Battery Optimization in Android Applications Using Smart Energy-Efficient Coding Techniques

Shafqat Hussain, Saima Munawar*, Nasir Naveed
Faculty of Computer Science and Information Technology, Virtual University of Pakistan, Lahore
*Correspondence: saima.munawar@vu.edu.pk

Citation | Hussain. S, Munawar. S, Naveed. N, "Battery Optimization in Android Applications Using Smart Energy-Efficient Coding Techniques", IJIST, Vol. 8 Issue. 5 pp 2225-2266, August 2026
DOI |
Received | July 31, 2026 **Revised |** Aug 22, 2026 **Accepted |** Aug 24, 2026 **Published |** Aug 29, 2026.

The proliferation of mobile applications mirrored the advancement of the digital world. However, battery life proved to be a major interruption to using these applications for long period of time without need of charging. Users impacted by the battery life limitation, faced performance issues and diminished sustainability of their devices. Advances in hardware and battery technology certainly have improved energy storage capabilities with the passage of time. However, a major cause of smartphone battery drain remains software inefficiencies, which persist today despite advances in hardware. Thus, to mitigate this problem, this study was carried out which investigates the potential benefits of smart coding techniques, including algorithm optimization, asynchronous programming, correct use of data structures and network throttling for enhancing android application battery performance without affecting application functionality. An experimental approach was used in the research to evaluate conventional coding against smart coding techniques. Open-source applications were selected from GitHub and cloned in Android Studio for android application development and editing. Selected five application projects from different category were OpenTasks (Productivity), AmazeFileManager (Utility), AntennaPod (Media Streaming), ThunderBird K-9 Mail (Communication), Fossify Gallery (Media Viewer). Consumption was measured before and after making code modification/optimizationand Battery Discharge Rate (BDR) analysis was performed by making use of Battery Historian. However, application responsiveness was also observed qualitatively and no lack was found. It was also carried out in debug mode on a physical device with a fixed real-world simulated scenario to measure each application's power consumption. According to the research results, specific software optimizations improved energy performance by 12.28–32.99%, with an overall average reduction of 23.40%. The average BDR decreased from 15.82 %/hr before optimization to 12.1210 %/hr after optimization, representing a mean reduction of 3.7035 %/hr. The individual improvements per application were 32.99% for Amaze File Manager, 22.05% for AntennaPod, 17.40% for Fossify Gallery, 12.28% for K-9 Mail, and 32.13% for OpenTasks, which means that coding improvements can significantly substitute for hardware solutions to improve sustainability. Statistical validation using paired-samples t-tests showed statistically significant reductions for Amaze File Manager ($t(3)=3.723$, $p=0.0337$), AntennaPod ($t(3)=4.108$, $p=0.0261$), Fossify Gallery ($t(3)=3.336$, $p=0.0445$), and OpenTasks ($t(3)=13.008$, $p=0.0010$), while K-9 Mail showed a reduction that was not statistically significant at the 0.05 level ($t(3)=3.098$, $p=0.0534$). Across the 20 paired observations, the overall difference was statistically significant ($t(19) = 7.618$, $p < 0.001$; 95% CI: 2.69–4.72 %/hr), which means that coding improvements can significantly substitute for hardware solutions to improve sustainability. . This study fills the gap in existing literature as it moves away from theoretical abstractions and profiling techniques for developers and instead focuses on developer-oriented energy optimization techniques at development time. The results showed the need for energy awareness in the process through IDE tools and CI/CD energy tracking and developer training for energy-efficient programming during the development phase. This developer centric approach supports green software engineering by promoting Energy Efficiency (EE) as a fundamental design principle in the modern development of mobile applications.

Keywords: Green Software Engineering, Energy-Efficient Programming, Smart Coding Techniques, Resource-Aware Programming, Battery Optimization, Asynchronous Programming, Android Studio, Android Application Development

Introduction:

In today's digital age, mobile devices have become an indispensable tool for communication, productivity, entertainment, and more. The main reason for excessive battery usage in mobile devices is due to absence of optimized software in mobile applications despite the advancement in battery technology [1]. With a dominating position whose market share is more than 70%, Android (developed by Google) is as world market leader. Most android applications suffer from inefficient coding practices that arise from poor threading control, repeated network requests and inadequate algorithm development [2]. As hardware technology has improved battery life, optimization of software for sustainable applications has not been enhanced at the same rate or pace. In previous research Energy-Aware Software Engineering (EASE) framework is proposed to enable developers to build energy-efficient applications and buffer framing was used in this research.

Recent research has increasingly demonstrated that software-level decisions can have a substantial influence on the energy consumption of Android applications [3]. Studies have investigated a range of approaches, including the identification and refactoring of Android energy code smells, energy-aware software design, automated static analysis, runtime energy profiling, intelligent task scheduling, and adaptive resource management [4][5][6]. These studies indicate that inefficient loops, inappropriate resource handling, prolonged wake locks, unnecessary background activity, excessive processor wake-ups, inefficient memory utilization, and poorly designed execution patterns can contribute significantly to battery drain. Recent developments have also introduced automated tools and intelligent optimization techniques that assist developers in detecting energy-intensive operations and improving application behavior without compromising functionality or responsiveness [7]. However, existing research often focuses on individual optimization techniques or specific application behaviors, leaving a need for a practical and integrated approach that combines multiple smart coding techniques and evaluates their effectiveness across different real-world android applications. Addressing this gap, the present research investigates software-level battery optimization through systematic identification, implementation, and evaluation of energy-efficient coding practices, with the objective of providing developers with practical guidelines for reducing unnecessary energy consumption while maintaining acceptable application performance and user experience.

Three main coding methods were used to reduce the energy expenditure of the framework through: Data optimization (or Processing), Network throttling and algorithm efficiency. Network scheduling adaptation is the main hot spot of power consumption [8]. The study develops a new framework that integrates barriers to knowledge during its development and validation. The framework offers developers a series of steps for the execution of advanced programming techniques that enhance the battery efficiency of android applications. Using a step-by-step process, the framework enables developers to create and test software behaviours with minimal power consumption.

The research motivation was drawn from the international drive to attain sustainability through energy-efficient technology solutions for global warming [9]. As we know mobile applications are the main tool that makes it easy to perform daily tasks. The use of inefficient software in mobile applications brings about two main concerns. First, users will become dissatisfied with mobile applications. Second, large-scale energy waste will take place [10]. The future of energy-efficient software development has dual benefits. They not only shield the environment but also save money. This is because they protect hardware from overheating, require less charging and don't create waste since the hardware will last longer. The main goal is to help developers find sustainable computing solutions through software-based solutions that relate theoretical energy coding approaches to actual system power use [11].

This research is mainly related to Sustainable Development Goal 7 (Affordable and Clean Energy) because it focuses on reducing the energy used by mobile applications. The study looks at how better coding and software development practices can reduce unnecessary power consumption and make mobile applications more energy efficient. The research is also connected with SDG 12 (Responsible Consumption and Production) because using less energy can improve the efficient use of computing resources and increase the life of mobile hardware. This may also help in reducing electronic waste. The study has a connection with SDG 13 (Climate Action), as lower energy consumption can help reduce the environmental impact related to energy production and use. The research also supports SDG 9 (Industry, Innovation and Infrastructure) by encouraging the use of improved and sustainable software development techniques.

Research Question:

In order to handle the problem identified above, the research explored the following research questions:

RQ 1: The impact of smart coding techniques on actual battery consumption while running existing mobile android applications?

RQ 2: In what way can the use of smart coding techniques (e.g. algorithm optimization, asynchronous execution, sensor/network management) reduce the battery consumption of an android application without a decline in performance?

Objectives of the Study:

Main Goals of This research are: -

To identify smart coding techniques that influence energy efficiency in mobile android applications.

To classify energy-efficient coding techniques according to their impact on power consumption and application performance.

To assess the energy consumption of real-world android applications using appropriate profiling and monitoring tools, including Battery Historian.

To compare the energy consumption of applications before and after applying selected smart coding techniques.

To develop practical guidelines and recommendations for Android developers to incorporate energy-efficient coding practices into application development.

Scope and Limitations:

Scope:

Android applications have more than 70% market share globally and have a very large database of open-source applications on GitHub. The prototypes of five applications are evaluated for this study. The first without changing (conventional methods implemented) application tested and, metrics while the second tests and metrics after employing innovative code techniques mentioned in the introduction. Their energy performances will be analyzed and results concluded how these techniques are effective.

Limitations:

The area of research is very huge and also, it's their scope, therefore hardware and kernel-level optimizations haven't been used in this case.

The outcome of the testing of apps may differ from one device to another due to even slight differences in specifications, even on same device because of complex variables and sensors involved.

The experiments conducted in this study were carried out on one Android device and an operating System (OS) version.

Literature Review:

This Literature review section discusses the existing work done in software-based energy optimization mainly in terms of coding techniques, algorithm design, profiling tools

and implementation at a developer's level. Total 32 papers were reviewed and focus was given to the latest published paper as the technology of android application is evolving in fast pace and graph of studies reviewed with year of publication is shown in Figure 1. This review of past research work that sustained the measurement system as its subject, follows a chronological progression of time: from early measurement studies to modern intelligent optimization frameworks. This showcases the progress, achievement and persisting gaps in the field and latest opportunities to enhance the practical work to make Applications less energy-intensiveand to mitigate requirements of energy at world level.

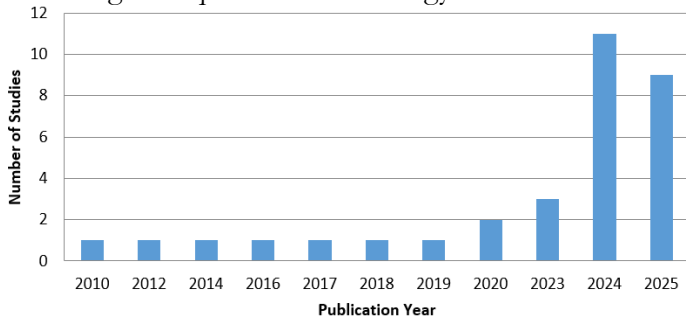

Figure 1. Graph of Number of Reviewed Studies by Year

The period of 2010-2012 Invention of Tools like Eprof:

In the early periods the work was among the earliest to assess Smartphone energy consumption with an empirical measurement. The CPU, display and wireless interfaces are dominant sources according to their analysis. Although the study provided basic data, it was fundamentally hardware-based. There was no software level tool to quantify power consumption. A power tool E Prof invented for energy consumption but editing code facility was unavailable [12].

Evolution toward Energy-Aware Software Design (2014-2016):

Invention of techniques for enhancement of energy efficiency in embedded systems was most important step. The study included strategies like Dynamic Voltage and frequency scaling (DVFS), cache optimization and hardware and software co-design. Focus of the research was at processor level. Insightful as it may be, the emphasis was largely on system-level and hardware solutions. Attention was specifically paid to limited coverage of coding practices. The source-level optimization trend started with proposed Java-level methods to lower energy consumption in mobile apps [13]. Through adaptive execution at runtime and optimized control flows their framework saved up to 50% energy. Despite its contribution, the research was limited in scope and not integrated with modern IDEs or CI pipelines. Thus, it resulted in remaining restricted real-world usability.

Profiling and Refactoring Tools for Energy Optimization:

Between 2017 and 2020, most of the progress was a contribution toward the creation of tools and frameworks that tried to bridge the gap between profiling and practice. E-Android was introduced, which can be used to find hidden energy hotspots in apps [14]. Although its effectiveness was proven for the diagnosis, it did not recommend any remedy. Next the proposed automated refactoring tool called Lea factor, which corrected common energy inefficiencies in open-source Android Apps [15]. While it was a great step towards developer-friendly solutions, it was still limited to a narrow set of energy anti-patterns and did not have real-time context awareness. In contrast, energy-aware design patterns, appeal to developers to incorporate sustainability during the early stages of software design [16]. Despite being a strong concept, this approach has not been validated on a large scale and has some its limitations.

Sustainable software engineering and introduced Progressive Web Apps (PWA) energy efficiency were introduced [17][18]. The attention gradually moved towards developer

behaviour, awareness, and tool integration. The energy optimization techniques in mobile applications were systematically mapped [19]. Suggestions include asynchronous programming, sensor throttling and runtime profiling. According to the researchers, developer-centric approaches can generate quantifiable energy benefits; however, they are not widely used due to their poor integration. Introduction of a Green Software Quality Framework (GSQF) and embedding of sustainability indicators directly into the software lifecycle was a major step in optimization [20]. Sustainability was promoted as a measurable quality attribute, but empirical validation was nil across many industrial projects. In 2024, researcher conducted a systematic literature review on environmentally sustainable software design [21]. Alignment between software architecture and energy efficiency was alarming. At the same time, exploration of ICT sustainability from a socio-technical perspective was digged-up [22]. As a result, they highlighted disconnect between top-down policies and bottom-up developer actions. Therefore, they called for practical frameworks at the developer level.

Adoption and Industry Practices in Real Life:

Studies have shown that many energy-efficient techniques exist, but few are used in industry. A survey of software companies reveals that due to the perceived complexity of energy optimization and a lack of demand from clients, as well as no economic incentives, most developers tend to devalue such optimization. Expanding and building on this, a systematic review of applied methods was performed, noting that methods like method profiling, refactoring and algorithmic optimization exist, however, their practical incorporation hardly occurs in mainstream development [16]. The researchers stressed the immediate necessity of training developers, IDE integration and standardized energy metrics.

Recent Advances in Smart Coding Techniques for Battery Optimization (2023–2025):

Recent research has increasingly established that software design and coding practices have a direct effect on the energy consumption of Android applications. Mobile energy-efficiency techniques were classified into many areas. These areas include algorithm optimization, asynchronous execution, network management, sensor control, memory optimization, and energy profiling [4]. Their findings demonstrate that energy efficiency should be considered throughout the software development process rather than only after application deployment. Android-specific code smells have received particular attention. In his research Gupta showed that inefficient practices, including slow loops and improper resource handling, can increase battery consumption [3]. Similarly, Android source-code smells were pointed out and it identified energy-related issues such as prolonged wake locks and inefficient background operations [7]. In further research identified and corrected android energy smells can produce measurable reductions in power consumption [23]. These studies support the use of code-level optimization as a practical approach to battery conservation.

Automated analysis and refactoring have also emerged as important approaches. A tool ecoCode was introduced, which detects Android energy smells through static analysis, while examination of relationship between design patterns, code smells, refactoring, and energy consumption also carried out [24][25]. EDATA tool proposed to support detailed energy debugging and identification of energy-intensive methods [26]. These studies provide a foundation for using profiling and quantitative measurements to evaluate software optimizations.

Chen applied Q-learning to Android display energy optimization, while Park investigated task wake-up sequences to reduce unnecessary processor activity [27][28]. Intelligent mobile operating-system scheduling carried out to balance power consumption and user-interface performance it extended energy-aware refactoring to sustainable software systems [29][26][30].

Overall, the literature shows a shift from conventional performance optimization toward energy-aware software engineering, combining code-smell detection, refactoring,

profiling, resource management, and intelligent scheduling. However, most studies evaluate individual techniques or specific energy problems rather than combining multiple coding optimizations across different real-world Android applications. This research addresses this gap by evaluating an integrated set of smart coding techniques including wake-lock optimization, elimination of unnecessary polling, efficient background execution, lazy loading, network optimization, and memory management across selected open-source android applications. The approach combines code-level modification with Battery Historian-based measurement to determine the practical impact of these techniques on battery consumption. The comparative analysis of studies' gaps and key findings are shown in Table No 1.

Table 1. Comparative Analysis of Reviewed Studies

Year	Study	Focus	Key Findings	Gaps / Challenges
2010	[1]	Smartphone power consumption	Demonstrated that different hardware/software activities contribute differently to smartphone energy consumption.	Limited to early smartphone platforms; did not address modern coding optimization techniques.
2012	[12]	Fine-grained smartphone power modeling	Showed that system-call-level behavior can be used to estimate and analyze power consumption.	Focused mainly on energy modeling rather than practical source-code optimization.
2014	[2]	Energy-efficient embedded computing	Techniques for improving energy efficiency through hardware and software approaches.	Broad embedded-systems focus; limited attention to Android application development.
2016	[13]	Source-level mobile energy optimization	Source-code modifications can reduce energy consumption in mobile applications.	Limited number of optimization techniques and older mobile development environments.
2017	[14]	Energy-aware Android analysis	Analysis techniques for identifying energy-consuming behavior in Android applications.	Mainly focused on detection/profiling rather than comprehensive code-level optimization.
2018	[15]	Automated Android refactoring	Demonstrated that automated refactoring can address energy-inefficient coding patterns.	Refactoring focused on specific patterns; broader optimization combinations were not investigated.
2019	[16]	Software energy efficiency	Systematically reviewed software energy-efficiency research and identified major optimization approaches.	Highlighted fragmentation of research and lack of standardized energy evaluation methods.
2020	[17]	Sustainable software engineering patterns	Patterns for incorporating sustainability into software engineering practices.	More conceptual and architectural; limited empirical Android-level validation.
2020	[18]	Energy efficiency in PWAs	Users' perceptions and energy-efficiency aspects of progressive web applications.	Focused on PWAs rather than native Android applications.
2023	[19]	Mobile application energy efficiency	Mapping of existing research and identified major techniques and research trends in mobile energy efficiency.	Need for stronger integration of energy optimization into everyday development practices.
2023	[4]	Energy-efficiency techniques for mobile apps	Taxonomy of techniques and systematically classified existing approaches.	Taxonomy identifies techniques but does not provide a unified experimental optimization framework.

2023	[25]	Automated detection and removal of energy smells in Android projects	ecoCode, a SonarQube-based approach for identifying energy-related code smells and supporting developers in reducing energy inefficiencies.	Focuses primarily on automated detection/static analysis; limited direct measurement of battery improvement across multiple real-world Android applications.
2024	[8]	Software-industry energy awareness	Examination of developers' attitudes toward software energy efficiency and identified barriers to adoption.	Developer awareness and organizational adoption remain significant challenges.
2024	[22]	ICT and sustainability	Examined relationships between ICT practices and sustainability approaches.	Broad sustainability perspective; limited application-level energy measurements.
2024	[20]	Green software quality and sustainability throughout the SDLC	Proposed a green software quality framework that embeds sustainability considerations into different stages of software development.	Provides a broad SDLC perspective but gives limited empirical evidence on Android battery consumption and specific coding-level optimizations.
2024	[21]	Environmentally sustainable software design and development	Systematically reviewed approaches for incorporating environmental sustainability into software design and development and highlighted the importance of energy-aware development practices.	Broad sustainability focus; limited experimental validation of individual coding techniques in real-world Android applications.
2024	[3]	Android code smells and battery consumption	Relationships between Android code smells and battery consumption and investigated solutions for reducing energy use.	Primarily centered on code smells; other optimization dimensions require further investigation.
2024	[5]	Mobile energy-consumption estimation	Systematically reviewed methods for estimating mobile software energy consumption.	Different measurement approaches make direct comparison between studies difficult.
2024	[6]	Mobile energy, runtime and memory	Conducted a large-scale empirical analysis linking mobile performance characteristics with energy and memory behavior.	Large-scale measurements provide evidence but do not establish a unified set of coding recommendations.
2024	[23]	Android energy-smell assessment	Proposed a developer-oriented framework for assessing power consumption associated with Android energy smells.	Mainly assessment-oriented; requires further validation through broader code-level interventions.
2024	[30]	Energy-aware code-smell refactoring	Investigated refactoring techniques for improving software sustainability and reducing energy-related inefficiencies.	Primarily focused on cloud software rather than mobile Android applications.
2024	[9]	Sustainable software engineering	How different teams contribute to sustainability in large IT organizations.	Organizational perspective; limited empirical analysis of application-level battery consumption.

2024	[10]	Energy concerns in software engineering	Review energy-related concerns across software engineering and emphasized the need to integrate energy considerations into development.	Broad software-engineering scope; limited focus on concrete Android coding techniques.
2025	[16]	Software energy-efficiency methods	Practical methods used to improve software energy efficiency and examined their application in practice.	Indicates continued lack of standardized approaches and widespread industry adoption.
2025	[7]	Android source-code smells	Systematically reviewed Android-specific source-code smells and their implications.	Identifies smells but further empirical evaluation of their battery impact is needed.
2025	[24]	Design patterns, code smells and refactoring	The effects of software design and refactoring practices on energy consumption.	Evidence remains heterogeneous; relationships between individual coding changes and battery savings require further validation.
2025	[27]	Android display energy optimization	Applying Q-learning to optimize display-related energy consumption in Android systems.	Focuses mainly on display/system-level optimization rather than general application source-code practices.
2025	[28]	Mobile task wakeups	Investigation toward wakeup sequences and their effect on energy consumption during mobile web browsing.	Specific to wakeup behavior and browsing; broader Android application optimization remains unexplored.
2025	[29]	Mobile OS scheduling	Intelligent scheduling to balance UI smoothness and power consumption.	Primarily OS/scheduling focused rather than developer-level coding optimization.
2025	[26]	Android energy debugging/testing	Introduced energy debugging and testing approaches for identifying energy-related problems in Android software.	Provides testing support but does not combine multiple coding optimizations in a controlled comparative experiment.
2025	[9]	Software design and energy performance	Systematically examination how software designs decisions influence energy performance.	Broad design-level perspective and limited direct experimentation across multiple Android applications.
2025	[11]	Green IoT software tailoring	Automatic software tailoring to improve energy efficiency under different hardware conditions.	Focused on IoT environments rather than Android applications and battery-powered mobile apps.

Gaps Identified in the Literature Review:

No doubt at early stages considerably works done related to this field but several limitations were noted:

Fragmented Approaches: In each study only one aspect is considered like in some studies only profiling of application discussed but, in some others, only refactoring or algorithm design is considered.

Limited IDE Integration: Integrated tools for profiling or optimization in IDEs are missing.

Lack of Context Awareness: No context aware coding in use like which can adjust its behavior by adopting user activity and device state.

Low Developer Adoption: Energy efficient practices are hindered and have less integration in academic curricula and professional training.

Statement of Problem:

Research indicates that coding logic implemented in different types of applications causes unnecessary consumption of power. These categories of inefficienciesare suboptimal algorithm implementation, wrong chosen data structures and power hungrypower-hungry background execution. Therefore, optimized coding techniques and application profiling data need to be connected with practical implementation of coding techniques.

Novelty of the Study:

The novelty of this study lies in its practical and integrated approach to reducing battery consumption through software-level coding optimizations in Android applications. Unlike studies that primarily examine individual energy smells, profiling techniques, or isolated optimization methods, this research combines multiple smart coding techniques within a single experimental framework. These techniques include wake-lock optimization, elimination of unnecessary polling, efficient background task execution, lazy loading, network optimization, and memory management.

A further contribution of this study is the evaluation of these techniques across multiple real-world, open-source Android applications rather than focusing on a single application or a single energy-consuming operation. The study compares application behavior before and after code-level modifications using measurable indicators such as battery discharge rate while monitoring CPU utilization, memory usage, and network activity. This provides a practical basis for determining which coding changes have the greatest effect on energy consumption. The study also establishes a direct relationship between specific code changes and observed energy savings. Instead of reporting only overall battery improvement, the research maps individual optimization techniques to their corresponding changes in resource consumption. This provides developers with practical evidence about which coding practices can be prioritized when optimizing android applications.

Finally, the study proposes a developer-oriented perspective in which battery efficiency is treated as an integral software quality consideration rather than a post-development concern. By combining source-code optimization with controlled profiling and comparative evaluation, the research provides a practical framework that can be replicated on other Android applications. The resulting guidelines can assist Android developers in identifying energy-intensive coding practices, implementing appropriate optimizations, and quantitatively evaluating their impact on battery usage.

Research Methodology:

As the usage of android application increased its battery consumption came under study because excessive power consumed by these applications worldwide became a greater concern regarding energy consumption. When study was carried out it revealed that during development phase power consumption was not considered at any Software Development Life Cycle (SDLC). There are limitations of hardware due to the compact size of hand-held android devices. So it is more feasible to make the application which runs on android devices

less energy-consuming The prevalence of energy-intensive applications motivated this research to investigate practical approaches for mitigating power consumption. so research to mitigate power issues was conducted. Methodology to research about the efficient coding is presented in this section. Quantitatively battery discharge rate was measured and reevaluated to validate results of code customization which are technically meaningful and for practical use.

Research design:

This research performs a methodology experiment in a quantitative manner with step by step.

Literature Research: Materials collection and studies related to the optimization of energy framework or strategy in a development level context.

Framework design: It refers to the development of a conceptual model for smart coding implementation.

Execution phase: Execution and running of five applications before and after code changes in Android Studio (IDE).

Profiling: Fetching power consumption data using Battery Historian

Analysis and Validation: Use of statistical tools to generate graphs for visualization.

Documentation: Legitimize well-structured documents to ensure safe designing.

For implementation of this study a layered framework was shown and followed during whole process of this experiment. Depiction of framework implemented can be seen in figure 2.

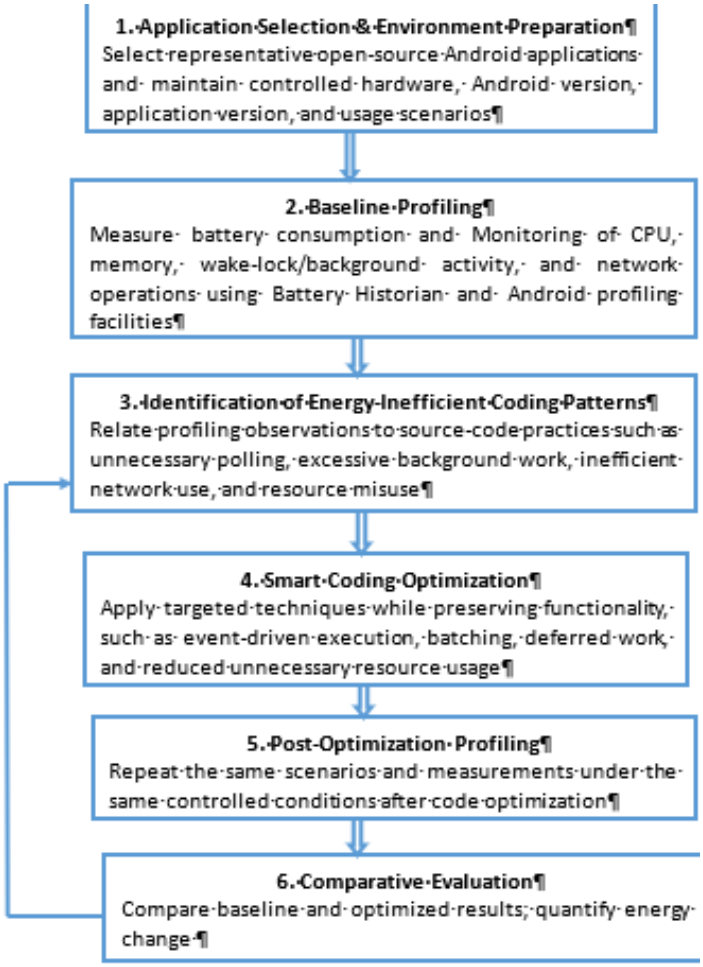

Figure 2. Implemented Framework of research for Smart Energy-Efficient Coding in Android Applications

Note: If an optimization does not provide sufficient energy improvement or affects performance, return to Layer 4; revise the intervention, and repeat profiling and evaluation.

Layer 1 Application Selection and Environment Preparation:

The first layer involves selecting representative open-source android applications which should be available with source code and preparing a controlled experimental environment. Applications selected according to their relevance to the study and the availability of their source code. The same hardware, Android version, application version, and usage scenarios maintained during baseline and post-optimization experiments to minimize external variations.

Layer 2 Baseline Profiling:

This layer establishes the original energy and resource-consumption behavior of each application before modification. The selected applications executed using predefined usage scenarios, while profiling tool Battery Historian used to collect measurements. Relevant indicators include battery consumption with proper monitoring of CPU utilization, memory usage, wake-lock activity, background activity, and network-related operations. The baseline results provide a reference against which the optimized implementation is compared.

Layer 3 Identification of Energy-Inefficient Coding Patterns:

The baseline profiling results and source-code analysis were used to identify coding practices that may contribute to unnecessary energy consumption. Examples include unnecessary polling, excessive background execution, inefficient network communication, inappropriate wake-lock usage, unnecessary memory operations, and inefficient processing. This layer connects observed energy behavior with specific source-code locations or programming practices.

Layer 4 Smart Coding Optimization:

In this layer, selected energy-inefficient code modified using appropriate smart coding techniques. The optimization focus remained on reducing unnecessary computation and resource utilization while maintaining the functional behavior of the application. For example, unnecessary polling can be replaced with event-driven mechanisms, background operations can be deferred or batched, and resource-intensive operations can be executed only when required. Each modification is documented so that the optimization process can be reproduced.

Layer 5 Post-Optimization Profiling:

After applying the selected coding techniques, the modified applications executed using the same experimental scenarios and environment used during baseline profiling. Energy and performance metrics collected again using the same measurement procedures. Maintaining identical testing conditions allows the results before and after optimization to be compared more reliably. While performance degradation monitored for any lack or latency.

Layer 6 Comparative Evaluations:

The baseline and post-optimization measurements statistically and descriptively compared to determine the effectiveness of each coding technique. Battery consumption is treated as the primary outcome. The percentage change in energy consumption calculated to quantify the improvement achieved by each optimization.

Selection of Applications:

Criteria for selection of Projects for customization: -

Written primarily in Java and Kotlin

Actively maintained or widely used

Represent different application categories (e.g., utilities, media, productivity)

Free from proprietary dependencies

Available on GitHub

Under study applications projects:

OpenTasks (Productivity)

AmazeFileManager (Utility)

AntennaPod (Media Streaming)
ThunderBird K-9 Mail (Communication)
Fossify Gallery (Media Viewer)

Selected Applications for This Research:

Amaze File Manager:
Category: Utility / File Management
GitHub:https://github.com/TeamAmaze/AmazeFileManager
Primary Language: Java
User Identifier (UID) When Application Run on a device: 10395
Why Selected:
Intensive file I/O operations
High CPU and storage interaction
Suitable for algorithm optimization, efficient data structures, and thread management

AntennaPod:
Category: Media / Podcast Player
GitHub:https://github.com/AntennaPod/AntennaPod
Primary Language: Java (core components)
User Identifier (UID) boundwith application when run-on physical android device: 10376
Why Selected:
Implementation of network throttling on network folders of email synchronization

FossifyGallery:
Category: Media / Image Viewer
GitHub: https://github.com/FossifyOrg/Gallery
Primary Language: Kotlin
User Identifier (UID) When Application Run on a device: 10397
Why Selected:
Image loading and efficient memory management
CPU and GPU interaction during image rendering and video playback
Suitable for assessing memory optimization and lazy loading techniques
Extensive use of RecyclerView for efficient media browsing

ThunderBird (K-9 Mail):
Category: Communication / Email Client
GitHub:https://github.com/thunderbird/thunderbird-android
Primary Language: Java
User Identifier (UID) When Application Run on a device: 10394
Why Selected:
Frequent background synchronization
Network Throttling

OpenTasks:
Category: Productivity / Task Management
GitHub:https://github.com/dmfs/opentasks
Primary Language: Java
User Identifier (UID) When Application Run on a device: 10371
Why Selected:
Uses background services and periodic synchronization
Involves database operations and notifications
Ideal for studying background execution, alarm scheduling, and asynchronous task optimization

Why These Apps Were Chosen (Methodological Justification):

Under study five application projects selected because these projects belong to different categories in term of real-world use cases. However, for compatibility issues only actively maintained projects chosen. Other consideration during selection of these projects was their full availability in open-source copy rights. This diversification of application considered to implement different coding techniques which are under discussion. To implement consistent environment only one physical Android device (Samsung A52) was used during the experiment. Mapping of every application with the coding technique to be implemented is shown in table 2.

How These Apps Map to Research Objectives:

Table 2. Mapping of Apps with research objectives

Apps Involved	Research Aspects and Coding Techniques
Amaze File Manager	Algorithm & File I/O Optimization
AntennaPod	Efficient Wi-Fi Lock Management
Fossify Gallery	Image Loading &Recycler View Optimization
ThunderBird (K-9 Mail)	Network Synchronization Throttling
OpenTasks	Wake-Up Alarm Optimization

Experimental Environment:

To implement consistent environment below conditions ensured:
Same physical Android device (Samsung A52)
API level: 34
Fixed OS version 14
Stable battery health
Nil other application running at time of experiment
Identical usage scenarios
Consistent room temperature it which android device runs
Controlled setup reflects a commitment to research integrity and accuracy.

Installation of Android Studio:

Android Studio version Quail 2 | 2026.1.2 downloaded and installed on HP Laptop 2570p (Window 10) with Intel Core i5 processor and 8Giga Byte (GB) Random Access memory (RAM) Overview of installed Android Studiois shown in figure 3 . Prerequisite which installed to smooth functioning of Android Studio are: -
Android Gradle Plugins (APG)
Java Development Kit (JDK) and environment variable JAVA_Home added in environment variable of system

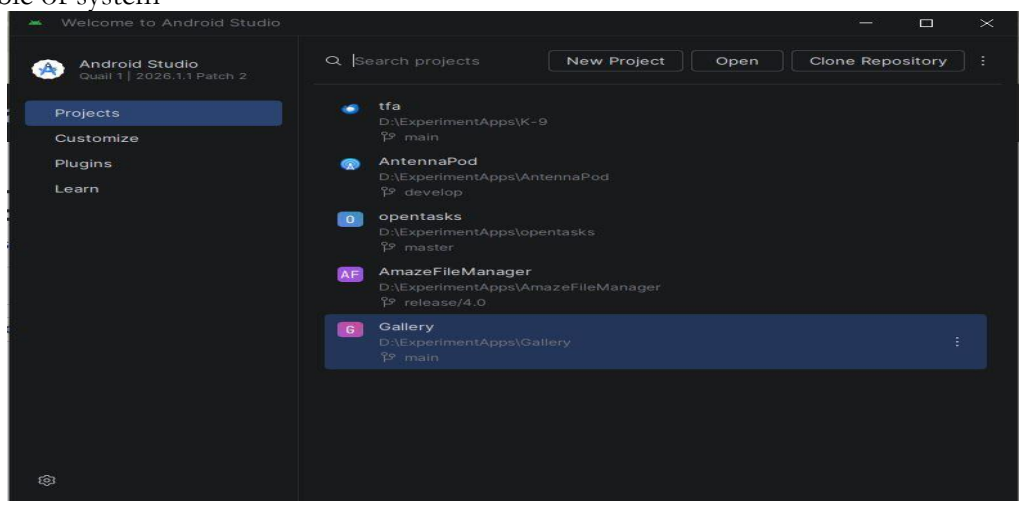

Figure 3. Android Studio Installed and Under Study Project are in Quick Open Window

Connection of Physical Device with Android Studio to Run Apps in Debug Mode:

Android Debug Bridge (ADB) installed in Android Studio to run application in debugging mode physical android device. Android device connected using Universal Serial Bus (USB) cable with Laptop.

Installation of Docker to run Battery Historian container:

Docker Desktop downloaded and installed at Laptop. After running docker server image of Battery Historian was fetched using command `docker run -p 9999:9999 itachi1706/battery-historian:latest` in command prompt and port set at 9999 (figure 4).

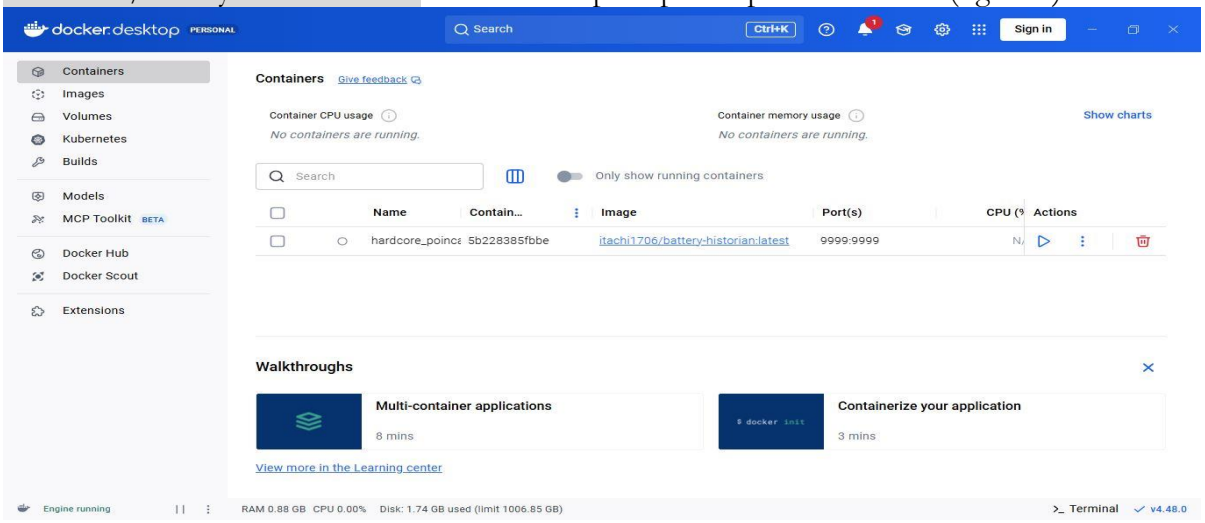

Figure 4. Docker Desktop Running with Battery Historian container

Fetching of Bugreport using Command Prompt with Android Debug Bridge (ADB**):**

ADB installed in folder C:\Program Files\platform-tools (figure 5) and under mentioned commands run in command prompt:-

cd C:\Program Files\platform-tools
adb devices
adb shell dumpsysbatterystats --reset
adbbugreport "C:\Users\SHAFQAT"

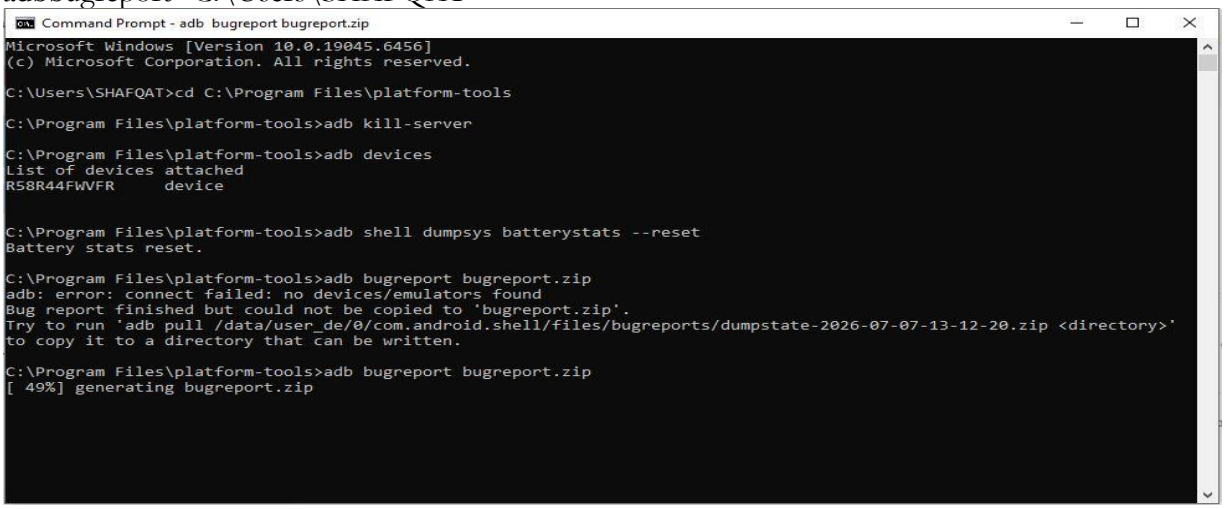

Figure 5. Command Prompt different commands to run ADB

Analysis of Bugreport Using Chrome:

When the status of Docker shows "listening on port 9999", in chrome browser battery historian was accessed using link http://localhost:9999/ and bugreport uploaded to analysis the battery graph (figure 6).

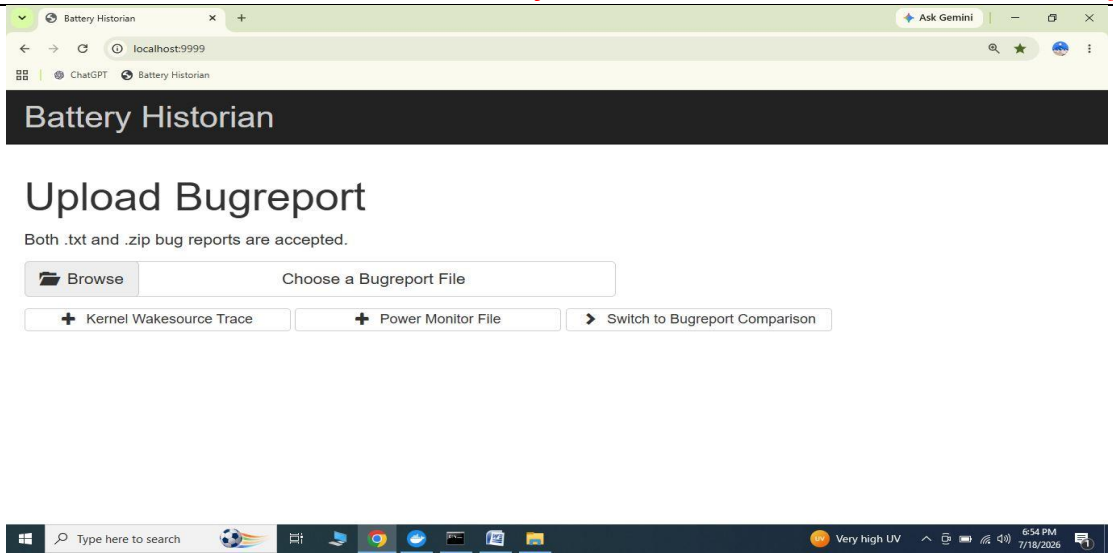

Figure 6. Fetching of Battery Historian report on localhost port 9999 in browser

Baseline Energy Profiling:

Before applying any optimizations, each application was analyzed in its original state. This baseline profiling phase established the reference energy-consumption behavior before code customization observes how ordinary coding decisions silently consume energy however these can be easily avoided.

Profiling Tools Used:

Android Studio Energy Profiler: For background activity analysis and checking for any lag in performance of apps during usage after application optimization. This was not part of experiment but used in additional to validate the performance lack during application running in debug mode.

Battery Historian: For battery discharge visualization. It was run using Docker software on Window 10 then Battery Historian image was fetched and bugreport was evaluated by accessing Battery Historian in chrome Browser at localhost Port No 9999

Metrics recorded during this phase included:

Battery Discharge Rate (BDR) which is %/hrs

Implementation of Smart Coding Techniques:

After intensive deeply research in project and looking in every line of code, smart coding techniques were applied. Each technique is discussed below separately.

Algorithm Efficiency:

Different algorithm which was power hungry and more computation replaced and it resulted in.

Eliminating redundant computations

Reducing unnecessary object creation using caching

Asynchronous and Event-Driven Programming:

Synchronization logics re-implemented by: Callbacks and listeners like Work Manager and Job Scheduler. This shifted from unnecessary CPU wake-ups to more power saving mode.

Context-Aware Execution:

Task performed in application adjusted by considering.

User activity

Network availability

Network Throttling:

It is technique used in networks using apps. Synchronization is bind with other application network request or deferred if it can be permitted by the feature implemented...

Post-Optimization Profiling and Evaluation:

After implantation of concerned coding techniques application re-run at physical android device and re-profiled under the same scenario in which previously application power consumption calculated. Comparisons between baseline and optimized versions provided quantifiable evidence of improvement which will be discussed application wise.

Application wise Discussion of Coding Change:

Amaze File Manager Application (UID 10395):

Change 1: Using Iterator.remove() instead of list.remove(i)

Inspection of project under study carried out. During inspection of Class LoadFilesListTask.java code it was observed that an element removal from an ArrayList using list.remove (i) was implemented and it caused shifting all remaining elements and updating indices which resulted in repeated size checking and it results in waste of resources if many null objects exist. So, list.remove line of code replaced with Iterator.remove () because this removes the current element safely without repeatedly recalculating indexes. This type of code change contributes to the correct use of data structures. Before and after code changes window of android studio are shown in figure 7 and 8.

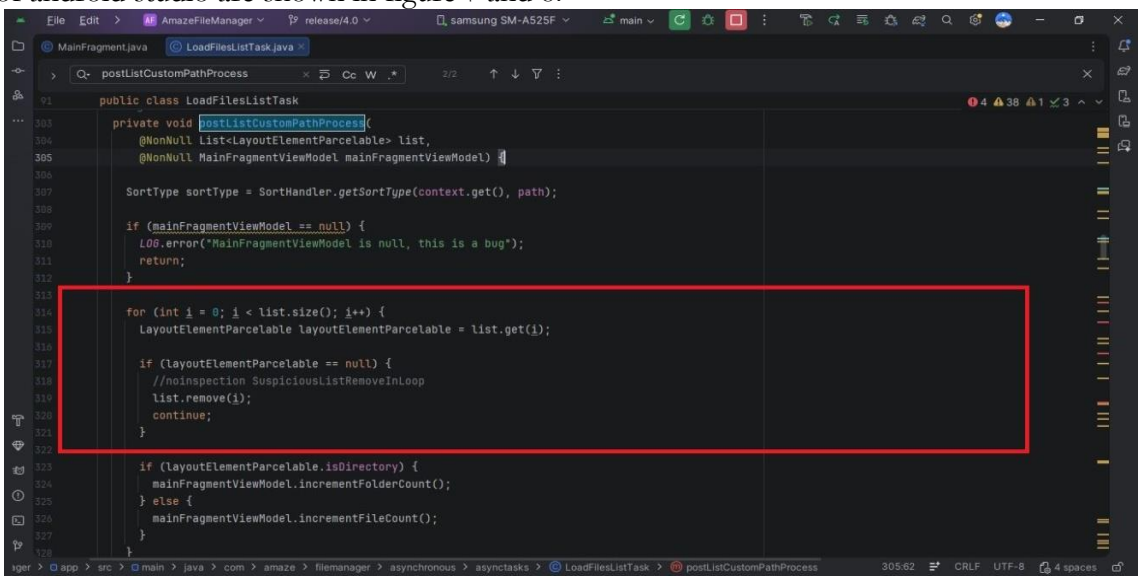

Figure 7. Code snapshot in Android Studio Before optimization

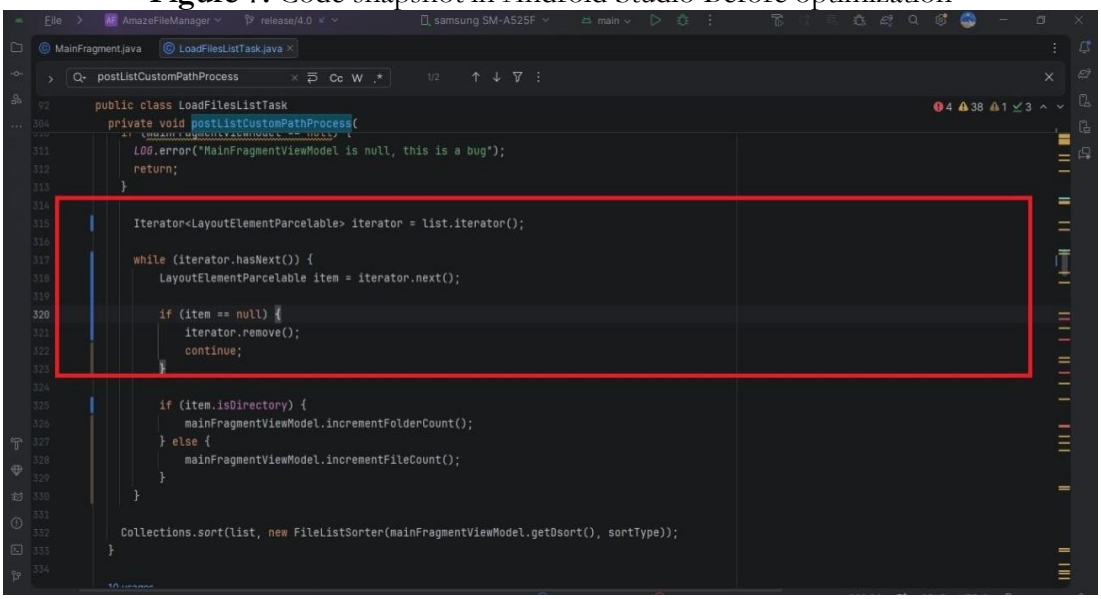

Figure 8. Code snapshot in Android Studio After optimization

Change 2: Reusing the same File Object File f = new File (path) at multiple Places:

During further observation in the Class LoadFilesListTask.java it was found that multiple objects are created and if efficiently managed only one object can be used so an object of type File is created and reused in all the others where it required. This showed the good example of optimization using data caching because in this case only one file object created in code instead of 2 instances of the file. As this application is a file manager app so if we suppose there are 5000 files so before 10000 file objects were created however now 5000 objects will be created. So that removed5,000 unnecessary allocations and these allocations matter in power consumption. Before and after code changes window of android studio are shown in figure 9 and 10.

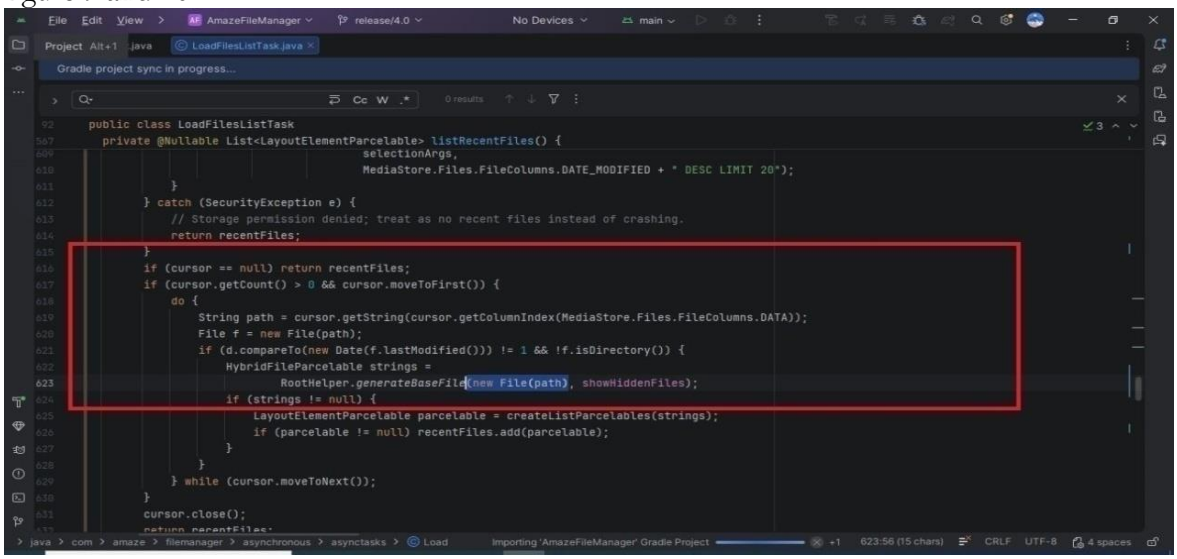

Figure 9. Code snapshot in Android Studio Before optimization

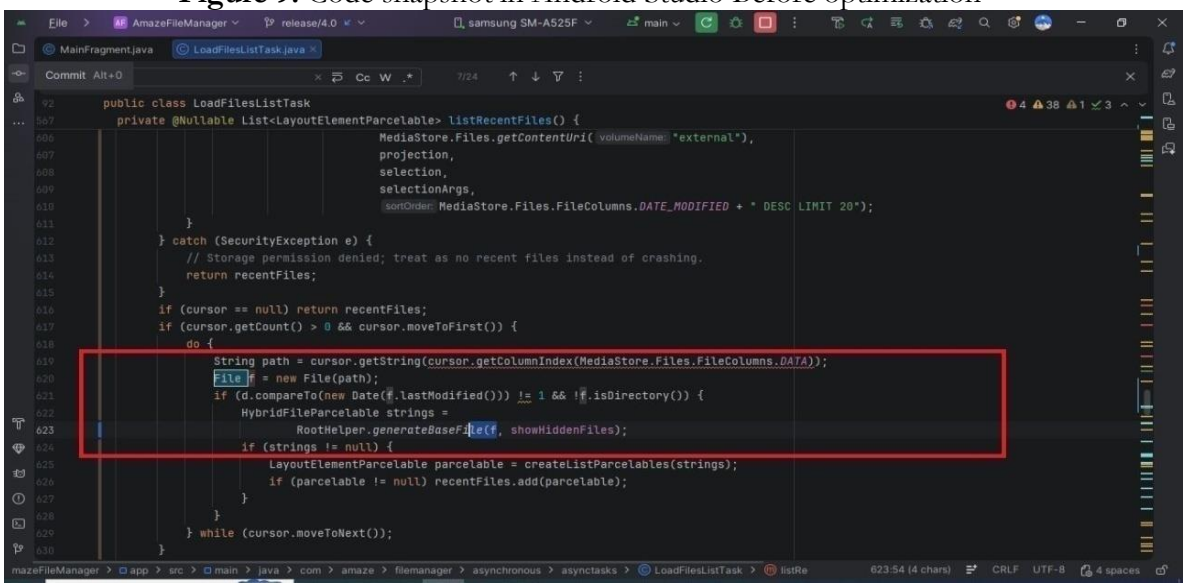

Figure 10. Code snapshot in Android Studio After optimization

Change 3: Caching Directory Metadatadir.length() and lastModified():

In the same class metadata related to directory was accessed ineffectively so that each method was recalculating metadata data. It can be cached using local variable in same class. It will prevent the repeated access of file system call which triggers a file system metadata lookup. Local variable created to save metadata instead of accessing the file system each time. It is evident that File system metadata access is much slower than reading a local variable and each repeated call follow this work flow in case of file system metadata:

Java →JNI→Linux Kernel→File system →Storage

This requires CPU activity and I/O. But caching avoids these repeated operations. Before and after code changes window of android studio are shown in figure 11 and 12.

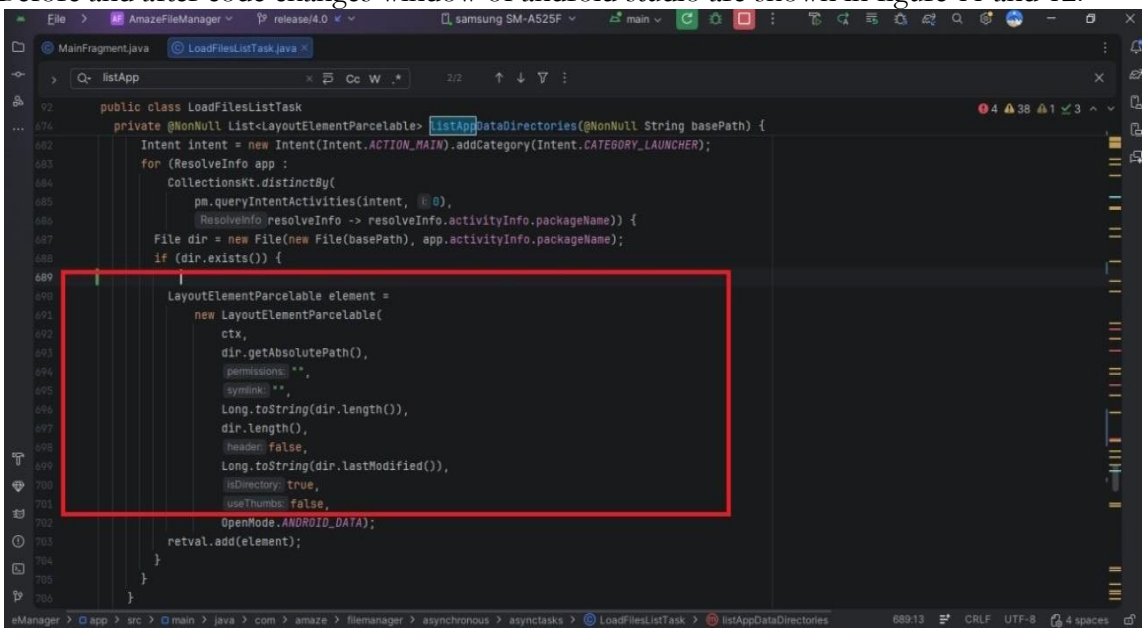

Figure 11. Code snapshot in Android Studio Before optimization

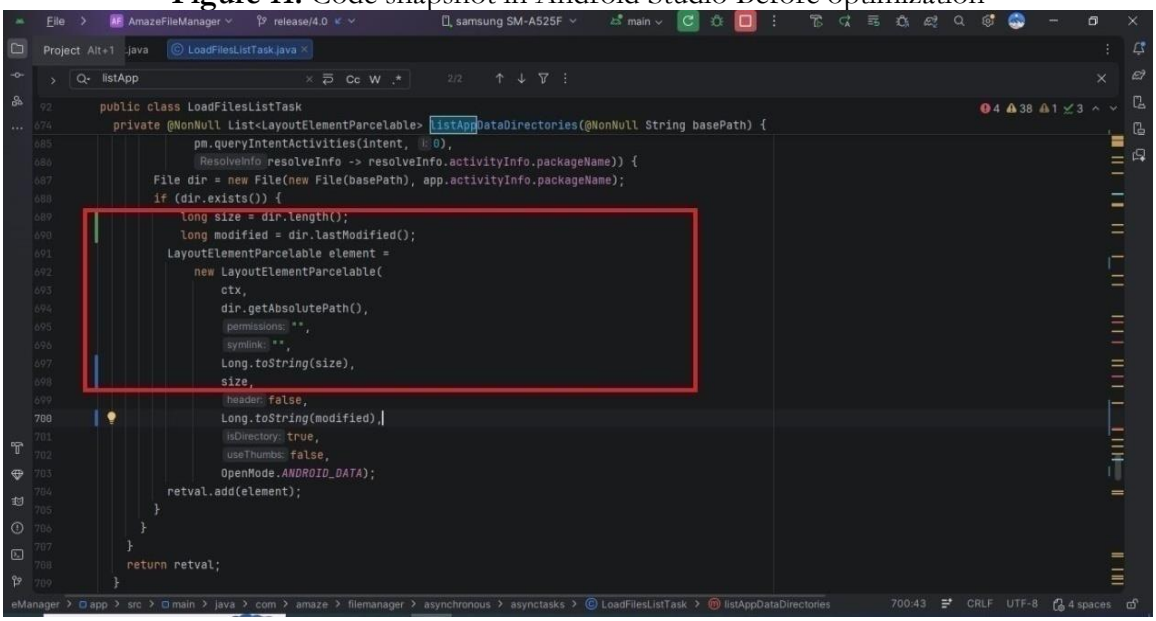

Figure 12. Code snapshot in Android Studio After optimization

Antenna Pod Application (UID 10376):

To optimize using Eliminating redundant Wi-Fi lock operations Application code review revealed that in Class PlaybackServiceMediaPlayer.java every call made in methodWifiLockIfNecessary() executes wifiLock.acquire() and acquire wifi lock even if the lock is already held. So condition was imposed in code to check if already lock is already held in the app. Wi-Fi lock is acquired only when it is not already held. It causes lowers synchronization overhead (Figure 13 and 14).

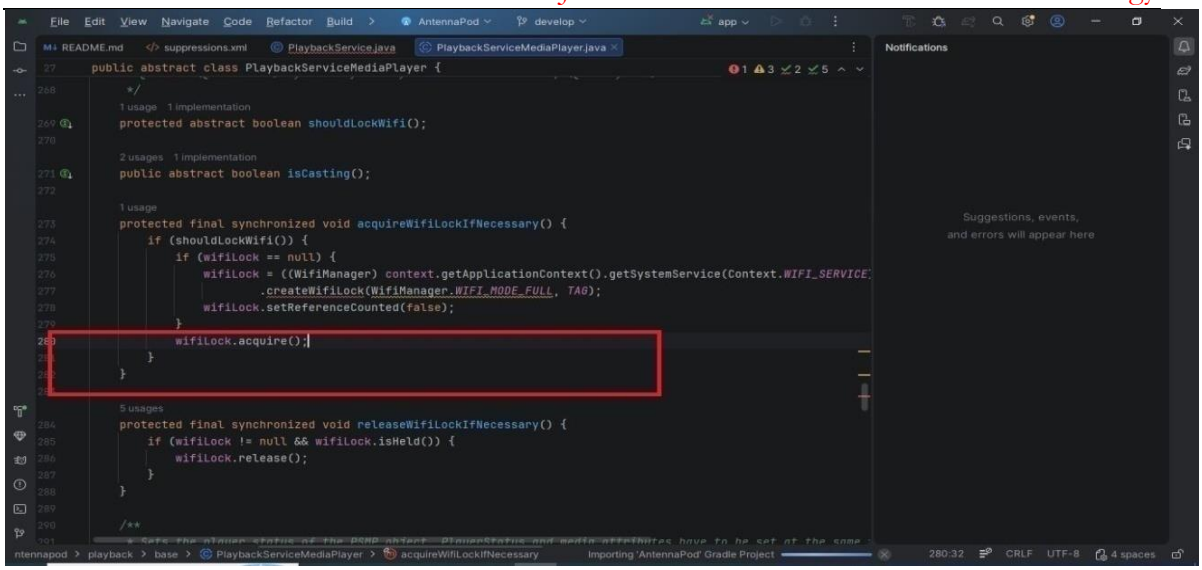

Figure 13. Code snapshot in Android Studio Before optimization

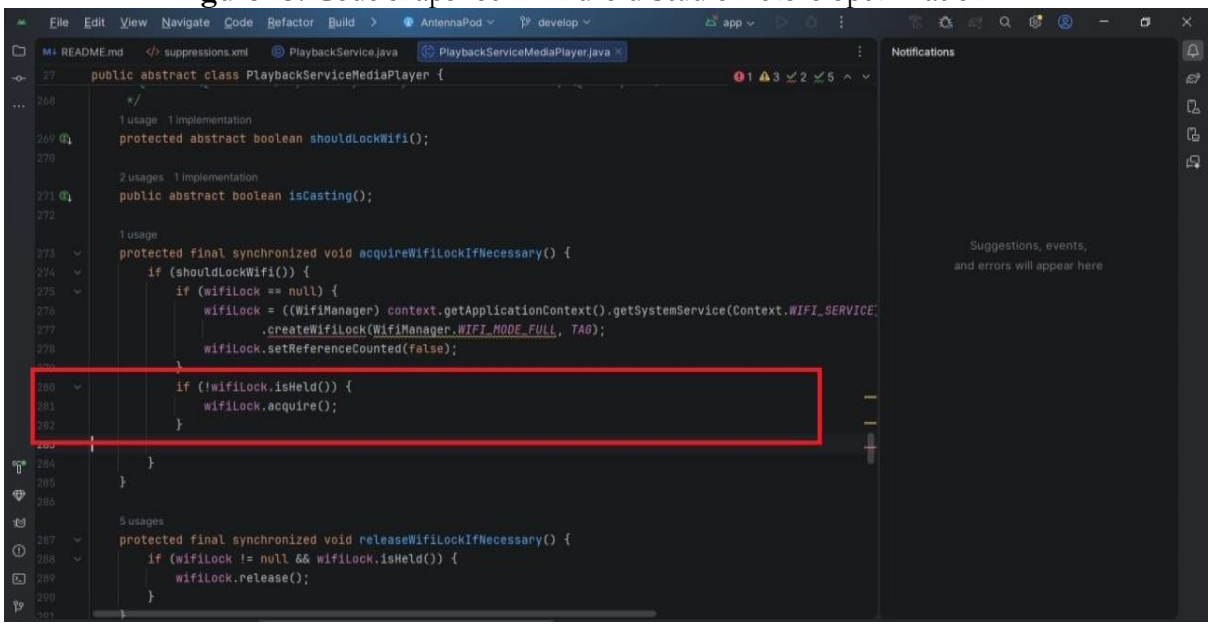

Figure 14. Code snapshot in Android Studio After optimization

Fossify Gallery (UID 10397):

Change 1 onViewRecycled() Optimization:

In android studio application clone made using GitHub link. During inspection and deeply searching for possibilities in media adapter class specially in recycler view, which used for implementation of scrolled list of interactive view, seen that implementation was more traversing as all child views (allViews.firstOrNull) which required iterating through the entire view hierarchy and Used Glide.with(activity) whose lifecycle is tied to the whole Activity. So to finds the thumbnail directly and findViewById () used. Moreover, Glide.with(thumbnail) which binds the request to the ImageView lifecycle only. It clears image requests immediately when the RecyclerView item is recycled .Before every recycled item performs under mentioned control flow during execution.

Iterate all child views → Search matching ID → Clear Glide request

This flow after new code implementation became:-

findViewById()→ Direct lookup → Clear request

This reduces unnecessary traversal of views in RecyclerView implemented in this Application (Figure 15 and 16).

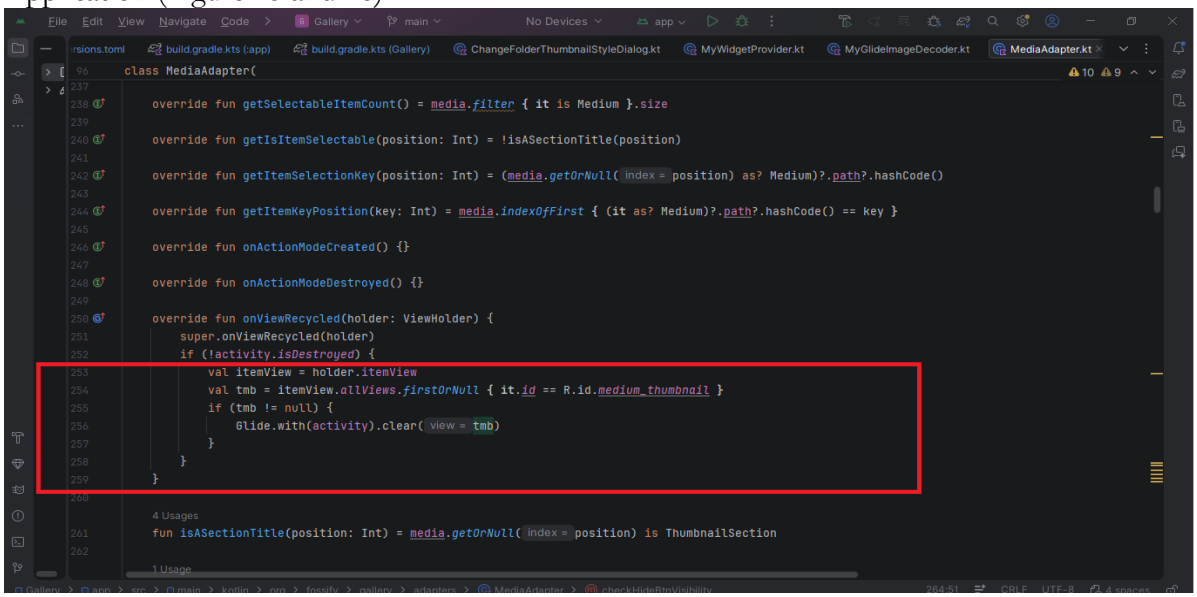

Figure 15. Code snapshot in Android Studio Before optimization

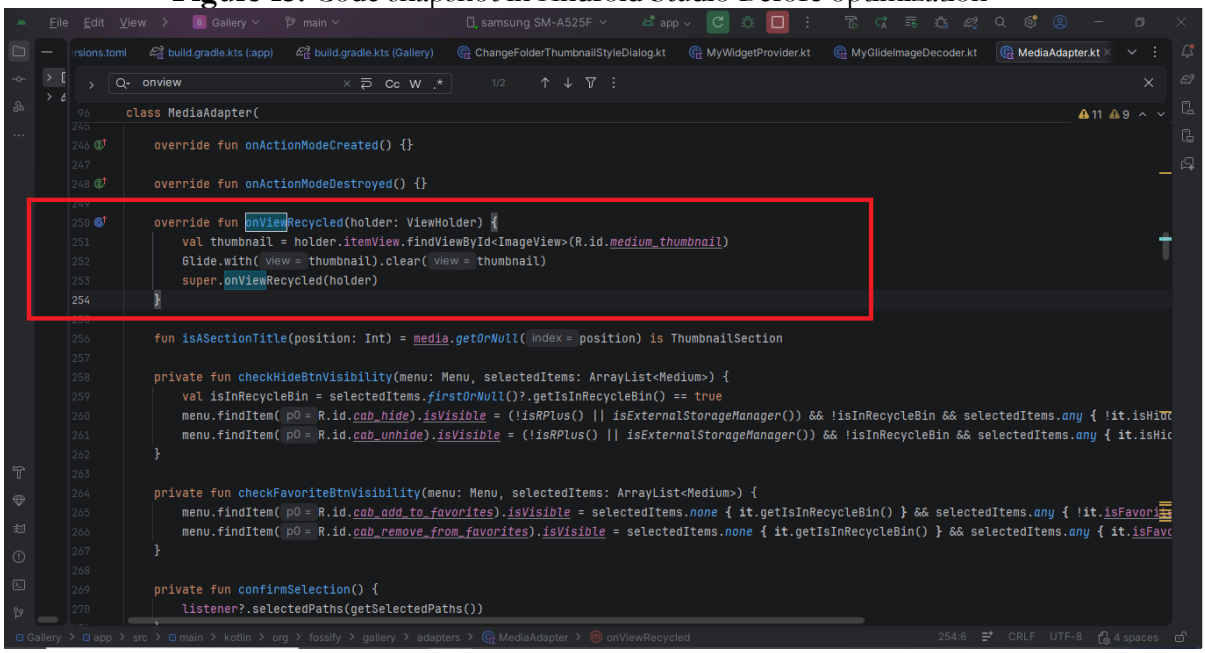

Figure 16. Code snapshot in Android Studio After optimization

Change 2 setupThumbnail() Optimization:

However in the same class multiple calls were made to medium.isVideo() and medium.isPortrait() and executed repeatedly. Then a variable is created to acquire the value. These values are used in logic. The values are computed once and this made repeated method calls to Boolean lookup once and re-used in the method everywhere which is much cheaper. Before and after code changes window of android studio are shown in figure 17 and 18.

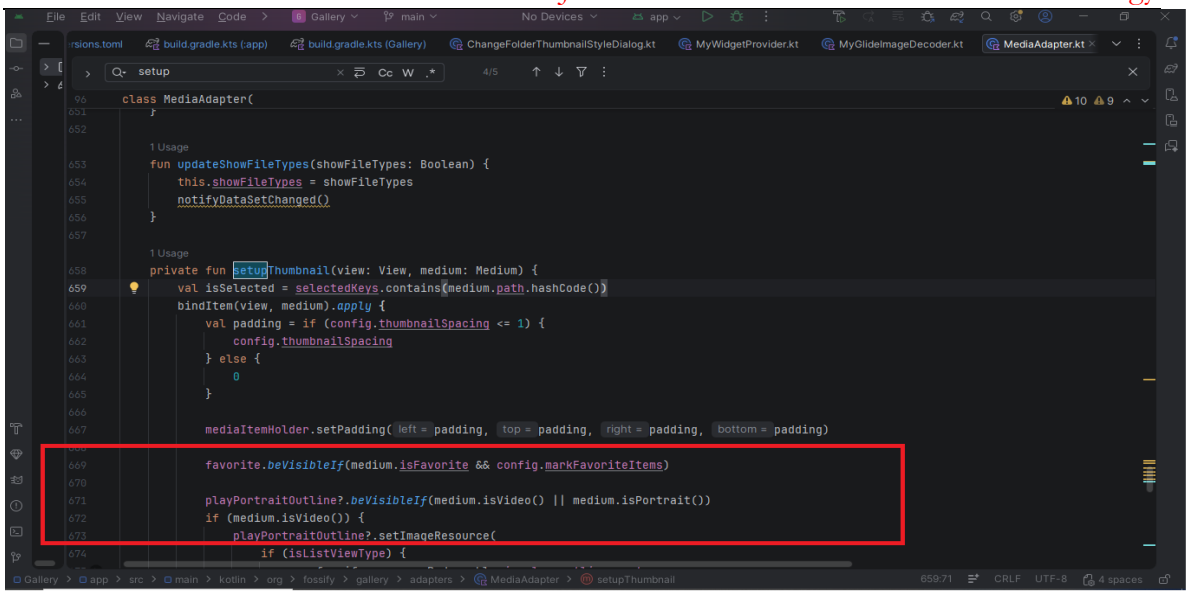

Figure 17. Code snapshot in Android Studio Before optimization

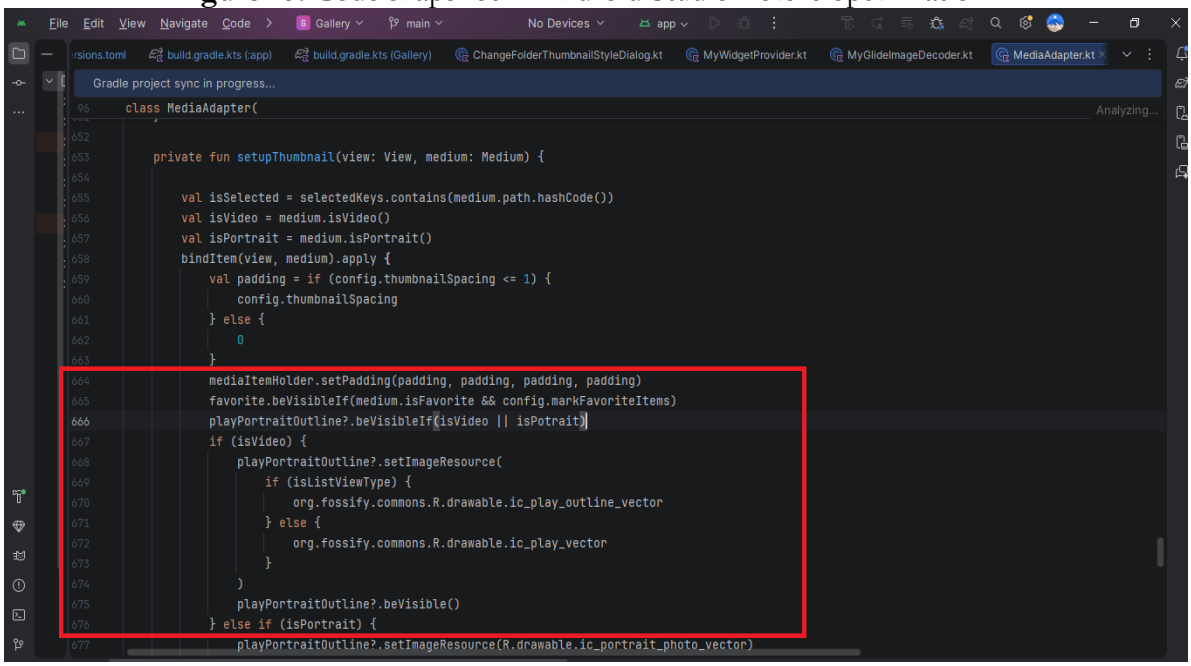

Figure 18. Code snapshot in Android Studio After optimization

Change 3 loadImageBase()OptimizationGlide.with(applicationContext):

In the same class request Glide.with(applicationContext)was tied to the Application lifecycle. Image requests could continue even after RecyclerView was recycled so it was changed toGlide.with(target) which will result in the request being tied to the ImageView lifecycle. When the thumbnail disappears,this new implemented method will automatically cancel the decoding, disk reads, bitmap transformation and release resources earlier.

Additional Optimization:

Additionally transitioncall.transition(getOptionalCrossFadeTransition (THUMBNAIL_FADE_DURATION_MS)) changed to .transition(getOptionalCrossFade Transition(0)) to remove the thumbnail fade animation to reduce GPUcomposition (Figure 19 and 20).

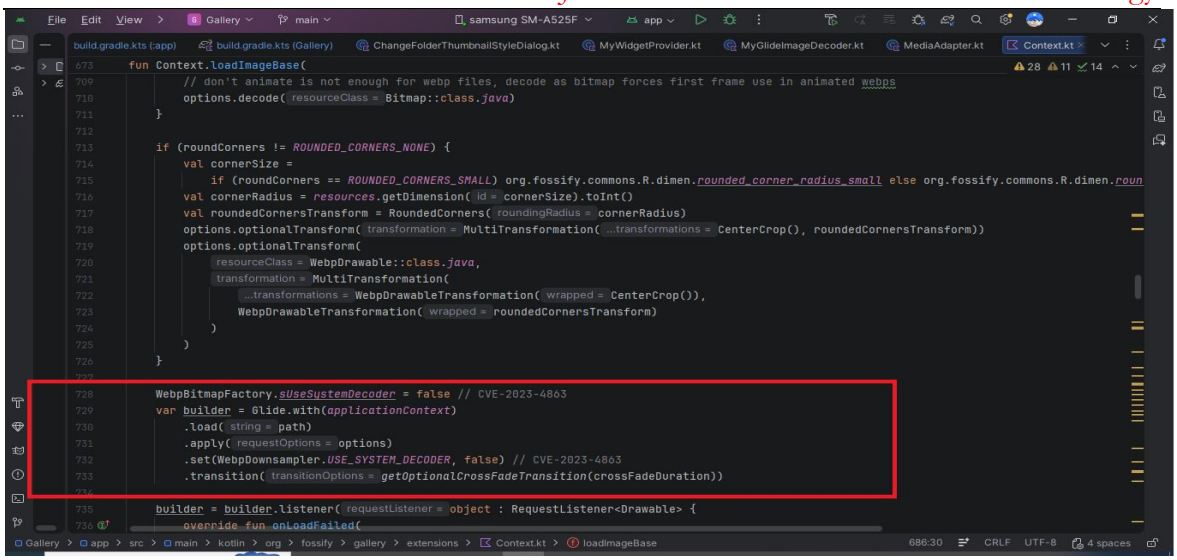

Figure 19. Code snapshot in Android Studio Before optimization

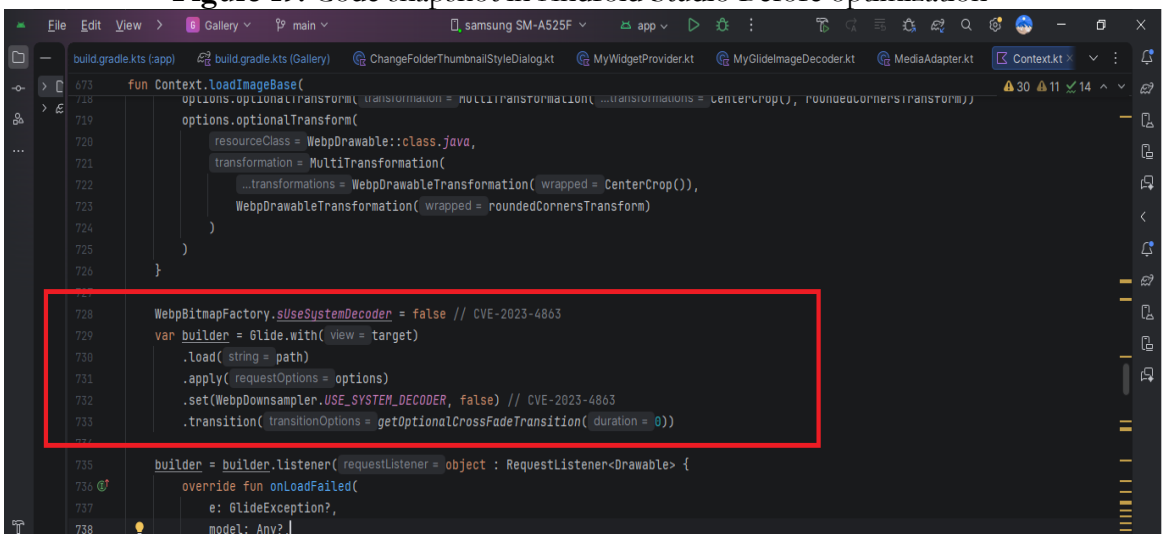

Figure 20. Code snapshot in Android Studio After optimization

ThunderBird (K-9 Mail) (UID 10394):

Network synchronization throttling was categorically reviewed as this application uses the network for synchronization of mail with server and local storage on the device. In Class ImapSync.kt synchronization request executes the complete synchronization process regardless of how recently the previous sync process occurred. These results in multiple sync requests occurring within a short period performed and requested for IMAP server connection which involve different categories like, authentication, folder opening, message count retrieval and Database updates even if synchronization had just completed. So for optimization purpose and reduce synchronization, interval check was added at the beginning of the method. 30 seconds was the enough periods for sync of emails. Now this condition prevents repeated synchronization requests within a 30-second interval. Each request of sync keeps the radio in a high-power state. And it is a known fact that network radios are among the largest contributors to battery consumption in communication applications. So after checking that within 30s there is synchronization occurred saves all android resources (Figure 21 and 22)

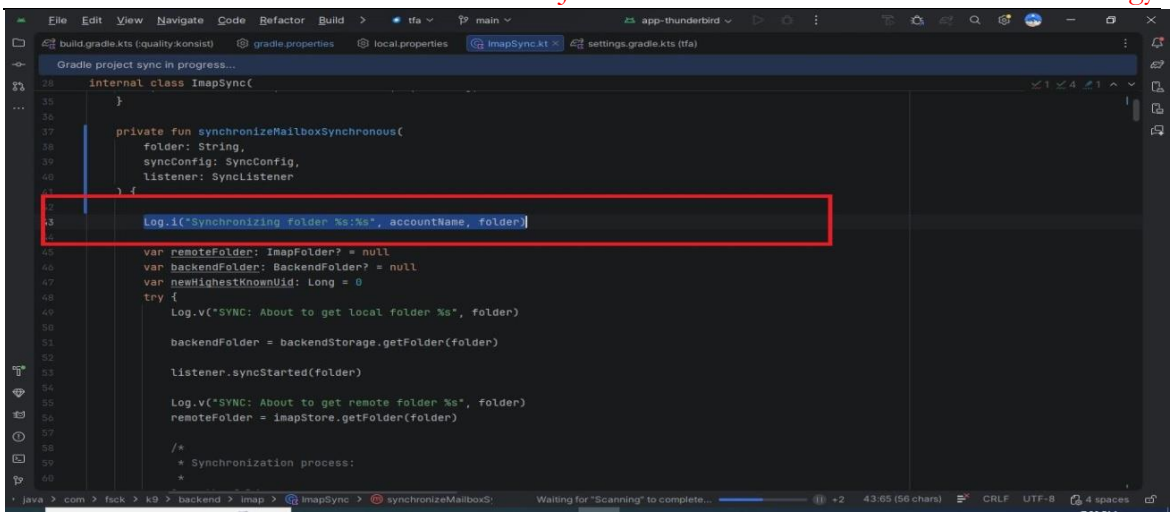

Figure 21. Code Snapshot in Android Studio Before Optimization

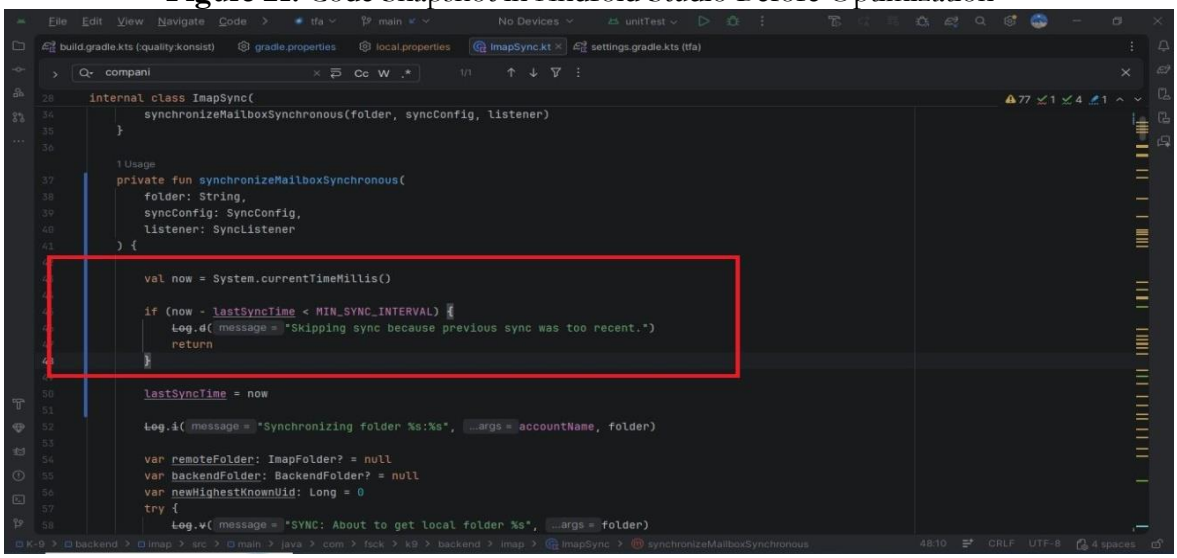

Figure 22. Code Snapshot in Android Studio After Optimization

OpenTask Application (UID10371):

Application clone made in Android Studio and research carried out regarding optimization of code removing unnecessary Wake-Up Alarms so when DelayedAction.java class studies alarm options procedure schedules an RTC_WAKEUP alarm. RTC_WAKEUP type of alarm triggers even when device is in sleep state and CPU exits its low-power state. Moreover, operating system delivers the broadcast immediately and background processing begins even if the user is not interacting with the device. Soothe option of Alarm changed from RTC_WAKEUP to RTC. Now optimized application schedules a normal RTC alarm and if the device is in sleep mode the CPU remains in its low-power state and alarm is delivered after the device naturally wakes resulting in no unnecessary wake-up occurring. Before and after code changes window of android studio are shown in figure 23 and 24.

Before Work Flow if an alarm generated:

Deep Sleep → RTC_WAKEUP → CPU Active → Execute Task → Return to Sleep

After Work Flow if an alarm generated:

Deep Sleep → Remain Sleeping → Normal Wake Event → Execute Task

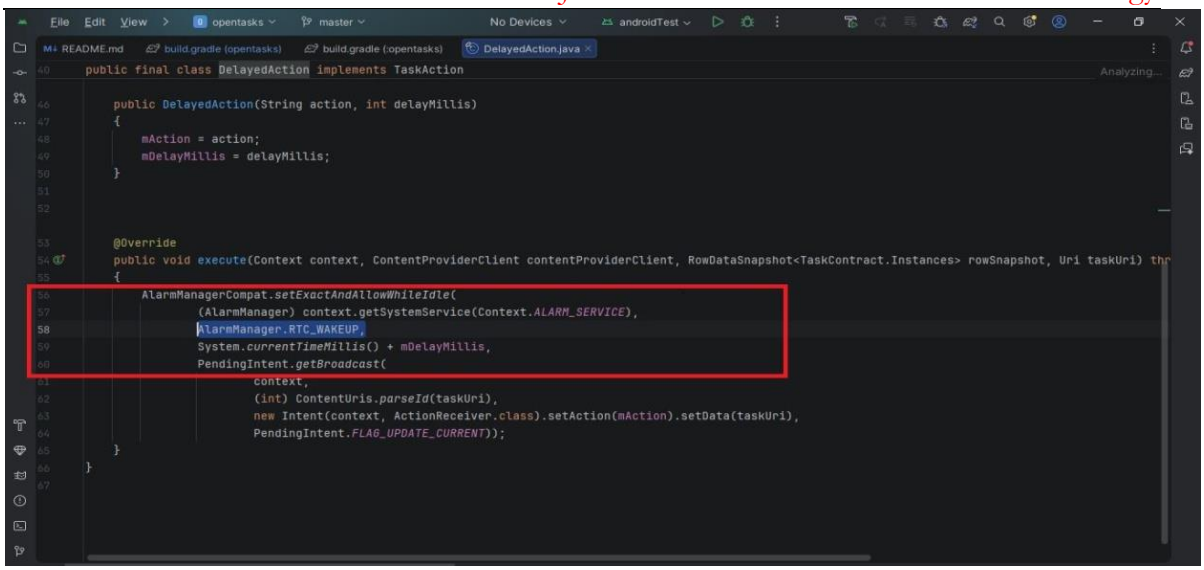

Figure 23. Code snapshot in Android Studio Before optimization

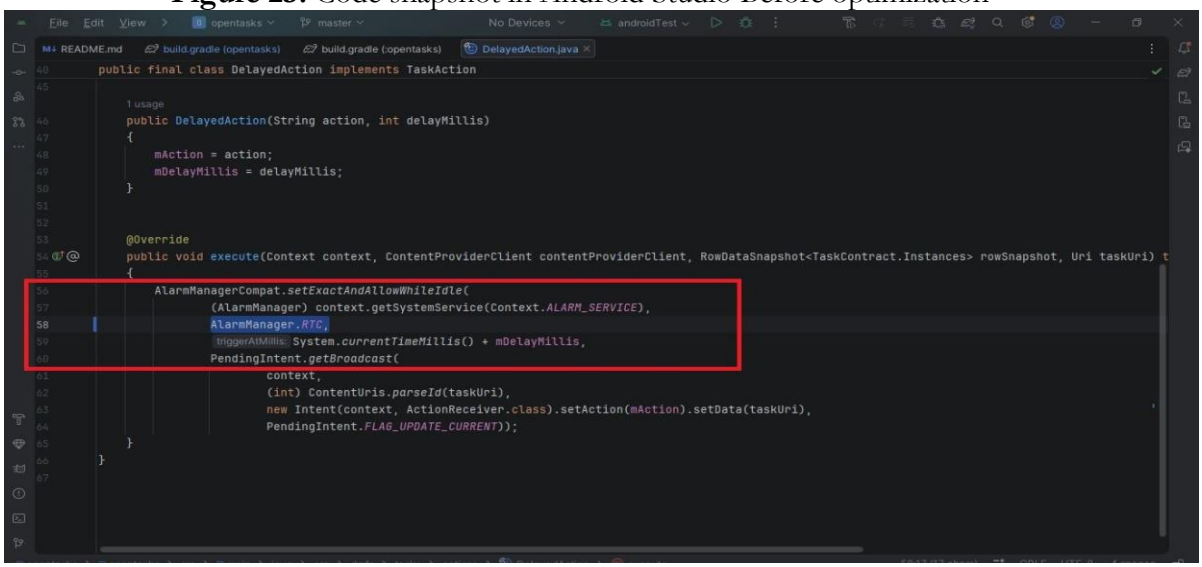

Figure 24. Code snapshot in Android Studio After optimization

Ethical Considerations:

All selected application projects were searched for any copy right issues. After confirmation of open-source rights, projects downloaded in android studio using link provided on GitHub. It is pertinent to mention that no user data was collected and strictly followed the ethical and privacy guidelines.

Results and Analysis:

Experimental Setup Overview:

Each project was cloned in android studio and application run in debug mode before and after code changing:

Baseline Version: Original unmodified code as Downloaded from GitHub

Optimized Version: Code changed by implementing concerning battery optimization technique.

Before any change in code of application project, it was run on android physical device (Samsung A52) with API level 34 and battery statistics noted down in identical environment after making changes in code and application re-run to check battery consumption. All apps installed are shown in the pasted snapshot of android device screen in figure 25.

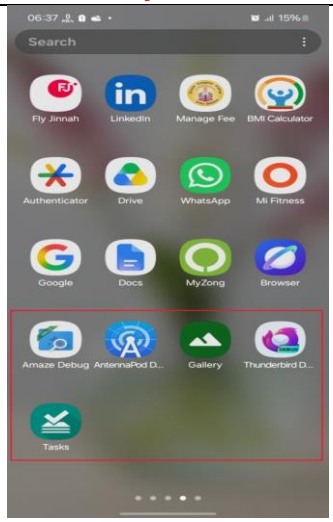

Figure 25. All Applications running on a physical device snapshot

Cross-Metric Trade-off Analysis:

In general concept it was doubt that battery optimization will degrade performance however after observation and interaction with application responsiveness was not affected. Furthermore, it resulted in reduction of CPU wake calls and these less requests to CPU made UI smoother. Efficiently use of network resource improved latency.

Battery Savings per Code Optimization Discussion:

Amaze File Manager Application (UID 10395):

Change 1: Using Iterator.remove() instead of list.remove(i)

Previous implementation:

Removing an element from an ArrayList using list.remove(i) caused shifting all remaining elements and Updating indices which resoled in repeated size checking If many null objects exist like

1000 files

100 nulls

100 shift operations

It has increased CPU work.

Optimized implementation:

Iterator.remove() removed the current element safely without repeatedly recalculating indexes. Benefits were fewer element shifts, less loop overhead and faster completion

Change 2: Reusing the same File ObjectFile f = new File(path) at multiple Places

Only one File object created in code instead of 2 instances of file

Energy Analysis:

Object creation itself is inexpensive, but repeated thousands of times it will cause:

Additional heap allocation during memory allocation

More garbage collection will be performed by procedure

Increased CPU cycles during execution

Extra memory traffic

Using one object instead of two reduces

Allocations by 50% which is huge saving of resources

GC frequency

Cache misses will be reduced

Less Garbage collection as less allocation was performed

Suppose there are 5000 files so before 10000 File object were created however now 5000 objects will be created. So that removes 5,000 unnecessary allocations.

Change 3: Caching Directory Metadatadir.length() and lastModified()

Each method call may trigger a filesystem metadata lookup. Local variable created to save Meta data instead of accessing the file system each time.Filesystem metadata access is much slower than reading a local variable. Each repeated call follows this work flow:-

Java →JNI→Linux Kernel→File system →Storage

This requires CPU activity and I/O. But caching avoids these repeated operations.

Results:

Calculation of battery consumption carried out by running bugreport and analyzing in Battery Historian Tool. Before optimization, out of four usages of the application with duration of 10 minute approximately noted Average Battery Discharge Rate (%/hrs) was 15.61. After Optimization of the code Average Battery Discharge Rate (%/hrs) was 10.46 which is 33 % less than the previous one so it proves the less consumption of battery due to code changes. Snapshot of 1st run is pasted as figure 26 and 27.

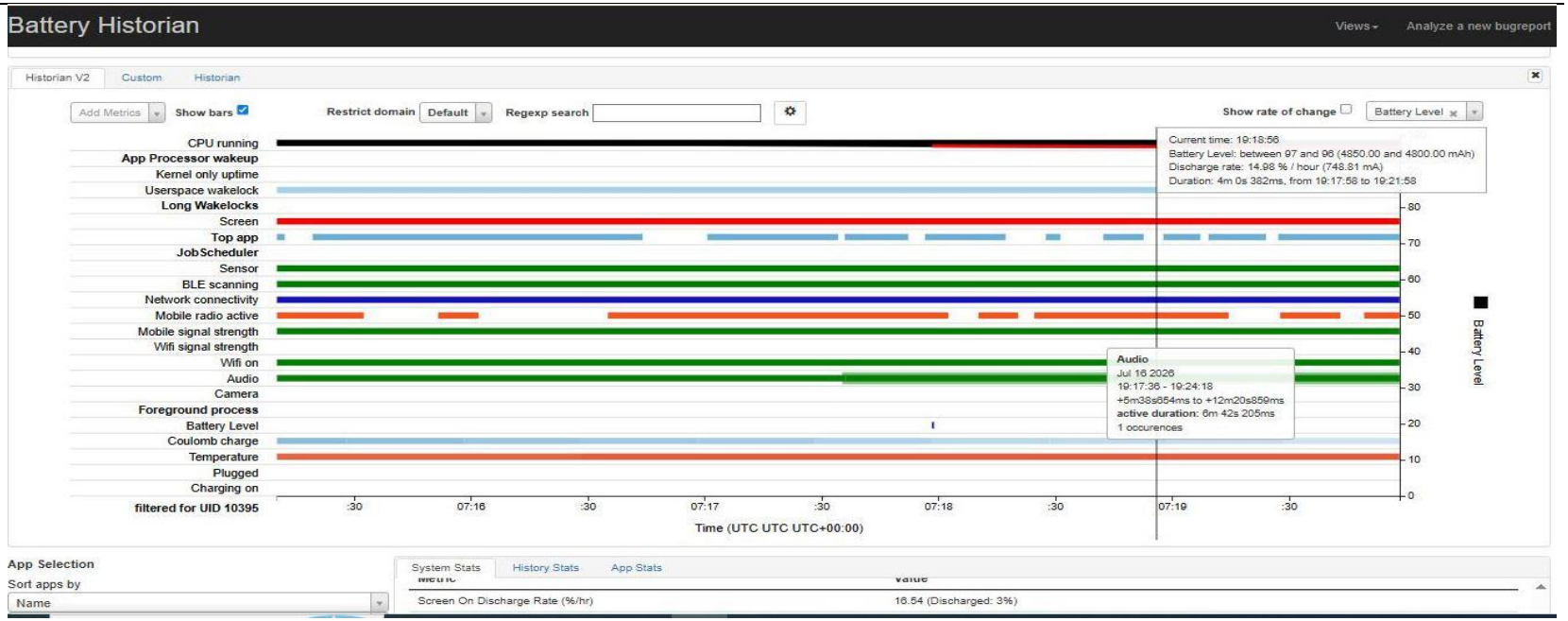

Figure 26. Before Implementation of Optimization Battery Historian Graph (UID 10395)

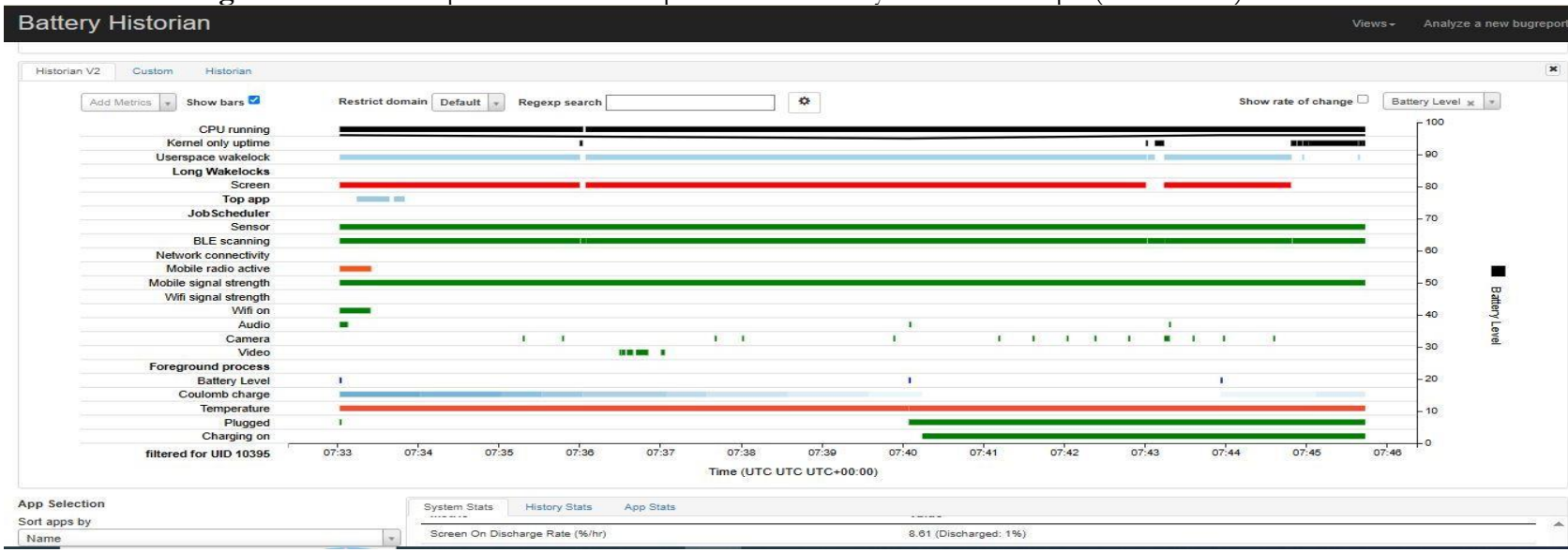

Figure 27. After Implementation of Optimization Battery Historian Graph (UID 10395)

Figure 28. Before Implementation of Optimization Battery Historian Graph (UID 10376)

Figure 29. After Implementation of Optimization Battery Historian Graph (UID 10376)

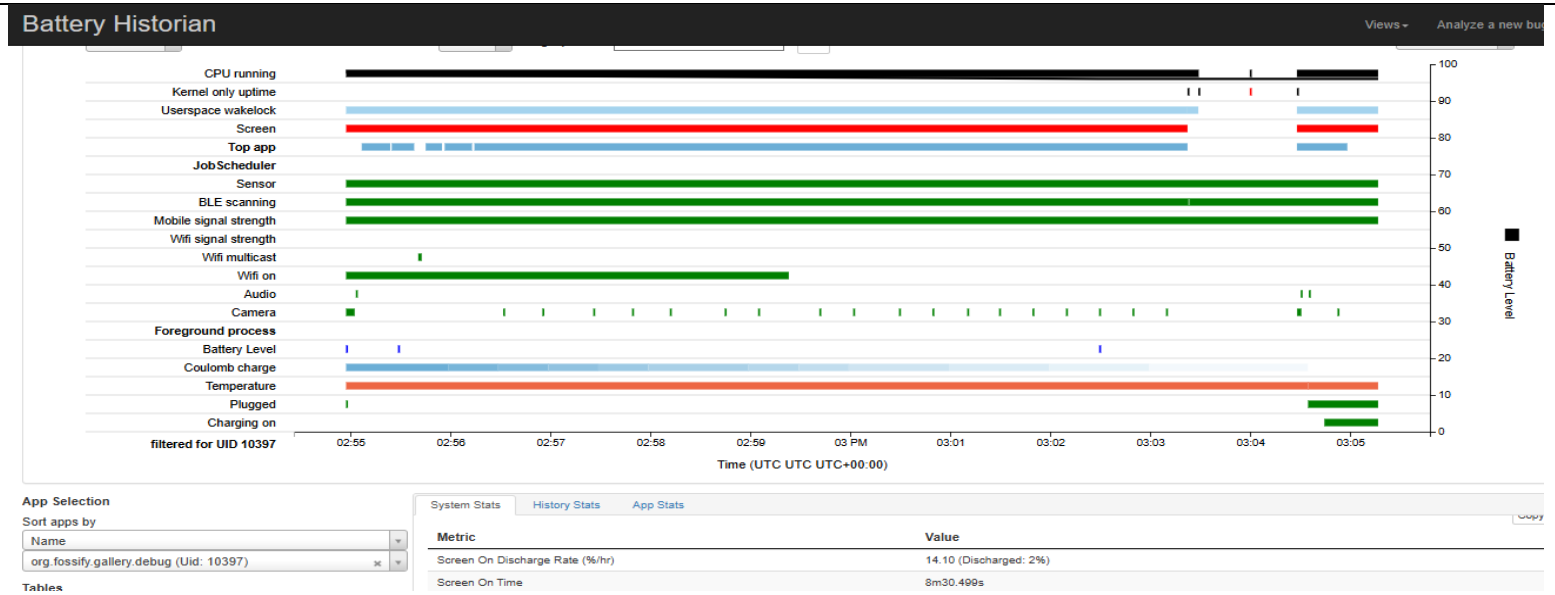

Figure 30. Before Implementation of Optimization Battery Historian Graph (UID 10397)

Figure 31. After Implementation of Optimization Battery Historian Graph (UID 10397)

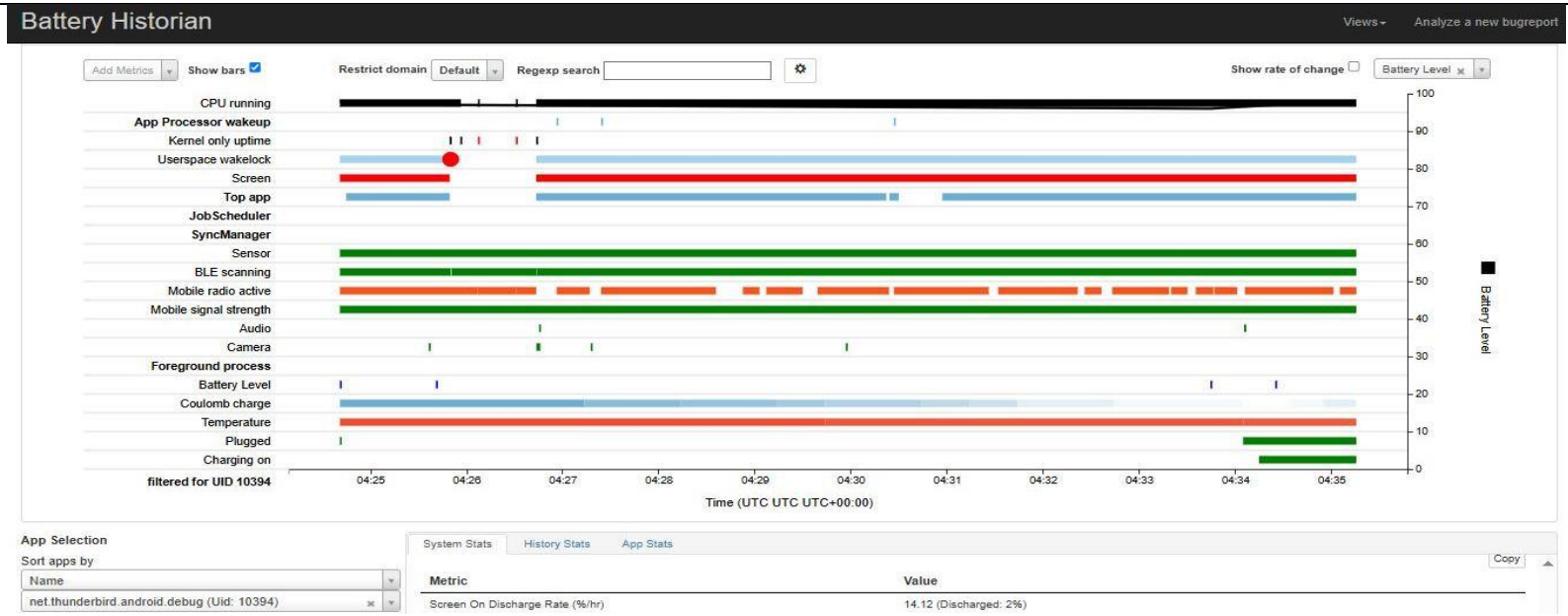

Figure 32. Before Implementation of Optimization Battery Historian Graph (UID 10394)

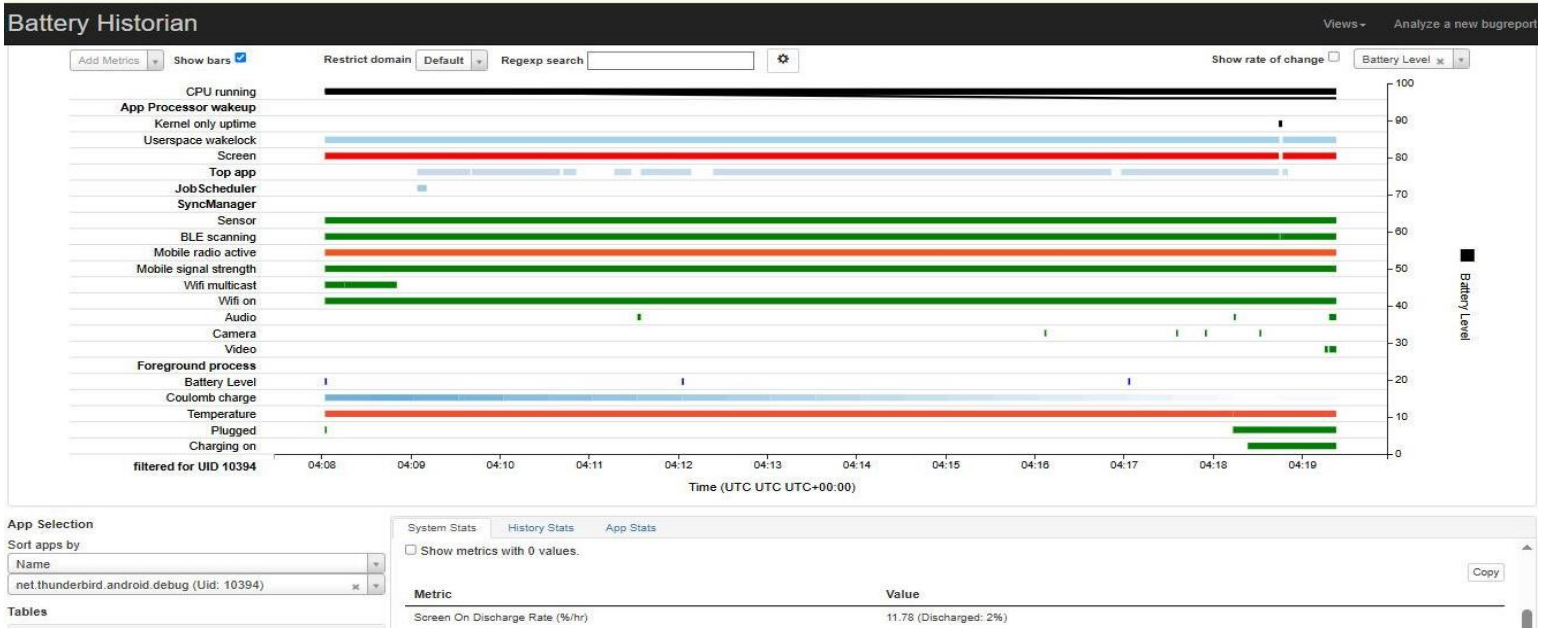

Figure 33. After Implementation of Optimization Battery Historian Graph (UID 10394)

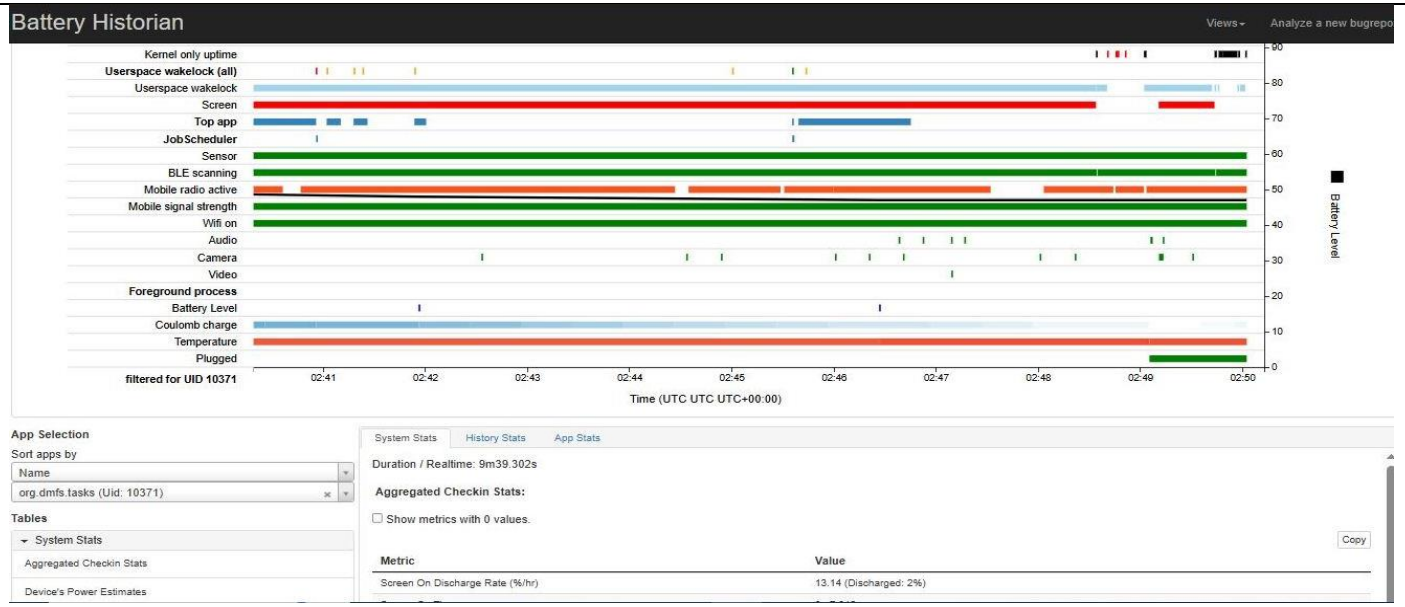

Figure 34. before Implementation of Optimization Battery Historian Graph (UID 10371)

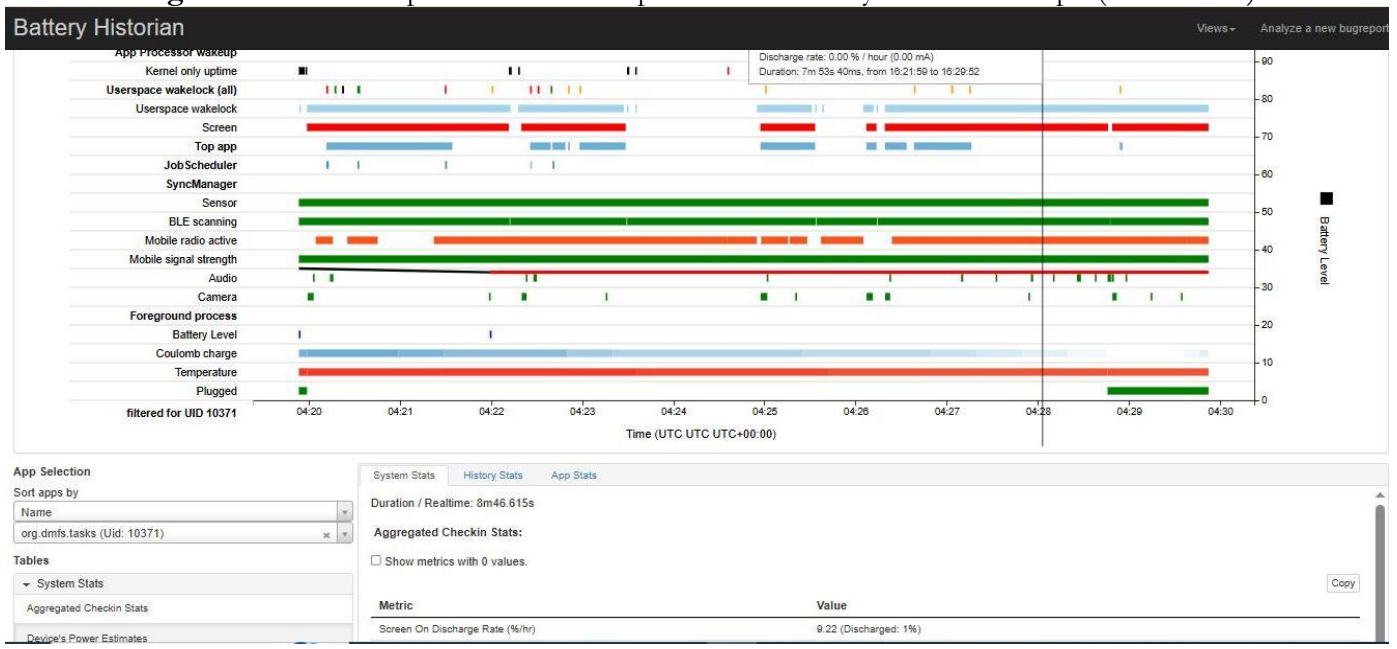

Figure 35. After Implementation of Optimization Battery Historian Graph (UID 10371)

Using wifiLock.acquire () with conditional execution

Energy Benefit:

Every call to acquireWifiLockIfNecessary() executes wifiLock.acquire(), even when the lock is already held. So, condition was imposed in code to check if already lock is held. Wi-Fi lock is acquired only when it is not already held. It caused: -

Reduces unnecessary CPU instructions.

Eliminates repeated framework processing.

Reduces Binder IPC transactions.

Results:

Calculation of battery consumption carried out by running bugreport and analyzing in Battery Historian Tool. Before optimization, out of four usages of application with duration of 10 minute approximately noted Average Battery Discharge Rate (%/hrs) was 22.28. After Optimization of code Average Battery Discharge Rate (%/hrs) was 17.37 which is 22.0 % less than the previous one so it proves the less consumption of battery due to code changes. Bugreport snapshot of 1st run is pasted as a figure 28 and 29.

AntennaPod Application (UID 10376)

FossifyGallery (UID 10397):

Change 1 onViewRecycled() Optimization

What Changed?

The previous implementation traversed all child views (allViews.firstOrNull) which required iterating through the entire view hierarchy and Used Glide.with(activity) whose lifecycle is tied to the whole Activity.

The optimized implementation:

Directly finds the thumbnail using findViewById()and Used Glide.with(thumbnail) which binds the request to the ImageView lifecycle only. It clears image requests immediately when the RecyclerView item is recycled. Before every recycled item performs

Iterate all child views→Search matching ID→Clear Glide request

It Time Complexity O(n) where n is the number of child views. However, after the lifecycle will be

findViewById()→Direct lookup→Clear request

This reduces unnecessary traversal.

Energy Impacts Acquired:

Clearing Glide immediately caused

Releases bitmap references earlier

Prevents recycled views from retaining images

Lowers memory pressure

Reduces Garbage Collection frequency

If the image is animated (GIF/WebP)Glide stops

Animation rendering

Frame decoding

GPU texture updates

As soon as the item leaves the screen cancelled image requests avoid

Additional thumbnail decoding

Unnecessary disk reads

Unnecessary cache access

This optimization benefits long scrolling sessions.

Change 2 setupThumbnail() Optimization

Multiple calls medium.isVideo() and medium.isPortrait() were executed repeatedly. Then a variable created to acquire the value. These values used in logics. The values are

computed once. Repeated method calls become Boolean lookup once and re-used in the method everywhere which is much cheaper.

Battery Impact:

This is a micro-optimization. For one item Savings are negligible. For 5000 thumbnails it avoids thousands of repeated function calls.

Change 3 loadImageBase() Optimization Glide.with(applicationContext)

The request was tied to the Application lifecycle. Image requests could continue even after:-

RecyclerView recycled

Activity destroyed

User closed gallery

So It was Changed with Glide.with(target). Now request is tied to the ImageView lifecycle. When the thumbnail disappears Glide automatically Cancel the decoding, disk reads, bitmap transformation and releases resources earlier.

Additional Optimization:

.transition(getOptionalCrossFadeTransition(
THUMBNAIL_FADE_DURATION_MS))

To:-

transition(getOptionalCrossFadeTransition(0))

Removing the thumbnail fade animation reduced

GPU composition

Animation frames

UI rendering work

This is another small but energy improvement during rapid scrolling.

Results:

Calculation of battery consumption carried out by running bugreport and analyzing in Battery Historian Tool. Before optimization, out of four usage of application with duration of 10 minute approximately noted Average Battery Discharge Rate (%/hrs) was 14.46. After Optimization of code Average Battery Discharge Rate (%/hrs) was 11.94 which is 17.4 % less than the previous one so it proves the less consumption of battery due to code changes. Bugreport snapshot of 1st run is pasted as figure 30 and 31.

ThunderBird (K-9 Mail) (UID 10394):

Optimization: Network Synchronization Throttling

Previously every synchronization request executed the complete synchronization process regardless of how recently the previous sync occurred. This meant that multiple sync requests occurring within a short period performed and resulted IMAP server connection, Authentication, folder opening, message count retrieval and Database updates even if synchronization had just completed.

After Optimization a synchronization interval check was added at the beginning of the method. In this case it prevents repeated synchronization requests within a 30-second interval.

Energy Impact Analysis:

Every request activated this action in a series

Wi-Fi radio or cellular modem

TCP connection establishment

TLS negotiation (when applicable)

IMAP communication

Packet transmission and reception

Each activation keeps the radio in a high-power state. As we know network radios are among the largest contributors to battery consumption in communication applications. So after checking that within 30s there is synchronization occurred saves all the mentioned resources.

Results:

Calculation of battery consumption carried out by running bugreport and analyzing in Battery Historian Tool. Before optimization, out of four usages of application with duration of 10 minute approximately noted Average Battery Discharge Rate (%/hrs) was 13.42. After Optimization of code Average Battery Discharge Rate (%/hrs) was 11.77 which is 12.3 % less than the previous one so it proves the less consumption of battery due to code changes. Bugreport snapshot of 1st run is pasted as figure 32 and 33.

OpenTask Application(UID 10371):

Optimization: Removing Unnecessary Wake-Up Alarms:

Previously the procedure schedules an RTC_WAKEUPalarm. When the alarm triggers:
Device wake from sleep,
CPU exited its low-power state,
The operating system delivered the broadcast immediately,
Background processing begins even if the user is not interacting with the device.
Now optimized application schedules a normal RTC alarm and if the device is asleep:
The CPU remains in its low-power state,
Alarm is delivered after the device naturally wakes,
No unnecessary wake-up occurs.

Energy Impact Analysis:

Before:

RTC_WAKEUP forces the processor to wake from suspend mode. This causes CPU wake-up and interrupt handling. Even a short wake-up consumes energy because the processor transitions from deep sleep to an active state.

After:

Using RTC allows the processor to remain asleep until another legitimate wake event occurs. Benefits include fewer CPU wake-ups, longer deep-sleep duration, reduced active processing time. The largest battery saving comes from allowing Android to remain in deep sleep.

Before Work Flow:

Deep Sleep→RTC_WAKEUP→CPU Active→Execute Task→Return to Sleep

After Work Flow:

Deep Sleep→Remain Sleeping→Normal Wake Event→Execute Task

Battery Consumption:

Modern Android devices consume very little power in deep sleep, but significantly more when awakened. So when the efficient use of wake lock acquired in logic it caused battery saving by allowing device to go to sleep when usage not required by user

Results:

Before any customization application battery consumption noted down. After that coding changes made to make the code more energy efficient. Post optimization results noted down in battery historian tool after fetching of bugreport. Optimized code implementation resulted averaged battery discharge rate (%/hrs) 11.77 which was previously averaged at 13.42. Comparing this battery savings showed that 12.3 % less battery consumed after new code implementation. It can be evident in below in figure 34 and 35.

Battery Consumption Analysis:

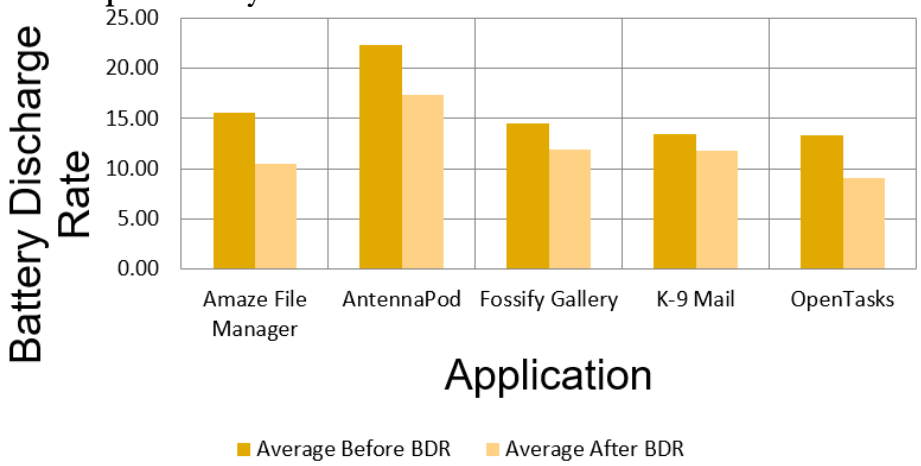

Figure 36. Average Battery Saved in applications graph

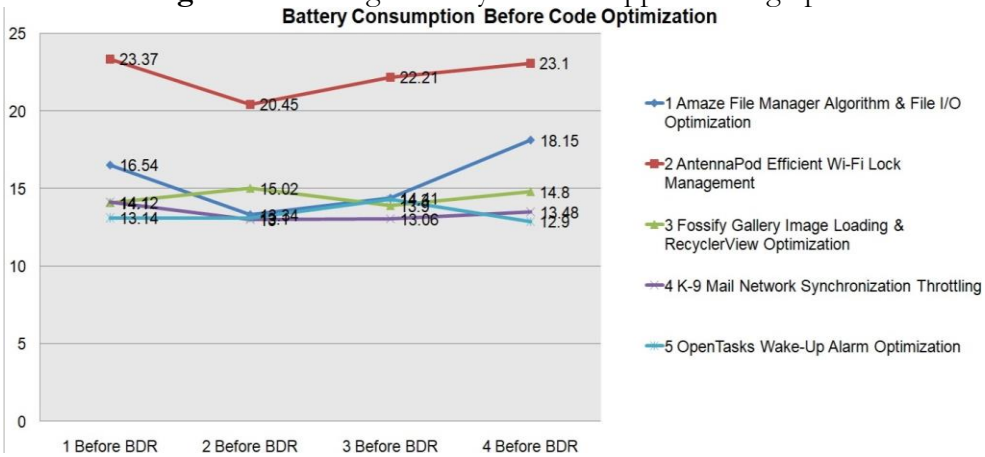

Figure 37. Graph Showing Battery Consumption (BDR %/hrs) Before Code Optimization in Four Runs

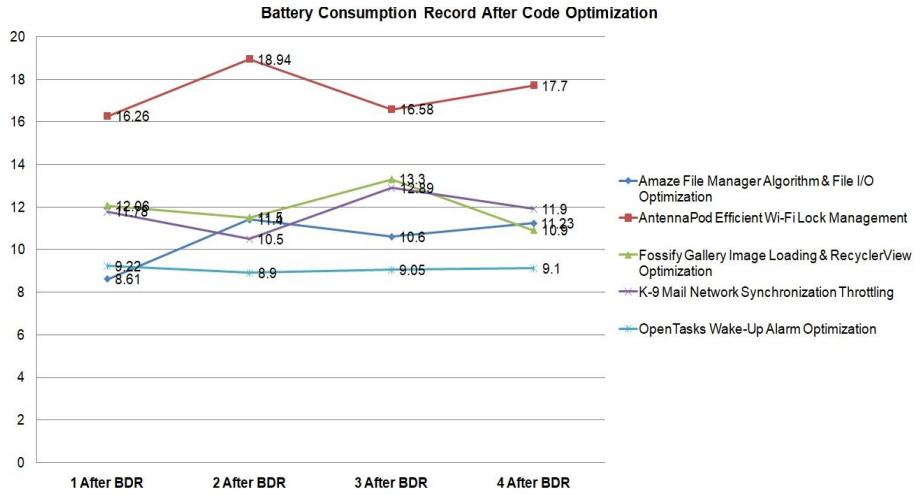

Figure 38. Graph Showing Battery Consumption (BDR %/hrs) After Code Optimization in Four Runs

Summary: Code Changes vs Battery Savings:

Table 3. Mapping of Code Changes and Techniques with Battery Savings

Application	Optimization Technique	Average BDR Before	Average BDR After	Battery Saving
Amaze File Manager	Algorithm & File I/O Optimization	15.61	10.46	32.99%
AntennaPod	Efficient Wi-Fi Lock Management	22.28	17.37	22.05%
Fossify Gallery	Image Loading & RecyclerView Optimization	14.46	11.94	17.40%
K-9 Mail	Network Synchronization Throttling	13.42	11.77	12.28%
OpenTasks	Wake-Up Alarm Optimization	13.36	9.07	32.13%

Cumulative Impact:

On average overall battery savings ranged between 12% to 33 % and varied across each application due to the unequal workload. This confirmed and validated that small code changes during development can drastically save power consumption and it resulted to prove the central hypothesis of this research. Results are shown in Table 3.

Statistical Validation of Battery Savings:

A paired-samples t-test was conducted to determine whether the observed reductions in Battery Discharge Rate (BDR) after applying the proposed coding optimizations were statistically significant. The four baseline measurements were paired with the corresponding four post-optimization measurements for each application. A significance level of $\alpha = 0.05$ was used. A 95% confidence interval (CI) was also calculated for the mean reduction in BDR.

Table 4. T Tests Results

Application	Mean Before BDR	Mean After BDR	Mean Reduction	Saving	t(3)	p-value	95% CI of Reduction	Significant?
Amaze File Manager	15.61	10.46	5.15	33.0%	3.723	0.0337	0.75–9.55	Yes
AntennaPod	22.28	17.37	4.91	22.0%	4.108	0.0261	1.11–8.72	Yes
Fossify Gallery	14.46	11.94	2.52	17.4%	3.336	0.0445	0.12–4.91	Yes
K-9 Mail	13.42	11.77	1.65	12.3%	3.098	0.0534	−0.04–3.34	Slightly
OpenTasks	13.36	9.07	4.29	32.1%	13.008	0.0010	3.24–5.34	Yes

The original measurements in the findings file are consistent with the reported average BDR values: Amaze File Manager decreased from 15.61 to 10.46, AntennaPod from 22.28 to 17.37, Fossify Gallery from 14.46 to 11.94, K-9 Mail from 13.42 to 11.77, and OpenTasks from 13.36 to 9.07.

Interpretation:

The statistical analysis shows (Table 4) that four of the five applications demonstrated statistically significant reductions in BDR at the 5% significance level. Amaze File Manager, AntennaPod, Fossify Gallery, and OpenTasks all produced $p < 0.05$. OpenTasks showed the strongest statistical evidence, with $t(3) = 13.008$ and $p = 0.0010$.

For K-9 Mail, although the observed BDR decreased by approximately 12.3%, the paired t-test produced $p = 0.0534$. Therefore, the reduction is not statistically significant at $\alpha = 0.05$. This is actually important to report because it demonstrates that the study does not simply treat every observed reduction as statistically significant.

Application Responsiveness: Application responsiveness was assessed during the experimental scenarios before and after the implementation of the proposed coding

optimizations. The assessment focused on observable application behavior, including UI interaction, navigation, loading behavior, and general responsiveness during the predefined usage scenarios. No noticeable degradation in application responsiveness was observed after optimization. However, the present study did not collect quantitative responsiveness metrics such as CPU utilization, memory consumption, frame rate, execution time, or UI latency. Therefore, the conclusion regarding responsiveness is based on qualitative observation rather than statistical performance validation.

Threats to Validity:

Utmost work done to simulate the real-world usage but uncertain conditions and complex variable involvement show slightly difference in energy patterns. Each time bug report generated almost same scenario but exact battery discharge rate results are impossible. These small differences can be neglected. There are more possibilities of code changes available in every application but time constraint hindered more customization. These all-available optimization chances can be explored and recalculated in advance studies.

Summary of Results:

In results section it was presented that code customization as suggested and implemented produced well satisfactory results. All the demonstration of applications experiments validates changes made in code and every change made the application more power efficient. So, the findings validate importance and integration of study into mainstream android developments. While running of application efficiently in controlled environment concluded the average battery savings of ~23%

These results validate the core hypothesis of this thesis:

Software-level decisions have a powerful and measurable impact on battery life.

Conclusion:

Mobile computing is developing rapidly and thus changing the way humans interact with technology on a daily basis. From the communication tool it was when first launched, smartphone has increasingly developed into a computing platform that has grown essential for education, healthcare, finance, navigation, entertainment and enterprise applications. Nonetheless, this expansion is encumbered by an ongoing, critical issue, a limited battery. Advancements have taken place in battery technology but not at a pace similar to processors, sensors, and power-aware hardware. Therefore, a lot depends on the software for the usability and energy consumption of the device.

The objective of the research was to find out whether a coding logic for developers can significantly make lower battery consumptionin Android apps without harming performance and user experience. The focus of the paper was to place software developers at the heart of the solution. In contrast, several previous works have only attempted hardware level optimizations or diagnostic profiling tools. It was recognized that at each step developers were making decisions about design on a daily basis which affected the usability time of the user's device. This research designed, implemented, and evaluated a set of energy-efficient coding techniques like implementation of efficient data structure, efficiently usage of network resources in 5 open-source android applications that are widely used. The optimization of algorithm design, executing asynchronously, reducing utilization of background tasks, using an efficient data structure, optimizing sensor and network utilization and lifecycle aware resource management are the main techniques. The research found that software level changes can result in large and measurable energy savings when the aforementioned techniques are applied at the source-code level and validated using profilers and tested in controlled experiments. The experiment shows that discharge of battery, CPU utilization, and network utilization can be minimized heavily if smart coding practice is employed. The optimized versions across the selected applications always outperformed their baseline. It must be energy

efficient, while ensuring it properly works fine and performs well at an acceptable level. In many cases, energy savings of 15% to 35% occurred.

In addition to the figures, this research added another critical insight. Energy efficiency has technical, human and social dimensions. It, therefore, raises questions of behaviour, acceptance and validity. When an application consumes an unreasonable amount of battery, it frustrates the user, reduces productivity and creates distrust. Developers, who are always under pressure to ship features under hurry factor and do not get to have information or feedback about their code's energy impact. This work will help bridge the gap between normal coding which is widely available in market and less resource consuming apps development.

Numerous Android apps have software glitches and inefficiencies that drain your battery substantially. Some glitches are resolved and fruitful result in the shape of less battery consumption achieved. This research presented and validated a feasible framework of smart and coding techniques for energy optimization.

Contributions to Research:

This study makes an important contribution to software engineering and mobile application development. It offers a developer perspective on optimizing energy research. Numerous previous works are dedicated to measurement and modeling, but in this study, focus was on, how developers themselves can change their coding practices for measurable energy savings. The suggested techniques are lightweight and practically applicable in real time Android projects. The study offered a reliable experiment protocol to estimate energy-efficiency enhancements employing easy tools and amendments after tests. The research will be valued for adoption and assessment by the industry without any hesitation.

Finally, it established that software sustainability begins at the source-code level and is mainly dependent on developer awareness and accountability, which contributes to the broad green and sustainable software engineering discussion.

Effect on Developers and Industry:

The proposed smart coding framework is designed to be applicable within the existing Android software development process rather than requiring a separate development environment. At the development stage, the framework can be integrated with Android Studio by incorporating energy-aware coding checks into the developer workflow. The optimization techniques evaluated in this study, including polling reduction, efficient background execution, network optimization, memory-aware image loading, and wake-up management, can be applied during source-code development and reviewed through profiling before and after implementation. Battery Historian and other available Android profiling facilities can subsequently be used to examine the energy behavior of the modified application.

The framework can also be extended to Continuous Integration and Continuous Delivery (CI/CD) pipelines. Energy-related checks can be incorporated after compilation and automated testing so that selected application builds can be evaluated for potential energy-intensive coding patterns. Baseline energy measurements can be maintained for important application functions, and subsequent builds can be compared against these baselines. When a substantial increase in energy consumption is detected, the build can be flagged for developer inspection. This approach would allow energy efficiency to become part of the software quality assessment process rather than being considered only after application development has been completed.

For commercial mobile applications, the framework can be applied incrementally without requiring developers to redesign an entire application. Existing applications can first be profiled to identify high-energy operations, followed by targeted optimization of the most significant energy-consuming components. The experimental results of this study demonstrate the practical potential of this approach, as the evaluated applications achieved BDR reductions ranging from 12.28% to 32.99% after applying targeted coding optimizations. Therefore, the

framework can support developers in prioritizing energy-related improvements according to the characteristics and workload of their applications.

It is particularly applicable to large commercial applications because energy optimization can be incorporated into existing development, testing, code review, and release processes. Rather than replacing existing software engineering practices, the framework adds energy efficiency as an additional quality consideration. However, direct integration with Android Studio and automated CI/CD energy gates was not implemented and experimentally validated in the present study. These integrations represent an extension of the proposed framework and an important direction for future research. Future implementations could provide automated energy warnings, build-level energy regression detection, and energy-related quality thresholds to support continuous energy-aware software development.

Limitations of this Study:

This study includes practical implementations but has several limitations. The tests were carried out in an environmentally controlled space and open-source applications. In order to mitigate variability, synthetic and repeatable workloads were used during running at physical device and maximum load put. However, real users and their behaviours can be much more complex and cannot be predicted. The study is only limited to android application too. The current work could act as a reference for researchers who are designing software's similar to that which is presented here.

Future Endeavors:

Future research should validate the proposed smart coding framework across a broader range of real-world environments to improve its external validity. In particular, the framework should be evaluated on commercial Android applications in addition to open-source applications to determine whether the identified optimization techniques remain effective in production-level software. Further experiments should be conducted on multiple Android devices with different hardware configurations, processors, memory capacities, battery technologies, and manufacturers to examine the consistency of the observed energy savings across device types. In addition, the framework should be tested on newer Android operating-system versions to assess its compatibility with evolving Android power-management mechanisms and platform-level optimizations. Such evaluations would help determine whether the reported improvements can be generalized beyond the specific applications, hardware, and Android environment used in the current study. Future studies could also employ larger sample sizes and longer experimental periods to provide stronger statistical validation and improve the reliability and generalizability of the proposed framework.

This study opens up many avenues for future investigations. Implementing energy trace feedback with development environments is one important direction. Subsequent enhancements could focus on providing IDE plug-ins that communicates notifications regarding energy effects in real-time while coding, like performance effect or linting effect and suggest code changing. An additional process where developers can expand is using a machine learning-based energy prediction that could suggest an energy-efficient alternative for inefficient code metrics. Recommendations can be customized as per the context of usage and the device in question. Further research will examine large-scale industrial validation of the proposed focus on code level battery optimization for commercial application in millions of users for long-term energy savings and impact on users. In education, which is related to android development, to be amended? In different android development phases small energy efficient techniques to be implemented and taught resulting in well aware and energy conscience next generation

References:

[1] "An analysis of power consumption in a smartphone | Proceedings of the 2010

USENIX conference on USENIX annual technical conference." Accessed: Aug. 14, 2026. [Online]. Available: https://dl.acm.org/doi/10.5555/1855840.1855861

[2] S. Mittal, "A survey of techniques for improving energy efficiency in embedded computing systems," *Int. J. Comput. Aided Eng. Technol.*, vol. 6, no. 4, pp. 440–459, Oct. 2014, doi: 10.1504/IJCAET.2014.065419.

[3] A. Gupta, B. Suri, D. Sharma, S. Misra, and L. Fernandez-Sanz, "Code smells analysis for android applications and a solution for less battery consumption," *Sci. Reports 2024 141*, vol. 14, no. 1, pp. 17683-, Jul. 2024, doi: 10.1038/s41598-024-67660-z.

[4] S. Huber, T. Lorey, and M. Felderer, "Techniques for Improving the Energy Efficiency of Mobile Apps: A Taxonomy and Systematic Literature Review," *Proc. - 2023 49th Euromicro Conf. Softw. Eng. Adv. Appl. SEAA 2023*, pp. 286–292, 2023, doi: 10.1109/SEAA60479.2023.00051.

[5] A. Schuler and G. Kotsis, "A systematic review on techniques and approaches to estimate mobile software energy consumption," *Sustain. Comput. Informatics Syst.*, vol. 41, p. 100919, Jan. 2024, doi: 10.1016/J.SUSCOM.2023.100919.

[6] R. Rua and J. Saraiva, "A large-scale empirical study on mobile performance: energy, run-time and memory," *Empir. Softw. Eng. 2023 291*, vol. 29, no. 1, pp. 31-, Dec. 2023, doi: 10.1007/S10664-023-10391-Y.

[7] M. Fawad, G. Rasool, and F. Palma, "Android Source Code Smells: A Systematic Literature Review," *Softw. - Pract. Exp.*, vol. 55, no. 5, pp. 847–882, May 2025, doi: 10.1002/SPE.3401;JOURNAL:JOURNAL:1097024X;WGROUP:STRING:PUBLICATION.

[8] W. Wysocki, "Why Don't Software Companies Care About Software Energy Efficiency? A Survey of Software Industry Developers.," *Procedia Comput. Sci.*, vol. 246, no. C, pp. 5054–5063, Jan. 2024, doi: 10.1016/J.PROCS.2024.09.589.

[9] A. Poth and P. Momen, "Sustainable software engineering—A contribution puzzle of different teams in large IT organizations," *J. Softw. Evol. Process*, vol. 36, no. 9, Sep. 2024, doi: 10.1002/SMR.2677;SERIALTOPIC:TOPIC:ACM-PUBTYPE.

[10] S. U. Lee, N. Fernando, K. Lee, and J. G. Schneider, "A survey of energy concerns for software engineering," *J. Syst. Softw.*, vol. 210, p. 111944, Apr. 2024, doi: 10.1016/J.JSS.2023.111944.

[11] J. M. Aragón-Jurado, J. C. de la Torre, P. Ruiz, and B. Dorronsoro, "Automatic software tailoring for Green Internet of Things," *Internet of Things*, vol. 30, p. 101521, Mar. 2025, doi: 10.1016/J.IOT.2025.101521.

[12] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y. M. Wang, "Fine-grained power modeling for smartphones using system call tracing," *EuroSys'11 - Proc. EuroSys 2011 Conf.*, pp. 153–167, Apr. 2011, doi: 10.1145/1966445.1966460;SUBPAGE:STRING:BASIC.

[13] X. Li and J. P. Gallagher, "A source-level energy optimization framework for mobile applications," *Proc. - 2016 IEEE 16th Int. Work. Conf. Source Code Anal. Manip. SCAM 2016*, pp. 31–40, Dec. 2016, doi: 10.1109/SCAM.2016.12.

[14] X. Gao, D. Liu, D. Liu, H. Wang, and A. Stavrou, "E-Android: A New Energy Profiling Tool for Smartphones," *Proc. - Int. Conf. Distrib. Comput. Syst.*, pp. 492–502, Jul. 2017, doi: 10.1109/ICDCS.2017.218.

[15] L. Cruz and R. Abreu, "Using Automatic Refactoring to Improve Energy Efficiency of Android Apps," *Av. en Ing. Softw. a Niv. Iberoam. CIbSE 2018*, pp. 163–176, Mar. 2018, Accessed: Aug. 14, 2026. [Online]. Available: https://arxiv.org/pdf/1803.05889

[16] W. Wysocki, I. Miciuła, and P. Plecka, "Methods of Improving Software Energy Efficiency: A Systematic Literature Review and the Current State of Applied Methods in Practice," *Electron. 2025, Vol. 14, Page 1331*, vol. 14, no. 7, p. 1331, Mar. 2025, doi:

10.3390/ELECTRONICS14071331.

[17] C. König, D. J. Lang, and I. Schaefer, "Sustainable Software Engineering: Concepts, Challenges, and Vision," *ACM Trans. Softw. Eng. Methodol.*, vol. 34, no. 5, May 2025, doi: 10.1145/3709352.

[18] S. Huber, L. Demetz, and M. Felderer, "A comparative study on the energy consumption of Progressive Web Apps," *Inf. Syst.*, vol. 108, p. 102017, Sep. 2022, doi: 10.1016/J.IS.2022.102017.

[19] J. S. Moreira, E. L. G. Alves, and W. L. Andrade, "A Systematic Mapping on Energy Efficiency Testing in Android Applications," *ACM Int. Conf. Proceeding Ser.*, Dec. 2020, doi: 10.1145/3439961.3439964.

[20] M. Muthu, K. Banuroopa, and S. Arunadevi, "Green and Sustainability in Software Development Lifecycle Process," *Sustain. Assess. 21st century*, Feb. 2020, doi: 10.5772/INTECHOPEN.88030.

[21] "Environmentally Sustainable Software Design and Development: A Systematic Literature Review." Accessed: Aug. 14, 2026. [Online]. Available: https://arxiv.org/html/2407.19901v1

[22] M. O. Adisa, S. Oyedeji, and J. Porras, "The nexus between ICT, top-down and bottom-up approaches for sustainability activities: A systematic mapping study," *J. Clean. Prod.*, vol. 449, p. 141768, Apr. 2024, doi: 10.1016/J.JCLEPRO.2024.141768.

[23] C. Groza, A. Dumitru-Cristian, M. Marcu, and R. Bogdan, "A Developer-Oriented Framework for Assessing Power Consumption in Mobile Applications: Android Energy Smells Case Study," *Sensors 2024, Vol. 24, Page 6469*, vol. 24, no. 19, p. 6469, Oct. 2024, doi: 10.3390/S24196469.

[24] Olivia Poy, M. Moraga, "Impact on energy consumption of design patterns, code smells and refactoring techniques: A systematic mapping study," vol. 222, p. 112303, 2025, doi: https://doi.org/10.1016/j.jss.2024.112303.

[25] Olivier Le Goaër, Julien Hertout, "ecoCode: a SonarQube Plugin to Remove Energy Smells from Android Projects," *ACM Int. Conf. Proceeding Ser.*, vol. 9, 2022, [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3559518

[26] E. Blokland, L. Cruz, and A. van Deursen, "EDATA: Energy Debugging And Testing for Android," *Proc. - 2025 IEEE/ACM 12th Int. Conf. Mob. Softw. Eng. Syst. MOBILESoft 2025*, pp. 94–104, 2025, doi: 10.1109/MOBILESOFT66462.2025.00017.

[27] Z. Chen, T. Wang, K. Zhang, Y. Guo, and Z. Shao, "A Q-Learning-Based Display Energy Optimization Scheme for Android Systems," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 44, no. 4, pp. 1262–1275, 2025, doi: 10.1109/TCAD.2024.3486244.

[28] S. Park, J. Kim, J. Lee, and H. Cha, "Ember: Task Wakeup Sequence–Based Energy Optimization for Mobile Web Browsing," *ACM Trans. Embed. Comput. Syst.*, vol. 24, no. 5s, Sep. 2025, doi: 10.1145/3757918;

[29] X. Dou, L. Liu, and L. Xiao, "An Intelligent Scheduling Approach on Mobile OS for Optimizing UI Smoothness and Power," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 1, Mar. 2025, doi: 10.1145/3674910;PAGE:STRING:ARTICLE/CHAPTER.

[30] A. Imran, T. Kosar, J. Zola, and M. F. Bulut, "Towards Sustainable Cloud Software Systems through Energy-Aware Code Smell Refactoring," *IEEE Int. Conf. Cloud Comput. CLOUD*, pp. 223–234, 2024, doi: 10.1109/CLOUD62652.2024.00034.