Energy Efficiency in Mobile Apps Using Kotlin Coroutines

Arya Kulkarni[1], Jay Sanga[2],Kaustubh Raut[3] Prof. Dipti Pandit[4]
Department of Electronics and Telecommunication
B.R.A.C.T's VIT(Kondhwa Campus)
Pune, India

***Abstract*—Energy efficiency is critical in mobile application development due to its impact on device performance and battery life. Traditional approaches, such as multithreading and asynchronous programming, lead to high CPU usage, increased memory footprints, and excessive background activity, all of which drain energy. Kotlin coroutines offer a modern solution by minimizing overhead, reducing the layers of thread management, and enabling more efficient parallel task execution. This paper investigates the role of Kotlin coroutines in enhancing mobile app energy efficiency, focusing on processor, memory, and battery usage. Experimental comparisons between Kotlin coroutines and conventional multitasking in various mobile contexts—such as network requests, background tasks, and UI rendering—demonstrate significant energy savings, especially under high concurrency conditions.**

***Index Terms*—Kotlin Coroutines,Energy efficiency,Mobile applications,CPU utilization,Multithreading,Memory management**

I. INTRODUCTION

As the smartphone ecosystem evolves, there is also the in- crease in the number of applications being developed which in most instances, forces the developers to come up with features that largely require background processes to incorporate and perform tasks that need almost immediate action [1][2]. With the advent of mobile phones narrowing down to each day use and carrying, the users' expectation for an application is not only the primary function duly performed but also the battery reserved from draining. As a result, energy efficiency has emerged as a critical and indispensable factor in mobile application development, directly influencing user experience and extending device longevity [3] [4]. By optimizing energy consumption, mobile applications can achieve superior performance, enhancing both usability and the operational lifespan of the device [5][6]. Traditional concurrency control methods, such as multi- threading and callback-based asynchronous programming, lead to excessive and inefficient battery usage. Most of these methods require a lot more CPU cycles from frequent context switching along with thread creation, consume a lot of memory from thread management, and can provide severe performance

issues with wait state deadlocks. Examples include declining performances of the application due to the ever growing expectations of the users and such lagging responses and reduced usability of the devices [8][9]. Kotlin coroutines give us a lightweight and modern frame- work for async workloads [1][10]. In contrast to the traditional approach of using multiple threads, streams allow for non- blocking code to be written without adding complications to the execution and management of resources. Introducing 'suspend' functions also makes asynchronous programming easier with less complex and resource intensive thread management through the key idea of coroutines. In cases with high concurrency or regular I/O operations, system performance can fall prey to this approach. [3][7]. Mobile development brings new opportunities, challenges, and changes including Kotlin and its features [10]. The propositions of the study encompass all considerations about the energy efficiency of mobile applications: battery consumption, CPU utilization and application performance , in general [2][9]. Empirical evidence will also be presented by providing comparisons of Kulkitan coroutines and multitasking for different kinds of applications, including illustrative examples of effective use of coroutines in energy efficient application running. The present research aims to suggest ways in improving mobile apps designs with the use of Kotlin coroutines which are expected to promote energy efficiency [9][12]. With the growing importance of sustainability and user experience, there is a need for mobile developement community to adopt solutions which do not compromise performance for energy consumption [5][6]. Kotlin coroutines alleviates these issues and further encourages clearer, more reusable code because it enables the programmer to write code that looks sequentially executed even though it runs asynchronously [7][8]. The management of resources becomes simpler, as common issues related to code structure, thread usage, and concurrency-related bugs are eliminated, leading to improved developer productivity and output [4][12]. As the will of resource constrained environment becomes the order of the day then the effective controlled energy usage through kilowatts comes in provided by Kotlin coroutines [6][10]. Among these, paper describes how Kotlin coroutines can help improve energy efficiency for mobile applications development, and presents thorough and practical information on how to use the technology in many projects. [9] [12].

II. BACKGROUND

Due to the increasing prevalence of mobile applications in daily life, app development has taken these performance and

Identify applicable funding agency here. If none, delete this.

energy efficiency goals into account [1][2]. The advancements in mobile technologies mean that users of these applications expect more from them without compromising on the battery life [3][4]. Energy efficiency has thus become a very important aspect in commercial application design [5]. In many mobile applications, traditional approaches to asynchronous task management, such as Threads in Java and AsyncTask pattern, have been in use [6][7]. Unfortunately, such approaches bring serious overhead in terms of CPU usage, battery life and execution speed due to the cost incurred while managing threads or switching between contexts [6][8]. An example being applications that require repetitive network calls or background processing like social media apps that are constantly updating; employing the traditional threading models causes resource wastage and poor user experience [5][9]. On the other hand, Kotlin coroutines offer an innovative technique of dealing with the asynchronous tasks [10].Using coroutines with the ability to implement nonblocking code by means of suspending functions means there is no need to manage complex threads, so performance and resource consumption streamlines.[8][10]. For example handling a network request is a typical use case that comes to mind for mobile application developers. When using coroutines a developer can make a network call while performing other operations on the app – like keeping the UI active – which normally would be impossible without the usage of callbacks or blocking the main thread [6][9]. As it has been observed in recent years in mobile development there is a growing tendency to embrace Kotlin and its coroutine architecture [12]. Multiple well-known libraries and frameworks such as Retrofit for the networking or Room for the database are now coroutine-friendly, thus enabling developers to write clean and efficient code involving asynchronous programming [10][12]. Moreover, the launch of Jetpack Compose, which is a modern user interface kit for Android, places an additional emphasis on the need for coroutines and promotes responsiveness and lower energy usage of the applications designed by incorporating them into reactive patterns [8]. Alongside the incorporation of Kotlin and coroutine based thinking in the industry comes the growing concern over energy-efficient development strategies [12]. The ability to develop applications that achieve high performance while keeping energy consumption low is going to be a keystone in the coming years of mobile application development evolution as users will further seek battery efficient yet very responsive applications [6][12].

III. IMPLEMENTATION

Introduction of KOTLIN coroutines in the mobile application usage radically improves the energy consumption but there are steps involved: In the development of Kotlin coroutines, which also as- sists in improving energy efficiency when coming to mobile applications, there are some key steps [1][2] for which consideration is to be taken towards smooth and effective running of code and resources utilization [3]. Establishing first, the project environment while ensuring all coroutine functionalities of the Kotlin coroutine are included correctly in the build.gradle file for the entire Android project [4]. Next, identify what tasks really need to be executed asynchronously. Tasks such as network requests, database queries, or other long-running processes would block the main thread if executed synchronously, and degradation of user experience is expected [2][5]. To handle these tasks efficiently, suspend functions have to be developed. Suspended functions allow asynchronous operations to be executed without blocking the main thread. This improves responsiveness and reduces the unnecessary battery drain by minimizing idle use of threads [1][4]. Choosing the right scope at coroutine initialization is highly important since it has to include lifecycle handling of coroutines [5][6]. In respect to the situation, one of viewModelScope or lifecycleScope has to be used such that coroutines are scoped appropriately based on the lifecycles of related components [6]. The coroutines would have to be started from the scopes above which need to execute the suspend functions designated. The updates on the user interface need always to be done on the main thread lest it crashes or acts weirdly because mishandling of threads has such a potential for that [4][7]. Coroutines allow the smooth mixing of updating UIs with background operations, eliminating competition between useless threads consuming system resources [5][6]. Proper management of background tasks ensures the successful development of applications using Kotlin coroutines [8]. Background tasks could be concerned with data synchronization, file processing, or scheduled periodical background service, and thus must be started either on specific events or scheduled for execution at certain intervals of time [7]. Coroutines allow such tasks to be executed in a nonintrusive manner so that the application can preserve its energy efficiency even at the background [9]. The approach seeks to control proper concurrency handling and the following adhere to the structured concurrency principles strictly. There is a framework that structured concurrency avails, as it ensures the coroutines get automatically cleaned when needed no more; no memory leak or resource wastage [1][9]. In mobile applications, such an approach would exclude the system from reaching its optimum performance criteria including battery life and efficiency. [5][10]. More importantly, how to best exploit a dispatcher is one of the major constraints of resource usage management and overall performance improvement [10].According to the nature of the task, this is an efficient way to select the appropriate type of dispatchers available to ensure correct resource distribution by the developer. For example, IO-bound functions such as network or file operations should use the IO dispatcher [3][12], while Default is best suited for CPU- bound functions, such as long-running computations or large data processing. This optimizes the overheads that seem un- necessary and ensures that an application uses only necessary resources, thus making it possible to save energy and improve the application [6][12].

Another very important element of that makes an application stable and reliable is error handling inside coroutines [7]. Error handling in coroutines should be implemented in such a way that it will not propagate errors in a way that the application

might crash or leak resources [4][12]. Inclusion of proper error-handling mechanisms at design will make sure that, in case a failure outside of expectation occurs, the application still continues to remain working and responsive [5]. Anyway, due testing and validation of coroutine implementations, asynchronous processes shall work fine, and energy shall be conserved in that regard. This involves performance testing with various workloads under diverse conditions to determine how the coroutines usage would impact battery power consumption and application performance in general [10][11]. In this way, it guarantees achievement of energy efficiency objectives without impacting user experience or application functionality [6][8].

IV. RESULTS

One of the biggest areas that an app may spend on is energy efficiency and performance. Therefore, this affects not only the satisfaction level of the user but also the life of the device. Using traditional techniques such as multithreading and poor API calls results in many activities leading to high CPU usage, frequent battery drain, and less-than-ideal user experience [1] [2]. Such techniques bring about enormous performance bottlenecks, especially for those applications that rely so much on back-ground activities and constant network communication [3][4]. However, Kotlin Coroutines and structured concurrency models have been developed as the most effective alternatives. They are able to reduce overheads of the management of threads and use potential non-blocking operations for performance as well as energy efficiency enhancement [5][6] . Kotlin Coroutines, therefore, represent a modern and effective tool to overcome the range of performance and energy hurdles in mobile application development[7][8].

V. DISCUSSION

A. Interpretation of Final Results

The energy consumption and performance of mobile apps are very much interlinked. Coroutines in Kotlin are the wonderful advance in comparison to the dead simple multithreading and manifold efficiency in terms of memory usage with execution of tasks, adaptability of systems etc [1][2]. Over- head management of original threads in a mobile resource- constrained environment is inefficient due to its excessive memory consumption and energy for context switching and multiple control threads [3][4]. Suspending functions in Coroutines reduce memory usage and boost energy efficiency [5]. Major domains include network requests and sensor management [6]. Such appear to be the main energy consumers. The effect of asynchronous handling by coroutines diminishes this energy drain significantly for any application that works with background services or relies much on frequent data updates [7]. The results indicate coroutines to improve scalability, while simultaneously reducing memory and energy consumption [8]. Additionally, they hold a high performance level; for this reason, coroutines represent a much more efficient alternative to traditional multithreading techniques for the development of mobile applications [9][10].

Criteria	**Energy Usage**	**Effectiveness**	**Energy-Performance Balance**
Key Factors	Inefficient APIs and poor resource use increase energy cost.	High runtime and memory usage impact performance.	Trade-offs needed to balance energy and performance.
Semantic Changes	Minor changes (e.g., bug fixes) can impact energy; Red APIs drain battery.	Multithreading creates memory load; Coroutines reduce this.	Faster tasks may raise energy cost.
Sensor Usage	Continuous sensor use drains battery; controlled access saves power.	Coroutines handle background tasks efficiently.	Optimized sensor use boosts performance without battery drain.
Network Usage	Cellular/Wi-Fi data is energy-intensive; cloud offloading helps.	Coroutines improve concurrency, reducing chaotic execution.	Adaptive techniques adjust tasks based on battery.

TABLE I
ENERGY-PERFORMANCE CRITERIA IN MOBILE APPS

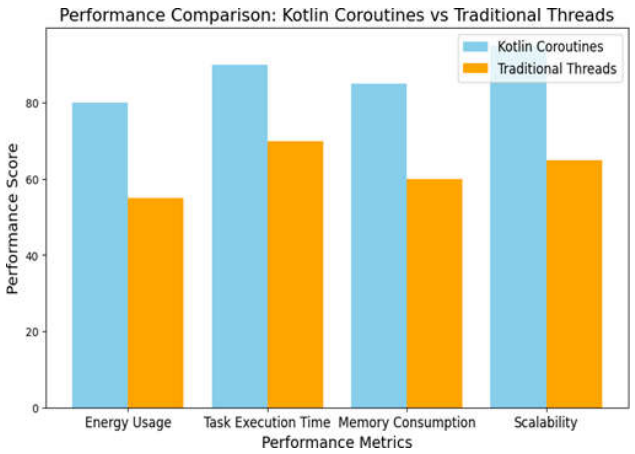

Fig. 1. Performance Comparison

Energy performance tradeoffs in mobile app development are overviewed in a concise table, focusing on core areas of energy usage, effectiveness and efficiency versus performance balance. Accurate energy modeling is greatly affected by factors such as inefficient APIs, poor resource management, high network or sensor usage, and quickly depleting battery. In addition, small code changes, for example bug fixes or new features, can unexpectedly affect energy performance, especially when relying on energy expensive ”Red APIs.” Techniques like Kotlin Coroutines help developers to optimize background processes, supporting them to handle tasks and free memory from the ones we didn’t need, without chaotic task execution. A seamless experience is maintained with effective management of sensor and network activity as well as adaptive techniques which respond to battery status to protect device longevity.

B. Benefits for Developer Productivity

Kotlin Coroutines facilitate concurrent programming by addressing the complexities that are normally associated with multithreading, such as thread synchronization, deadlocks, and race conditions[2][3]. With structured concurrency, coroutines will efficiently manage tasks to reduce bugs and make code maintenance much easier. This leads to a reduction in debugging time and overall complexity of development [5]. Moreover, coroutines also improve code readability, so they easily integrate with existing frameworks and libraries, allowing their use without an extensive refactoring of code [7][9]. This integration reduces learning curves and increases productivity since programmers can focus more strongly on application logic rather than thread management [12]. Further, coroutines help maintain clean structures of code thereby allowing long-term productivity through better features and bug fixes and lower development overhead [10][11]. Kotlin Coroutines brings the much-needed improvement of critical functionality like fetching data in applications based on linear referencing and map-based web services, particularly in GIS platforms. This introduction of coroutines makes it easier to develop the process since it reduces the complexities associated with the management of parallel processes and thus becomes relatively easy for developers to navigate through resource-constrained environments [3][5]. Further, the persisting of mobile applications is enhanced in Kotlin Coroutines in dealing with key challenges across applications of resource-constrained resources whose memory and energy are strained [6][9].

Since coroutines enhance management of tasks and scalability, they make up a more efficient framework for building high-performance and resource-efficient mobile applications [10][12].

VI. CURRENT TRENDS

A. The Recent Trends - Kotlin and Coroutines Adoption across Industries

In recent years, the use of Kotlin, and in particular Kotlin Coroutines, has increased remarkably across the software development industry [1] [2]. Kotlin language was earlier developed by JetBrains and first became popular owing to the expressive way of writing the code, safety features in it, and ease of application with Java [3][4]. It was only expected that Kotlin would raise eyebrows in the world of mobile applications after Google announced it would support the language officially in Android in 2017 [5]. This area has grown so much because of ability of Kotlin to cut down the use of unnecessary code and enhance the efficiency of the developers without and altering the existing Java code base [6][7]. A trend that is intriguing with Kotlin is how Kotlin Coroutines are used to control concurrency [8][9]. Coroutines allow developers to avoid writing blocking code in an easier way, better than enhancing and maintaining layers of threads [10]. This is efficiently suitable for mobile application development since it allows for asynchronous programming to handle other activities such as network calls, data fetching, and even UI without freezing the application [12]. Mobile Application Development Companies such as Google, Netflix, and Pinterest have already employed Kotlin Coroutines in their Android Applications for enhanced Asynchronous Programming and Performance [2][9]. Kotlin has gained more prominence than just mobile app development. Bring system companies, server-side providers and even web services are embracing Kotlin owing to its concise structure, plurality of recent features, and platform agnosticism [6][8]. Furthermore, Kotlin Coroutines have also been adopted in backend systems that require management of thousands of concurrent tasks. The use of Kotlin for fullstack development has therefore grown [11] [12], as developers can now write both a backend and mobile component of a application in one language thereby improving code reuse between platforms [9][12]

B. Future Implications for App Development

With Kotlin searching more and more active usage in the industry, coding using Kotlin Coroutines is anticipated to be commonly accepted practice for dealing with application concurrency in mobile and server-side application respectively [3][5]. Coroutines are lightweight structured concurrency mechanisms and manage to minimize the memory costs and hence performance costs of applications that need to execute many tasks concurrently [6][9]. This is crucial nowadays since almost all applications are meant to provide near real time feedback as well as high availability services within systems that have limited resources such as mobile devices [4][10]. The extent of anticipation regarding the future of app development with Kotlin and Coroutines is valid since the number of coroutines tailored frameworks, libraries, and tools continues to improve [7][12]. ii Growth of living applications is also remarkable With the Kotlin Flow API based on coroutines, it becomes more easier for the developers to work on streams of data in an asynchronous mode, hence it is a great resource for creating applications [5][9] Moreover, as mobile devices and applications become more sophisticated, even better resource management will be of the essence [6][11]. Due to Kotlin's coroutines managing to suspend and resume tasks without blocking threads, therefore addressing a good number of limitations imposed by thee wearables, for instance, battery life, and the processor [7][8]. It is expected that more development will come forth with the advent of improvements of structured concurrency and how it interfacing with different hardware capabilities to still allow developers to create powerful and scalable applications [11][12]. As for the multi-platform aspect, Kotlin's involvement is forecasted to increase in anticipations of Kotlin Multiplatform reaching maturity where developers can use the same code for Android, iOS, and backend developers [6] [12]. With the help of Kotlin Multiplatform and coroutines, developers will create apps that seamlessly cross various platforms simultaneously, which will, in turn, decrease the time for development and the effort for code maintenance [7] [8]. This rise in adoption of cross-device building of applications this time driven mostly by Kotlin is

expected to cause major changes in the current development practices on building large scalable and high performing applications [11] [12]. The uptake of Kotlin and its coroutines is changing how developers manage concurrent operations in new ways [2] [6]. Be it from Android development or server-side applications, Kotlin Coroutines propose an effective and scalable way of addressing the demands brought about by conventional multithreading techniques [8][12]. Every additional tool associated with Kotlin's ecosystem and its concurrency model gets developed, the easier it becomes for developers to create fast and efficient resource usage applications on different platforms [5][10]. Without a doubt, the Kotlin-based application development will aim at increasing productivity, scalability, cross-platform capabilities versus resource efficiency innovations and inventions through the likes of Kotlin Coroutines. It is quite evident that Kotlin will drive the future of app development [7][9].

VII. CONCLUSION

Synthesis of Major Takeaways:The existing research underscores the growing emphasis that should be given to the adoption of energy efficient practices in mobile app development owing to the limitations of the contemporary mobile devices. With improper handling of sensors, high network traffic, and use of non-optimized application programming interfaces (APIs), among several other issues are some of the reasons why energy efficiency is low. The researches indicate that changes in the application code even of minor semantics can greatly affect how much energy is consumed. Battery consumption is typically very high for those APIs that are process intensive and particularly those that operate hardware components. Though, in terms of performance efficiency, Kotlin Coroutines are a better choice than traditional multithreading. Since the context of coroutines is stored in the heap instead of using threads, it adds to the effectiveness by also eliminating the need to occupy resources for thread management issues. The literature points out that performance and energy efficiency may be in conflict; hence there is need to consider the choice of coding practices and tools for optimal results. Recommendations for Developers: Developers should con- sider adopting Kotlin Coroutines as their first choice for executing asynchronous tasks in the application to increase the performance, as well as the scalability of the application. Due to the fact that coroutines can effectively replace traditional forms of multithreading, a developer is able to cut down the memory overhead considerably as well as increase the system's responsiveness, more so in the scenarios that involve I/O bound interactions and high levels of concurrency. In addition, the developers should be careful of the power consumption trends in their applications by ensuring that the use of sensors in the application is done well, the network calls done are further kept low and the APIs used are of low, power consuming. It is also possible to use self-adaptive designs in order to achieve a favorable relative performance and energy efficiency with respect to the prevailing conditions. Also, assessing the coding standards deployed and profiling of the application performance on different devices from the very beginning are vital in addressing the power issues in development processes.

REFERENCES

[1] K. Chauhan, S. Kumar, D. Sethia, and M. N. Alam, "Performance Analysis of Kotlin Coroutines on Android in a Model-View-Intent Architecture Pattern," 2021 2nd International Conference for Emerging Technology (INCET), Belagavi, India, 2021, pp. 1-5.

[2] D. Beronic´, L. Modric´, B. Mihaljevic´, and A. Radovan, "Comparison of Structured Concurrency Constructs in Java and Kotlin – Virtual Threads and Coroutines," in MIPRO 2022 - 45th Jubilee International Convention, Opatija, Croatia, 2022, pp. 960-965.

[3] R. Rua and J. Saraiva, "A Large-Scale Empirical Study on Mobile Performance: Energy, Run-Time, and Memory," Empirical Software Engineering, vol. 29, no. 31, pp. 1-34, 2024.

[4] S. Huber, T. Lorey, and M. Felderer, "Techniques for Improving the Energy Efficiency of Mobile Apps: A Taxonomy and Systematic Literature Review," arXiv preprint arXiv:2308.08292, 2023. [Online]. Available: https://arxiv.org/abs/2308.08292.

[5] G. A. Almeida, "Concurrency - Kotlin Coroutines – An Android Development Case Study Report," Master's thesis, School of Technology and Management, Polytechnic Institute of Leiria, Portugal, 2019.

[6] D. Furian, S. Azzopardi, Y. Falcone, and G. Schneider, "Runtime Verification of Kotlin Coroutines," in Proceedings of the Conference on Runtime Verification (RV'18), Springer LNCS, vol. 11237, pp. 64-89, 2022.

[7] M. Wu and D. Brumley, "Exploiting Concurrency Vulnerabilities in Web Applications," 2011 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 2011, pp. 251-265.

[8] S. Kim, D. Lee, and M. Song, "Energy-efficient Task Scheduling for Mobile Applications Using Improved Genetic Algorithm," IEEE Access, vol. 7, pp. 12414-12425, 2019.

[9] Y. Kim, "A Performance Comparison of Coroutine-based Programming Models in High-Concurrency Mobile Applications," ACM Mobile Applications Conference (MobApp), 2022.

[10] J. Wang, J. Li, and R. Lee, "Energy-Aware Task Scheduling for Mobile Devices: A Survey," Journal of Computer Science and Technology, vol. 36, pp. 55-69, 2021.

[11] Z. Zhang, Y. Liu, and W. Wu, "Optimizing Energy Consumption in Mobile Applications Using AI-based Task Scheduling," IEEE Mobile Cloud Conference, 2020, pp. 233-241.

[12] D. Y. Sun, "Energy Optimization in High-Concurrency Mobile Applications: A Comprehensive Study," IEEE Transactions on Mobile Com +puting, vol. 22, pp. 40-49, 2023.