Online-Journals.org

iJIM International Journal of Interactive Mobile Technologies

iJIM | eISSN: 1865-7923 | Vol. 19 No. 15 (2025) | OPEN ACCESS

https://doi.org/10.3991/ijim.v19i15.55713

PAPER

Optimizing Memory Usage in Android Smartphones: A Comparative Analysis of Data Structures Across Different Hardware Architectures

Lucia Nugraheni Harnaningrum(✉), **Pius Dian Widi Anggoro**, **Asyahri Hadi Nasyuha**, **Muchamad Mafmudin**, **Akbar Wijaya**

Digital Teknologi Indonesia University, Yogyakarta, Indonesia

ningrum@utdi.ac.id

ABSTRACT

Efficient memory management is a critical factor in enhancing the performance of mobile applications, particularly in resource-constrained environments. This study comprehensively evaluates memory consumption across various data structures on Android smartphones with different hardware architectures, including Snapdragon 732G, Snapdragon 805, and Dimensity 9300. The analysis employs statistical metrics such as standard deviation, minimum, median, and maximum memory usage to assess different data structures' efficiency. Empirical results demonstrate that primitive data structures exhibit significantly lower memory overhead than more complex structures such as *LinkedList* and *ArrayList*, which tend to increase memory fragmentation and garbage collection (GC) overhead. A significant change is the testing of the Primitive Array data structure with API 30 to API 33, which experienced a decrease in memory usage of almost 61%. These findings offer valuable insights for Android developers, enabling them to make informed decisions in selecting optimal data structures to enhance memory efficiency, reduce application latency, and improve overall user experience.

KEYWORDS

memory optimization, Android smartphones, data structures, hardware architectures, mobile performance

1 INTRODUCTION

The rapid evolution of mobile technology has significantly increased the demand for optimized memory management in mobile applications. As smartphones continue to advance in terms of hardware capabilities, the complexity of applications also grows, leading to higher memory consumption. The use of smartphones can also be used for mobile transactions, one of which is by using NFC. The transaction process requires security with security algorithms requiring memory space on

Harnaningrum, L. N., Anggoro, P. D. W., Nasyuha, A. H., Mafmudin, M., Wijaya, A. (2025). Optimizing Memory Usage in Android Smartphones: A Comparative Analysis of Data Structures Across Different Hardware Architectures. *International Journal of Interactive Mobile Technologies (iJIM)*, 19(15), pp. 97–109. https://doi.org/10.3991/ijim.v19i15.55713

Article submitted 2025-04-05. Revision uploaded 2025-06-02. Final acceptance 2025-06-05.

the smartphone [1], [2]. Inefficient memory allocation depletes available RAM and degrades overall system performance, resulting in slow response times, increased power consumption, and frequent application crashes. Consequently, efficient memory utilization has become crucial for developers aiming to enhance user experience and ensure smooth application execution across different hardware configurations.

The rapid evolution of mobile technology has significantly increased the demand for optimized memory management in mobile applications. As smartphones continue to advance in terms of hardware capabilities, the complexity of applications also grows, leading to higher memory consumption. Inefficient memory allocation depletes available RAM and degrades overall system performance, resulting in slow response times, increased power consumption, and frequent application crashes [3], [4]. Consequently, efficient memory utilization has become crucial for developers aiming to enhance user experience and ensure smooth application execution across different hardware configurations.

Memory management in Android smartphones is inherently challenging due to the platform's reliance on garbage collection (GC) and dynamic memory allocation. Unlike iOS, which adopts a more controlled memory allocation mechanism, Android applications operate within a virtual machine environment, such as the Android Runtime (ART), which dynamically manages memory resources [5], [6]. However, excessive reliance on automatic GC can introduce performance bottlenecks, particularly in resource-constrained environments. Poor memory management can lead to high GC overhead, affecting frame rendering times and increasing CPU utilization [7], [8]. Various studies have examined individual techniques for optimizing memory usage in Android applications [9], [10]. Prior research has focused on strategies such as object pooling, lazy loading, and manual memory deallocation to minimize heap fragmentation. However, limited research has been conducted on the comparative analysis of memory consumption across different data structures in diverse hardware architectures. Since data structures form the backbone of application logic, their selection can significantly influence memory footprint, CPU cycles, and cache efficiency [11], [12].

This study aims to bridge this gap by conducting an extensive comparative analysis of memory consumption across different data structures in Android smartphones powered by three distinct hardware architectures: Snapdragon 732G, Snapdragon 805, and Dimensity 9300. By evaluating common data structures such as ArrayList, LinkedList, HashMap, and primitive arrays, we aim to provide quantitative insights into their memory usage patterns in real-world application scenarios. Additionally, we employ statistical measures, including standard deviation, median, and maximum memory usage, to offer a more granular understanding of memory efficiency across different environments. Our findings contribute to optimizing mobile applications by offering evidence-based recommendations for developers [13]. Selecting an appropriate data structure can lead to lower memory fragmentation, reduced GC invocations, and improved runtime efficiency. As mobile applications demand higher computational power, understanding the interplay between data structures and hardware architectures becomes essential for sustaining optimal performance.

2 RELATED WORK

Memory optimization in mobile applications has been extensively studied, with researchers exploring various approaches such as GC improvements, efficient data structure selection, and memory-aware algorithms. Efficient memory management is essential for enhancing mobile application performance [14], [15], [16], particularly in resource-constrained environments where memory leaks and excessive allocations can degrade responsiveness [17]–[19] Investigated the impact of non-functional optimization techniques on mobile applications [20], [21], [22], emphasizing aspects such as energy consumption, memory usage, and CPU efficiency. Their study highlighted the trade-offs between execution speed and memory footprint, demonstrating that certain optimization strategies can significantly improve performance. However, their analysis primarily focused on broad software optimization techniques rather than a granular evaluation of memory consumption across different data structures [23]. The following studies show significant progress in optimizing mobile application performance. Chen et al. developed QLEO, a Q-learning-based system to adaptively conserve Android screen energy [9], presented a comprehensive survey of 156 publications on non-functional optimizations such as energy, memory, and application response time; while [10] introduced SmartSplit, a genetic algorithm that splits CNN processing between the smartphone and the cloud to simultaneously optimize latency, memory, and energy consumption. These three approaches reinforce the case for resource efficiency in smartphones.

A framework for thread-oriented memory management is implemented in mobile edge computing [24], [25]. The approach aimed to reduce memory contention by dynamically allocating memory based on workload distribution [26]. While effective in distributed computing scenarios, their study did not address the specific role of data structures in memory consumption within standalone Android applications [27], [28]. Other research efforts have explored GC enhancements to improve memory efficiency in Android applications and analyzed the impact of different GC algorithms on application performance, demonstrating that frequent memory allocations in complex data structures led to increased GC overhead. Their findings underscore the need to select data structures to minimize unnecessary memory churn carefully.

Conducted a comparative analysis of Java and Kotlin memory management in Android applications [29], [30], showing that improper object retention in heap memory can lead to performance degradation [31], [32]. While their study emphasized language-level optimizations, it did not evaluate how specific data structures perform under different hardware conditions. Despite these advancements, a comprehensive comparative study evaluating the memory efficiency of different data structures across Android hardware architectures remains limited. Most existing research focuses on general optimization strategies or language-based memory improvements, overlooking the hardware-specific memory implications of selecting different data structures. Addressing this gap, our study provides an empirical comparative analysis of memory consumption across fundamental data structures (*ArrayList, LinkedList, HashMap, primitive arrays*) on Android smartphones with diverse hardware configurations (Snapdragon 732G, Snapdragon 805, and Dimensity 9300). By bridging this study gap, we aim to offer practical recommendations for Android developers seeking to optimize memory utilization across different devices.

3 METHODOLOGY

The study evaluates memory consumption across ArrayList, LinkedList, HashMap, and primitive arrays on Android smartphones with Snapdragon 732G, Snapdragon 805, and Dimensity 9300 chipsets. A custom benchmarking app developed in Java and Kotlin, it executes insertion, retrieval, and deletion operations while monitoring heap usage, GC frequency, and allocation efficiency using Android Profiler and Memory Analyzer Tool (MAT). The dataset consists of randomly generated numerical and string values to ensure uniform testing conditions. Statistical analysis, including mean, standard deviation, and ANOVA tests, identifies significant differences in memory consumption across hardware variations. The results provide actionable insights for optimizing memory efficiency in Android applications.

3.1 Experimental setup

In assessing the impact of hardware architecture on memory consumption, this study conducts benchmarking experiments on three Android smartphones with distinct hardware configurations: Snapdragon 732G (8GB RAM, Adreno GPU), Snapdragon 805 (6GB RAM, Adreno GPU), and Dimensity 9300 (12GB RAM, Immortalis GPU). These devices represent different processing capabilities and memory handling mechanisms, allowing for a comprehensive analysis of memory efficiency across varying architectures. A custom benchmarking application developed in Java and Kotlin evaluated memory allocation patterns during everyday data structure operations, including insertion, retrieval, and deletion. The study focuses on the memory footprint of the following data structures:

1. List-based Structures: List, ArrayList, MutableList, LinkedList
2. Primitive Data Structures: Primitive List, Primitive Array, Primitive ArrayList, Primitive MutableList, Primitive LinkedList

Memory profiling was conducted using Android Studio's Profiler tool, capturing real-time memory usage, heap allocation, and GC frequency. Each test was executed under identical conditions, with predefined dataset sizes and controlled background processes to minimize external interference. The recorded memory allocation size metrics, peak memory usage, and standard deviation were analyzed to identify patterns in memory efficiency across different data structures and hardware architectures. The experimental results offer insights into optimizing memory management in Android applications by guiding developers toward more efficient data structure choices based on device specifications.

3.2 Data analysis

We analyzed key statistical measures, including standard deviation, minimum and maximum memory consumption, and median values, to evaluate memory efficiency across data structures and hardware configurations. These metrics help understand memory allocation variability and identify Android applications' most memory-efficient data structures.

1. Standard Deviation: Measures the variance in memory allocation, indicating stability or fluctuation in memory consumption across different hardware architectures.

2. Minimum and Maximum Values: Identify the lowest and highest memory usage, helping to determine data structures that consume excessive memory.
3. Median: Represents the typical memory allocation trend, providing insight into central memory usage tendencies.

Memory profiling was performed on Snapdragon 732G, Snapdragon 805, and Dimensity 9300 chipsets, covering list-based and primitive-based data structures. The results in Table 1 reveal that primitive-based data structures consistently exhibit lower memory consumption compared to more complex structures such as LinkedList and ArrayList. Primitive List and Primitive Array achieve the lowest mean memory usage, while LinkedList and List demonstrate the highest memory overhead.

Table 1. Memory consumption (in KB) across different data structures and hardware architectures

Data Type	Snap 732G	Snap 805	Dim 9300	Mean
List	459.17	462.60	523.1	481.62
ArrayList	458.90	452.10	521	477.33
Mutable List	458.00	451.50	509	472.83
LinkedList	463.11	450.73	526.1	479.98
Primitive List	91.53	84.00	143.6	106.38
Primitive Array	94.00	85.40	171.2	116.87
Primitive Array List	91.95	85.60	142.4	106.65
Primitive Mutable List	90.90	85.10	148.8	108.27
Primitive Linked List	101.40	83.70	149.2	111.43

The statistical analysis in Table 1 highlights notable differences in memory consumption across various hardware architectures, with Dimensity 9300 generally exhibiting higher memory usage, particularly in list-based data structures. Meanwhile, the Snapdragon 805 demonstrates the lowest overall memory allocation, suggesting a more efficient memory management mechanism. To provide a more precise visualization of these findings, Figure 2 presents a comparative analysis of memory consumption across the three hardware configurations, emphasizing how different chipsets handle memory-intensive data structures.

Figure 1 illustrates the differences in memory consumption across Snapdragon 732G, Snapdragon 805, and Dimensity 9300 when executing operations on various data structures. The results reinforce the significance of hardware-aware memory optimization, as different chipsets exhibit varying levels of efficiency in handling list-based and primitive-based data structures.

Building on the comparative memory analysis across different hardware configurations, it is essential to examine the performance of individual data structures in each chipset. While Table 1 and Figure 1 provide an overview of memory consumption trends, Figure 2 presents a more granular visualization, illustrating how each data structure performs across Snapdragon 732G, Snapdragon 805, and Dimensity 9300. The results reinforce the significant impact of hardware architecture on memory allocation efficiency, emphasizing the need for developers to consider chipset-specific optimizations when selecting data structures.

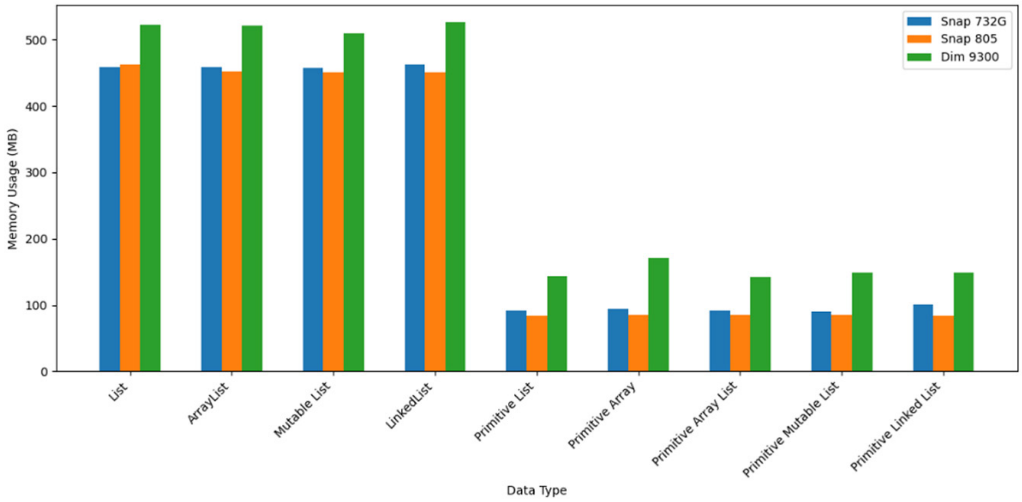

Fig. 1. Comparative memory usage across different smartphone chipsets

Figure 1 compares memory usage for each data structure across Snapdragon 732G, Snapdragon 805, and Dimensity 9300. The results indicate that list-based structures (List, ArrayList, LinkedList) exhibit higher memory consumption, particularly on Dimensity 9300. In contrast, primitive-based structures (Primitive List, Primitive Array, and similar others.) maintain significantly lower memory overhead. These findings highlight the need for efficient data structure selection based on hardware-specific performance characteristics.

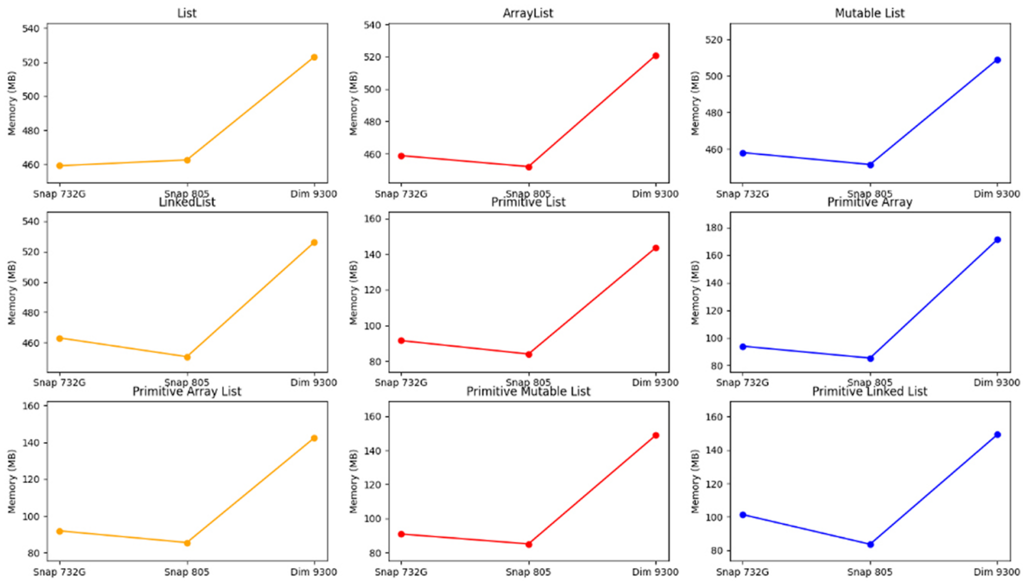

Fig. 2. Memory consumption of different data structures across chipsets

The analysis demonstrates that hardware variations influence memory usage patterns, with the Dimensity 9300 chipset exhibiting the highest memory consumption across all data structures. This suggests that higher RAM availability may lead to increased memory allocation, possibly due to internal memory management optimizations. Conversely, the Snapdragon 805 consistently shows the lowest memory consumption, indicating more efficient memory allocation mechanisms in mid-range devices. Furthermore, primitive-based data structures outperform list-based structures in terms of memory efficiency. Primitive Mutable Lists and Primitive Lists consume the least memory, making them ideal choices for applications requiring optimized memory usage. The results underscore the importance of selecting lightweight data structures to enhance performance and minimize GC overhead in Android applications. These insights provide valuable guidelines for Android developers in choosing the most suitable data structures based on device specifications, optimizing performance while reducing memory overhead. Future work will explore the impact of real-world application scenarios and concurrent execution on memory allocation efficiency.

4 RESULT AND DISCUSSION

The comparative analysis of memory consumption across different data structures and hardware architectures reveals significant variations in memory allocation efficiency. The Dimensity 9300 chipset consistently exhibited the highest memory usage, while the Snapdragon 805 was the most memory-efficient, highlighting the role of hardware architecture in memory management. The results indicate that list-based structures, such as List, ArrayList, and LinkedList, have the highest memory footprint, particularly on Dimensity 9300, suggesting that these structures introduce considerable overhead in memory allocation. Conversely, primitive data structures (Primitive Array, Primitive List, and Primitive Mutable List) demonstrated significantly lower memory consumption, with reduced variance across different hardware configurations. These structures are memory-efficient because they avoid list-based structures' additional object wrapping and reference overhead. Notably, the Mutable List and Primitive Mutable List displayed varying efficiency depending on hardware specifications, reinforcing the hypothesis that hardware architecture influences memory allocation efficiency. The standard deviation of memory usage across these data structures further highlights that more complex structures, such as LinkedList, exhibit unpredictable memory consumption, making them less ideal for applications targeting lower-end devices with limited RAM. Developers should consider hardware-aware memory optimization strategies to ensure stable and efficient performance across diverse smartphone models.

The API version also influences memory allocation patterns, as seen in Table 2, where API 33 generally reduces memory consumption across all data structures compared to API 30. The most notable reduction is observed in Primitive Array, which sees a nearly 61% decrease in memory usage from API 30 to API 33. This indicates that Android's memory management improvements in newer APIs contribute to better memory efficiency, further validating the importance of optimizing data structure selection based on hardware and software constraints.

Table 2. Memory consumption (in KB) across android API versions

Data Type	API 30	API 33
List	721.598,02	327.470,37
Array List	721.519,09	474.390,21
Mutable List	360.483,00	271.162,04
Linked List	267.713,00	259.963,73
Primitive Array	41.986,24	16.273,62

(Continued)

Table 2. Memory consumption (in KB) across android API versions *(Continued)*

Data Type	API 30	API 33
Primitive List	79.943,14	23.500,97
Primitive Mutable List	25.650,49	31.732,63
Primitive Array List	53.174,11	42.531,88
Primitive Link Lits	69.483,67	34.096,00

The analysis in Table 2 demonstrates a significant reduction in memory consumption across most data structures when transitioning from API 30 to API 33, particularly in primitive-based structures such as Primitive Array and Primitive List. This suggests that Android's memory management improvements in newer API versions effectively optimize memory allocation and reduce overall heap usage. Figure 3 visually compares memory performance between API 30 and API 33 to illustrate these trends further, highlighting the extent of memory efficiency gains across different data structures.

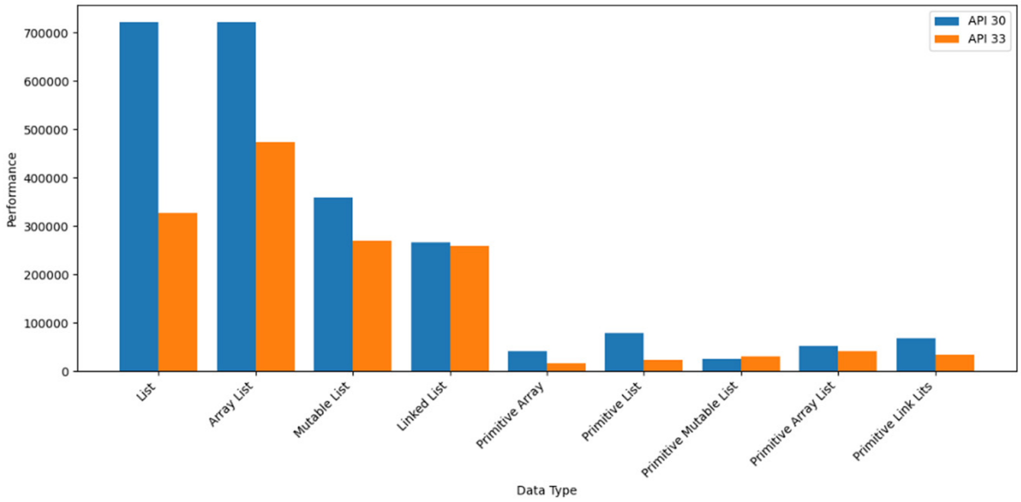

Fig. 3. Comparison of memory performance between API 30 and API 33

These findings provide practical insights for Android developers, demonstrating that selecting primitive-based data structures can significantly enhance memory efficiency, particularly on lower-end devices and newer Android versions. Future research can further explore real-world application performance trade-offs, integrating multi-threading scenarios and complex UI interactions to validate the impact of memory optimization strategies in dynamic execution environments.

4.1 Implications for mobile development

The findings of this study have significant implications for Android application development, particularly in optimizing memory efficiency across different hardware architectures. As mobile applications demand computational power, developers must adopt memory-conscious programming strategies to enhance performance, minimize GC overhead, and ensure a smoother user experience across various device configurations. Based on our analysis, we highlight the following key takeaways:

1. Utilizing Primitive Data Structures for Memory Efficiency
   Our results demonstrate that primitive-based data structures (e.g., Primitive Array, Primitive List, and Primitive Mutable List) consume less memory than object-based structures such as List, ArrayList, and LinkedList. It is particularly beneficial for applications running on low-end and mid-range devices, where RAM limitations often lead to performance bottlenecks. Developers should prioritize primitive structures whenever possible to reduce memory fragmentation, lower heap usage, and enhance GC efficiency.
2. Device-Specific Optimization is Crucial
   Memory consumption behaviors vary across different hardware architectures, indicating that a one-size-fits-all approach to memory management may not be optimal. For example, Snapdragon 805 consistently exhibited lower memory usage across all tested data structures, while Dimensity 9300 showed significantly higher memory allocation. Developers should implement hardware-aware optimizations, such as adaptive data structure selection based on device specifications. Profiling tools should be used early in development to tailor memory management strategies according to the target hardware.
3. Combining GC Strategies with Data Structure Optimization
   The study reinforces the need to balance GC mechanisms with efficient data structure selection. Frequent object allocations in list-based structures trigger higher GC activity, increasing CPU overhead. By choosing lighter data structures, developers can reduce the need for excessive GC cycles. Manual memory management techniques, such as object pooling and lazy loading, can improve memory stability, particularly in memory-intensive applications like gaming, multimedia processing, and real-time data streaming.
4. Hybrid Approaches for Dynamic Memory Management
   A hybrid memory optimization strategy could improve adaptability by dynamically switching between primitive and object-based data structures based on runtime conditions. For example, applications running on devices with limited RAM could default to primitive structures, whereas high-end devices could utilize LinkedList and ArrayList in dynamic resizing scenarios. Implementing such adaptive strategies can optimize memory usage and processing speed, enhancing overall app efficiency across different hardware environments.
5. Leveraging Android API Enhancements for Memory Optimization
   Our comparative analysis across API 30 and API 33 indicates that newer Android versions introduce more efficient memory management mechanisms, particularly for primitive-based data structures. Developers should take advantage of Android's evolving memory management features, ensuring that applications are optimized for the latest API versions while maintaining backward compatibility. It can significantly reduce memory overhead and enhance app stability across various devices.

The key takeaways from this study highlight essential strategies for memory-efficient Android development, emphasizing the impact of data structure selection, hardware-specific optimization, GC strategies, hybrid memory management, and leveraging Android API improvements. These factors collectively enhance application performance, minimize memory overhead, and improve user experience. To summarize the implications of these findings, Figure 4 visualizes the relative importance of each optimization approach in Android memory management.

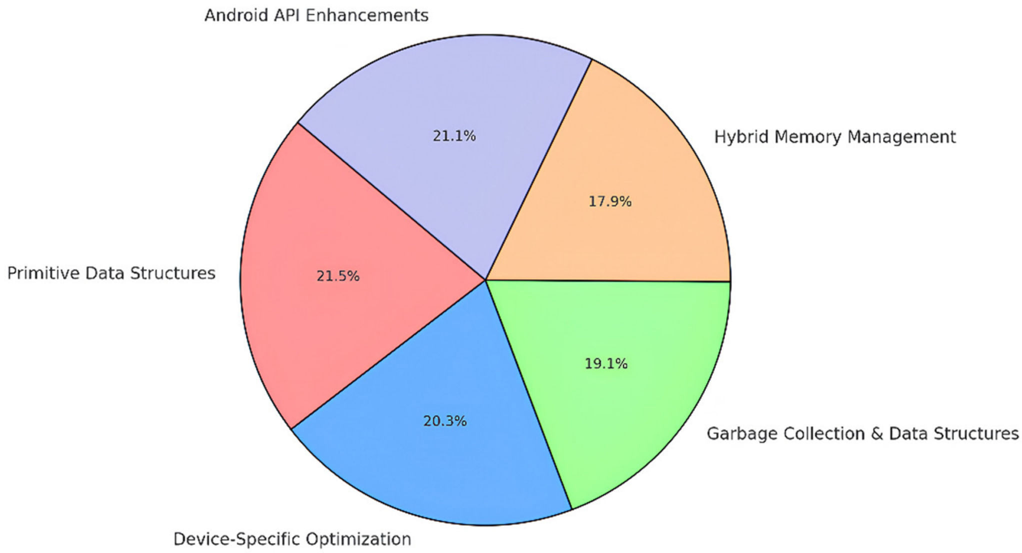

Fig. 4. Key implications for android memory optimization

Figure 4 represents the critical factors influencing memory optimization in Android development. The findings emphasize the importance of selecting primitive-based data structures, tailoring optimizations to specific hardware, effectively managing garbage collection, implementing hybrid memory strategies, and utilizing newer Android API enhancements for improved efficiency.

5 CONCLUSION

This study provides empirical evidence that selecting the appropriate data structure is crucial in optimizing memory consumption in Android applications. Through extensive comparative analysis across different hardware architectures (Snapdragon 732G, Snapdragon 805, and Dimensity 9300) and Android API versions, we demonstrated that primitive-based data structures significantly outperform traditional list-based structures in terms of memory efficiency. The findings highlight that hardware-specific memory management strategies are essential, as memory behavior varies across chipsets and Android versions. The results indicate that LinkedList, List, and ArrayList consistently exhibit higher memory overhead. At the same time, Primitive List, Primitive Array, and Primitive Mutable List offer significantly lower memory usage, making them ideal choices for memory-efficient application development. Moreover, the impact of hardware architecture on memory allocation efficiency underscores the importance of device-aware optimizations, as memory-intensive structures that perform well on one chipset may not necessarily be efficient on another. Additionally, the study finds that newer Android API versions introduce improvements in memory management, further validating the need for developers to leverage API-level optimizations while maintaining backward compatibility.

Future research can explore the impact of modern programming paradigms, such as Jetpack Compose versus traditional Views, on memory efficiency, providing insights for developers transitioning to declarative UI frameworks. Additionally, investigating adaptive GC strategies, including real-time profiling and just-in-time (JIT) memory adjustments, could further enhance memory management.

Another promising direction is analyzing the energy efficiency implications of memory-optimized applications, as reducing memory overhead can minimize CPU usage and battery drain. Moreover, studying the behavior of data structures under multi-threading and concurrent execution will help developers optimize applications requiring high-performance memory management. Finally, future research can examine memory consumption in resource-intensive applications such as mobile gaming, augmented reality, and AI-driven applications to determine the best memory-efficient structures for high-load scenarios.

6 REFERENCES

[1] H. Altuwaijri and S. Ghouzali, "Android data storage security: A review," *Journal of King Saud University – Computer and Information Science*, vol. 32, no. 5, pp. 543–552, 2020. https://doi.org/10.1016/j.jksuci.2018.07.004

[2] M. Hort, M. Kechagia, F. Sarro, and M. Harman, "A survey of performance optimization for mobile applications," *IEEE Transactions Software Engineering*, vol. 48, no. 8, pp. 2879–2904, 2022. https://doi.org/10.1109/TSE.2021.3071193

[3] S. Lee and H. Bahn, "Characterization of android memory references and implication to hybrid memory management," *IEEE Access*, vol. 9, pp. 60997–61009, 2021. https://doi.org/10.1109/ACCESS.2021.3074179

[4] A. Degu, "Android application memory and energy performance: Systematic literature review," *IOSR J. Comp. Eng.*, vol. 21, no. 3, pp. 20–32, 2019. https://doi.org/10.9790/0661-2103052032

[5] J. Simão, S. Esteves, A. Pires, and L. Veiga, "GC-Wise: A Self-adaptive approach for memory-performance efficiency in Java VMs," *Future Generation Computer Systems*, vol. 100, pp. 674–688, 2019. https://doi.org/10.1016/j.future.2019.05.027

[6] D. M. Adinew, Z. Shijie, and Y. Liao, "Spark performance optimization analysis in memory tuning on GC overhead for big data analytics," in *Proceedings of the 2019 8th International Conference on Networks, Communication and Computing*, 2020, pp. 75–78. https://doi.org/10.1145/3375998.3376039

[7] A. Pupykina and G. Agosta, "Survey of memory management techniques for HPC and cloud computing," *IEEE Access*, vol. 7, pp. 167351–167373, 2019. https://doi.org/10.1109/ACCESS.2019.2954169

[8] W. Weng, A. Uta, and J. S. Rellermeyer, "Brug: An adaptive memory (re-)allocator," in *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, Philadelphia, PA, USA, 2024, pp. 67–76. https://doi.org/10.1109/CCGrid59990.2024.00017

[9] Z. Chen, T. Wang, K. Zhang, Y. Guo, and Z. Shao, "A Q-Learning-based display energy optimization scheme for android systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits Systems*, vol. 44, no. 4, pp. 1262–1275, 2025. https://doi.org/10.1109/TCAD.2024.3486244

[10] I. Prakash, A. Bansal, R. Verma, and R. Shorey, "SmartSplit: Latency-energy-memory optimisation for CNN splitting on smartphone environment," in *2022 14th International Conference on COMmunication Systems and NETworkS (COMSNETS)*, Bangalore, India, 2022, pp. 549–557. https://doi.org/10.1109/COMSNETS53615.2022.9668610

[11] L. Chen *et al.*, "Unified holistic memory management supporting multiple big data processing frameworks over hybrid memories," *ACM Transsction on Computure Systems (TOCS)*, vol. 39, no. 2, pp. 1–38, 2022. https://doi.org/10.1145/3511211

[12] A. Singhvi *et al.*, "CliqueMap: Productionizing an RMA-based distributed caching system," in *SIGCOMM 2021 – Proc. ACM SIGCOMM 2021 Conference*, 2021, pp. 93–105. https://doi.org/10.1145/3452296.3472934

[13] J. Wang, "Optimizing learning path design in mobile learning platforms for online courses," *International Journal of Interactive Mobile Technologies (iJIM),* vol. 19, no. 1, pp. 4–17, 2025. https://doi.org/10.3991/ijim.v19i01.53407

[14] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Int. Conf. Archit. Support Program. Lang. Oper. Syst. – ASPLOS*, 2019, pp. 331–345. https://doi.org/10.1145/3297858.3304024

[15] A. Abu Taleb, Q. Abu Al-Haija, and A. Odeh, "Performance analysis of sink mobility models for wireless sensor networks: A comparative study," *International Journal of Interactive Mobile Technologies*, vol. 17, no. 18, pp. 129–142, 2023. https://doi.org/10.3991/ijim.v17i18.42121

[16] D. Kondratyuk *et al.*, "MoViNets : Mobile video networks for efficient video recognition," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Kuala Lumpur, Malaysia, 2021, pp. 16020–16030.

[17] D. Amalfitano, V. Riccio, P. Tramontana, and A. R. Fasolino, "Do memories haunt you? An automated black box testing approach for detecting memory leaks in android apps," *IEEE Access*, vol. 8, pp. 12217–12231, 2020. https://doi.org/10.1109/ACCESS.2020.2966522

[18] C. Lou *et al.*, "RESIN: A holistic service for dealing with memory leaks in production cloud infrastructure," in *Proc. 16th USENIX Symp. Oper. Syst. Des. Implementation, OSDI 2022*, 2022, pp. 109–125.

[19] J. Vilk and E. D. Berger, "BLeak: Automatically debugging memory leaks in web applications," *Communication of the ACM*, vol. 63, no. 11, pp. 146–153, 2020. https://doi.org/10.1145/3422598

[20] A. E. Yahya, A. Gharbi, W. M. S. Yafooz, and A. Al-Dhaqm, "A novel hybrid deep learning model for detecting and classifying non-functional requirements of mobile apps issues," *Electronics*, vol. 12, no. 5, p. 1258, 2023. https://doi.org/10.3390/electronics12051258

[21] Y. Tian, "The impact of mobile interactive technology on international chinese learning: A case study of AI-driven applications," *International Journal of Interactive Mobile Technologies*, vol. 19, no. 8, pp. 125–139, 2025. https://doi.org/10.3991/ijim.v19i08.55337

[22] A. Ahmad, C. Feng, K. Li, S. M. Asim, and T. Sun, "Toward empirically investigating non-functional requirements of iOS developers on stack overflow," *IEEE Access*, vol. 7, no. 1, pp. 61145–61169, 2019. https://doi.org/10.1109/ACCESS.2019.2914429

[23] B. Li, Q. Zhao, S. Jiao, and X. Liu, "DroidPerf: Profiling memory objects on android devices," in *Proc. Annu. Int. Conf. Mob. Comput. Networking, MOBICOM*, 2023, pp. 75–89. https://doi.org/10.1145/3570361.3592503

[24] N. Naveen Kumar, S. Saravana, S. Balamurugan, P. Seshu Kumar, and S. Suresh, "Optimized memory allocation in edge-PLCs using Deep Q-Networks and bidirectional LSTM with quantum genetic algorithm," *e-Prime – Advances Electrical Engineering, Electronics and Energy*, vol. 10, p. 100762, 2024. https://doi.org/10.1016/j.prime.2024.100762

[25] M. Singh, P. K. Shrivastava, A. Al Hashmi, A. Gupta, and A. Ansari, "Reduce device memory using centralized common resource pool," in *2020 10th International Conference on Cloud Computing, Data Science and Engineering (Conference),* Noida, India, 2020, pp. 565–569. https://doi.org/10.1109/Confluence47617.2020.9058226

[26] S. Sudhakaran, A. Ali-Gombe, A. Orgah, A. Case, and G. G. Richard, "Ampledroid recovering large object files from android application memory," *2020 IEEE International Workshop on Information Forensics and Security (WIFS)*, New York, NY, USA, 2020, pp. 1–6. https://doi.org/10.1109/WIFS49906.2020.9360906

[27] H. S. Musa, M. Krichen, A. A. Altun, and M. Ammi, "Survey on blockchain-based data storage security for android mobile applications," *Sensors*, vol. 23, no. 21, p. 8749, 2023. https://doi.org/10.3390/s23218749

[28] A. Ali-Gombe, S. Sudhakaran, A. Case, and G. G. Richard, "DroidScraper: A tool for Android in-memory object recovery and reconstruction," in *RAID 2019 Proc. – 22nd Int. Symp. Res. Attacks, Intrusions Defenses*, 2019, pp. 547–559.

[29] J. S. Espitia Acero, "Empirical testing for establishing benchmarks : Process review and comparison between Java, Kotlin and dart's performance," Universidad los Andes, 2020.

[30] O. Hamdi, A. Ouni, E. A. Alomar, M. O. Cinneide, and M. W. Mkaouer, "An empirical study on the impact of refactoring on quality metrics in android applications," in *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*, Madrid, Spain, 2021, pp. 28–39. https://doi.org/10.1109/MobileSoft52590.2021.00010

[31] M. Peters, G. L. Scoccia, and I. Malavolta, "How does migrating to kotlin impact the run-time efficiency of android apps?" in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Luxembourg, 2021, pp. 36–46. https://doi.org/10.1109/SCAM52516.2021.00014

[32] K. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, "Top of the heap: Efficient memory error protection of safe heap objects," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, vol. 1, no. 5, 2024, pp. 1330–1344. https://doi.org/10.1145/3658644.3690310

7 AUTHORS

Lucia Nugraheni Harnaningrum is with the Digital Teknologi Indonesia University, Yogyakarta, Indonesia (E-mail: ningrum@utdi.ac.id).

Pius Dian Widi Anggoro is with the Digital Teknologi Indonesia University, Yogyakarta, Indonesia.

Asyahri Hadi Nasyuha is with the Digital Teknologi Indonesia University, Yogyakarta, Indonesia.

Muchamad Mafmudin is with the Digital Teknologi Indonesia University, Yogyakarta, Indonesia.

Akbar Wijaya is with the Digital Teknologi Indonesia University, Yogyakarta, Indonesia.