PhantomSwap: A Hybrid Memory Swapping System for Mobile Devices

Cheng-Yu Hu
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
r12921a09@ntu.edu.tw

Sheng-De Wang
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan
sdwang@ntu.edu.tw

***Abstract*—Mobile devices face a persistent conflict between the growing memory demands of sophisticated applications and their limited physical DRAM. This conflict leads to aggressive process terminations by the Low Memory Killer (LMK), which degrades the multitasking user experience. While prior object-level swapping frameworks like Marvin and Fleet have sought to address this by optimizing for single performance vectors, such as reclamation speed or hot-launch latency, they might exhibit performance cliffs under complex, unpredictable workloads.**

This paper introduces PhantomSwap, a hybrid memory swapping system designed to deliver a more holistic improvement in system fluidity and resilience. Its core innovation is a reactive, three-tiered memory hierarchy (DRAM, ZRAM, Flash) managed by an intelligent Unified Swap Controller using a lightweight Object Aging Algorithm. This architecture is deeply co-designed with the Android Runtime's Garbage Collector, employing a Bookmark-style mechanism to ensure memory correctness without expensive fault-ins.

Our evaluation demonstrates the effectiveness of this approach. Experimental results show that PhantomSwap significantly increases application cache capacity, supporting up to 19 concurrent commercial applications compared to 15 in the Android baseline. Critically, while maintaining highly competitive median hot-launch performance, PhantomSwap exhibits superior tail latency, reducing the 95th-percentile launch time by up to 46% against state-of-the-art frameworks in complex gaming workloads. The ZRAM tier acts as a crucial buffer, "softening" the performance penalty of memory misses and preventing the severe stalls inherent in two-tiered systems.

***Index Terms*—Memory Management, Mobile Devices, Operating Systems**

I. Introduction

In today's digital era, mobile devices have become indispensable. This ubiquity stems from a rich ecosystem of sophisticated applications, which have become increasingly memory-intensive. Many popular social media, gaming, and productivity apps each consume several hundred megabytes of RAM, with some exceeding a gigabyte [1]. This trend places significant strain on the limited DRAM resources of mobile platforms.

The disparity between these soaring app memory demands and fixed physical memory often leads to frustrating user experiences [2]. For example, a user navigating with a map app might switch to reply to a message, only to find upon return that the map app was terminated by Android's Low Memory Killer (LMK) [3] and must be slowly reloaded. Similarly, gamers may find their game state lost after pausing to check an online guide. These issues stem from the LMK aggressively terminating background processes to free memory.

To address this challenge, prior object-level swapping frameworks like Marvin [4] and Fleet [5] have been proposed. While the object-level paradigm offers significant advantages over traditional page-based swapping, as summarized in Table I, existing frameworks tend to focus on a single performance vector, such as reclamation speed or hot-launch latency, and can exhibit performance cliffs under complex, unpredictable workloads.

TABLE I: Comparison of Swapping Approaches

Approach	Granularity	Strength	Weakness
Traditional	Page-level	Simplicity & OS-level transparency	Not object-aware, causes I/O amplification
Object-level	Object-level	Fine-grained control & runtime-awareness	High complexity & runtime overhead

In light of these limitations, we introduce PhantomSwap, a novel framework that adopts a *reactive philosophy* to deliver a more *holistic improvement* in system fluidity. Rather than attempting to predict future accesses, PhantomSwap reacts to observed access patterns. Its core innovation is a *three-tier memory hierarchy* (DRAM, ZRAM, Flash) [6] that extends prior two-tier models by adding a fast, in-memory compression tier, which provides for more nuanced placement decisions.

The key contributions of this work are as follows:

- **A Reactive Three-Tiered Architecture:** A system architecture built around a reactive philosophy and a three-tier memory hierarchy that optimizes for overall system fluidity and ensures graceful performance degradation.
- **Increased Application Cache Capacity:** By leveraging the intermediate ZRAM tier, PhantomSwap significantly increases the number of applications the system can keep readily available compared to two-tier systems.
- **Efficient Object Aging and GC Integration:** We developed an "Object Aging Algorithm" and integrated a robust bookmarking mechanism into the garbage collector

to maintain memory correctness without incurring expensive fault-ins.

II. Related Work

A fundamental limitation of traditional page-based swapping in a managed runtime such as Android is the conflict with Garbage Collector (GC) [7]. A standard GC cycle must traverse the entire live object graph, inevitably touching objects on swapped-out pages. This triggers a "swap-in storm": a cascade of page faults that forces pages back into DRAM merely to satisfy the GC's traversal, not for application use. This phenomenon severely undermines the effectiveness of swapping and can paradoxically increase memory pressure, accelerating LMK interventions [5]. Mitigating this GC-swap conflict is a primary motivation for the object-aware frameworks discussed next.

Recognizing this problem, researchers have proposed object-level swapping frameworks. Marvin [4] proactively checkpoints infrequently accessed ("cold") objects to flash storage before memory becomes critical. This predictive, two-tier (DRAM/Flash) approach enables rapid memory reclamation. In contrast, Fleet [5] focuses on improving app launch latency by differentiating between foreground and background object allocations, preferentially swapping out background objects in a reactive, two-tier manner. While effective, both frameworks optimize for a single performance vector and can exhibit performance cliffs, which motivates our holistic three-tiered approach.

III. PhantomSwap Design

A. Overview

PhantomSwap adopts a modular, three-layered architecture, as illustrated in Figure 1. This design clearly separates the system's concerns:

- **The Policy Layer** (the "brain"): Makes high-level strategic decisions about which objects to swap and when.
- **The Execution Layer** (the "muscle"): Performs the low-level swapping operations, interacting with the kernel and the Java heap.
- **The Coordination Layer** (the "nervous system"): Ensures correctness and robustness by integrating with the GC and managing concurrency.

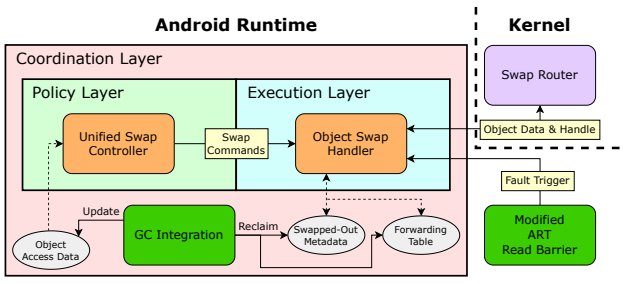


Fig. 1: The architecture of the PhantomSwap framework.

B. The Policy Layer: Unified Swap Controller

The Policy Layer is driven by the Unified Swap Controller, which employs a nuanced temperature-based model. To track object temperature, we developed the **Object Aging Algorithm**, a lightweight, lock-free mechanism whose workflow is shown in Figure 2. It uses thread-local buffers to record object accesses without introducing contention. During a GC safepoint, a dedicated "Aging Pass" processes these records: an object's age counter is reset to zero if recently accessed, or incremented otherwise.

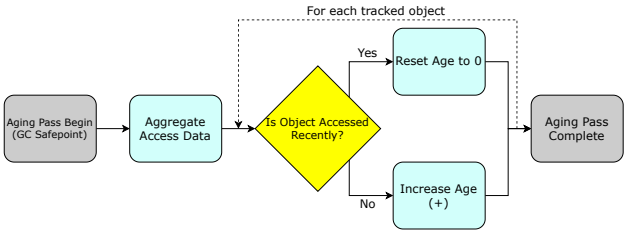


Fig. 2: The workflow of the Object Aging Algorithm.

This age score directly informs our **Multi-Tiered Swapping Policy**, illustrated in Figure 3. The controller uses configurable age thresholds to map an object to the appropriate memory tier:

- **Hot Objects** (low age) are retained in DRAM.
- **Warm Objects** (medium age) are compressed and migrated to the ZRAM tier.
- **Cold Objects** (high age) are evicted to Flash storage.

The controller also dynamically manages ZRAM capacity, migrating the coldest objects from ZRAM to Flash when pressure is high.

To ensure system stability and efficiency, the controller performs two additional functions. First, it applies **candidate filtering** to exclude certain objects from swapping, such as VM-internal objects, objects with native resources, and those smaller than a 2KB threshold. Second, it provides **dynamic resource management** by continuously monitoring ZRAM usage. When ZRAM occupancy exceeds a high-water mark, the controller proactively migrates the coldest objects from ZRAM to Flash, ensuring that space remains available for newly identified warm objects.

C. The Execution Layer: Swapping and Fault-Handling

The Execution Layer performs the low-level mechanics of moving objects between memory tiers. The swap-out process, initiated by the `Object Swap Handler`, replaces a target object on the heap with a minimal, content-free placeholder, termed the *Phantom Stub*. It then registers the object's state and metadata, including a CRC32 [8] checksum for integrity, in the `g_swapped_info` map. The fault-in process is subsequently triggered when the *Modified ART Read Barrier* intercepts an application's attempt to access a `Phantom Stub`.

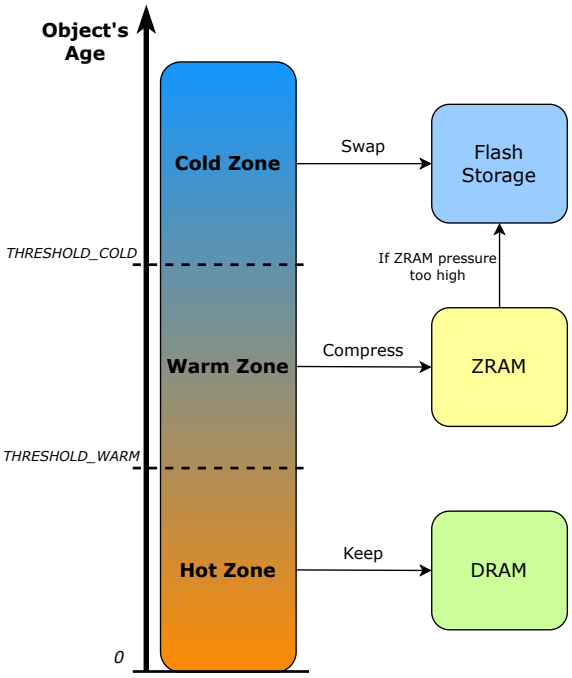


Fig. 3: The Multi-Tiered Swapping Policy dictates object placement based on age.

Upon a fault, the **Relocatable Swap-In** mechanism, illustrated in Figure 4, is invoked. This process is "relocatable" because it restores the object to a **new** memory location, a critical feature for enabling true memory reclamation. A forwarding pointer is installed in the `g_forwarding_map` to redirect subsequent accesses. The object's `SwappedInfo` entry then acts as a temporary "tombstone" to guarantee consistency. Data integrity is ensured by re-calculating and verifying the CRC32 checksum during restoration.

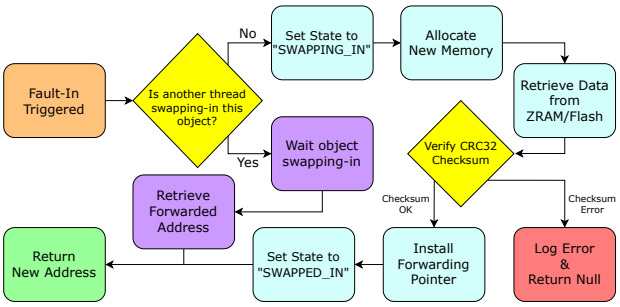


Fig. 4: The workflow of the Relocatable Swap-In mechanism, detailing the synchronization and restoration process.

D. The Coordination Layer: GC Integration

The Coordination Layer's primary role is to ensure system-wide correctness and robustness. Its most critical function is to solve the GC-swap conflict. To prevent the phenomenon of "swap-in storm", PhantomSwap implements a **bookmark-style GC integration**. When the GC's traversal encounters a `Phantom Stub`, it retrieves the object's pre-saved list of references from its metadata entry in `g_swapped_info` and pushes them onto the mark stack. This approach preserves object reachability without incurring expensive fault-ins and allows the GC to correctly reclaim the storage of any swapped-out objects that have become garbage.

Furthermore, this layer guarantees robustness through well-defined protocols. **Concurrency control** for fault-in operations is managed by a fine-grained, per-object locking mechanism utilizing a mutex and a condition variable. To prevent race conditions during the swap-out phase, all such operations are performed **atomically** at an ART safepoint, where application threads are safely paused.

E. Implementation Overview

Our implementation follows a modular design, separating core logic into new self-contained modules and integrating them into the system via targeted hooks. This strategy, which ensures high cohesion and low coupling, is summarized in Table II.

TABLE II: Overview of PhantomSwap Implementation

Category	Component	Implementation Summary
New Self-Contained Modules	Unified Swap Controller	New C++ module to implement the aging algorithm and make swap policy decisions.
	Object Swap Handler	New C++ module to execute I/O operations and manage object forwarding.
	Kernel Swap Router	New kernel driver to process 'ioctl' calls from ART for storage operations.
Targeted System Modifications	ART Read Barrier	Hooked to intercept memory reads and trigger the fault-in process.
	ART Garbage Collector	Hooked into the GC lifecycle for object aging, bookmark-style marking, and cleanup.

IV. Evaluation

A. Experimental Setup

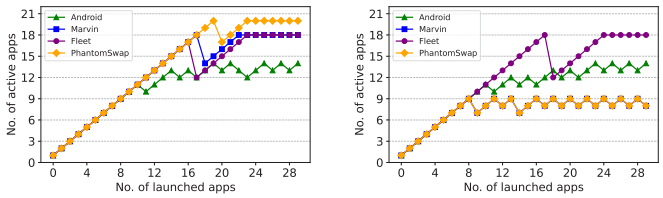
Our evaluation was conducted on a Google Pixel 3 with 4GB of RAM, running a modified Android 10 build. This environment was chosen to ensure a direct and rigorous comparison with the state-of-the-art Fleet framework [5]. We compare PhantomSwap against three baselines: the stock Android 10 with ZRAM, and our re-implementations of Marvin [4] and Fleet [5]. Our aging thresholds were set to `THRESHOLD_WARM=10` and `THRESHOLD_COLD=30` based on a preliminary sensitivity analysis (detailed in subsubsection IV-B5). We used both synthetic micro-benchmarks and a macro-benchmark suite of 20 popular commercial applications, listed in Table III, to evaluate the system.

TABLE III: The Commercial Applications for Macro-benchmarks

App Type	App Description
Communication	Twitter (X), Facebook, Instagram, Telegram, Line
Multimedia	YouTube, TikTok, Spotify, Twitch, Rave, BigoLive
Tools & Utilities	Amazon Shopping, Google Maps, Chrome, Firefox, LinkedIn, Gmail
Games	Angry Birds Classic, Candy Crush Saga, PUBG Mobile

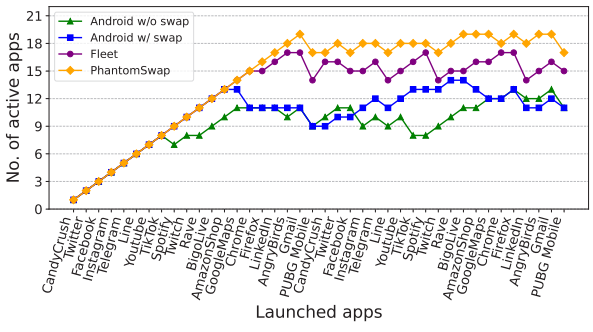
B. Results

1) Application Cache Capacity: We first measure the Application Cache Capacity, the number of apps that can remain active in the background. The detailed results across various workloads are presented in Figure 5.



(a) Large object (2048B) apps

(b) Small object (512B) apps



(c) Commercial apps

Fig. 5: Application Cache Capacity results across different workloads.

The synthetic large object workload (Figure 5a) shows that PhantomSwap achieves the most stable capacity, while the small object test (Figure 5b) validates our design trade-off of ignoring small objects to avoid excessive overhead. Most critically, in realistic commercial application workloads (Figure 5c), PhantomSwap demonstrates clear superiority.

The summary of maximum cache capacity is presented in Figure 6. In the extended commercial test, PhantomSwap supports up to 19 concurrent apps, a 26% increase over the stock Android system (15 apps) and also surpasses Fleet (17 apps). This advantage arises from our intermediate ZRAM tier, which efficiently handles the "warm" objects typical in real-world multitasking scenarios.

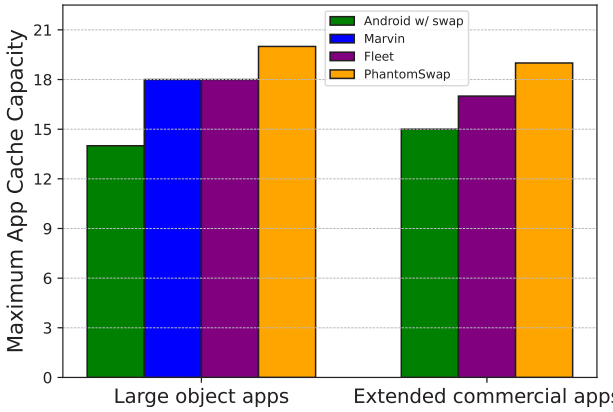
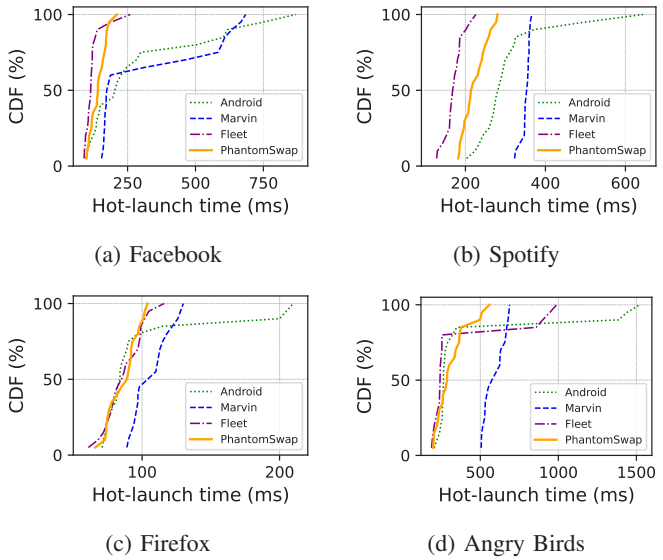


Fig. 6: Maximum application cache capacity across key workloads.

2) Application Hot-Launch Performance: Next, we evaluate hot-launch latency. The full performance distribution, presented as a Cumulative Distribution Function (CDF) in Figure 7, reveals the nuanced behavior between the frameworks. For applications with simple and highly predictable hot-launch working sets, such as *Firefox*, both PhantomSwap and Fleet deliver exceptional, near-identical performance, with their CDF curves being extremely steep and closely aligned. However, the difference in design philosophy becomes most apparent in more complex, hard-to-predict workloads like *Facebook* and Angry *Birds*. In these cases, all frameworks exhibit a performance tail, but PhantomSwap's degradation is substantially more graceful. This indicates that while some critical objects are inevitably swapped out under heavy memory pressure, PhantomSwap manages these scenarios more effectively.

To quantify the worst-case user experience, we focus on the 95th-percentile (P95) tail latency, summarized in Figure 8. The chart highlights that PhantomSwap delivers superior tail latency across all applications. In the complex *Angry Birds* workload, PhantomSwap reduces P95 latency by 46% compared to Fleet (512ms vs. 948ms). This demonstrates the effectiveness of the ZRAM tier as a "performance safety net" that softens the penalty of memory misses when a policy decision is imperfect.



(a) Facebook (b) Spotify

(c) Firefox (d) Angry Birds

Fig. 7: CDF of hot-launch times for four representative commercial applications under high memory pressure. A curve further to the left indicates better performance.

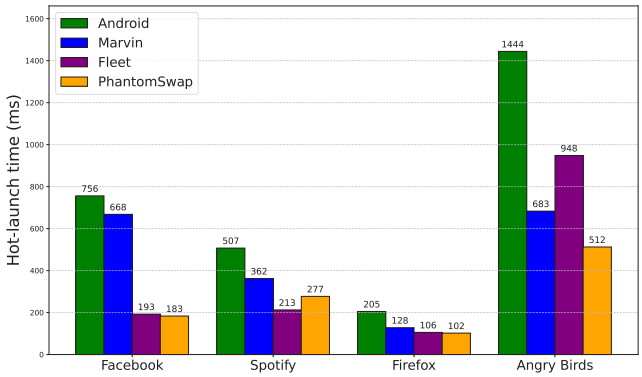
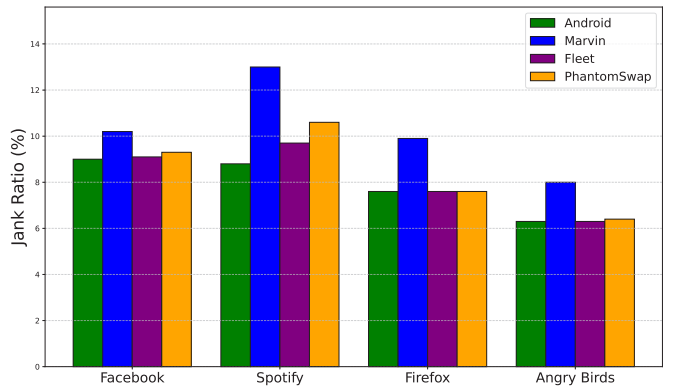


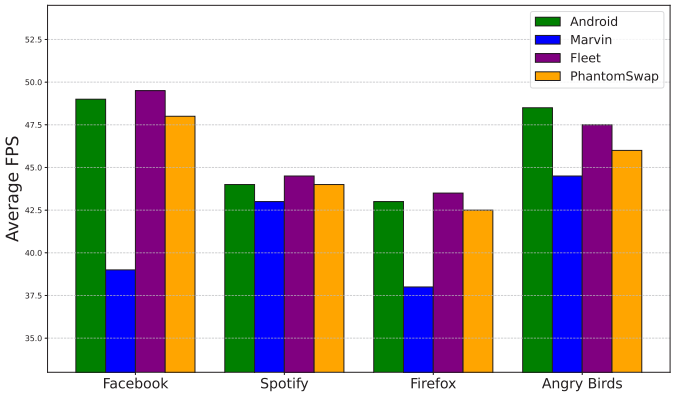
Fig. 8: Tail (P95) hot-launch latency comparison, indicating worst-case performance.

3) Runtime Fluidity: To ensure our background mechanisms do not degrade the foreground user experience, For measuring runtime fluidity, we used the Perfetto [9] tool to trace Jank Ratio [10] (lower is better) and average Frames Per Second (FPS, higher is better). As shown in Figure 9, PhantomSwap's impact on runtime fluidity is minimal. Compared to the Android baseline and Fleet, PhantomSwap shows a slight increase in Jank Ratio of about 1% and a decrease in average FPS of 2-4 frames across most applications. However, its performance in both metrics is consistently better than Marvin's. This critical result indicates that the substantial gains in application cache capacity are achieved with only a marginal and acceptable overhead on the interactive performance of the foreground application.

4) System Overheads: Finally, we measured the system costs. PhantomSwap introduces a modest CPU overhead of approximately 1.27% compared to the Android baseline and



(a) Jank Ratio



(b) Average FPS

Fig. 9: Runtime fluidity comparison across representative applications. The left chart shows the Jank Ratio, and the right chart shows the average FPS.

requires 65-100 MB of DRAM for its metadata. We conclude that these costs represent a favorable trade-off for the significant gains in system capacity and resilience.

5) Parameter Sensitivity Analysis: The aging thresholds used in our experiments, `THRESHOLD_WARM=10` and `THRESHOLD_COLD=30`, were determined through a preliminary sensitivity analysis. Our tuning revealed a clear trade-off: setting `THRESHOLD_WARM` too low (e.g., < 5) resulted in aggressive swapping of temporarily idle objects, harming hot-launch latency, while setting it too high (e.g., > 20) was too conservative, reducing cache capacity. Similarly, a `THRESHOLD_COLD` value set too close to `THRESHOLD_WARM` (e.g., 15) caused premature eviction to Flash, negating the benefit of the ZRAM buffer. Therefore, the chosen values represent a balanced trade-off for our target device.

V. Discussion

Our experimental evaluation confirms that PhantomSwap achieves its primary goal of enhancing system resilience. The framework substantially increases application cache capacity and, most critically, demonstrates superior tail latency on complex workloads, all while maintaining system fluidity

and incurring only modest overhead. This confirms that our approach provides a more holistic performance improvement compared to prior work, which often optimizes for a single performance vector at the expense of others.

The key to PhantomSwap's success lies in its three-tiered memory hierarchy. The intermediate ZRAM layer acts as a crucial "performance safety net," a feature absent in two-tiered (DRAM/Flash) systems like Marvin and Fleet. When a policy inevitably makes a mistake (e.g., misclassifying a required object), two-tiered systems pay a severe penalty due to high-latency Flash I/O. PhantomSwap's ZRAM tier effectively "softens" this penalty, servicing the vast majority of such memory misses with fast microsecond-level decompression. This design provides graceful performance degradation and is the core reason for the superior system resilience we observed.

Although our results are promising, we acknowledge the following limitations, which also point to valuable directions for future work.

- **Evaluation Environment:** Our evaluation was carried out on an older hardware and software platform (a Google Pixel 3 running Android 10) to ensure a rigorous comparison with the state-of-the-art Fleet framework. Future work is needed to validate our findings on modern devices with different resource constraints.
- **Workload Selection:** The suite of commercial applications, though representative, cannot cover all possible usage patterns. Performance in other specialized application categories could be explored.
- **Static Parameters:** The aging thresholds remain static during runtime. A promising direction is to develop a dynamic policy in which these thresholds adapt to the changing state of the real-time system, such as ZRAM usage or battery status.

VI. Conclusion

This paper proposed a reactive three-tier memory swapping framework, called PhantomSwap, that enhances multitasking resilience on mobile devices. Our evaluation demonstrates that PhantomSwap significantly increases application cache capacity and improves worst-case hot-launch latency, with only modest system overheads. By leveraging an intermediate ZRAM tier and a co-designed Garbage Collector (GC) integration, our work presents a robust and practical solution for delivering a more consistent and fluid user experience in memory-constrained mobile environments. We also identify several promising directions for future work, including hybrid policy controllers and asynchronous prefetching.

Acknowledgment

The work was partially supported by the National Council of Science and Technology, Taiwan, Under the grant no. 113-2221-E-002-177.

References

[1] D. Nunez, S. Z. Guyer, and E. D. Berger, "Prioritized garbage collection: explicit gc support for software caches," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 695–710. [Online]. Available: https://doi.org/10.1145/2983990.2984028

[2] I. A. Qazi, Z. A. Qazi, T. A. Benson, G. Murtaza, E. Latif, A. Manan, and A. Tariq, "Mobile web browse under memory pressure," *SIGCOMM Comput. Commun. Rev.*, vol. 50, no. 4, p. 35–48, Oct. 2020. [Online]. Available: https://doi.org/10.1145/3431832.3431837

[3] Android Developers, "Low memory killer (lmk)," https://developer.android.com/games/optimize/vitals/lmk, 2024, accessed: 2025-06-19.

[4] N. Lebeck, A. Krishnamurthy, H. M. Levy, and I. Zhang, "End the senseless killing: improving memory management for mobile operating systems," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC'20. USA: USENIX Association, 2020.

[5] J. Huang, Y. Zhang, J. Qiu, Y. Liang, R. Ausavarungnirun, Q. Li, and C. J. Xue, "More apps, faster hot-launch on mobile devices via fore/background-aware gc-swap co-design," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 654–670. [Online]. Available: https://doi.org/10.1145/3620666.3651377

[6] Android Developers, "Memory allocation among processes," https://developer.android.com/topic/performance/memory-management, 2024, accessed: 2025-06-19.

[7] Android Open Source Project, "Debug art garbage collection," https://source.android.com/docs/core/runtime/gc-debug, 2024, accessed: 2025-06-19.

[8] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection," *Proceedings of the IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[9] Google, "Perfetto - system profiling, tracing and diagnostics," https://perfetto.dev/, 2024, accessed: 2025-06-19.

[10] Android Developers, "Detect and fix jank in android studio," https://developer.android.com/studio/profile/jank-detection, 2024, accessed: 2025-06-19.