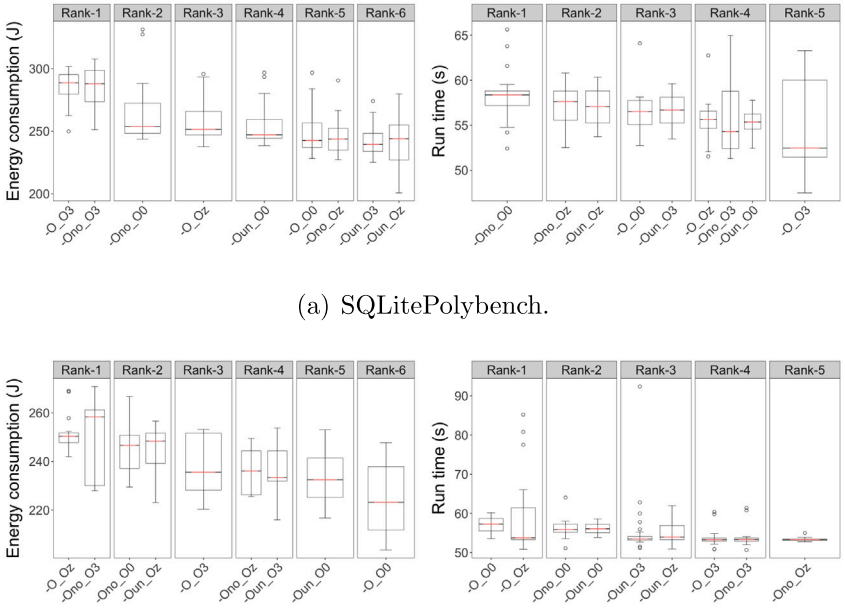
Regarding FileManagement, Fig. 4(b) illustrates that the combination of optimizations resulting in the shortest apparent runtime is `-Onone` with `-Oz`, leading to a 4.5408% improvement over the non-optimized version. In this case, unlike SQLitePolybench, it also comes with a reduction in energy consumption by 4.2829%. On the other hand, the optimization combination with the lowest apparent energy consumption is `-O` with `-O0`, providing a 9.5194% improvement in energy usage compared to the baseline. However, for this specific application, runtime increases by 2.4304%.



(a) SQLitePolybench.

(b) FileManagement.

Fig. 4. Comparison of energy and runtime for disk benchmarks compiled with selected generic flags using `swiftc` and LLVM. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)



(a) SQLitePolybench.

(b) FileManagement.

Fig. 5. NPSK rankings of energy and runtime for disk benchmarks compiled with selected generic flags in `swiftc` and LLVM, ranked by highest energy and longest runtime.

Concerning the NPSK ranking results, Fig. 5(a) depicts the performance of SQLitePolybench in both runtime and energy consumption. It is important to note that combinations appearing in the same ranking are not statistically distinguishable from each other, and that higher ranks represent better results. We can note that the combination with the shortest runtime, `-O` with `-O3`, also has the highest energy consumption. Similarly, the two combinations that achieve the lowest energy consumption results, `-Ounchecked` with `-O3` and

`-Ounchecked` with `-Oz`, rank lower in terms of execution time, appearing in rank 3 and rank 2, respectively. This suggests that the most aggressive optimization combinations that reduce runtime tend to negatively impact energy consumption. Nevertheless, the inverse does not hold, as the combinations with the lowest energy consumption do not achieve the shortest runtime but still show improvements.

The ranking results for FileManagement are presented in Fig. 5(b). In this case, the optimization combination with the shortest runtime, `-Onone` with `-Oz`, does not rank among the worst for energy consumption but is instead positioned at rank 4, enhancing the results obtained for SQLitePolybench. This is not the only inconsistent result found between the two applications, as the optimization combination with the lowest energy consumption for FileManagement, `-O` with `-O0`, is also the one with the longest runtime. This may indicate that optimizations that have a more aggressive impact on energy consumption tend to negatively affect runtime. Nevertheless, for both applications studied in this category, an optimization that achieves good runtime results tends to show poor results in energy consumption. Finally, it was found an uneven impact of the optimizations on these two applications, as the best optimizations for SQLitePolybench for runtime and consumption are among the worst ones for FileManagement, and vice versa.

6.1.2. Deep learning models

In Fig. 6(a), the boxplots expose the measurements for both energy consumption and runtime across the different optimization combinations considered for ClassificationInference. The combination of `-Ounchecked` with `-O3` appears to achieve the lowest energy consumption, 13.4654% lower than the non-optimized application, with a slight 0.3649% runtime increase. In contrast, when focusing on runtime, the combination of `-Ounchecked` with `-Oz` seems to perform best, reducing the baseline runtime by 6.2258%, although with an increase in energy consumption by 6.7421%.

For SegmentationInference, the measurement results are illustrated in Fig. 6(b). In this case, the combination that seems to achieve the lowest energy consumption is `-O` with `-O0`, obtaining an improvement of 7.2364% but with an increase of 3.1955% in runtime compared to the baseline application. Regarding runtime, the combination `-Onone` with `-Oz` stands out, reducing it by 2.9913% while also slightly lowering energy consumption by 0.8447%.

The NPSK ranking results for both energy consumption and runtime for ClassificationInference are shown in Fig. 7(a). The two best combinations in terms of energy consumption are `-Ounchecked` with `-O3` and `-Ounchecked` with `-O0`, but these also rank among the worst in runtime, placed in rank 3 and rank 1, respectively. The reverse is also true: the combination with the fastest runtime, `-Ounchecked` with `-Oz`, ranks the worst for energy consumption. Additionally, for this specific application, the `-Ounchecked` optimization flag from `swiftc` appears in both scenarios, with its impact varying on energy consumption and runtime depending on the LLVM flag used.

Regarding SegmentationInference, the ranking results are outlined in Fig. 7(b). The best combinations in terms of energy consumption are `-Ounchecked` with `-O3` and `-O` with `-O0`, both of which again rank among the worst in runtime, at ranks 3 and 1, respectively. Interestingly, the combination `-Ounchecked` with `-O3` holds the same rank for both energy consumption and runtime in ClassificationInference and SegmentationInference. In terms of runtime, the combination that achieves the fastest result is `-Onone` with `-Oz`, though in this case, it is not among the worst for energy consumption, occupying rank 5, on par with the non-optimized application.

6.1.3. Video games

Regarding the studied applications in the video games category, Fig. 8(a) illustrates the boxplots of measurements for both energy consumption and runtime for FlappyFlyBird. It is evident that the combination with the shortest runtime is `-O` with `-O0`, with improvements of 1.7574% and 3.0693% on runtime and consumption, respectively, over the non-optimized application. For energy consumption, the best combination is `-O` with `-O3`, with an improvement of 8.8667%, and reducing runtime by 0.4013%.

The results for GhostRun are depicted in Fig. 8(b), where it can be seen that the combination with the shortest runtime is `-Ounchecked` with `-Oz`, reducing runtime by 9.5370%, though at the cost of highly increasing energy consumption by 34.1913%. On the other hand, `-O` with `-O3` appears to achieve the lowest energy consumption, as for FlappyFlyBird, with an improvement of 0.4573% while also enhancing runtime by 0.2528%.

Concerning the NPSK ranking results, Fig. 9(a) displays the results for both energy consumption and runtime for FlappyFlyBird. In this case, we notice that the best combinations for energy consumption are also among the best for runtime. Similarly, the top combination for runtime, `-O` with `-O0`, ranks 4 in terms of energy consumption. Therefore, for this specific application, improvements in energy consumption generally do not have a major impact in runtime, and vice versa.

Finally, the NPSK ranking results for GhostRun are shown in Fig. 9(b). Here, again, we can see that the best combination for energy consumption `-O` with `-O3`, ranks among the worst for runtime. Conversely, the top combination for runtime, `-Ounchecked` with `-Oz`, is among the worst for energy consumption. Moreover, it is noteworthy that none of the optimization flag combinations improve the energy consumption compared to the non-optimized application, with all but `-O` with `-O3` worsening it.

6.2. Energy efficiency of the different optimized applications versions

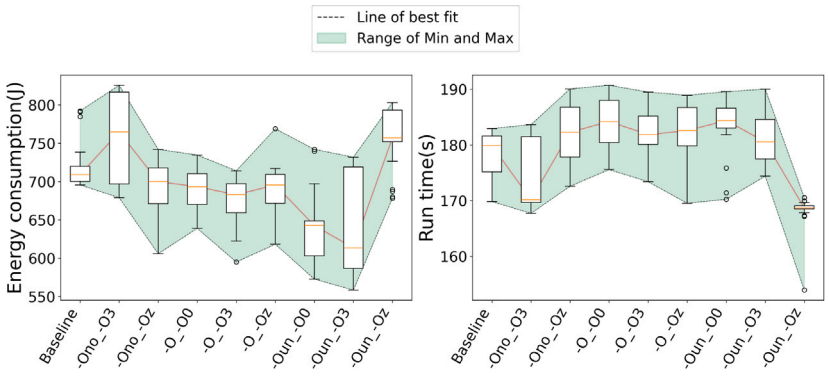
We apply in this section the GPS-UP metrics to find the energy-efficiency category for every optimized version of the considered applications. For that, the median of the 30 independent measurements for both energy consumption and runtime was used. Based on these values, and using the non-optimized application, `-Onone` with `-O0`, as the baseline, each metric was calculated, and the different versions were categorized into one of the nine possible categories.

6.2.1. Evaluation of GPS-UP metrics and software categories for considered applications

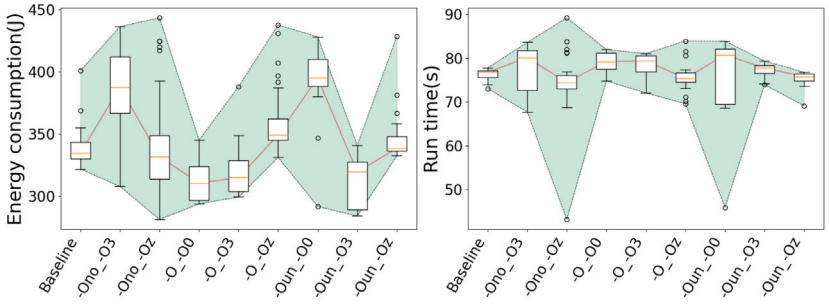
We analyze in this section the GPS-UP categories of the different optimized versions of the studied applications. Fig. 10 provides an overall view of the results, while a detailed analysis of the runtime and energy efficiency performance of the different applications and their optimized versions is provided in Section S1 of the supplementary material.[2] The figure presents the GPS-UP Software Energy Efficiency Quadrant Chart of the different optimized versions for every application considered in this study. As it can be seen, there is no clear conclusion on which optimizations can lead to energy efficient solutions, not even within the same software category, as they produce greener software in some cases and worsen the energy efficient performance in others. As an illustration, it can be seen how combinations `-O` with `-O0` and `-Ounchecked` with `-O3` fall into Category 1 for FlappyFlyBird and into Category 8 for the other studied game (see Figs. 10(e) and 10(f)).

Another interesting observation is that the combination `-Onone` `-O3` consistently leads to non energy efficient versions, falling into Categories 5 or 8 in all cases. Therefore, using `-O3` LLVM compilation flag may benefit runtime (it happens in 4 out of the 6 cases), but at the cost of energy efficiency performance loss in all cases. It is also interesting that no optimized version falls into Categories 2, 6, and 7. The cases of Categories 2 and 7 could be expected, as it is difficult to get and improvement or loss in speedup, respectively, at exactly the

[2] At the disposal of the reviewer at the end of this document. It will be publicly available at the project's GitHub, upon acceptance of this work for publication.

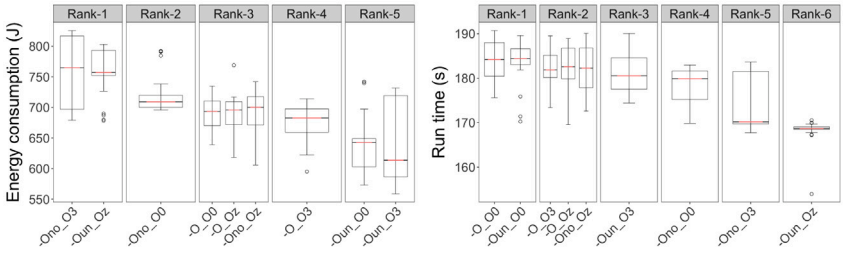


(a) ClassificationInference.

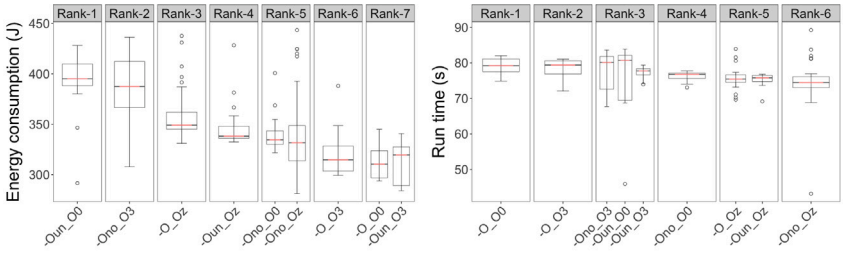


(b) SegmentationInference.

Fig. 6. Comparison of energy and runtime for ML benchmarks compiled with selected generic flags using `swiftc` and LLVM.



(a) ClassificationInference.

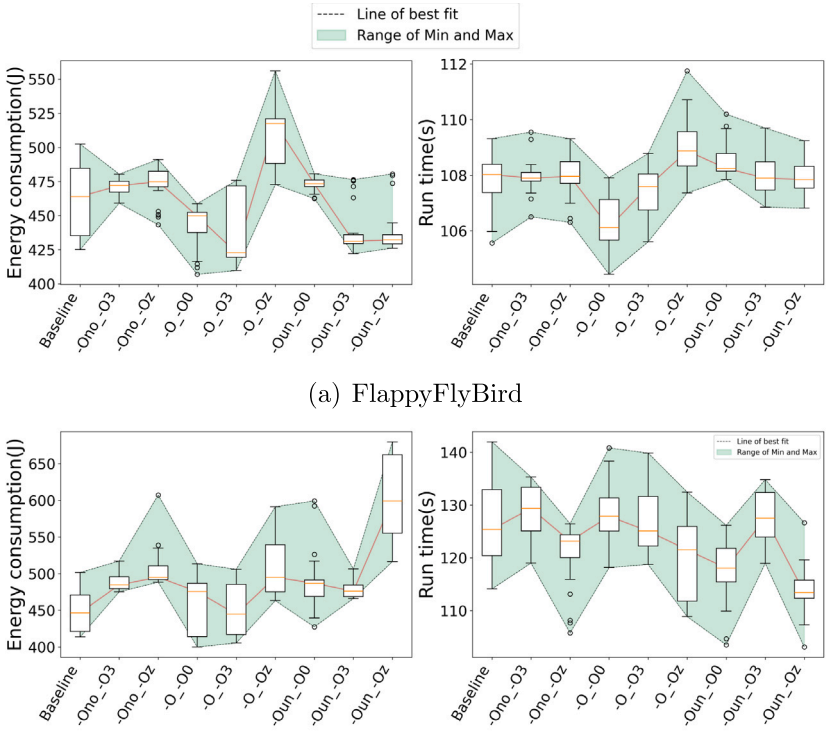


(b) SegmentationInference.

Fig. 7. NPSK rankings of energy and runtime for ML benchmarks compiled with selected generic flags in `swiftc` and LLVM, ranked by highest energy and longest runtime.

same consumption value. However, the case of Category 6 means that it never happens that speed up decrease was lower than power up loss.

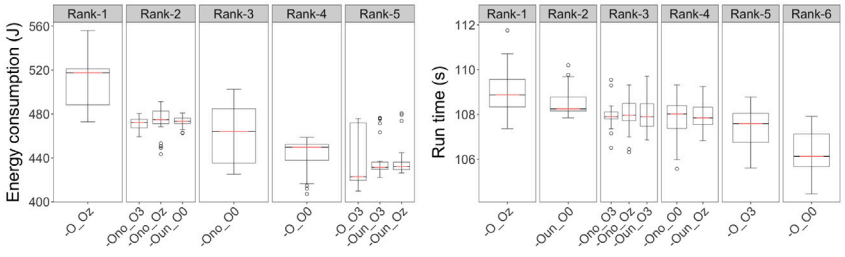
To end this section, we would like to remark that 8 optimized versions were generated for each considered application in this work, making it 48 optimized versions in total, and only 26 of them were found to be greener than the original application according to GPS-UP categorization. However, a high 45.83% of the optimized versions (namely, 22 versions), were found to offer a lower power up performance than the original applications. For the different application kinds considered, the percentage of optimization versions that worsen the



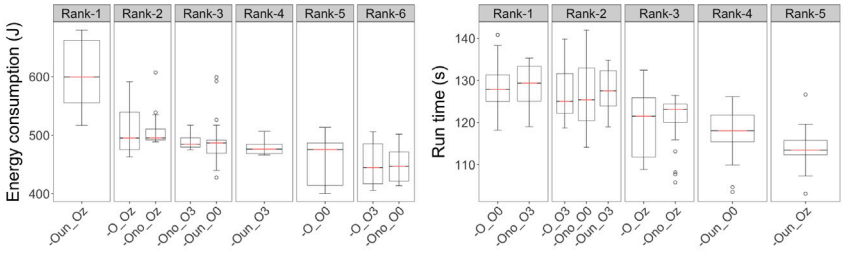
(a) FlappyFlyBird

(b) GhostRun

Fig. 8. Comparison of energy and runtime for video games benchmarks compiled with selected generic flags using `swiftc` and LLVM.

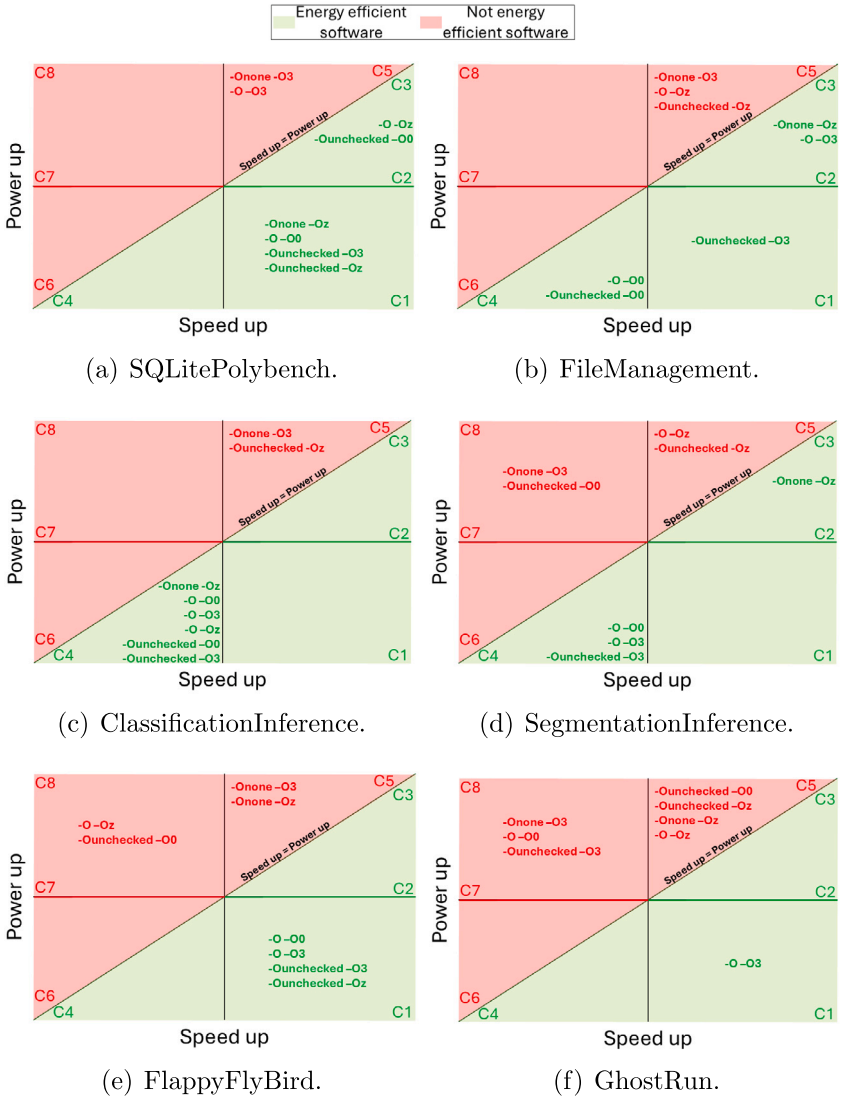


(a) FlappyFlyBird.



(b) GhostRun.

Fig. 9. NPSK rankings of energy and runtime for video games compiled with selected generic flags in `swiftc` and LLVM, ranked by highest energy and longest runtime.



(a) SQLitePolybench. (b) FileManagement.

(c) ClassificationInference. (d) SegmentationInference.

(e) FlappyFlyBird. (f) GhostRun.

Fig. 10. Software energy efficiency quadrant charts for the applications compiled with the selected optimization flags from `swiftc` and LLVM using median values after 30 runs.

power up performance of the original application is 21.25% for disk operations, 37.5% for deep learning models and a high 68.75% for video game applications.

6.2.2. Analysis of the consumption time-series obtained for different GPS-UP software categories

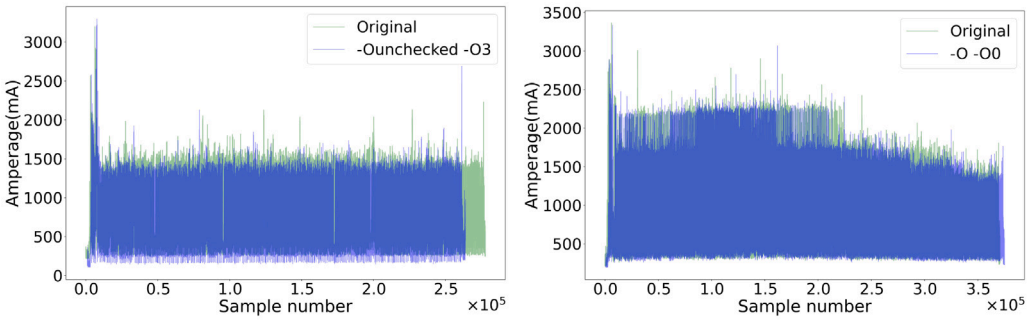
To analyze in detail how the different optimized versions of the applications behave in terms of energy consumption, we compare in this section the original applications against an optimized version in terms of their instant consumption. In order to show representative cases, the four most frequently occurring categories from the GPS-UP Software Energy Efficiency Quadrant Chart were selected: 1, 4, 5, and 8. From each of these categories, an optimized version of an application was chosen, and a randomly selected signal from the 30 independent samples was selected for the optimized and original versions of the application. This allows us to observe the differences in both signal duration and power consumption.

Fig. 11(a) shows the signal for the optimized version of FileManagement employing the combination of `-Ounchecked` with `-O3`, which falls into Category 1. It is evident that the signal from the optimized version has a significantly shorter duration. Additionally, throughout the entire run of the application, most current values are lower than in the non-optimized version. These two issues lead to a substantial reduction in energy consumption.

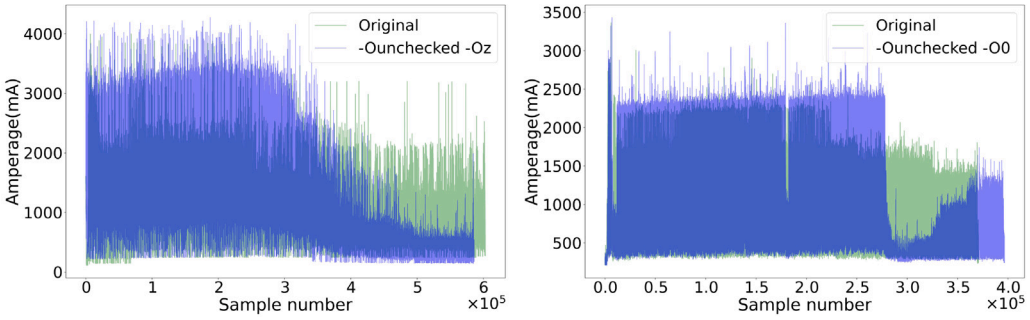
For Category 4, we selected the optimized version of the SegmentationInference application applying `-O` with `-O0`, as shown in Fig. 11(b). In this case, the signal from the optimized version has a longer duration compared to the non-optimized version, indicating a speed up value less than 1. However, we can once again observe that the current values are lower for most timestamps, which results in a reduction in energy consumption despite the increase in runtime.

Regarding the red categories, Fig. 11(c) illustrates the signal obtained for the optimized version of GhostRun using `-Ounchecked` with `-Oz`, which falls into Category 5. We can appreciate that the signal duration of the optimized version is clearly shorter compared to the non-optimized version. However, the power consumed during its execution is significantly higher, with much greater current values over time compared to the non-optimized application. As a result, the energy consumption of the optimized version ends up being higher.

Finally, for Category 8, the optimized version of SegmentationInference was selected, applying `-Ounchecked` with `-O0`, as detailed in Fig. 11(d). In this plot, we notice a longer run compared to the non-optimized application, along with higher current values at most timestamps. Consequently, the optimized version of the application is less energy-efficient. It is noteworthy that the signal of the optimized application towards the end exhibits significantly different current intensity values, underscoring the substantial influence of optimizations on the energy footprint of the application.

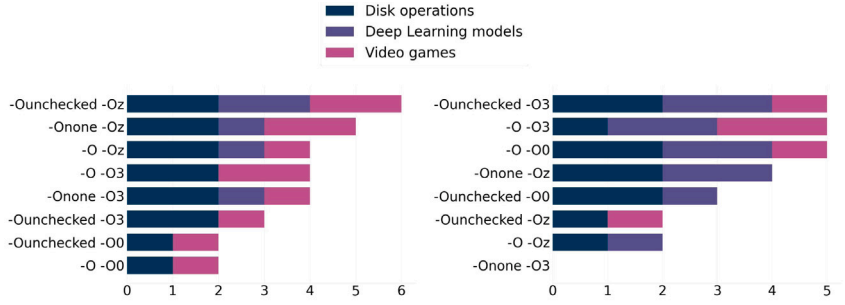


(a) C1: $S_{up}\uparrow P_{up}\downarrow$ - FileManagement. (b) C4: $S_{up}\downarrow P_{up}\downarrow$ - SegmentationInference.



(c) C5: $S_{up}\uparrow P_{up}\uparrow$ - GhostRun. (d) C8: $S_{up}\downarrow P_{up}\uparrow$ - SegmentationInference.

Fig. 11. Comparison of signal consumption for different GPS-UP categories obtained.



(a) Speed up. (b) Green up.

Fig. 12. Number of times a speed up/green up greater than one have been achieved, categorized by application type, for each of the optimization combinations studied.

7. Discussion

In this section, we analyze and address the various research questions that motivated this study. **RQ1** investigates whether the general belief that applying compiler optimizations always leads to improvements in runtime holds true for iOS applications. **RQ2** examines whether these same optimizations also result in improvements in energy consumption. **RQ3** explores whether there is a direct correlation between improvements in runtime and energy consumption when applying compiler optimizations to iOS applications.

RQ1. Do the most aggressive optimization flags improve the runtime of the iOS applications?

To examine this question, we must analyze the speedup values obtained for each optimized version of the different applications under consideration. Fig. 12(a) presents the frequency with which a speedup greater than one was achieved for each combination of optimizations, categorized by application type.

It is evident that the only combination that consistently enhances runtime across all applications is `-Ounchecked` combined with `-Oz`. This behavior is rational for the `swiftc` compiler optimization, as `-Ounchecked` represents its most aggressive option prioritizing runtime at the expense of removing implicit type and overflow checks. However, the other two combinations utilizing `-Ounchecked` rank lower in the graph, resulting in improved runtime for only two applications with `-Ounchecked` and `-O0`, and three with `-Ounchecked` and `-O3`. Notably, neither of these combinations achieves an improvement for applications within the deep learning models category.

Regarding the LLVM optimization `-Oz`, it is typically employed to reduce the size of the compiled application and energy consumption, not to optimize runtime. Nevertheless, the three best combinations, in terms of achieving speed ups greater than one, all include this optimization. Thus, we conclude that `-Oz` has a notable impact on runtime across the applications considered.

When discussing the results of optimizations typically employed to improve runtime using the `-O` flag in `swiftc`, we see that the outcomes are inconsistent and highly dependent on the specific LLVM transformations applied. In fact, without LLVM transformations,

`swiftc` optimizations generally fail to enhance runtime, with no speed ups greater than one observed in the deep learning models category. On the other hand, while `-O3` is the LLVM optimization traditionally used to reduce runtime, all combinations that use it rank high in the graph but still perform below those that include `-Oz`.

The results indicate that improvements in runtime for each of the applications studied are clearly dependent on the LLVM transformation used, while the influence of `swiftc` transformations is generally less significant. Surprisingly, the most consistent transformation has been `-Oz` from LLVM, yielding the best results whether combined with or without `swiftc` compiler optimizations. Moreover, it is the only transformation that managed to enhance the runtime of at least one application in the deep learning models category across all combinations. In consequence, although `-O3` is typically the LLVM transformation used to achieve better runtimes, the results of this study show that for the applications considered, `-Oz` is more consistent across different types of iOS applications considered.

> LLVM compiler optimizations, especially `-Oz`, significantly improve runtime of the studied applications, while `swiftc` optimizations have a weaker impact.

RQ2. Do the most aggressive optimization flags improve the energy consumption of the iOS applications?

Compiler optimizations are typically focused on improving execution time and/or application size, so in battery-powered devices like smartphones, it is important to evaluate the energy consumption caused by these optimizations. For this reason, Fig. 12(b) shows the number of times we achieved a green up greater than one for each of the combinations considered, categorized by application type.

None of the combinations studied succeed in improving energy consumption for all the applications considered, highlighting that these compiler transformations are not actually focused on reducing energy usage, despite the importance of this task. Three of the combinations manage to improve energy consumption in all applications except one: `-Ounchecked` with `-O3` and `-O` with `-O0` fails for GhostRun, while `-O` with `-O3` does not succeed for SQLitePolybench.

As we can see, the most consistent LLVM transformation is `-O3`, which is typically aimed at improving runtime. However, its impact is highly dependent on the `swiftc` optimization used in conjunction with it; when used alone, it fails to improve energy consumption in any of the applications considered. On the other hand, the most consistent `swiftc` optimization is `-O`, which is also generally focused on runtime improvements. Similarly, its performance depends on the LLVM optimization with which it is combined. For instance, when used with `-Oz`, it only achieves green up values greater than one in two applications.

The category of applications where it is most challenging to achieve improvements in energy consumption is video games, with `-O` combined with `-O3` being the only combination that succeeds in both applications considered. This contrasts with the speed up values, where the most difficult category was deep learning models. This suggests that with more screen usage, compiler optimizations find it more challenging to achieve energy savings.

Based on the performed study, improvements in energy consumption are even more dependent on the combination of optimizations used, requiring both `swiftc` and `LLVM` optimizations. Although the `-Oz` transformation is intended to reduce application size and occasionally lower software energy consumption, it is the most inconsistent LLVM transformation for the applications studied. The most consistent transformations are those typically aimed at improving runtime: `-O3` from LLVM and `-O` from `swiftc`. When combined, they achieve green up values greater than one in most applications considered. These results differ from those obtained for speed up, as, despite being focused on improving runtime, they are not the most consistent in the studied iOS applications. However, they do consistently improve energy consumption, highlighting the fact that an improvement in runtime does not always correspond to a reduction in energy consumption, and vice versa.

> Combining `swiftc` and LLVM optimizations, in particular `-O3` and `-O`, reliably improves energy consumption, though these optimizations do not consistently reduce runtime of the studied iOS applications.

RQ3. Does an improvement in runtime correlate with an improvement in energy consumption for the iOS applications?

Throughout this work, we have observed various situations regarding how runtime and energy consumption behave under different combinations of optimizations. Typically, in most studies in the literature, an improvement in the runtime of an application or software is expected to lead to a reduction in energy consumption, as the power consumed by the application during execution usually remains constant. For example, in FlappyFlyBird, this behavior is evident for most of the combinations studied, where improving runtime results in a reduced energy footprint and vice versa.

However, there are instances, both within the FlappyFlyBird application and the other applications, where an improvement in runtime leads to increased energy consumption due to a rise in power usage during execution. In contrast, there are cases where a decrease or worsening of runtime results in improved energy consumption. Given these results, it is clear that energy consumption is not solely dependent on runtime but is also influenced by a multitude of other factors that directly affect power consumption throughout the execution of the application.

Furthermore, if we examine how many times the different optimizations achieved speed up and green up values greater than one, we can see that the most consistent optimization combinations do not align, and the best for runtime are not the same as those for energy consumption. In fact, the results from the NPSK rankings indicate that often the best compiler optimization for energy consumption is the one that performs worst in terms of runtime, and vice versa.

To analyze the relationship between runtime and energy consumption of the various optimized versions of the considered applications, Spearman correlation coefficients have been calculated for each version using a confidence level of 95%, as detailed in Table 2. A coefficient value close to one indicates a strong correlation between the variables, while a value close to zero suggests that there is no correlation. The last row reflects the Spearman correlation coefficient for all measurements of each application, regardless of the optimization flags used during compilation.

Most of the obtained coefficients are negative, with very low p-values, indicating that an increase in runtime usually leads to a decrease in energy consumption, and vice versa. However, when analyzing the overall correlation coefficients calculated for all measurements of the different optimized versions of the same application, FlappyFlyBird achieves a value of 0.5842 with p-value lower than 0.05. This suggests that for this particular application, a reduction in runtime typically results in a decrease in energy consumption, and vice versa.

Among the correlation coefficients obtained for the different studied combinations, most weak correlations, ranging from 0.3 to –0.3, tend to have higher p-values. Therefore, we cannot conclude that the variables are correlated in these cases at a confidence level of 95%, as may be seen with `-Onone -O3` for FlappyFlyBird and `-O -O3` for FileManagement, among others. Nevertheless, in each specific case, there are again numerous strong negative correlations with very low p-values for the different optimized versions of each application, with the exception of FlappyFlyBird.

Table 2
Spearman's correlation coefficient and corresponding p-values between runtime and energy consumption for each optimized version of the applications, based on 30 independent measurements.

	Disk operations		Deep learning models		Video games	
	SQLPoly	FileMng	ClassInf	SegInf	FlyBird	GhostR
`-Onone -O0`	−0.3757 4.07e−02	−0.7726 5.64e−07	−0.8745 2.68e−10	−0.9132 1.96e−12	0.6489 1.00e−04	−0.9808 1.91e−21
`-Onone -O3`	−0.5123 3.80e−03	0.2360 2.09e−01	−0.8456 4.05e−09	−0.3330 7.21e−02	0.2116 2.62e−01	−0.9648 8.68e−18
`-Onone -Oz`	−0.7571 1.28e−06	0.3339 7.13e−02	−0.9653 7.28e−18	−0.5403 2.00e−03	0.5128 3.70e−03	−0.7428 2.58e−06
`-O -O0`	−0.4852 6.56e−03	−0.9719 3.81e−19	−0.9929 2.01e−27	−0.9768 2.67e−20	0.3904 3.29e−02	−0.9769 2.67−20
`-O -O3`	−0.2476 1.87e−01	0.2908 1.19e−01	−0.9644 1.03e−17	−0.9728 2.44e−19	0.5408 2.00e−03	−0.9755 5.82e−20
`-O -Oz`	−0.4234 1.97e−02	−0.0376 8.44e−01	−0.9368 2.68e−14	−0.4669 9.20e−03	0.6849 2.96e−05	−0.9622 2.37e−17
`-Ounchecked -O0`	−0.3508 5.73e−02	−0.9551 2.54e−16	−0.2941 1.15e−01	−0.5679 1.00e−03	−0.1559 4.11e−01	−0.3775 3.97e−02
`-Ounchecked -O3`	−0.8007 1.07e−07	−0.0141 9.39e−01	−0.9359 3.24e−14	−0.7695 6.69e−07	0.0839 6.59e−01	−0.9684 1.99e−18
`-Ounchecked -Oz`	−0.8541 1.94e−09	−0.5310 2.50e−03	0.5150 3.50e−03	−0.9123 2.24e−12	−0.0318 8.67e−01	0.2801 1.34e−01
Overall	−0.4897 5.85e−59	−0.2021 4.17e−26	−0.8358 9.39e−72	−0.3167 1.04e−07	0.5842 4.17e−26	−0.7903 5.85e−59

These findings highlight the complexity of automatically achieving greener software versions, requiring a trade-off or balance between improvements in runtime and energy consumption for the application in question. It is essential to study and optimize energy consumption as an independent variable, as it may, in some cases, be completely opposed to runtime. This approach is crucial for both development and research efforts aimed at producing increasingly greener applications.

> Greener software requires balancing runtime and energy optimizations, as energy consumption often opposes runtime improvements for the studied iOS applications.

8. Threats to validity

Construct validity

Generally, using benchmarks on different types of applications and operations does not accurately reflect the actual usage patterns of a typical user when interacting with an application or smartphone, especially in the absence of studies on the behavior of the average user. Moreover, shared iOS libraries such as SQLite, CoreML, or SpriteKit are not being optimized. Therefore, future work should focus on studying the effects of optimizations not only on benchmarks focused on specific types of applications or operations but also on shared libraries and real-world usage patterns.

Our benchmark corpus, composed of small, self-contained applications, was deliberately designed to create a controlled experimental setting that isolates the impact of compiler optimizations on developer-controlled code. While this approach allows for precise measurement of compiler-level effects, we acknowledge that it may not fully capture the diversity of library reuse practices typically found in larger, production-grade iOS applications. We now explicitly recognize this as a limitation of our current study and propose expanding the analysis in future work to include production-scale applications with more extensive library reuse.

Additionally, both LLVM and `swiftc` compilers offer a wide range of compilation options aimed at both analysis and optimization. In the development of a real application intended for production, many more alternative compilation options would be considered beyond those examined here, where only the most aggressive optimization flags were used.

Internal validity

One of the main concerns is the accuracy and reliability of the measurement system, specifically the error from the Monsoon Power Monitor and the synchronization methodology. Moreover, smartphone services, including the Wi-Fi connection, remain active as they are necessary for deploying different versions of the applications. Nonetheless, to mitigate uncertainty, all results have been calculated based on 30 independent measurements.

External validity

The applications considered, as well as the benchmarks built upon them, do not necessarily generalize or represent the effects that these optimizations might have on other applications, even within the same category. A extensive analysis of the resources required by the application and the actions performed within it is necessary.

Additionally, all experimentation was performed using LLVM 15.0.0 and `swiftc` 5.10. Therefore, the same behavior may not apply to other versions. Furthermore, all tests were conducted on an iPhone running iOS 13.4.1, so altering the version of the operating system or the DUT could significantly impact the results.

Conclusion validity

To ensure the robustness of our statistical analysis, the NPSK test at a 95% confidence level to group values based on statistically significant differences is applied. This test is particularly suitable for comparing multiple groups without assuming normality. Additionally, the Spearman correlation coefficient to evaluate the strength and direction of monotonic relationships between variables is computed.

To mitigate potential threats to conclusion validity, the appropriateness of the statistical tests for the dataset was verified, and it was also ensured that the sample size (30 independent measurements per application) provides sufficient energy to detect meaningful differences. Moreover, possible biases were accounted for by maintaining consistent experimental conditions across all runs.

9. Conclusion and future work

In this paper, we present an empirical analysis of the combined effect of the most aggressive Swift and LLVM optimizations on a total of six different iOS applications, belonging to three categories: *(1)* Disk operations, *(2)* Deep learning models, and *(3)* Video games. For each application, a benchmark is created to intensively utilize the resources and libraries that characterize them. The results obtained for each combination of optimizations are analyzed statistically, characterizing the several versions of each application using the GPS-UP metrics from the literature. The results challenge the common assumption that improving runtime always leads to reduced energy consumption. In fact, the data shows that energy consumption can worsen by up to 34.1913% compared to the non-optimized version, despite achieving a 9.5370% improvement in runtime. Moreover, for the considered applications, it is generally observed that the best optimization for energy consumption often leads to the worst performance in terms of runtime, and vice versa.

To collect energy consumption measurements, a new version of the iGreenMiner energy consumption measurement system is introduced, featuring a synchronization approach that is independent of the IDE. This synchronization approach employs an intermediate application that generates signal patterns through CPU loops, allowing synchronization without relying on temporal debugging information.

The results demonstrate that, for the studied applications performing a specific task over a defined period, optimizations such as `-O` from Swift and `-O3` from LLVM consistently reduce energy consumption. However, despite being designed to improve runtime performance, these optimizations do not always achieve that goal for the studied iOS applications. In contrast, optimizations focused on code size, like `-Oz` from LLVM, tend to be more consistent in obtaining runtime improvements for the considered applications. These findings outline the complexity of optimizing software energy consumption, highlighting the need to find a balance between runtime improvements and energy efficiency to achieve increasingly greener and more sustainable software, as well as the development of a new generation of smarter compilers capable of determining this balance.

As possible future research directions, we plan to extend the study by analyzing a larger number of iOS applications, including additional categories beyond disk operations, deep learning models, and video games. This expansion aims to improve the generalizability of our findings and capture a broader spectrum of real-world workloads. Furthermore, we intend to incorporate real user behavior patterns across multiple applications as well as considering other types of applications. Additionally, we aim to analyze the impact of other compiler flags used in smartphone application development. We also intend to model a multi-objective combinatorial optimization problem with several compilation flags to identify pseudo-optimal versions of the application that minimize both energy consumption and runtime. This approach would allow selecting the most appropriate version by determining the trade-off between improvements in runtime and energy consumption.

CRediT authorship contribution statement

José Miguel Aragón-Jurado: Writing – review & editing, Writing – original draft, Visualization, Methodology, Investigation, Data curation, Conceptualization. **Abdul Ali Bangash:** Writing – review & editing, Software, Resources. **Bernabé Dorronsoro:** Writing – review & editing, Supervision, Conceptualization. **Karim Ali:** Writing – review & editing, Resources. **Abram Hindle:** Writing – review & editing, Supervision, Methodology. **Patricia Ruiz:** Writing – review & editing, Supervision, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This publication is part of project PID2022-137858OB-I00 funded by MICIU/AEI/10.13039/501100011033 and by "ERDF A way of making Europe", and eFracWare (TED2021-131880B-I00) funded by MICIN/AEI/10.13039/501100011033 and the European Union NextGeneration EU/PRTR. We also would like to thank the Spanish Ministry of Science and Innovation and ERDF for the support under RED2022-134647-T. J.M. Aragón-Jurado would like to acknowledge the Spanish Ministerio de Ciencia, Innovación y Universidades for the support through FPU21/02026 grant. Abram Hindle is supported by the NSERC Discovery Grant program.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.suscom.2025.101166.

Data availability

The source code for the different experiments performed, along with the developed benchmarks, is available at https://doi.org/10.5281/zenodo.14888155.

References

[1] B.X. Lee, F. Kjaerulf, S. Turner, L. Cohen, P.D. Donnelly, R. Muggah, R. Davis, A. Realini, B. Kieselbach, L.S. MacGregor, et al., Transforming our world: Implementing the 2030 agenda through sustainable development goal indicators, J. Public Health Policy 37 (2016) 13–31.

[2] T. Johann, M. Dick, E. Kern, S. Naumann, Sustainable development, sustainable software, and sustainable software engineering: an integrated approach, in: 2011 International Symposium on Humanities, Science and Engineering Research, IEEE, 2011, pp. 34–39.

[3] M.E. Ibrahim, M. Rupp, S.-D. Habib, Compiler-based optimizations impact on embedded software power consumption, in: 2009 Joint IEEE North-East Workshop on Circuits and Systems and TAISA Conference, IEEE, 2009, pp. 1–4.

[4] J.M. Aragón-Jurado, J.C. de la Torre, P. Ruiz, P.L. Galindo, A.Y. Zomaya, B. Dorronsoro, Automatic software tailoring for optimal performance, IEEE Trans. Sustain. Comput. 9 (3) (2024) 464–481, http://dx.doi.org/10.1109/TSUSC.2023.3330671.

[5] Apple, Apple Inc., 2025, https://www.apple.com, (accessed 01 November 2024).

[6] Apple, iOS - Apple, 2025, https://www.apple.com/ios/, (accessed 01 November 2024).

[7] Apple, Swift.org - Welcome to Swift.org, 2025, https://swift.org/, (accessed 01 November 2024).

[8] D. Domínguez-Álvarez, A. Gorla, J. Caballero, On the usage of programming languages in the iOS ecosystem, in: 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation, SCAM, IEEE, 2022, pp. 176–180.

[9] A.A. Bangash, D. Tiganov, K. Ali, A. Hindle, Energy efficient guidelines for iOS core location framework, in: 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2021, pp. 320–331.

[10] A.A. Bangash, K. Eng, Q. Jamal, K. Ali, A. Hindle, Energy consumption estimation of API-usage in smartphone apps via static analysis, in: 2023 IEEE/ACM 20th International Conference on Mining Software Repositories, MSR, IEEE, 2023, pp. 272–283.

[11] LLVM, The LLVM compiler infrastructure, 2025, https://llvm.org/, (accessed 01 November 2024).

[12] Apple, Xcode - Apple developer, 2025, https://developer.apple.com/xcode/, (accessed 01 November 2024).

[13] A. Schuler, G. Kotsis, A systematic review on techniques and approaches to estimate mobile software energy consumption, Sustain. Comput.: Inform. Syst. (2023) 100919.

[14] M. Ciman, O. Gaggi, An empirical analysis of energy consumption of cross-platform frameworks for mobile development, Pervasive Mob. Comput. 39 (2017) 214–230.

[15] A. Hindle, A. Wilson, K. Rasmussen, E.J. Barlow, J.C. Campbell, S. Romansky, GreenMiner: A hardware based mining software repositories software energy consumption framework, in: Proceedings of the 11th Working Conference on Mining Software Repositories, 2014, pp. 12–21.

[16] L.M. Fischer, L.B. de Brisolara, J.C.B. De Mattos, SEMA: An approach based on internal measurement to evaluate energy efficiency of android applications, in: 2015 Brazilian Symposium on Computing Systems Engineering, SBESC, IEEE, 2015, pp. 48–53.

[17] A. Carette, M.A.A. Younes, G. Hecht, N. Moha, R. Rouvoy, Investigating the energy impact of Android smells, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2017, pp. 115–126.

[18] M.A. Hoque, M. Siekkinen, K.N. Khan, Y. Xiao, S. Tarkoma, Modeling, profiling, and debugging the energy consumption of mobile devices, ACM Comput. Surv. 48 (3) (2015) 1–40.

[19] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, A. De Lucia, Software-based energy profiling of Android apps: Simple, efficient and reliable? in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER, IEEE, 2017, pp. 103–114.

[20] C. Wang, F. Yan, Y. Guo, X. Chen, Power estimation for mobile applications with profile-driven battery traces, in: International Symposium on Low Power Electronics and Design, ISLPED, IEEE, 2013, pp. 120–125.

[21] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R.P. Dick, Z.M. Mao, L. Yang, Accurate online power estimation and automatic battery behavior based power model generation for smartphones, in: Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, 2010, pp. 105–114.

[22] L. Corral, A.B. Georgiev, A. Sillitti, G. Succi, A method for characterizing energy consumption in Android smartphones, in: 2013 2nd International Workshop on Green and Sustainable Software, GREENS, IEEE, 2013, pp. 38–45.

[23] S.A. Chowdhury, A. Hindle, GreenOracle: Estimating software energy consumption with energy measurement corpora, in: Proceedings of the 13th International Conference on Mining Software Repositories, 2016, pp. 49–60.

[24] A. Pathak, Y.C. Hu, M. Zhang, P. Bahl, Y.-M. Wang, Fine-grained power modeling for smartphones using system call tracing, in: Proceedings of the Sixth Conference on Computer Systems, 2011, pp. 153–168.

[25] S. Romansky, N.C. Borle, S. Chowdhury, A. Hindle, R. Greiner, Deep green: Modelling time-series of software energy consumption, in: 2017 IEEE International Conference on Software Maintenance and Evolution, ICSME, IEEE, 2017, pp. 273–283.

[26] L. Li, X. Wang, F. Qin, EnergyDx: Diagnosing energy anomaly in mobile apps by identifying the manifestation point, in: 2020 IEEE 40th International Conference on Distributed Computing Systems, ICDCS, IEEE, 2020, pp. 256–266.

[27] D. Li, S. Hao, W.G. Halfond, R. Govindan, Calculating source line level energy information for Android applications, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, 2013, pp. 78–89.

[28] X. Li, Y. Yang, Y. Liu, J.P. Gallagher, K. Wu, Detecting and diagnosing energy issues for mobile applications, in: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020, pp. 115–127.

[29] G. Huang, H. Cai, M. Swiech, Y. Zhang, X. Liu, P.A. Dinda, DelayDroid: an instrumented approach to reducing tail-time energy of Android apps, Sci. China Inf. Sci. 60 (1) (2017) 12106.

[30] R.W. Ahmad, A. Naveed, J.J. Rodrigues, A. Gani, S.A. Madani, J. Shuja, T. Maqsood, S. Saeed, Enhancement and assessment of a code-analysis-based energy estimation framework, IEEE Syst. J. 13 (1) (2018) 1052–1059.

[31] S. Abdulsalam, Z. Zong, Q. Gu, M. Qiu, Using the Greenup, Powerup, and Speedup metrics to evaluate software energy efficiency, in: 2015 Sixth International Green and Sustainable Computing Conference, IGSC, IEEE, 2015, pp. 1–8.

[32] C.L. Goues, T. Nguyen, S. Forrest, W. Weimer, GenProg: A generic method for automatic software repair, IEEE Trans. Softw. Eng. 38 (1) (2012) 54–72.

[33] J.W. Nimmer, Automatic Generation and Checking of Program Specifications (Ph.D. thesis), Massachusetts Institute of Technology (MIT), Boston, USA, 2002.

[34] A. Blot, J. Petke, A comprehensive survey of benchmarks for automated improvement of software's non-functional properties, 2022, arXiv:2212.08540.

[35] B. Dorronsoro, J.M. Aragón-Jurado, J. Jareño, J.C. de la Torre, P. Ruiz, A survey on automatic source code transformation for green software generation, in: M.A. Abraham (Ed.), Encyclopedia of Sustainable Technologies (Second Edition), second ed., Elsevier, Oxford, 2024, pp. 765–779, http://dx.doi.org/10.1016/B978-0-323-90386-8.00122-4.

[36] R. Verdecchia, R.A. Saez, G. Procaccianti, P. Lago, Empirical evaluation of the energy impact of refactoring code smells, in: International Conference on ICT for Sustainability, 2018, pp. 365–383.

[37] B. Diniz, D. Guedes, W. Meira, R. Bianchini, Limiting the power consumption of main memory, SIGARCH Comput. Archit. News 35 (2) (2007) 290–301.

[38] M. Bokhari, B. Alexander, A hybrid distributed EA approach for energy optimisation on smartphones, Empir. Softw. Eng. 27 (2022) http://dx.doi.org/10.1007/s10664-022-10188-5.

[39] Y. Hakimi, R. Baghdadi, Y. Challal, A hybrid machine learning model for code optimization, Int. J. Parallel Program. 51 (2023) 309–331, http://dx.doi.org/10.1007/s10766-023-00758-5.

[40] W.B. Langdon, M. Harman, Optimizing existing software with genetic programming, IEEE Trans. Evol. Comput. 19 (1) (2015) 118–135.

[41] B.R. Bruce, J. Petke, M. Harman, Reducing energy consumption using genetic improvement, in: ACM Conference on Genetic and Evolutionary Computing, GECCO, 2015, pp. 1327–1334.

[42] A.A. Bangash, K. Ali, A. Hindle, Black box technique to reduce energy consumption of Android Apps, in: 2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE-NIER, 2022, pp. 1–5, http://dx.doi.org/10.1145/3510455.3512795.

[43] M. Kandemir, N. Vijaykrishnan, M.J. Irwin, Power Aware Computing, Springer, 2002.

[44] V. Venkatachalam, M. Franz, Power reduction techniques for microprocessor systems, ACM Comput. Surv. 37 (2005) 195–237, http://dx.doi.org/10.1145/1108956.1108957.

[45] A. Sachan, P. Srivastav, B. Ghoshal, Learning based application driven energy aware compilation for GPU, Microprocess. Microsyst. 94 (2022) 104664, http://dx.doi.org/10.1016/j.micpro.2022.104664.

[46] J.C. de la Torre, P. Ruiz, B. Dorronsoro, P.L. Galindo, Analyzing the influence of LLVM code optimization passes on software performance, in: International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, Springer, 2018, pp. 272–283.

[47] K.D. Cooper, P.J. Schielke, D. Subramanian, Optimizing for reduced code space using genetic algorithms, in: Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems, LCTES, ACM, 1999, pp. 1–9.

[48] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, J. Wawrzynek, AutoPhase: Compiler phase-ordering for HLS with deep reinforcement learning, in: 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM, 2019, p. 308, http://dx.doi.org/10.1109/FCCM.2019.00049.

[49] U. Garciarena, R. Santana, Evolutionary optimization of compiler flag selection by learning and exploiting flags interactions, in: Proceedings of the 2016 on Genetic and Evolutionary Computation Conference, GECCO, ACM, 2016, pp. 1159–1166.

[50] J. Chen, N. Xu, P. Chen, H. Zhang, Efficient compiler autotuning via Bayesian optimization, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE, 2021, pp. 1198–1209.

[51] J.M. Aragón-Jurado, J.C. de la Torre, P. Ruiz, B. Dorronsoro, Automatic software tailoring for Green Internet of Things, Internet Things (2025) 101521.

[52] J.C. de la Torre, J. Jareño, J.M. Aragón-Jurado, S. Varrette, B. Dorronsoro, Source code obfuscation with genetic algorithms using LLVM code optimizations, Log. J. IGPL (2024) jzae069, http://dx.doi.org/10.1093/jigpal/jzae069.

[53] J. Pallister, S.J. Hollis, J. Bennett, Identifying compiler options to minimize energy consumption for embedded platforms, Comput. J. 58 (1) (2015) 95–109.

[54] E. Damasceno, F. Queiroz, L. Siqueira, T. Rodrigues, M. Amaris, Comparative analysis of compiler efficiency: Energy consumption metrics in high-performance computing domains, in: Simpósio em Sistemas Computacionais de Alto Desempenho, SSCAD, SBC, 2024, pp. 252–263.

[55] J.M. Aragón-Jurado, P. Ruiz, B. Dorronsoro, R. Thawonmas, Green gaming: Automated energy consumption reduction for doom engine, IEEE Consum. Electron. Mag. (2025) 1–6, http://dx.doi.org/10.1109/MCE.2025.3565227.

[56] N. van Kempen, H.-J. Kwon, D.T. Nguyen, E.D. Berger, It's not easy being green: On the energy efficiency of programming languages, 2024, arXiv preprint arXiv:2410.05460.

[57] E. Jiménez, A. Gordillo, C. Calero, M.Á. Moraga, F. García, Does the compiler or interpreter version influence the energy consumption of programming languages? Sci. Comput. Program. (2025) 103270.

[58] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J.P. Fernandes, J. Saraiva, Ranking programming languages by energy efficiency, Sci. Comput. Program. 205 (2021) 102609.

[59] N. Marini, L. Pampaloni, F. Di Martino, R. Verdecchia, E. Vicario, Green AI: Which programming language consumes the most?, 2024, arXiv preprint arXiv:2501.14776.

[60] D. Gadioli, R. Nobre, P. Pinto, E. Vitali, A.H. Ashouri, G. Palermo, J. Cardoso, C. Silvano, SOCRATES - A seamless online compiler and system runtime autotuning framework for energy-aware applications, in: Proceedings of the 2018 Design, Automation and Test in Europe Conference and Exhibition, DATE 2018, Vol. 2018-January, 2018, pp. 1143–1146, http://dx.doi.org/10.23919/DATE.2018.8342183.

[61] Ios-deploy, 2023, https://github.com/ios-control/ios-deploy, (accessed 01 November 2024).

[62] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, IEEE Trans. Softw. Eng. 43 (1) (2016) 1–18.

[63] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto, The impact of automated parameter optimization on defect prediction models, IEEE Trans. Softw. Eng. 45 (7) (2018) 683–711.

[64] A. Roy, S.M. Rumble, R. Stutsman, P. Levis, D. Mazieres, N. Zeldovich, Energy management in mobile devices with the cinder operating system, in: Proceedings of the Sixth Conference on Computer Systems, 2011, pp. 139–152.

[65] V.M.F. Jacques, N. Alizadeh, F. Castor, A study on the battery usage of Deep Learning frameworks on iOS devices, in: Proceedings of the IEEE/ACM 11th International Conference on Mobile Software Engineering and Systems, 2024, pp. 1–11.

[66] Y. Choi, S. Park, S. Jeon, R. Ha, H. Cha, Optimizing energy consumption of mobile games, IEEE Trans. Mob. Comput. 21 (10) (2021) 3744–3756.
[67] B. Zhou, I. Neamtiu, R. Gupta, A cross-platform analysis of bugs and bug-fixing in open source projects: desktop vs. Android vs. iOS, in: Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering, 2015, pp. 1–10.
[68] M. Böhmer, A. Krüger, A study on icon arrangement by smartphone users, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, 2013, pp. 2137–2146.
[69] T. Shakya, FileManagement_Practice, 2022, https://github.com/Tilak1028-st/FileManagement_Practice, (accessed 01 November 2024).
[70] A. McIntosh, S. Hassan, A. Hindle, What can Android mobile app developers do about the energy consumption of machine learning? Empir. Softw. Eng. 24 (2019) 562–601.
[71] tucan9389, SimpleClassification-CreateML-CoreML, 2018, https://github.com/tucan9389/SimpleClassification-CreateML-CoreML, (accessed 01 November 2024).
[72] Photoroom, Coreml-benchmark, 2019, https://github.com/Photoroom/coreml-benchmark, (accessed 01 November 2024).
[73] Y. Seo, R. Dolan, M. Buchanan-Oliver, Playing games: advancing research on online and mobile gaming consumption, Internet Res. 29 (2) (2019) 289–292.
[74] F. Mäyrä, K. Alha, Mobile gaming, in: The Video Game Debate 2, Routledge, 2020, pp. 107–120.
[75] A. Eleev, flappy-fly-bird, 2023, https://github.com/eleev/flappy-fly-bird, (accessed 01 November 2024).
[76] G. Kunz, ghost-run-game, 2020, https://github.com/gabrielkunz/ghost-run-game, (accessed 01 November 2024).
[77] J.M. Aragon-Jurado, Data from does faster mean greener? Runtime and energy trade-offs in iOS applications with compiler optimizations, 2025, http://dx.doi.org/10.5281/zenodo.14888155.