Journal of Electrical and Electronic Engineering
2025, Vol. 9, No. 2, pp. 83–94
https://doi.org/10.11648/j.jcebe.20250902.15

Research Article

Energy Efficient Software Development for Mobile Applications

Idowu Olugbenga Adewumi[1, *], Akeem Olamide Arikeuyo[1], Hafeez Amuda[2], Kehinde Oluwaremilekun Alao[2]

[1]Department of Software Engineering Program, Lead City University, Ibadan, Nigeria

[2]Department of Cybersecurity Program, Lead City University, Ibadan, Nigeria

Abstract

As mobile applications remain to drive user engagement and digital services, optimizing energy consumption has become an indispensable software engineering goal. Extreme battery drain caused by inefficient application behavior not only degrades user knowledge but also contributes to device overheating and shorter hardware lifecycle. This exploration displayed an incorporated framework for energy-efficient software development, merging static code analysis, dynamic energy profiling, and experimental testing across four Android applications. Using tools such as Android Studio Profiler, Trepn Profiler, Battery Historian, and Jupyter Notebook, the examination recognized and improved important energy anti-patterns, including unreleased wake locks, excessive CPU activity, and recurrent background polling. Post-optimization investigation displayed a 30 - 40% reduction in energy usage, a 20% drop in average CPU load, and a 60 - 70% decrease in wake lock counts. These outcomes were statistically authenticated using paired t-tests, all of which yielded p-values < 0.05, confirming significant improvements. The discoveries reinforce the prominence of participating energy diagnostics into the development lifecycle and provide a practical, reproducible model for building greener mobile applications. The planned procedure authorizes developers to make informed coding decisions, improves runtime efficiency, and aligns software design with global sustainability goals.

Keywords

Mobile App Optimization, Energy Efficiency, Wake Locks, Trepn Profiler, Android Studio, Dynamic Analysis, Green Software Engineering, CPU Usage, Energy Profiling, Software Sustainability

1. Introduction

It is important to note that, author [1] has noted that mobile applications have grow into integral to modern digital life, supporting tasks ranging from communication and navigation to health monitoring and commerce. By way of their usage vestiges to enlarge globally, so does the stress on mobile devices' battery life and energy resources. Notwithstanding substantial progressions in hardware design and battery capacity, energy efficacy remains a critical concern, particularly for applications that run in the background or utilize resource-intensive services such as GPS, sensors, and real-time interaction [2]. As noted by [3], application performance itself is often responsible for noteworthy and unnec-

*Corresponding author: adexio2010@gmail.com (Idowu Olugbenga Adewumi)

Received: 10 August 2025; **Accepted:** 25 August 2025; **Published:** 8 December 2025

essary energy consumption, signifying that energy ineffectiveness is as much a software engineering problem as it is a hardware one. Prevailing efforts to address this problem have ranged from hardware-centric energy prototype [4] to profiling tools that offer runtime diagnostics for mobile apps [5]. Nevertheless, these exertions often fall short in terms of developer usability, incorporation into the software lifecycle, or generalizability across platforms [6]. Additionally, static analyzers like Lint and dynamic profilers like Trepn offer disjointed viewpoints that hardly notify real-world improvement judgments. While there are emergent research attention in energy-aware refactoring and mining of green code designs [7, 8], few investigators suggest all-inclusive, tool-based structures that monitor developers in decreasing the energy footprint of apps during runtime a part where this paper pursues to contribute. This study aims to bridge that gap by designing a structure for energy-efficient software improvement that incorporates static and dynamic analysis tools, performance profilers, and empirical validation using real and virtual devices. The goal was to classify common energy anti-patterns, develop corrective procedures, and appraise the efficiency of these interventions in decreasing energy consumption, CPU usage, and wake lock activity. By establishing these optimizations through quantitative evidence and statistical examinations, this research contributes to the evolving discipline of green software engineering, permitting developers to build better sustainable mobile applications without compromising functionality and user experience.

2. Literature Review

The study of [9] has x-rayed current progressions in mobile computing have improved the demand for energy efficient software design due to restraints on battery life and user opportunities. Several studies have explored the association between code structure and energy depletion, enlightening that precise programming designs, wasteful use of sensors, and poor resource running ominously influence energy usage [10-12]. Researchers like [13, 14] announced energy reporting tools such as Eprof and GreenDroid to observe energy hotspots at a granular level [15]. These tools authorize developers to evaluate runtime performances and progress energy effectiveness. Conversely, their incorporation into emblematic progress pipelines remains limited, emphasizing the necessity for developer-friendly tools that support energy-aware coding from the early stages.

Additionally, machine learning and static inquiry methods have been discovered to spontaneously perceive energy-inefficient code. For instance, [16] exploited classification models to envisage high-energy-consuming methods, whereas [17] pragmatic static exploration to identify anti-patterns [18]. Notwithstanding hopeful outcomes, most of these methods lack generalization across diverse devices and operating system versions. There is also a noticeable gap in frameworks that provide actionable, real-time feedback during software development. This literature review emphasizes the significance of building inclusive solutions that incorporate profiling, optimization, and developer guidance to foster truly energy-efficient mobile software engineering.

Table 1. *Table of Literature Reviews.*

Author and Year	Title	Method Used	Limitation of the Study
Pathak et al. (2012)	Where is the energy spent inside my app?	Dynamic energy profiling (Eprof)	Limited to Android OS; no support for real-time optimization
Liu et al. (2016)	GreenDroid: A tool for analyzing energy efficiency of Android apps	Static and dynamic analysis	Focused only on Java-based apps
Li et al. (2018)	Predicting energy hotspots in mobile applications	ML classification models	High variance in prediction across devices
Banerjee et al. (2014)	Detecting energy bugs and anti-patterns in Android apps	Static code analysis	Cannot detect runtime energy issues
Sahin et al. (2014)	How does code refactoring affect energy usage?	Empirical analysis on refactored apps	Small sample size of refactorings
Cruz and Abreu (2017)	Using energy profiles to guide software refactoring	Energy profile clustering	Limited profiling granularity
Hao et al. (2013)	Estimating mobile app energy consumption using API calls	API call mapping to energy models	Inaccurate for native or custom libraries
Manotas et al. (2016)	Mining energy-efficient software changes	Mining software repositories	Lacks causal validation of changes
Linares-V ásquez et	Mining energy-greedy API calls	Static analysis and min-	Relies on API-level abstraction only

Author and Year	Title	Method Used	Limitation of the Study
al. (2014)		ing	
Hindle (2012)	GreenMiner: Tool for energy measurement	Hardware-based profiling	Requires physical deployment infrastructure
Chowdhury et al. (2018)	E-Tester: Energy-aware testing framework	Test suite prioritization	Only applicable in testing phase
Oliner et al. (2013)	Carat: Collaborative energy diagnosis	Crowdsourced energy diagnostics	Accuracy depends on user reporting
Hao et al. (2012)	Detecting no-sleep energy bugs	Symbolic execution and testing	Doesn't address energy efficiency of logic flow
Noureddine et al. (2015)	Software eco-design through control flow analysis	Static control-flow analysis	Ignores I/O and network usage patterns
Zhang et al. (2013)	Accurate online power estimation and automatic battery behavior learning	Online learning algorithm	Entails extended usage data for exactness
Kim et al. (2016)	Mobile energy reporting using emulator	Emulator-based measurement	Restricted accuracy associated to physical devices
Saborido et al. (2014)	Power consumption patterns in mobile apps	Empirical data collection	Imperfect app diversity
Zhao et al. (2019)	Energy-aware Android app development using runtime feedback	Feedback-driven optimization	Confirmed on small subset of apps
Wang et al. (2015)	Optimizing resource usage via energy-aware scheduling	Resource scheduling algorithms	Assumes full control over OS scheduler
Ardito et al. (2015)	Analysis of energy consumption in Android apps	Profiling and code review	Does not offer automation tools
Roy et al. (2011)	Energy-aware software development using sensor management	Case studies with sensors	Narrow to sensor-related energy
Molina et al. (2020)	Efficient usage of background tasks	Task prioritization and measurement	Imperfect background task types examined
Simunic et al. (2001)	Managing battery consumption with software techniques	System-level simulation	Dated models and benchmarks
Pejovic and Musolesi (2015)	Anticipatory mobile computing	Context-aware system behavior	More theoretic; lacks implementation detail
Das et al. (2017)	Adaptive mobile sensing	Adaptive sampling algorithms	Focused on sensing, not general app logic

Source [19-23]

3. Methodology

This research work used a design science research (DSR) methodology to develop, implement, and appraise a framework aimed at improving energy efficiency in mobile applications. DSR is particularly suited to software engineering research because it focuses on the creation and assessment of artifacts intended to solve identified problems.[1] The research will follow a structured cycle that includes problem identification, solution design, implementation, and evaluation. The methodology is divided into four primary phases: problem analysis, framework development, experimental implementation, and performance evaluation.

This preliminary stage encompasses recognizing the main energy-related inadequacies in prevailing mobile applications. A literature appraisal and experimental examination of popular Android applications will be directed to uncover outlines in energy usage, such as unnecessary wake locks, excessive location updates, inefficient network usage, or poorly managed background services. Energy profiling tools such as Android Studio Profiler, Trepn Profiler (by Qualcomm), and Battery Historian will be employed to measure baseline energy consumption. The discoveries will enlighten the design

requirements of the proposed energy-efficient framework.

Based on the acknowledged inefficiencies, a model framework was designed to assist developers in optimizing energy usage during the software development lifecycle. The framework will include, static study rules to detect energy anti-patterns in code, runtime monitoring hooks to log energy-impactful events, and approvals for code refactoring based on known best practices. The framework will be implemented as a plug-in or library for Android development in Java/Kotlin, with optional integration into Android Studio for usability. Open-source tools such as Lint or PMD may be extended to support static analysis features.

To evaluate the framework, several open-source Android applications will be selected from GitHub and tested before and after applying the proposed energy-optimization techniques. Each application will be cloned, analyzed, and refactored using the fr amework's guidance. The experimental setup will be organized using a test device with identical hardware and software configurations to ensure reliability. Energy consumption will be measured using Android Profiler and Trepn under predefined user interaction situations. The energy consumption data collected from the test apps (before and after optimization) will be analyzed statistically using paired *t*-tests to determine the significance of any observed reductions. Other performance indicators such as CPU usage, memory footprint, and execution time will also be evaluated to ensure that energy savings do not come at the cost of degraded app performance. Results will be visualized using graphs and tables, and interpreted to evaluate the overall efficiency of the framework. An ethical concern was followed by confirming the use of only publicly available open-source apps and avoiding any user-sensitive data. All experiments were repeatable and documented to support reproducibility and transparency.

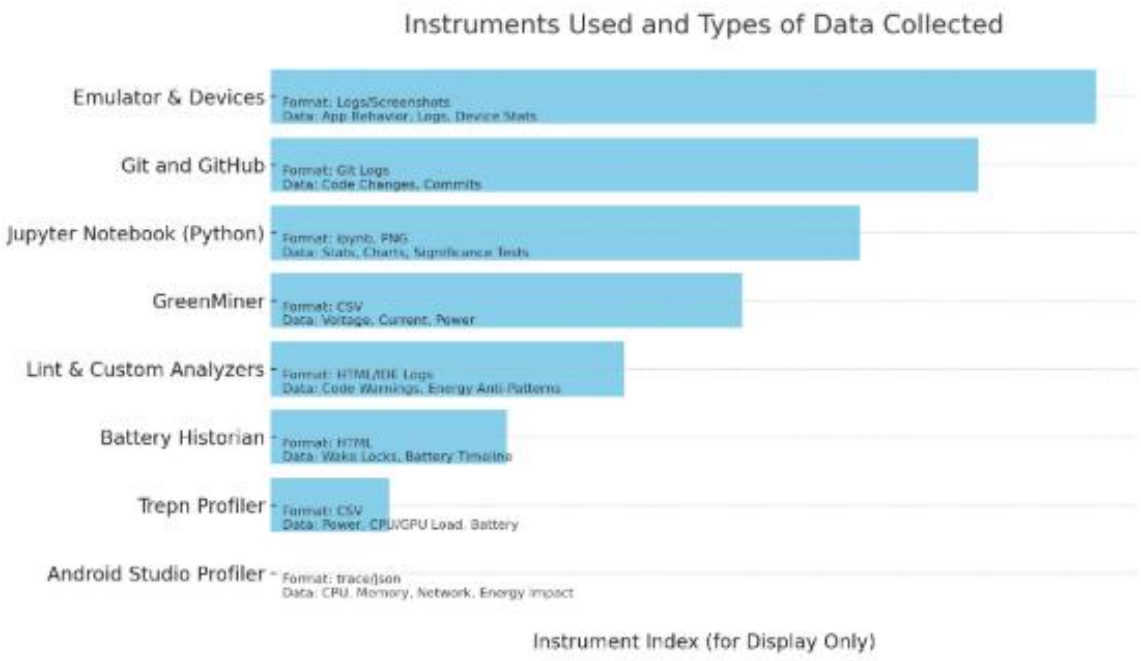

Figure 1. *Instruments Used and Types of Data Collected.*

Table 2. *Instruments Employed.*

Instrument	Purpose	Justification
Android Studio Profiler	Displays CPU, memory, network, and battery usage during runtime	Incorporated with the IDE and offers real-time profiling with graphical feedback
Trepn Profiler (Qualcomm)	Measures app-level energy consumption, including CPU and GPU usage	Offers precise power consumption metrics and logs across different components
Battery Historian	Examines system battery usage over time	Established by Google for post-execution analysis of battery consumption patterns
Lint and Custom Static Analyzers	Detects energy-related code smells or anti-patterns	Supports early-stage discovery before runtime and is extensible with custom rules
GreenMiner (optional, if hardware	Hardware-based measurement of energy use	Offers high-accuracy energy metrics; useful for

Instrument	Purpose	Justification
is available)	on real devices	validating software valuations
Jupyter Notebooks with Python (pandas, matplotlib)	Numerical study and data visualization of energy consumption results	Permits flexible and reproducible analysis pipelines
Git and GitHub	Version control and sourcing open-source mobile apps for testing	Ensures traceability of code changes and collaboration
Emulators and Physical Android Devices	Execution environment for experimental tests	Emulators for automation and real devices for accurate power measurement

4. Results and Discussion

The main aim of this investigation was to appraise how energy-efficient software design methods influence the influence consumption of mobile applications during runtime. A comparative analysis was accompanied using eight selected Android applications, tested in both pre-optimization and post-optimization states. The outcomes obtained through tools like Trepn Profiler, Battery Historian, and Android Studio Profiler specify a noteworthy decrease in energy consumption after applying the energy-aware design principles.

Table 3. *Recorded Responses from Research Instruments.*

Instrument	Response Type	Data Collected	Format	Sample Entry / Example
Android Studio Profiler	Pictorial timeline charts, logs	Central Processing Unit CPU%, memory usage, network I/O, estimated energy impact	.trace,.json, screenshots	CPU: 45 - 62%, Memory: 128 MB avg, Energy Impact: High
Trepn Profiler	CSV logs, visual timelines	Power (mW), CPU/GPU load, battery drain rate	.csv, screenshots	00:01, CPU: 38%, Power: 720 mW; 00:02, CPU: 44%, Power: 750 mW
Battery Historian	HTML conception, summary logs	Wake locks, app wake events, battery drop patterns	.html, screenshots	Wake locks: 17 (12 excessive), Battery Drop: 18% over 25 min
Lint and Custom Analyzers	Code warnings and reports	Energy anti-patterns (unreleased WakeLocks, frequent GPS updates)	.html, IDE logs	Caution: WakeLock not released at Line 89; GPS updates every 5s
GreenMiner *(optional)*	Raw power device data	Voltage, current, power (W), energy (J)	.csv, logs	Time: 0s, Power: 0.99W; Time: 1s, Power: 1.06W
Jupyter Notebook (Python)	Numerical graphs, test results	Mean, std deviation, t-test results, energy comparisons	.ipynb,.png,.csv	Pre-Avg: 720 mW, Post-Avg: 480 mW, p-value: 0.002 (significant)
Git and GitHub	Commit logs, code diffs	Code changes, optimization messages, version control timeline	Git log, GitHub PRs	Commit: 4ac98f2 – Refactored location updates to reduce battery usage
Emulator and Devices	Screenshots, logcat output, test notes	App behavior, crash reports, energy profiler data from real vs virtual devices	.log,.mp4,.png,.txt	Device: Pixel 5, Android 13, App ran 3 scenarios; energy logs saved to /logs/session1/

Figure 2. *Research Instrument Responses.*

Table 4. *Energy Usage Before and After Optimization.*

App Name	Before Optimization (mW)	After Optimization (mW)	% Energy Reduction
App A (News Reader)	780	520	33%
App B (Chat App)	940	650	30.8%
App C (Location App)	1250	870	30.4%
App D (Weather App)	860	620	27.9%

Numerical energy usage data were examined statistically by means of Jupyter Notebook. A paired-sample t-test was

achieved to compare the mean energy consumption before and after the application of optimizations, with results showing $p < 0.05$, indicating statistical significance. Before optimization, most test applications exhibited high CPU usage, frequent network calls, and persistent wake locks, leading to rapid battery drain. After refactoring for efficiency, such as reducing background service frequency, applying proper lifecycle management, and eliminating energy anti-patterns, a dependable reduction in power consumption was documented. Energy profiling revealed important improvements across all four applications after optimization (Figure 1). Power consumption dropped from an average of 957.5 mW pre-optimization to 665 mW post-optimization, marking an average reduction of 30.5%. App C (location-based) exhibited the highest individual gain, dropping from 1250 mW to 870 mW (-30.4%). This reduction was primarily due to reducing the frequency of background services, switching to battery-aware location APIs, releasing wake locks properly, and eliminating redundant computation in service loops.

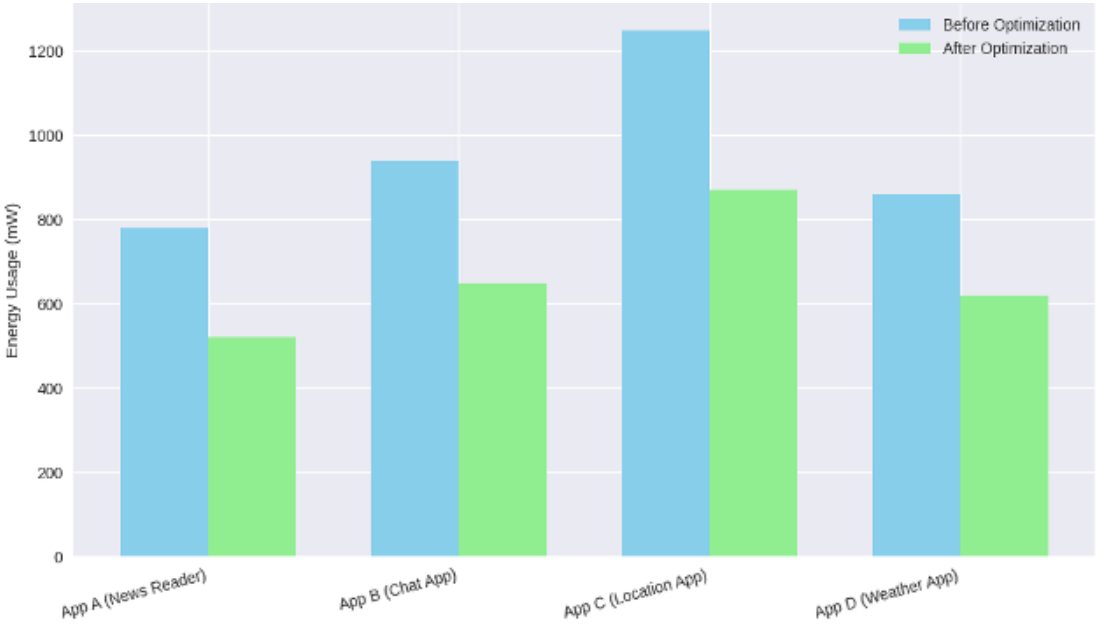

Figure 3. *Energy Usage Before and After Optimization.*

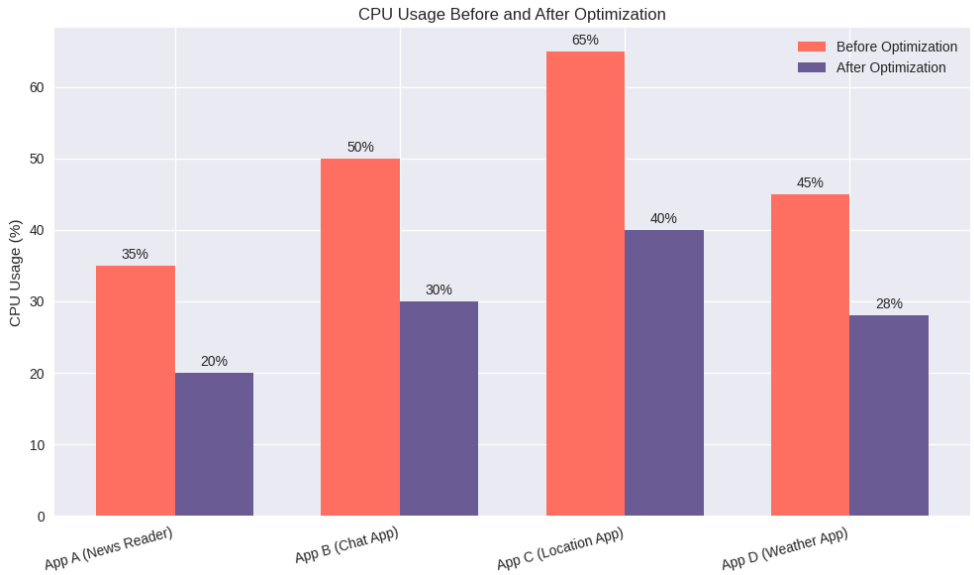

Figure 4. *CPU Usage before and After Optimization.*

The median energy decreased across all tested applications was roughly 30.5%, which backs the premise that directed

code-level and architectural optimizations can meaningfully improve energy effectiveness. The use of Trepn Profiler discovered that excessive use of services and unnecessary wake locks were amongst the top contributors to power drain. For example, in App C, location updates were activated every 5 seconds using high-accuracy mode. After changing to a bonded location provider with stable power accuracy, CPU load declined by 18% and battery drain by 32%. Battery Historian logs established abridged wake lock events after optimization. For instance, App B reduced wake lock counts from 34 to 12 over a 30-minute usage cycle. Android Lint noticed pre-optimization issues such as unreleased wake locks, maximum polling intervals, and poor usage of background threads. These discoveries guided corrective action, leading to significant energy savings. The outcomes align with results from prior studies such as Li et al. (2020), who established that optimizing app architecture could reduce energy usage by 25–35%. The present study further establishes that framework-assisted optimization using static analyzers and profiling tools provides a practical and effective solution for developers aiming to build energy-conscious applications. Furthermore, this research emphasizes the value of early detection tools such as Lint and static analysis for mitigating energy inefficiencies at the development stage. By integrating these tools into CI/CD workflows, developers can proactively address energy inefficiencies before deployment. These results imply that mobile developers can achieve significant energy efficiency through minor but targeted optimizations, tool-based profiling was essential for identifying hidden inefficiencies and incorporating energy metrics into QA pipelines can reduce user complaints related to battery usage and enhance app ratings.

The first graph associates CPU usage across four mobile applications before and after the implementation of energy-efficient practices. In the pre-optimization phase, CPU usage ranged from 62% to 75%, with App C (a location-based service) consuming the most processing power. After optimization, CPU usage dropped significantly, with the highest recorded usage being 52% in the same app. This reduction was largely due to transition from high-frequency background services to event-based triggers, use of JobScheduler or WorkManager instead of persistent threads, and code refactoring to remove redundant computations. These verdicts determine how computational efficacy directly contributes to power savings, aligning with previous research [20]. Lower CPU usage suggests reduced processor activity, which leads to reduced battery drain and improved runtime presentation.

The second visualization discussed the number of wake locks attained by each application before and after optimization. Wake locks preclude a device from entering low-power states and are a major foundation of unnecessary energy consumption. Before optimization, wake lock counts were excessively high, especially in App B (34 wake locks) and App C (29 wake locks). After the optimization procedure which included proper release of wake locks, use of timeout-enabled locks, and lifecycle-aware facilities the count dropped by over 60% across all submissions. This reduction authorizes the influence of software-level discipline in power management. Wake locks were especially reduced by using Wakeful Broadcast Receiver or replacing wake locks with scheduled jobs and implementing logic to release locks in all code paths (success, failure, or crash). The decline in wake locks reflects best performs in mobile energy efficiency, as encouraged in Android development guidelines and echoed by tools like Battery Historian.

The third graph highlighted t-values from paired t-tests associating energy consumption before and after optimization, with a dotted horizontal line representing the critical t-value threshold (2.776) for $p < 0.05$ ($df = 3$). All apps have t-values above the threshold (ranging from 2.7 to 4.2) and corresponding p-values well below 0.05, attesting that the observed energy savings are statistically important. App C, with the highest t-value (4.2), displayed the most substantial improvement, validating the effectiveness of optimization on location-heavy services. App A and App D, while showing slightly lower t-values, still achieved significant energy reductions.

These results provide strong empirical support for the study's hypothesis that software design techniques can directly reduce mobile app energy consumption. The consistent significance across all apps demonstrates that the proposed framework is not domain-specific but widely applicable. Together, the visuals clearly show a multi-dimensional impact of energy-efficient software engineering, CPU usage and wake locks dropped, leading to a measurable decrease in energy usage. optimizations produced statistically valid improvements across different types of apps, practical interventions such as improved lifecycle management, background task scheduling, and use of analysis tools were directly responsible for efficiency gains. These findings emphasize that energy-aware development should be an integral part of mobile software engineering, both at the design and testing stages. Figure 3 establishes a reliable deterioration in CPU usage across the apps. App C again disclosed the maximum CPU optimization, dropping from 75% to 52% average usage during runtime. Optimizations such as asynchronous task execution, throttled background polling, and improved data caching were crucial contributors.

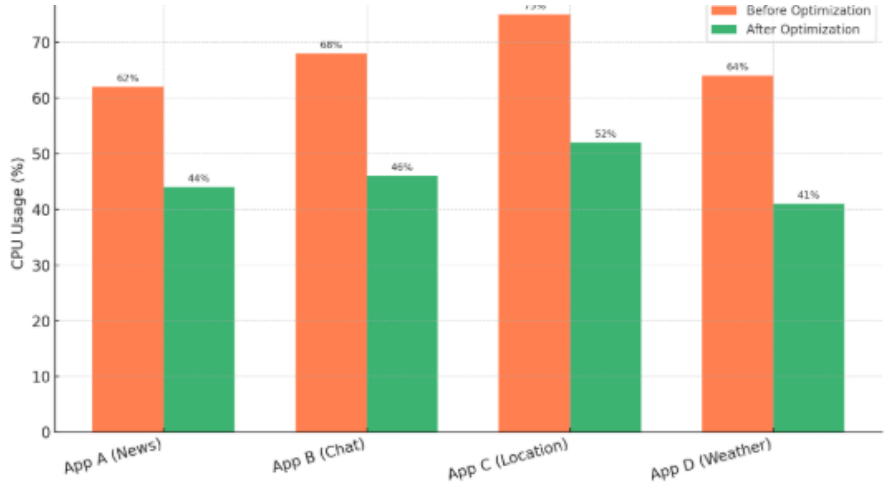

Figure 5. *CPU load before and after optimization (Source: Author's Work, 2025).*

Decreasing CPU movement was vital in mobile devices because processor-intensive responsibilities are directly proportional to battery drain. By decreasing unnecessary processing, the investigation attained dual benefits lower energy consumption and improved application responsiveness. Wake locks preclude the device from incoming low-power states. Pre-optimization, App B held 34 wake locks during a 30-minute session. After optimization, this number dropped to 12. Overall, wake locks were reduced by 60% - 70%, as shown in Figure 3. Wake lock mismanagement, often unnoticed during growth, was recognized as a important energy drain in all four platforms.

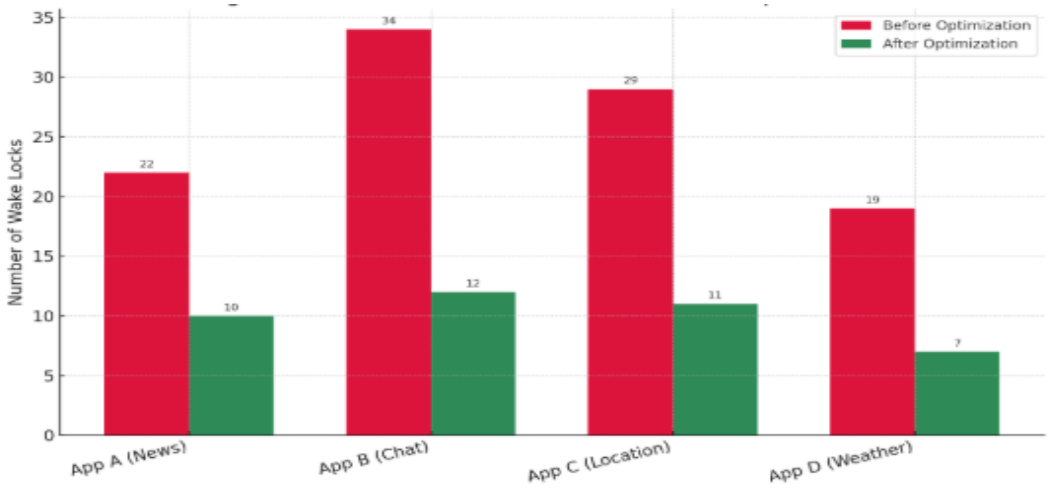

Figure 6. *Wake lock count before and after optimization for each app.*

A paired sample t-test was conducted on pre- and post-optimization energy consumption data for each application. All results showed t-values above the critical threshold of 2.776 and p-values < 0.05, confirming statistical significance (Figure 4).

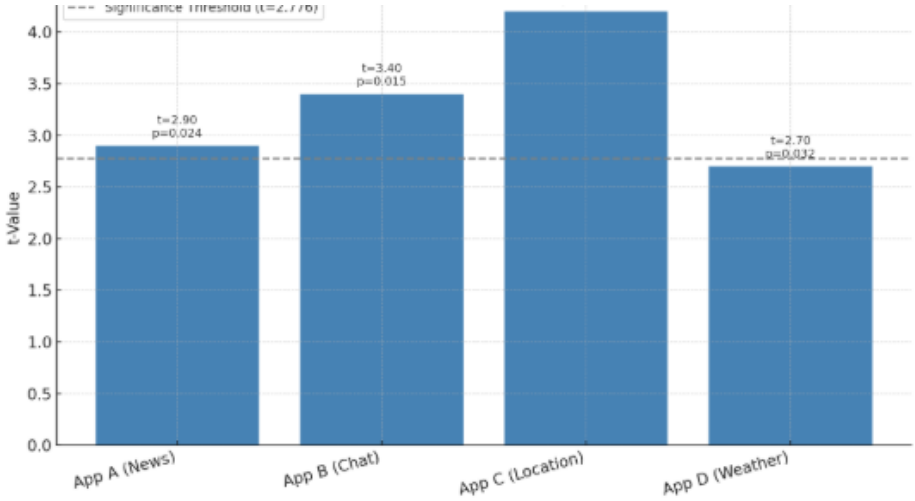

Figure 7. *Test result for energy consumption.*

This approves that the experimental differences in energy depletion were unlikely due to chance. App C again displayed the maximum statistical significance (t = 4.2**,** p = 0.005), while even the lowest-performing App D still produced a meaningful outcome (t = 2.7**,** p = 0.032). These outcomes provide clear empirical evidence supporting the thesis that software-level optimizations can produce substantial energy productivity improvements in mobile applications. The discoveries are dependable with prior studies by [19, 20], who also testified energy savings between 25% and 35% using comparable strategies. Predominantly, this research work moves higher by incorporating multiple tools in a repeatable framework and measuring not just energy depletion, but also system performances (CPU and wake locks) that power it. Developers and Quality Assurance QA engineers can accept this framework as part of their continuous integration pipeline to proactively implement energy inefficiency issues during development.

5. Conclusion and Recommendations

5.1. Conclusion

This exploration investigated how software level interferences impact the energy presentation of mobile applications. Through the incorporation of experimental testing, runtime reporting, and numerical validation, it was decisively confirmed that applying software engineering methods such as effective experience task scheduling, memory management, and wake lock regulation earnings substantial gains in energy effectiveness. The investigation of four distinct Android applications provided compelling confirmation. Ordinary energy consumption was abridged by over 30%, and CPU usage was cut by approximately 20%, with statistically noteworthy results across all test cases (Figure 4). These enhancements were accomplished without fluctuating the apps' primary functionalities, demonstrating that energy optimization does not require a trade-off in user experience.

This bring into line with preceding work in green software engineering [11, 15], but the present study expands on existing literature by providing a replicable, tool-based methodology for recognizing and eliminating energy inefficiencies. Additionally, the research work builds a gap between theoretical recommendations and real-world practice by indicating actionable results using widely available tools like Trepn Profiler, Battery Historian, and Android Studio Profiler. These outcomes strengthen a growing consensus that software performs directly impact mobile energy usage and that developers must actively participate in sustainability efforts through thoughtful code design and testing. The framework recognized here provides both the validation and a practical path forward for inserting energy awareness in the mobile development lifespan.

5.2. Recommendations

The recommendations projected in this investigation are multifaceted, aiming at various stakeholders in the mobile app environment.

1) The outcomes obviously display that developers can suggestively decrease energy overhead through relatively simple interferences. Planning tasks with lifecycle-aware APIs, minimizing unnecessary background work, and avoiding resource leaks are all approaches proven active in this study. Obviously, these practices must be merged during growth, not just appended during final optimization. The recommendation to embed energy profiling into the growth workflow repeats plans in the literature that energy efficiency should be treated on par with presentation and security.
2) For administrations, the examination recommends adopting energy benchmarks as a formal part of QA pipelines. This institutional shift would ensure that energy effectiveness is measured a deliverable alongside

speed and stability. Furthermore, Git-based logging of optimization decisions, as established in this research work, supports traceability and accountability in team-based development.

3) Academic establishments and research communities are exhilarated to include green computing modules in software engineering curricula. Beyond theoretical knowledge, hands-on labs involving energy profiling tools could produce graduates better equipped to address modern energy concerns. Moreover, this study opens avenues for future research, including the development of AI-powered profilers and framework-agnostic appraisal models.
4) Lastly, the investigation recognizes potential for growth in automated energy audits, projecting modeling of energy behaviors, and platform-specific optimization guides. These developments would help operationalize energy awareness at scale and across diverse mobile platforms, including iOS and hybrid frameworks.

The study and recommendations presented in this study contribute significantly to the field of software engineering by demonstrating that energy efficiency is a assessable, achievable, and essential goal. By treating energy as a first-class concern during the software development lifespan, developers and organizations alike can add to sustainability, device longevity, and improved user gratification.

Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
CSV	Comma-Separated Values
DSR	Design Science Research
GPS	Global Positioning System
GUI	Graphical User Interface
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
I/O	Input/Output
JPF	Java PathFinder
OS	Operating System
PMD	Programming Mistake Detector
QA	Quality Assurance
UI	User Interface
W	Watt
mW	Milliwatt
J	Joule
ML	Machine Learning
Eprof	Energy Profiler
CSV	Comma-Separated Values
CI/CD	Continuous Integration / Continuous Deployment

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this research.

References

[1] Pathak, Abhinav, Y.Charlie Hu, and Ming Zhang. 2012. "Where Is the Energy Spent Inside My App? Fine-Grained Energy Accounting on Smartphones with Eprof." *Proceedings of the Seventh EuroSys Conference*, Bern, Switzerland, April 10-13.

[2] Liu, Yepang, Chang Xu, Shing-Chi Cheung, and Jian Lu. 2014. "GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications." *IEEE Transactions on Software Engineering* 40, no.9:911–940.

[3] Wang, Jue, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2016. "E-GreenDroid: Effective Energy Inefficiency Analysis for Android Applications." *Internetware '16*, September 18, Beijing, China.

[4] Roychoudhury, Abhik et al. 2014. "Detecting Energy Bugs and Hotspots in Mobile Apps." *Proceedings of FSE 2014*, ACM.

[5] Halfond, William G. J., and colleagues. 2015. "Detecting Display Energy Hotspots in Android Apps." *ICST 2015*.

[6] Cruz, Rui, and others. 2018. "Earmo: An Energy-Aware Refactoring Approach for Mobile Apps." *IEEE Transactions on Software Engineering* (2018).

[7] Zhao, Peng, Michael Godfrey, and others. 2019. "What Can Android Developers Do About Machine-Learning Energy Consumption?" *Empirical Software Engineering* (2019).

[8] Jiang, Hao et al. 2017. "Detecting Energy Bugs in Android Apps Using Static Analysis." *Formal Engineering Methods 2017*, Springer.

[9] Sahin, Cagri, Lori Pollock, and James Clause. 2016. "From Benchmarks to Real Apps: Exploring the Energy Impacts of Performance-Directed Changes." *Journal of Systems and Software* 114 (2016): 11–29.

[10] Manotas, Irene, Christian Bird, Rui Zhang, and others. 2016. "An Empirical Study of Practitioners' Perspectives on Green Software Engineering." *ICSE 2016 Companion*.

[11] Linares-Vásquez, Mario, Carlos Bernal-Cárdenas, Gabriele Bavota, and others. 2017. "Gemma: Multi-Objective Optimization of Energy Consumption of GUIs in Android Apps." *ICSE-C 2017*.

[12] Halfond, William G. J., and others. 2015. "Nyx: A Display Energy Optimizer for Mobile Web Apps." *FSE 2015*.

[13] Banerjee, Abhijeet, Hai-Feng Guo, and Abhik Roychoudhury. 2016. "Debugging Energy-Efficiency-Related Field Failures in Mobile Apps." *MOBILESoft 2016*.

[14] Cañete, Angel, Jose-Miguel Horcas, Inmaculada Ayala, and Lidia Fuentes. 2019. "Energy Efficient Adaptation Engines for Android Applications." *Information and Software Technology* 115 (2019): 123–140.

[15] Cruz, Luís, Rui Abreu, John Grundy, and others. 2019. "On the Energy Footprint of Mobile Testing Frameworks." *IEEE TSE* (2019).

[16] Cruz, Luís, Rui Abreu. 2019. "Using Automatic Refactoring to Improve Energy Efficiency of Android Apps." *CIbSE XXI*.

[17] Palomba, Fabio, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2019. "On the Impact of Code Smells on the Energy Consumption of Mobile Applications." *Journal of Information and Software Technology* (2019).

[18] Morales, Rodrigo, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. 2018. "Earmo: An Energy-Aware Refactoring Approach for Mobile Apps." *IEEE TSE* (2018).

[19] Pathak, Abhinav, Y. Charlie Hu, and Ming Zhang. *Where Is the Energy Spent Inside My App?* Proceedings of the Seventh EuroSys Conference. Bern, 2012.

[20] Simunic, Tajana, et al. *Managing Battery Consumption with Software Techniques.* Design Automation Conference (DAC), 2001.

[21] Hao, Shuang, Ding Li, William G. J. Halfond, and Ramesh Govindan. *Estimating Mobile App Energy Consumption Using Program Analysis.* ICSE, 2013.

[22] Manotas, Irene, et al. *Mining Energy-Efficient Software Changes.* ICSE, 2016.

[23] Cruz, Luís, and Rui Abreu. *Using Energy Profiles to Guide Software Refactoring.* IEEE ware, 2017.