More Apps, Faster Hot-Launch on Mobile Devices via Fore/Background-aware GC-Swap Co-design

Jiacheng Huang
jiacheng.huang@my.cityu.edu.hk
City University of Hong Kong &
Wuhan University
Hong Kong, China

Yunmo Zhang
yunmo.zhang@my.cityu.edu.hk
City University of Hong Kong
Hong Kong, China

Junqiao Qiu
junqiqiu@cityu.edu.hk
City University of Hong Kong
Hong Kong, China

Yu Liang
yulianglenny@gmail.com
ETH Zürich
Zürich, Switzerland

Rachata Ausavarungnirun
r.ausavarungnirun@gmail.com
King Mongkut's University of
Technology North Bangkok
Bangkok, Thailand

Qingan Li[*]
qingan@whu.edu.cn
Wuhan University
Wuhan, China

Chun Jason Xue
jason.xue@mbzuai.ac.ae
Mohamed bin Zayed University of
Artificial Intelligence
Abu Dhabi, United Arab Emirates

Abstract

Faster app launching is crucial for the user experience on mobile devices. Apps launched from a background cached state, called hot-launching, have much better performance than apps launched from scratch. To increase the number of hot-launches, leading mobile vendors now cache more apps in the background by enabling swap. Recent work also proposed reducing the Java heap to increase the number of cached apps. However, this paper found that existing methods deteriorate app hot-launch performance while increasing the number of cached apps. To simultaneously improve the number of cached apps and hot-launch performance, this paper proposes Fleet, a foreground/background-aware GC-swap co-design framework. To enhance app-caching capacity, Fleet limits the tracing range of GC to background objects only, avoiding touching long-lifetime foreground objects. To improve hot-launch performance, Fleet identifies objects that will be accessed during the next hot-launch and uses runtime information to guide the swap scheme in the OS. In addition, Fleet aggregates small objects with similar access patterns into the same pages to improve swap efficiency. We implemented Fleet in AOSP and evaluated its performance with different types of apps. Experimental results show that Fleet achieves a 1.59× faster hot-launch time and caches 1.21× more apps than Android.

CCS Concepts: • **Software and its engineering**; • **Computer systems organization** → *Embedded systems*;

Keywords: Garbage Collection, Operating Systems, Memory Management, Mobile Systems

ACM Reference Format:
Jiacheng Huang, Yunmo Zhang, Junqiao Qiu, Yu Liang, Rachata Ausavarungnirun, Qingan Li, and Chun Jason Xue. 2024. More Apps, Faster Hot-Launch on Mobile Devices via Fore/Background-aware GC-Swap Co-design. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620666.3651377

[*]Corresponding Author

1 Introduction

Mobile devices have become an indispensable part of daily life for everyone today. Mobile device users often switch among apps. The app launch performance, therefore, is a crucial metric of the user experience [41]. To enhance the app launch performance, Android caches apps in the background whenever there is sufficient memory [32, 33], while interacting with users with the app in the foreground, as shown in Figure 1. When the user switches to an app already cached in the background, Android directly moves

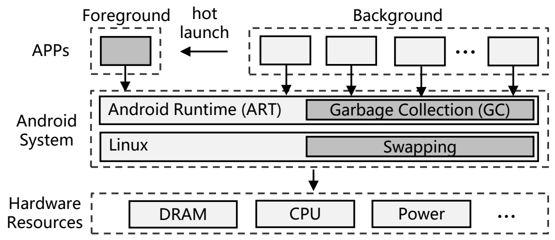


Figure 1. Architecture of Android systems.

it to the foreground, called *hot-launch* [41]. Hot-launch is much faster than launching an app by creating a new process, called *cold-launch* because the former does not require additional initialization work.

To benefit more from the hot-launch, there have been two directions. First, enabling the swap provided by the Linux kernel to cache more apps. The swap extends the available memory resources by offloading the least-recently-used (LRU) pages from the main memory to the swap partition [4–6, 13, 33, 54]. Second, recent work Marvin [32] proposes to cache more apps by separating the reference information from the object and keeping it in memory after the object has been swapped out. However, while both enabling swap and Marvin increase the number of cached apps, this paper found that the hot-launch time deteriorates significantly, especially for tail latency. This is because, existing works essentially use the native LRU-based swap mechanism, which is agnostic of app re-launching. It may swap out memory pages that are least recently used but required during the hot-launch stage. This paper aims to improve hot-launch performance while simultaneously caching more apps.

To maximize app caching capacity and improve hot-launch performance, this paper proposes Fleet, a fore/background-aware GC-swap co-design framework. The idea is motivated by two key fore/background-related observations, as depicted in §4: (i) The lifetime of foreground objects (i.e., objects allocated when the owner app runs in the foreground) is much longer than that of background objects (i.e., objects allocated when the owner app runs in the background). Hence it will be more efficient to conduct GC on background objects only when an app is in the background; (ii) Upon hot-launches, objects allocated in the immediately preceding foreground state and objects close to the GC roots are more likely to be re-accessed and can be identified through runtime information. These objects can be cached in memory to facilitate hot-launches.

Based on these observations, Fleet is proposed, which consists of two components: (i) Background-object GC (BGC), which limits the tracing range of the garbage collector to background objects only, avoiding touching the foreground objects. This approach minimizes the conflict between GC and the swap, maximizing the caching capacity for apps; (ii) Runtime-Guided Swap (RGS), which adaptively guides the swap scheme using runtime information. It identifies objects that will be re-accessed during the next hot-launch and adjusts the swap scheme to cache these identified objects in memory.

We implement Fleet on Android. The experimental results show that, on average, Fleet can increase the number of maximum cached apps by 21% compared to the original Android. Compared to Marvin, Fleet improves the tail hot-launch performance by 4.45× on average. Additionally, Fleet has a better ability to cache apps by 2× for apps with small objects. This paper makes the following contributions:

- We observe that when an app is in the background, its foreground objects occupy a larger memory footprint and have a longer lifetime compared to its background objects;
- We observe that some special objects occupy a large proportion of object re-accesses during the hot-launch, and these special objects can be determined;
- Based on the first observation, we propose a GC method that restricts the tracing range to background objects only to mitigate the conflict between GC and the swap mechanism, to cache more apps in memory;
- Based on the second observation, we propose a swap scheme that selectively keeps objects likely to be used during hot-launch in memory, to improve the hot-launch performance.

2 Background

In this section, we introduce how Android launches apps (§2.1) and the memory management involved in app launching, including GC and swap (§2.2).

2.1 Hot-launch is crucial for mobile user experience

Android typically has one launched app in the foreground, which allows interaction with the user [26, 34, 41]. There are two ways to launch apps: *hot-launch* and *cold-launch*. As shown in Figure 1, hot-launch refers to the app being directly switched from the background to the foreground, while cold-launch refers to the app starting with a new process creation. Hot-launch is more desirable in terms of user experience because it offers shorter launch latency. Using the experimental setup detailed in §6, we measure the duration of hot-launch and cold-launch for our tested apps. The launch time is the duration until the first frame is displayed, which is measured by the Android Debug Bridge (ADB) tool [7, 8]. The cold-launch time is obtained by explicitly terminating apps before the launch. We repeat the launch 20 times for each test case and calculate the average and standard deviation. Figure 2 shows that hot-launch significantly outperforms cold-launch. For example, the average hot-launch time for Twitter is 273 ms, while the cold-launch time is 2390 ms, which is 8.75× longer than the hot-launch.

The speedup from the hot launch is crucial for the mobile user experience. Previous studies on user experience in the

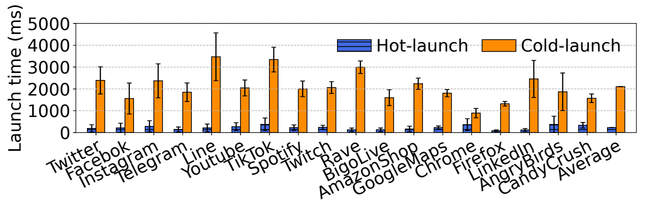


Figure 2. The average hot-launch and cold-launch times. The hot-launch is significantly faster than the cold-launch for all tested apps.

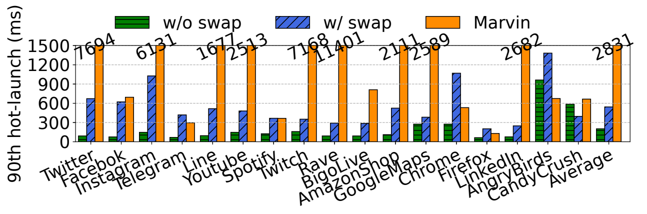


Figure 3. The 90th percentile tail hot-launch time of Marvin and Linux swap. Both of these methods deteriorate the hot-launch performance compared to the baseline without swap.

mobile context [1, 2, 22] report that delays in the range of hundreds of milliseconds are perceivable by users. According to a quantified study on the impact of response time on user experience in the human-computer interaction community [35, 44], user productivity remains unaffected only when the response time is below 150 milliseconds. If the response time exceeds 150 milliseconds, users become aware of the delay [44]. The issue of hundred-millisecond delays has also become central to the memory management and scheduling efforts of many mobile devices [23, 29, 45].

2.2 Two-layer memory management

Android uses the Android Runtime (ART) to execute apps and utilizes Linux to manage hardware resources (Figure 1). Each app runs in an independent ART instance. ART is an implementation of a Java virtual machine that incorporates a garbage collection (GC) mechanism to manage the app's dynamic memory [20, 52]. At the same time, Linux extends memory capacity through the swap mechanism, which can offload cold memory to a swap partition. This section will introduce the GC and swap separately.

Garbage collection. ART employs a tracing-based concurrent copying GC [10]. The GC performs liveness analysis of objects by traversing the object reference graph and copies live objects to a new memory location. Here is a list of GC terminologies used in the paper:

- *Region:* A segment of continuous memory. The region from which the GC copies objects is called the *from-region*, while the region to which the GC copies objects is called the *to-region*;
- *Roots:* A group of special objects that directly or indirectly reference all objects used by the program;
- *Mutator thread:* All app threads, excluding the GC thread;
- *Read barrier:* A piece of code executed by mutator threads concurrently with the GC thread. It could copy objects to the *to-region* whenever accessing an object;
- *Write barrier:* A piece of code that is executed whenever a mutator thread writes an object;
- *Card table:* An array where each byte represents some objects corresponding to a range of continuous addresses.

Linux swap. As described in §2.1, Android aims to benefit from hot-launches by caching apps in memory. However, caching more apps is challenging due to the limited memory capacity of mobile devices. Therefore, Android utilizes swap to enable more memory than the physical DRAM capacity available. When memory pressure becomes high, the swap mechanism offloads the least-recently-used pages to the swap partition [34, 51].

Nowadays, the number of apps is rapidly increasing, and each app is occupying more memory [39, 41]. The increasing speed of DRAM capacity cannot match the significantly growing memory requirements of mobile apps [53]. In order to cache more apps in the background, enabling swap has become the default and widespread setup for Android devices. Mainstream mobile device vendors, such as Samsung, Huawei, and OPPO, have enabled swap in their smartphones [4, 6, 13].

However, swap cannot efficiently work with GC in Android to achieve a desirable caching capacity because the pages swapped out would be swapped back due to the tracing procedure of GC. To address this issue, Marvin [32] proposes to mitigate the conflict between GC and swap using a bookmarking GC [27]. Marvin saves corresponding references for each swapped-out object, allowing it to locate live objects based on the references without touching and swapping them back. Therefore, Marvin can improve the caching capacity of Android.

3 Motivation

In this section, we study the limitations of existing works (§3.1) and identify two issues of the Linux swap that lead to their inferior performance on Android (§3.2).

3.1 The performance of hot-launch deteriorates when more cached apps are enabled

There are two methods to cache more apps in Android: (i) Linux swap and (ii) Marvin [32]. While these methods enable more apps to be cached in the background, they sacrifice the hot-launch performance. Using the experimental setup described in §7.2, we measure the tail hot-launch performance of these two methods.

Tail hot-launch in Linux swap. As depicted in Figure 3, the hot-launch time is slowed by several hundred milliseconds due to the swapped pages. For example, Instagram takes 1027 ms to complete a hot-launch with swap enabled, while it only takes 147 ms without swap. The hot-launch becomes 7× slower after enabling swap. In the following §3.2, we will conduct a detailed analysis to identify the root causes of the low hot-launch performance when Android enables swap.

Tail hot-launch in Marvin. Figure 3 also demonstrates that Marvin could experience several seconds of delay during tail hot-launch. Despite its ability to cache more apps, Marvin fails to achieve the desired hot-launch performance. This is due to three drawbacks of Marvin: (i) It requires a long stop-the-world (STW) pause time to maintain consistency between the separated reference information and objects; (ii) Marvin induces swap amplification due to analyzing at object granularity while swapping at page granularity; (iii) Marvin employs an LRU-based swap scheme that is agnostic to whether or not the evicted page will be used during the hot-launch stage.

In summary, the ultimate challenge is: ***How can we maintain or even improve the hot-launch performance while caching more apps?*** This is the aim of this paper.

3.2 Analysis of the Linux swap in Android

Android directly utilizes the existing LRU-based swap scheme in Linux. However, the LRU-based scheme is designed for general workloads from the Linux perspective and neglects the specific characteristics of Android. In this section, we will analyze the challenges of enabling Linux swap in Android.

We conducted a motivational experiment, as shown in Figure 4. First, we started the Amazon shop in the foreground. Then, we switch it to the background after 20 seconds. Finally, we switch it back to the foreground at 53 seconds (through a hot-launch). During this experiment, we assign an increasing object ID to each object based on their allocation order and sample the object access every 100 accesses. Based on the results, we uncovered two issues using the LRU-based swap scheme for Android apps.

First, the GC may offset the effects of swapping in terms of caching more apps. Figure 4 illustrates that when the app is in the background, only a small portion of objects are accessed in memory, while most other objects are least recently used. However, when a GC is triggered (at around 37 seconds), there is a spike in accessed objects due to the tracing work of GC, even though they are not necessarily used by the app mutator threads. Due to GC, the LRU pages will be swapped back into memory, leading to high memory pressure, which may induce terminations of cached apps.

Second, the hot-launch process may be delayed as the necessary pages could be swapped out beforehand by the LRU-based swap mechanism. Figure 4 shows that when the app is in the background, there are many LRU objects, and

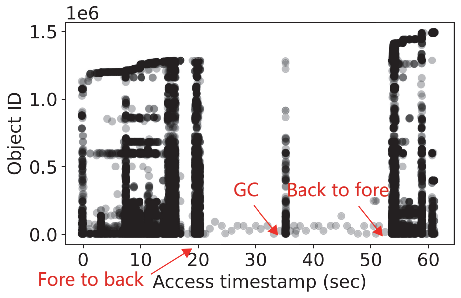


Figure 4. The accessed objects over time. Initially, we start the Amazon shop in the foreground. Then, we switch it to the background at 20 seconds. During its run in the background, a GC occurs at 37 seconds. Finally, we switch it back to the foreground at 53 seconds.

most of the memory could be safely swapped out according to the LRU assumption. However, during the hot-launch stage (at around 53 seconds), the app needs to access many objects that haven't been used recently. Then, the hot-launch process may be blocked due to accessing multiple pages that have been swapped out.

The on-demand swap-in from the swap partition is significantly slower than reads from the DRAM. We measure the performance of both the DRAM and the swap partition. We utilize tinymembench [16] and FIO [17] to test the read bandwidth of the DRAM and the flash-based swap partition. The experimental results indicate that the DRAM bandwidth is 9182.7 MB/s, whereas the read bandwidth of the swap partition is 20.3 MB/s. The performance of the DRAM is approximately 452× faster than that of the swap partition.

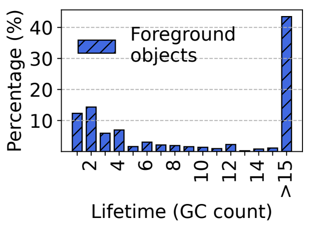
In summary, the LRU-based swap in Linux does not work well for Android apps, neither for caching more apps nor for achieving fast hot-launch. The Marvin work partially addresses the first issue but fails to address the second one. As a result, existing works cannot simultaneously maximize app caching capacity and improve hot-launch performance.

4 Observations

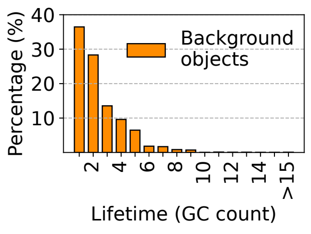
As stated in the analysis in §3.2, app state in mobile devices is either *foreground* or *background*, which the prior Linux swap is agnostic to. In this section, we present three observations of Android apps related to their foreground and background states.

4.1 Fore/background object characteristics

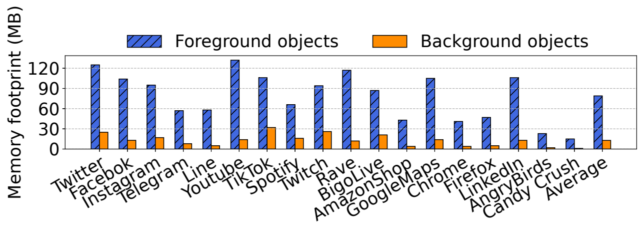
Due to the significant change in an app's behavior when it switches from the foreground to the background (as shown in Figure 4), we examine the distinct states of objects before and after the switch to the background in this section. We categorize all objects into two types according to the owner app's state when they are allocated:



(a) Foreground object lifetime



(b) Background object lifetime



(c) Memory footprint

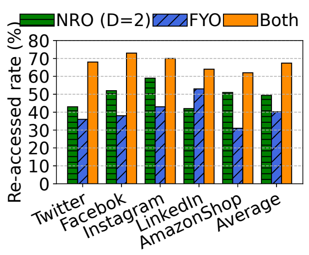
Figure 5. Foreground and background objects have different lifetime and memory footprints. Foreground objects have longer lifetimes and occupy the majority of memory. In (a) and (b), the last bar indicates the percent of alive objects after 15 GC cycles.

- *Foreground objects (FGO):* The objects that are allocated when the owner app is in the foreground state;
- *Background objects (BGO):* The objects that are allocated when the owner app is in the background state.

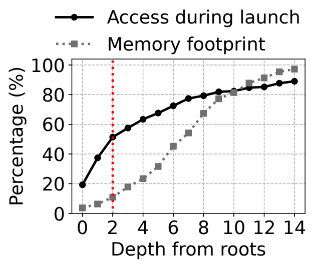
At the moment that an app switches to the background, all existing objects are considered FGO, while all newly allocated objects after the switching are classified as BGO.

We measure the lifetime (GC count) of these two types of objects while the app is in the background. Using Twitter as an example, we start and use it for 10 minutes in the foreground. Then, we switch it to the background and perform an explicit GC every 15 seconds. After each GC, we calculate the lifetime of each object as the number of GC cycles that it survived. Finally, we generate the lifetime distribution of FGO (Figure 5a) and BGO (Figure 5b), respectively. These results show a significant disparity in the lifetime of FGO and BGO. Most BGO are reclaimed within the first several GCs, while more than 40% FGO still survive after 15 GC cycles. One possible reason is that all BGO are newly allocated in the background, which are younger and thus more likely to become garbage [42]. Additionally, the number of FGO is far more than BGO, as shown in Figure 5c. This is because a background app is mostly inactive and only allocates a few objects. Therefore, we draw the first key fore/background related observation: ***When an app is in the background, its FGO occupy a larger memory footprint and have a longer lifetime compared to its BGO.***

The existing ART uses a GC scheme that treats FGO and BGO equally. However, based on our observation, the FGO



(a) Re-accessed objects



(b) Depth analysis of NRO

Figure 6. The re-access analysis of NRO and FYO during hot-launches. The larger the depth, the higher the re-access ratio covered by the NRO, while the memory footprint also increases. Because the re-access ratio increases faster than the memory footprint for small depth values, we can select a relatively small depth parameter to cover more re-access with minimal memory footprint.

should not be visited by the GC as often as the BGO, since most of them have a long lifetime.

4.2 Fore/background object access pattern

As shown in Figure 4, a subset of FGO (accessed before 20 seconds) are re-accessed at 53 seconds during the hot-launch procedure. In accordance with the nature of the Android launch procedure, we identify two special types of FGO that tend to be re-accessed during the hot-launches:

- *Near roots objects (NRO):* The objects for which the shortest path to the roots is smaller than a depth parameter (defined as D) in the reference graph are defined as NRO with the depth distance. NRO could be different when selecting different depth values;
- *Foreground young objects (FYO):* The objects allocated just before an app is switched to the background.

We conducted an experiment to analyze the object re-accessed during hot-launches. We start each tested app and use it in the foreground for 10 minutes. Then, we switch the app to the background. We obtain the FYO and NRO during the first GC when the app is in the background (depicted in §5.3). After 30 seconds, we switch the app back to the foreground and record all accessed objects during the hot-launch time. Finally, we calculate the re-access ratio of NRO and FYO, as shown in Figure 6a. For these five tested apps, NRO occupies around 50% re-access objects on average and FYO is around 40%. NRO and FYO, combined, account for 68% of the total re-access objects on average.

We further analyzed the memory footprints of these two types of objects for the five apps in Figure 6a. The average memory percentages of NRO and FYO are 10.4% and 9.3%, respectively, with a depth of 2. Together, they occupy 15.5% of the heap memory because they share common objects. Therefore, these two types of launch objects (NRO and FYO)

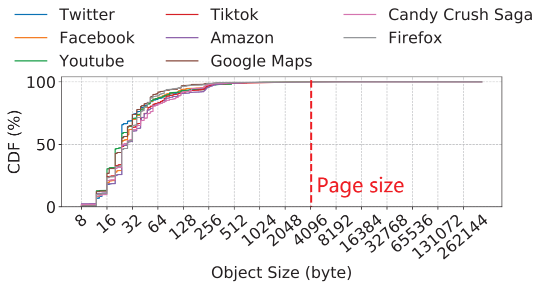


Figure 7. The object size distribution. Due to the significant size difference between objects and pages, co-designing GC and swap is challenging.

can cover the majority of demanded objects with a small memory footprint. The second key fore/background related observation is: ***NRO and FYO occupy a large proportion of object re-accesses during hot-launches with a small memory footprint.***

We also analyzed the effect of the depth parameter for NRO using Twitter, as shown in Figure 6b. With a larger depth parameter, the re-access ratio of NRO is higher, but its memory footprint also becomes larger. Regarding the depth parameter, one key insight is that the re-access ratio increases faster than the memory footprint for small depth values. We can select a relatively small depth parameter to cover more re-access with a small memory footprint.

NRO are likely to be reaccessed during the hot-launches for two reasons. First, NRO is often accessed in the intermediate path of many other objects. Since the roots include all objects an app could directly access, objects closer to the root are more likely to be accessed when accessing other objects. Second, GC usually starts simultaneously with a hot-launch, and it will also access NRO. Android uses a dynamic memory threshold to control GC. If the current memory usage is close to the threshold, a new GC will be triggered. When an app is in the background, the threshold is set to a value close to the memory usage. However, during a hot-launch, many new objects are created quickly, resulting in memory usage larger than the threshold and triggering a new GC. This GC will also access NRO.

FYO are also likely to be re-accessed because of their associated foreground tasks. These tasks are apt to be suspended in the background. When the app switches back to the foreground, these suspended tasks resume and continue to access the recently allocated young objects. Furthermore, the young objects will be accessed more frequently than the others [18] in general. Thus, FYO are also highly probable to be accessed by the resumed foreground tasks during the hot-launch time.

4.3 The size mismatch between GC and swap

Another feature of Android is that most objects are small. We analyzed the object size distribution for several popular commercial apps in the app store market. As shown in Figure 7, all tested apps exhibit similar object size distributions. The swap mechanism operates at the granularity of a fixed-size page, commonly 4 KB [12]. The majority of objects are significantly smaller than the page size. Due to the significant size difference between objects and pages, co-designing GC and swap is challenging. To address this challenge, our insight is to group objects with similar access patterns into the same page based on runtime information.

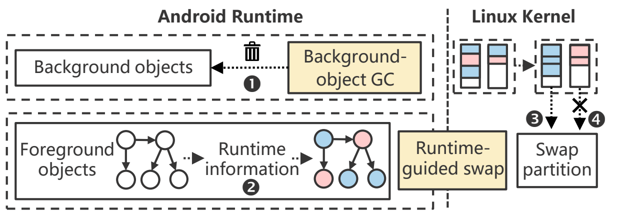


Figure 8. Overview of Fleet.

5 Fleet Design

5.1 Overview

Fleet is a cross-layer memory management framework that aims to cache more apps and enable faster hot-launches simultaneously. Figure 8 illustrates the two main components in Fleet's design: (i) Background-object GC, a GC method that collaborates with swap, and (ii) Runtime-guided swap, a swap scheme that leverages runtime information to optimize hot-launches.

Key idea. Driven by the observations presented in §4, Fleet is designed based on two key ideas: (i) Since BGO is more likely to become garbage than FGO, Fleet focuses on BGO in background GC. The long-lifetime FGO, which occupies most of the memory, can be safely swapped out. Therefore, GC can minimize access to the swapped pages while freeing garbage objects on time; (ii) Fleet identifies NRO, FYO, and currently used objects based on the runtime information in ART. The underlying swap can adjust its scheme to facilitate access to these NRO and FYO, optimizing the performance of hot-launches.

Background-object GC (§5.2). Background-object GC (BGC) is designed to free garbage objects while avoiding touching the swap pages when an app is in the background. Unlike the existing GC in ART, which traces all objects, BGC only traces BGO ❶.

Runtime-guided swap (§5.3). Runtime-guided swap (RGS) is proposed to swap pages in order to extend memory capacity while retaining hot-launch performance. RGS consists of two steps. First, it utilizes a copy-based GC to identify the NRO and FYO after an app switches to the background ❷. During this GC, objects are also grouped into pages. Second, RGS conveys the information of the grouped pages to the Linux kernel. Guided by the runtime information, Linux can actively swap out the grouped pages ❸, while caching pages containing NRO and FYO in the main memory ❹.

Workflow. Fleet starts working after an app is switched to the background. First, Fleet waits for a specific time (defined as T_B) to ensure the app runs in a stable background state. Then, Fleet performs a full GC, classifying objects into different types, such as NRO and FYO, based on runtime information. Objects are then grouped into pages according to their classified types. Afterward, Fleet applies RGS to guide the swap process. While running in the background, the app employs BGC to free garbage objects. NRO and FYO objects are cached until there are no other pages to be swapped out. If a hot-launch occurs during this period, it would be fast because NRO and FYO objects are grouped and cached in the main memory. Once the app is switched to the foreground, Fleet waits for a specific time (defined as T_F) to ensure that the app runs in a stable foreground state. Finally, Fleet stops, and the foreground app executes the same as a default Android app.

5.2 Background-object GC

Existing ART mainly utilizes two types of GC: *(i)* Minor GC, which frees garbage objects from newly allocated regions after the last GC; *(ii)* Major GC, which frees garbage objects from the entire Java heap. The existing major GC is not suitable for background apps because it targets all objects and calculates the live ratio of regions by fully tracing. We propose BGC (Figure 9) to replace the existing major GC when an app is in the background. BGC aims to free garbage objects only from BGO to minimize access to the FGO. In this section, we present the design of BGC. First, we explain how to separate FGO and BGO. Then, we introduce a carefully designed *card table*. Finally, we present BGC's execution procedure.

FGO & BGO separation. Fleet divides all objects into FGO and BGO by adding a region-type flag to the metadata of regions, indicating whether these regions contain FGO. After an app is switched to the background, Fleet uses a full GC to compact all FGO into specific regions (detailed in §5.3.1). Simultaneously, Fleet also sets the region-type flag of the regions to which all FGO are copied. Afterward, all objects allocated while the app is in the background are classified as BGO. These objects will be allocated in new regions that are different from those already containing FGO. Therefore, FGO and BGO can be organized in separate regions.

Card table for tracing BGO. To conduct liveness analysis for BGO, we need to track all references from FGO to BGO. BGC can identify these references by recording modifications made to FGO, as all BGO are allocated after FGO.

To achieve this, we propose the use of an additional card table to record all modified FGO, as shown in Figure 9. BGC uses a shift instruction with the CARD_SHIFT value to translate between the FGO address and the offset of the corresponding byte in the card table. BGC records the modified

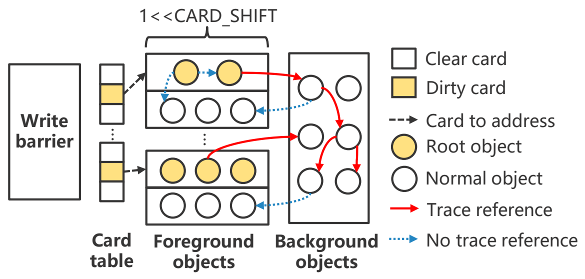


Figure 9. The BGC Framework. The app's total objects are divided into two categories: BGO and FGO. By employing a card table, the BGC restricts the GC range to the BGO.

FGO using a designed *write barrier*. Whenever an object is written, the write barrier first checks if the object belongs to the FGO. If the object belongs to the FGO, the write barrier code will find the corresponding byte in the card table by using a shift instruction on the object's address. Then, the write barrier code will mark the byte as DIRTY. Finally, during BGC execution, it can obtain all references from FGO to BGO by checking the objects corresponding to the bytes marked as DIRTY in the card table.

GC execution. As shown in Figure 9, the execution follows these steps: Once FGO and BGO are separated into different regions, BGC initializes its card table to empty. While the app runs in the background, modified objects are recorded in the card table using the designed write barrier. When a GC starts, it scans all dirty bytes in the card table to identify the FGO that have been modified while the apps are in the background. The GC then adds all these modified FGO to the root set. Next, the GC performs a tracing procedure to find all objects that can be reached from the root set. During the tracing procedure, if the BGC encounters a reference to an object belonging to FGO, it considers this object as a live object and does not access it. At the same time, it copies all live BGO to new to-regions. Finally, the BGC releases the memory of all from-regions containing the BGO. By following this process, the BGC can free garbage objects in BGO while minimizing access to the FGO.

Discussion on memory leak. BGC can efficiently free garbage objects in background apps on time due to the long lifetime and inactive nature of FGO, as observed in the lifetime distribution discussed in §4.1. In some cases, if necessary, Fleet can resort to the original Android method of using full tracing to clear garbage objects from the entire Java heap. Nevertheless, even in such worst-case scenarios, Fleet still benefits from reducing the frequency of full heap tracing for background apps.

5.3 Runtime-guided swap

The existing Android system directly utilizes the native LRU-based swap scheme provided by Linux without considering the mobile-specific characteristics of hot-launch performance. The runtime-guided swap (RGS) is designed to propose a new swap scheme that aims to achieve faster hot-launch performance by taking into account runtime information, including NRO and FYO. RGS consists of two steps: (i) Object grouping, a full GC process that classifies objects based on runtime information and groups them into pages; (ii) Page swapping, a system call that conveys information about the grouped pages to the Linux kernel, thereby achieving a hot-launch-friendly swap scheme.

5.3.1 Object grouping. RGS utilizes a full GC that considers the access pattern of objects to group them. The object grouping operation begins after the app has been switched to the background for a specified window of time (T_B).

Classification. We will identify the NRO and FYO for the next hot-launch, based on our observations in §4.2. Additionally, in this section, we will introduce a new type of object called working set (WS) objects. These WS objects are the objects currently being used by the app. The WS type ensures that the necessary objects required for the app to run in the background are not selected to be swapped out.

As shown in Figure 10, unlike the existing GC in Android, which uses a depth-first search (DFS) algorithm to trace the object reference graph, we utilize a breadth-first search (BFS) algorithm with a first-in-first-out *mark queue*. Based on the BFS algorithm starting from the roots, we perform a *mark* operation for each traversed object.

Similar to the existing concurrent GC, we enable the read barrier mechanism during the BFS-based traversal. Concurrently with the tracing and mark procedure of the GC thread, mutator threads also perform the mark operation through the read barrier whenever they access an object. The WS objects can be differentiated by the thread that triggers the mark operation. If an object is marked by mutator threads (in their read barriers), it will be classified as a WS object.

In summary, during this GC procedure, the mark operation can classify objects into the following categories:

- *NRO:* We add a *depth delimiter* to the mark queue during graph traversal. Additionally, we maintain a *depth counter* throughout the traversal procedure. The depth counter keeps track of the depth of the currently marked object from the roots. If the depth counter is smaller than the NRO threshold, we classify the marked object as an NRO;
- *FYO:* Because the group operation of RGS is the first GC after the app is switched to the background, all objects newly allocated between the group operation and the last GC are considered FYO. In ART, there is a flag in the metadata of each region that indicates whether the region is newly allocated after the last GC. By utilizing this flag, we can determine whether an object is an FYO;
- *WS objects:* We obtain the thread ID of the current thread during the mark operation. We compare the thread ID with that of the GC thread. If these thread IDs are different, we identify the marked object as a WS object;

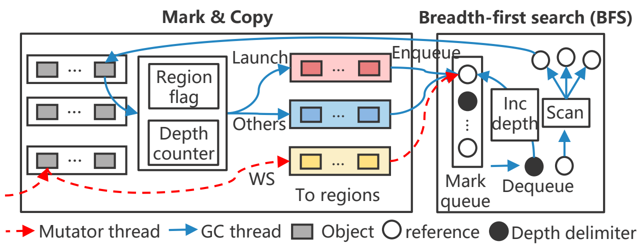


Figure 10. The object grouping procedure of the RGS. It groups objects into pages through a full GC.

- *Cold objects:* All other objects are classified as cold objects.

Group into regions. As shown in Figure 10, a *copy* operation is imposed on every object marked by the full GC. The copy operation copies all reachable objects from from-regions to to-regions. Unlike the existing ART, which treats all to-regions equally, we propose three types of to-regions corresponding to the classification of the object:

- *Launch region:* This region contains objects that will be used in the next hot launch. We copy all NRO and FYO objects to the launch region;
- *WS region:* This region contains objects that will be used when the app is in the background. We copy all WS objects to the WS region;
- *Cold region:* This region contains objects that will not be used during the background and the hot launch. We copy all cold objects to the cold region.

Based on the object classification, the copy operation selects the corresponding type of region as the destination to-region for every marked object. The ART uses the bump-pointer allocator [10, 19] to continuously allocate objects in these regions, resulting in objects with the same classification being compactly grouped in their respective regions. Finally, we obtain the rearranged pages by utilizing the memory pages of the regions.

5.3.2 Page swapping. Based on the three types of regions in the object grouping procedure, all FGO are grouped into three types of pages: (i) Launch pages, (ii) WS pages, and (iii) Cold pages. To achieve the interaction between ART and Linux, we modified the existing `madvise` system call in the Linux kernel and added two new options: `COLD_RUNTIME` and `HOT_RUNTIME`. Therefore, we can convey the information of these different types of pages to the underlying Linux through the modified `madvise` system call.

Actively swap-out. After the object grouping procedure finishes, RGS will make a `madvise` system call with the `COLD_RUNTIME` option and provide the memory address range of all cold regions as input values. The kernel part of RGS will then identify all physical pages corresponding to these specified memory address ranges. These physical pages will be actively swapped to the swap partition to preserve memory ahead of memory-pressure situations. As a

Table 1. Comparison methods.

Methods	GC approach	Swap approach	
		Granularity	Scheme
Android [10]	Native GC	Page	LRU
Marvin [32]	Bookmark GC	Object	Object LRU
Fleet	Background-object GC (§5.2)	Grouped page (§5.3.1)	Runtime-guided swap (§5.3)

Table 2. Fleet's default parameters.

Parameter description	Symbol	Setting
Maximum depth to the roots for NRO	D	2
Wait time to start Fleet in the background	T_B	10 seconds
Wait time to stop Fleet in the foreground	T_F	3 seconds
`CART_SHIFT` for card address conversion	-	10
Region size of the Java heap	-	256 KB

Table 3. The commercial apps for evaluation.

App type	App Description
Communication	Twitter (X), Facebook, Instagram, Telegram, Line
Multi-media	Youtube, Tiktok, Spotify, Twitch, Rave, BigoLive
Tools & utilities	AmazonShop, GoogleMaps, Chrome, Firefox, LinkedIn
Games	Angry birds classic, Candy crush saga

result, RGS optimizes the swap-out mechanism by actively swapping out cold regions when an app switches to the background.

Cache the launch pages. While an app is in the background, RGS will periodically execute the `madvise` system call with the `HOT_RUNTIME` option on the memory of the hot regions. RGS will find the corresponding physical pages for these hot regions and move these pages to a highly used position in the LRU queue in the Linux kernel. As a result, these launch pages are unlikely to be swapped out, optimizing the hot-launch performance.

6 Evaluation Methodology

Comparison schemes. We compare Fleet to default Android [10] and Marvin [32], as summarized in Table 1. The key differences between these methods are as follows:

Default Android: Android uses GC and swap as two independent mechanisms. As mentioned in §3.2, this method has two problems: (i) Low caching capacity due to conflicts between GC and swap, and (ii) Slow hot-launch due to sudden swapping of multiple pages during launch.

Marvin: This method also co-designs GC and swap. Marvin records the reference information of each object before swapping it out. By utilizing the recorded reference information, it does not need to access the objects during GC. However, Marvin could not achieve desirable hot-launch performance for three reasons: (i) Marvin uses object-granularity swap, which is inefficient for swapping objects smaller than the page size, (ii) maintaining the reference information requires long stop-the-world pauses, and (iii) it does not consider the suddenly accessed objects during hot-launch.

Fleet: This method is a collaborative design for GC and swap. Fleet groups small objects into pages according to the runtime information before swapping. Additionally, Fleet resolves the conflict between GC and swap by restricting the GC range only to the background objects.

Parameter setting. Both Fleet and Marvin use the default parameters, which are identical to the default Android parameters, in addition to the parameters proposed by their method. Marvin includes a large object threshold parameter. Based on this threshold parameter value, Marvin only processes objects that exceed the threshold size. For our evaluation, we set the threshold parameter to 1024 bytes. The configurations for Fleet are listed in Table 2.

Experimental platform. We use a Google Pixel 3 smartphone with 4 GB of Micron LPDDR4X RAM and an Octa-core Qualcomm Snapdragon 845 processor. For the swap partition, we employ a 2 GB flash-based block device. We use the android-10.0.0_r1 branch of the Android Open Source Project (AOSP) and the android-msm-crosshatch-4.9-android10 branch of the Linux kernel.

Workloads. We utilize two types of apps: (i) The manually created apps downloaded from the Marvin project [3]. These apps have two parameters: object size and memory footprint. Once these apps start, they allocate objects of the same size, determined by the object size parameter, to occupy the specified memory footprint. We configure two types of object sizes: small object apps (512 bytes) and large object apps (2048 bytes). The memory footprint of each app is set to 180 MB. (ii) The most popular commercial apps from the Google Play store, which cover four categories as shown in Table 3.

7 Evaluation Results

The evaluation of Fleet answers the following questions:

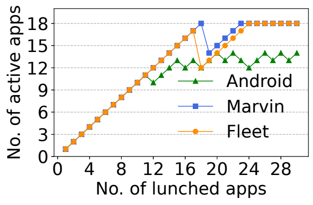
Q1: How many cached apps does Fleet improve by reducing the GC working set when an app is in the background? (§7.1)

Q2: How much does Fleet improve the hot-launch performance for different categories of apps? (§7.2)

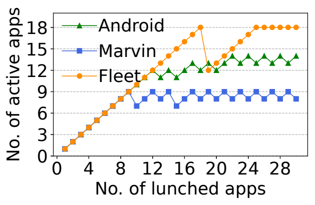
Q3: What are the effects of Fleet on other crucial system indicators such as user experience, CPU usage, memory overhead, and energy consumption? (§7.3)

7.1 App caching capacity

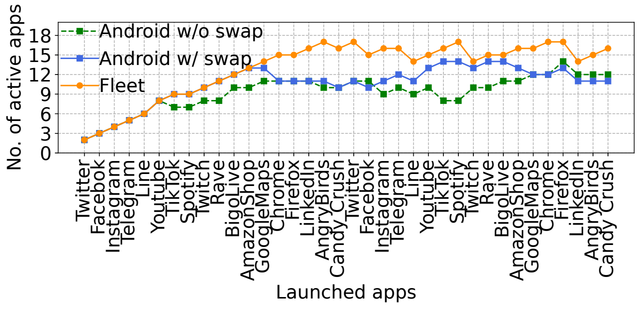
To test the caching capacity, we evaluate manually created apps to avoid any bias caused by differences in app implementations. Specifically, we continuously launch one additional



(a) Large object apps



(b) Small object apps



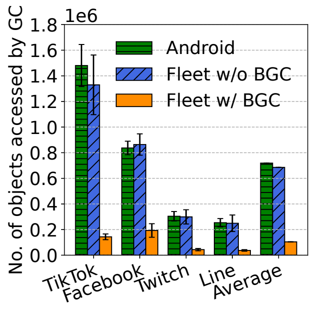
(c) Commercial apps

Figure 11. Caching capacity for different types of apps. (a) shows that Fleet and Marvin have similar caching capacity for manually created apps with large-size objects. (b) demonstrates that Fleet has better caching capacity for manually created apps with small-size objects. (c) illustrates that Fleet outperforms Android in terms of caching capacity for commercial apps.

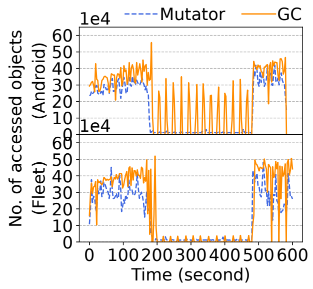
app and count the number of remaining active apps after each launch.

Since Marvin is only evaluated on large object apps, we first present the caching capacity results for the large object apps, as shown in Figure 11a. Fleet achieves a comparable caching capacity to Marvin, as both can cache up to 18 apps, which is approximately 1.3× the number of cached apps in Android. This improvement is because Marvin and Fleet can overcome the drawback of Android, where GC traces the swapped objects. In comparison, Android starts to kill apps when there are 11 cached apps and caches a maximum of 14 apps. Additionally, Figure 11b shows the caching capacity of small object apps. For these apps, Fleet outperforms Marvin by 2× in terms of maximum caching. Fleet can cache up to 18 small object apps, which is the same as caching large object apps. However, Marvin cannot handle small objects smaller than its configured large object threshold and caches a maximum of 9 small object apps. This result shows that Fleet is not sensitive to object sizes, which can be attributed to Fleet's design that groups small objects into pages.

We also evaluate Fleet on commercial apps, as shown in Table 3. To ensure a fair comparison, we only compare Fleet against Android, as Marvin's prototype does not fully support commercial apps. According to our tests, Marvin



(a) GC range in the background



(b) Accesses analysis of Twitch

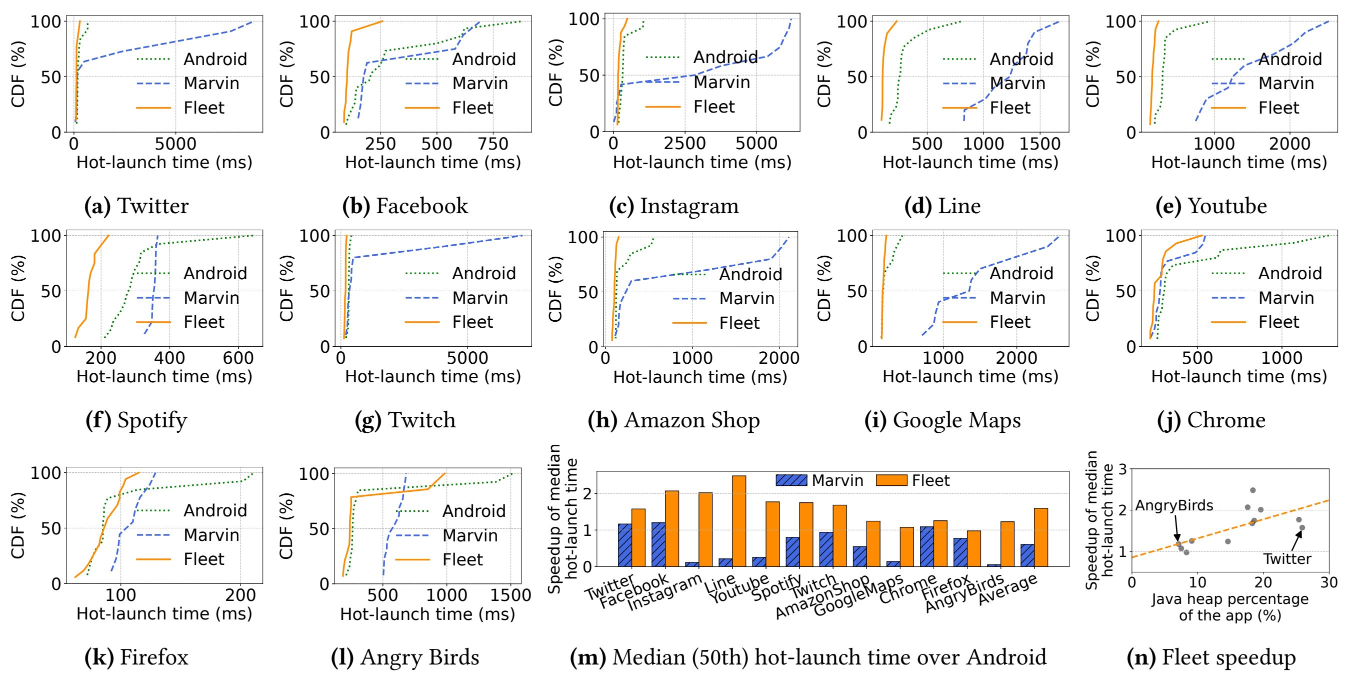
Figure 12. The analysis of GC working set. Fleet reduces the GC range when an app is in the background.

crashes after launching approximately four apps. The original paper [32] also acknowledges that Marvin easily crashes when running commercial apps. In this test, we sequentially launch these apps in a round-robin sequence (from Twitter to Candy Crush, as indicated on the x-axis of Figure 11c). We use each app for 30 seconds and record the number of active apps cached in the background. We report the results from two cycles of launches. The results show that Fleet can cache more apps than Android, both on average and at best, regardless of whether Android enables swap or not. Fleet can cache a maximum of 17 apps, which is 1.21× better than Android with swap. This improvement validates the effectiveness of Fleet's strategies. Additionally, commercial apps often consist of many small objects, which Fleet's object grouping mechanism can handle efficiently.

GC working set. To assess how Fleet improves caching capacity by reducing the working set of background GC, we measure the working sets of GC threads when an app is in the background. We measure the number of objects accessed by the GC thread during a single GC execution. Figure 12a illustrates that, on average, the GC thread accesses approximately 7×10^5 objects for Android. With BGC, Fleet can reduce the working set of GC to 10^5 on average. Fleet's reduction is approximately 7× of Android. Figure 12b presents a more detailed measurement of accessed objects for Twitch. We perform the same usage pattern on both Android and Fleet. During these experiments, we switch Twitch to the background at around 180 seconds and bring it back to the foreground at about 480 seconds. This figure demonstrates that when the app is in the background (180 to 480 seconds), Fleet can significantly reduce the number of objects accessed by the GC thread compared to Android.

7.2 Hot-launch performance

To assess Fleet's hot-launch performance, we test the commercial apps under memory pressure with about 10 background apps. We measure the hot-launch time as the duration from the start to the first frame display, which can



(a) Twitter **(b)** Facebook **(c)** Instagram **(d)** Line **(e)** Youtube

(f) Spotify **(g)** Twitch **(h)** Amazon Shop **(i)** Google Maps **(j)** Chrome

(k) Firefox **(l)** Angry Birds **(m)** Median (50th) hot-launch time over Android **(n)** Fleet speedup

Figure 13. The hot-launch performance of commercial apps under high memory pressure.

be obtained using the ADB tool [7, 8]. We conduct 20 hot-launches for each app. Additionally, between two launches, we use other apps for 30 seconds to simulate the intermittent usage pattern of users. We present the hot-launch times of 12 representative commercial apps in Figure 13. These 12 apps are selected to cover all app types in Table 3. We also provide other hot-launch data, including the other 6 apps and additional statistical characteristics in Appendix A.

From these results, we make three observations. First, Fleet achieves the best hot-launch time among these three approaches for almost all apps. As shown in Figure 13m, the average speedup of the median hot launch time of Fleet is 1.59× compared to Android and 2.62× compared to Marvin. Taking Facebook as an example, the median hot launch time for Fleet is 101 ms, which is 1.72× faster than Marvin (174 ms) and 2.07× faster than Android (209 ms). Second, Fleet displays significantly shorter tail launch times. The 90th percentile tail hot launch time of Fleet is 2.56× and 4.45× compared to Android and Marvin, respectively (More details are shown in Figure 15a in Appendix A). For instance, Android's 90th percentile hot-launch time on Facebook is 622 ms, whereas Fleet's 90th percentile launch time is 121 ms, which is 5.14× faster. Third, Figure 13n demonstrates a positive correlation between the speedup achieved by Fleet and the Java heap percentage of apps. This implies that Fleet can provide greater benefits for apps with a higher Java heap ratio. In summary, Fleet's design, which involves swapping with runtime information, enables it to achieve faster hot launch latency.

7.3 Runtime performance

Frame rendering. In this part, we present the impact of Fleet on the user experience, as measured by the jank ratio [15] and frames per second (FPS). For this experiment, we use all tested apps in the foreground state for one minute. While using the app, we simulate human interaction by continuously swiping the screen using the ADB [7] tool, following a predefined script. We profile performance information during the experiment using the system trace [9] and analyze the trace file using Perfetto [14]. To determine the number of janks, we count occurrences where the time between two rendered frames exceeds 16.7 milliseconds (equivalent to 60 frames per second) [15]. The jank ratio is calculated as the proportion of janks to the total number of frames. Additionally, we calculate the FPS results by dividing the number of rendered frames by the duration. Figure 14 displays the jank ratio and FPS results, respectively. Compared to Android, Fleet exhibits a nearly identical average jank ratio and FPS. Compared to Marvin, Fleet demonstrates a 19.9% improvement in the jank ratio and a 20.3% improvement in FPS on average. Therefore, Fleet can deliver a user experience comparable to the original Android.

CPU usage. We also measure the CPU usage of each app. To do this, we launch an app, use it for 30 seconds, switch it to the background for 30 seconds, and repeat this procedure. Throughout this process, we utilize the system trace [9] to record the CPU usage. In terms of total CPU time cost, Fleet performs worse than Android by an average of 0.18%, but it performs better than Marvin by an average of 3.21%.

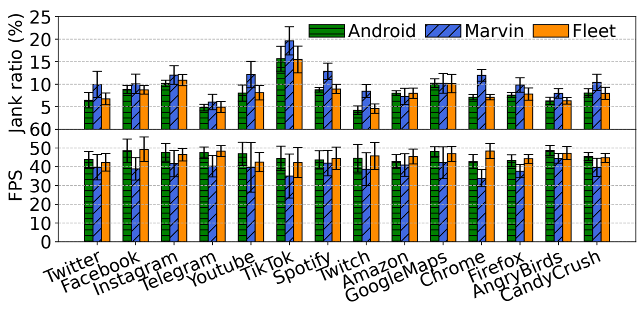


Figure 14. Frame rendering performance analysis with jank ratio and frame per second (FPS).

Most of the CPU overhead is attributed to the GC threads, where Fleet incurs an additional 0.16% CPU time compared to Android on average.

Memory overhead. Fleet introduces an additional card table fixed at 4 MB, which is proportional to the 4 GB heap size. The memory occupancy of the card table can be reduced by reclaiming its empty pages. On the other hand, Fleet can efficiently offload cold objects that are larger than the card table. As a result, Fleet can decrease the overall memory usage with minimal memory overhead.

Power consumption. We use the Monsoon Power Monitor [11] to measure the power consumption of Fleet and Android for our tested apps. Each app is used for one minute in the foreground and one minute in the background, and we measure the average energy consumption during usage. The average power consumption of Fleet is 1851 ± 143 mW. For Android, it is 1817 ± 197 mW. Within the standard error, the power consumption of Fleet and Android is similar.

7.4 Sensitivity study on the background heap size

ART utilizes a dynamic heap size scheme that adjusts the heap limit based on the currently allocated memory size after GC. In this section, we analyze the impact of the heap size by setting the updated heap size to 1.1× and 2× the allocated memory size after GC when the app is in the background. We evaluate the caching capacity and hot launch performance for these two heap size schemes in Fleet and Android.

Regarding caching capacity, a relatively smaller heap size leads to higher app caching improvement in Fleet. When the heap size scheme is configured to 1.1×, Fleet's caching capacity improves by approximately 20%. However, when the scheme is configured to 2×, Fleet's caching capacity becomes similar to that of Android. This is because a larger heap size limit allows for more garbage background objects, resulting in inefficient BGC in Fleet.

As for the hot launch time, Fleet demonstrates consistent performance across these two heap size schemes. In contrast, Android exhibits significant variation in hot launch times across these two heap size schemes. For instance, compared to the heap size configured to 2×, the 1.1× configuration is about 31% faster in Android. Therefore, Fleet's hot launch performance is more robust than that of Android when considering different heap size configurations. This is because Fleet accurately identifies and swaps objects in the runtime layer, considering the next hot launch, whereas the original Android uses a generic swap approach.

8 Related Work

GC in the runtime. Fleet addresses four challenges in the design of its GC method: (i) Avoid paging during GC. There are primarily two approaches to coordinating GC and swap. First, one method involves recording the outcome references of swapped objects [21, 27, 32, 43]. These methods require recording the reference information before swapping objects. During the tracing procedure, these methods utilize the recorded reference information to traverse the object reference graph without accessing the objects. Second, another approach is to control the range of GC [30, 46, 47]. Fleet is a method that controls the range of GC. To the best of authors' knowledge, Fleet is the first method that controls the range of GC by employing a scheme that differs between foreground and background states; (ii) Identify hotness for objects [21, 28, 43, 50]. The key distinction between Fleet and these methods lies in the consideration of the distinct hotness patterns observed in Android, such as foreground and background usage; (iii) Reorganize objects through copy-based GC [28, 50]. These works aim to consolidate small objects with similar access patterns. Fleet groups the objects that need to be accessed during the hot-launch; (iv) Application-aware GC. This term refers to various efforts that aim to co-design the GC scheme with specific applications. Examples of such works include those focused on big data systems [37, 38], disaggregated memory [24, 36, 46, 47], and non-volatile memory [40, 49]. The objective of these methods is to design an improved GC tailored to the specific workload and requirements of each application. Fleet shares a similar objective but focuses specifically on mobile apps.

Swap in mobile devices. Swap mechanisms are widely used in mobile devices. SEAL [33] introduces a two-level swap mechanism for mobile devices. Kwon et al. [31] propose a compressed swap scheme for mobile GPU memory to increase the caching capacity of apps. ASAP [41] improves hot-launch performance by employing a prefetching method. However, these methods consider GC threads as regular program execution and fail to address the adverse effects of GC, which is a crucial issue in Android devices. Since GC moves objects to different pages, estimating access patterns at a page level will be interference.

Cross layer memory management. Several works propose a cross-layer design for memory management [25, 32, 34, 48]. Marvin [32] is the first to co-design GC and swapping for mobile systems. Marvin utilizes a stub-based method to record references to swapped-out objects, which improves the caching capacity for apps with large objects. However, it causes significant degradation in hot-launch performance due to two problems. First, it cannot efficiently process small objects, which constitute the majority of objects in Android apps. Second, it requires long stop-the-world pauses to maintain the reference information. Acclaim [34] proposes a cross-layer solution by passing the app state to the kernel and adjusting memory management.

9 Conclusion

The app launch time is critical to the user experience on mobile devices. However, existing approaches fall short in optimizing the hot-launch performance while maximizing the number of cached apps. Based on the observation of foreground/background object characteristics, this paper presents Fleet, a fore/background-aware GC-swap co-design framework that guides the swap with the foreground object access pattern and restricts the GC tracing range to the background objects. The evaluation reveals that, on average, Fleet achieves a 1.59× faster hot-launch time and caches 1.21× more apps than Android.

Acknowledgments

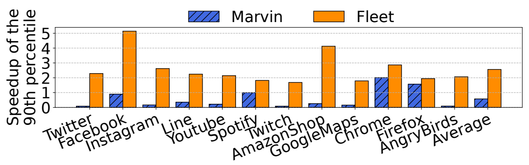
We are grateful to our shepherd and the anonymous reviewers for their constructive comments and suggestions. The work was supported in part by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CityU 11209122), the Thailand MHESI Research Grant for New Scholars (RGNS 64-091), the National Key Research and Development Program of China (No. 2022YFB3104502), the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant CARCH A20211, the Wuhan Science and Technology Joint Project for Building a Strong Transportation Country (No. 2023-2-7), the CCF-Phytium Fund, and the National Science Foundation (NSF) Grant (No. 2105006).

A Hot Launch Performance Appendix

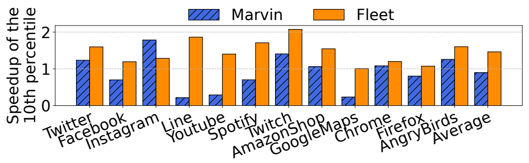
In this appendix, we present 3 additional characteristics of the hot launch time (Figure 15) and hot launch distributions for the remaining 6 commercial apps (Figure 16).

Figure 15 supplements 3 statistical characteristics: (a) the tail hot-launch time at the 90th percentile, (b) the minimum hot-launch time at the 10th percentile, and (c) the mean hot-launch time and its corresponding standard deviation. We calculate the performance speedup relative to Android. The results indicate that Fleet improves the hot-launch performance for all 3 statistical measures. Specifically, Fleet optimizes the tail hot launch time more effectively. Fleet shows an average improvement of more than 2.5× in optimizing the 90th percentile tail hot-launch time.

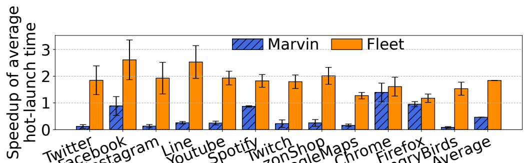
Figure 16 shows that Fleet provides faster hot launch times for most apps, except for Candy Crush Saga. This discrepancy is because Candy Crush Saga has a small percentage of Java heap (only 4%). Fleet is primarily optimized for memory in the Java heap, making it more suitable for apps with a larger percentage of Java heap.



(a) The 90th percentile hot-launch time

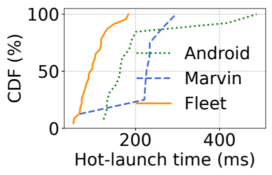


(b) The 10th percentile hot-launch time

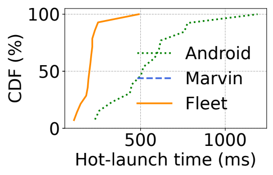


(c) Average hot-launch time with standard deviation

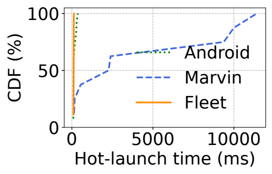
Figure 15. The hot-launch speedup over Android under high memory pressure.



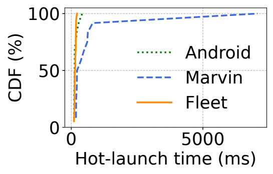
(a) Telegram



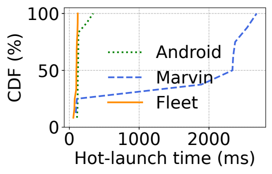
(b) TikTok



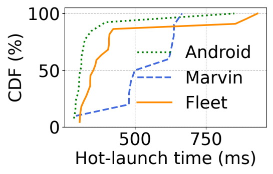
(c) Rave



(d) BigoLive



(e) LinkedIn



(f) Candy crush soga

Figure 16. The hot-launch performance distribution of the 6 other commercial apps under high memory pressure.

B Artifact Appendix

B.1 Abstract

Fleet is a co-design of Android Runtime (ART) and Linux kernel, enabling more efficient memory management for mobile devices. The artifact provides the source code of our modified ART and Linux kernel. Additionally, this artifact includes testing scripts and Jupyter notebooks for collecting data and reproducing the key experimental results reported in the paper. These results encompass app caching capacity, hot launch performance, and runtime performance.

For the evaluation, it is necessary to have a Pixel 3 mobile device and a development workstation running 64-bit Linux distribution with more than 64 GB DRAM and 800 GB free disk space. The mobile device should be connected to the workstation using a USB data cable.

B.2 Artifact check-list (meta-information)

- **Program:** Modified ART and Linux kernel, Python and Bash scripts, Jupyter Notebooks.
- **Compilation:** Java Development Kit 8, Python 3, GNU C Library.
- **Run-time environment:** Android 10 for the device and Ubuntu 22.04 for the development workstation.
- **Hardware:** Pixel 3 mobile phone.
- **Metrics:** App caching capacity, GC working set size, hot launch performance, CPU usage, and frame rendering performance.
- **Output:** The console and system trace tool output raw data, and Jupyter Notebook is used to analyze and generate visualized results.
- **Experiments:** Python and Bash scripts are used for collecting data, and Jupyter Notebooks are used for analysis.
- **How much disk space required (approximately)?:** About 800 GB, mainly for building Fleet and other compared Android systems.
- **How much time is needed to prepare workflow (approximately)?:** About 12 hours.
- **How much time is needed to complete experiments (approximately)?:** About 12 hours.
- **Publicly available?:** Yes.
- **Archived(provide DOI)?:**https://doi.org/10.5281/zenodo.10878114

B.3 Description

B.3.1 How to access. In addition to the archived version, we maintain a public GitHub repository: https://github.com/jiachengh/Fleet

B.3.2 Hardware dependencies. To reproduce the results reported in the paper, we require a Google Pixel 3 mobile device. Additionally, a development workstation is necessary for building Android images and interacting with the device. It is recommended that the development workstation have 64 GB of DRAM and 800 GB of storage space, primarily for building Android systems. The workstation should be an AMD64 architecture machine.

B.3.3 Software dependencies. We recommend that the development workstation run on Ubuntu 22.04, but other similar 64-bit Linux distributions should also work. The development environment should include GNU C Library (glibc) 2.17 or later. To interact with the device, the workstation should also have some development tools, including Android Debug Bridge (ADB) and fastboot. Additionally, the Pixel 3 mobile device should be configured with its bootloader unlocked.

B.3.4 Data sets. The artifact provides installation packages for all the tested commercial apps listed in Table 3. These apps can be found in the `evaluation/APK` folder. Additionally, there are installation scripts available in the `evaluation/install-apk` folder to assist users in installing these apps on their mobile devices.

B.4 Installation

Our artifact includes a `README.md` file that provides a more detailed procedure for building and installing Fleet and the other baselines. The installation process for building Android systems is divided into three main steps.

First, we need to obtain the source code of Android by downloading the basic source code and making our modifications. The artifact provides our modified code in the `src` folder. To incorporate these files into the original AOSP and kernel, we can run the `apply-modification.sh` script.

Second, after updating the source code, we need to build Android system images from it. This involves building the kernel and the AOSP. We can utilize the scripts named `build_*.sh` provided in the artifact.

Third, once we have built Android images, users can use the scripts provided in the artifact to flash the compiled images to the Pixel 3 device using the `fastboot*.sh` scripts. Additionally, we can use corresponding scripts provided by the artifact to set up the configuration for our system, including configuring the swap partition and installing the Google services framework and apps.

Moreover, the artifact also provides pre-built system images. Users can directly flash the corresponding pre-built images to the device using the `fastboot*.sh` scripts.

B.5 Experiment workflow

The artifact includes all experiment-related files in the `evaluation` folder. Additionally, there is a `EXPERIMENTS.md` document that provides a more detailed description of the evaluation. To reproduce the key results reported in the paper, this section will include the following three experiments.

B.5.1 App caching and GC working set. All the relevant files are located in the `evaluation/exp-cache-app` folder.
Figure 11c. To begin, run the following command from the `evaluation` folder:

```
python ./exp-cache-app/run-cache-commercial.py
```

This Python script will continuously start the apps and count the number of active apps. The number of active apps can be observed in the console after the keyword `CachingNUM=`. When the script finishes, it will print the summary of the cached app numbers in the last line, which begins with `cached_numbers=`. Finally, users need to paste these summary results into the `exp-run-caching-commercial.ipynb` notebook to generate the result figure.

Figure 12. This result requires additional profile information, which can be obtained by enabling the debug flag before compiling Fleet. Therefore, users need to rebuild Fleet to enable profile information. For each comparison of configuration, users should collect the working set alongside the execution time using two terminals. One terminal is used to save the logs printed by `adb logcat`. The other is used to execute the app-running scripts `run-gc-working-set.py`.

To produce Figure 12a, users should retrieve the background GC working set logs from the log files saved in the first terminal and calculate the statistical results. Finally, users could input these results to the `exp-gc-working-set.ipynb` notebook and plot the figure.

To produce Figure 12b, the artifact provides some analysis code blocks in the `exp-gc-working-set.ipynb` notebook. Users need to update the log paths of enabling BGC and disabling BGC in the notebook. Then the notebook can automatically analyze the log files and plot the result figure.

B.5.2 Hot launch performance.

Users could use the following command to continuously launch apps and obtain the hot and cold launch times in the console:

```
python ./exp-hot-launch/run-launch.py
```

If there is not enough hot launch data for some apps, users can adjust the app startup sequence and number of startups in the script.

Once the script finishes, it will print a summary of the hot launch times for every app. Users need to copy the summary hot launch time from the console and paste it into the `exp-hot-launch.ipynb` Jupyter Notebook.

Figure 13a-m. After all the hot launch records in `exp-hot-launch.ipynb` have been updated, we can run the all code blocks in the notebook. The notebook will generate the CDF graph of hot launch times for all apps and the statistical graph.

Figure 13n. Users can use the `check-heap.sh` script to get the Java heap size and the total memory footprint of apps. Then, the ratio of the Java heap can be calculated by dividing the Java heap size by the total memory of the app.

Finally, users could write the ratio of Java heap and the hot launch speedup in the corresponding code block in `exp-hot-launch.ipynb`, which will generate Figure 13n.

B.5.3 Runtime performance.

To obtain the runtime information of apps, the experiments mainly consist of two steps: (i) utilize the `system trace` tool to capture traces; (ii) analyze these captured traces by the Perfetto Python library.

CPU usage. First, users should turn on the `system trace` while simultaneously executing `run-once-cpu.py`. When the Python script finishes, users should close the `system trace` and fetch the captured trace from the mobile device to the development workstation. After obtaining the trace file, users can update the `TRACE_PATH` variable in the `exp-runtime-performance.ipynb` notebook. Finally, users can execute blocks of the notebook, which analyze the trace file using Perfetto and output the CPU usage result in the last code block.

Figure 14. Similarly, users should start the `system trace` while initiating the `run-once-foreground.py` script. After the script finishes, users close the `system trace` and fetch the trace file from the device. Then, users can use the `exp-runtime-performance.ipynb` notebook to analyze the trace by updating the `TRACE_PATH` to the trace file. After running all the blocks in the notebook, it will generate the jank ratio and FPS results. Users could paste these results into the `exp-runtime-performance-plot.ipynb` notebook which incorporates the script to plot the resulting figure.

B.6 Evaluation and expected results

During these experiments, users can obtain raw data from the console and traces captured by the system trace tool. Furthermore, the artifact provides Jupyter notebooks to analyze the raw data and generate figures.

Please note that due to differences in app usage, such as daily content and individual accounts, we do not anticipate identical results, but we expect the results to exhibit similar trends as those reported in the paper.

References

[1] Milliseconds make millions: A study on how improvements in mobile site speed positively affect a brand's bottom line. https://www2.deloitte.com/content/dam/Deloitte/ie/Documents/Consulting/Milliseconds_Make_Millions_report.pdf.

[2] Response times: The 3 important limits. https://www.nngroup.com/articles/response-times-3-important-limits/.

[3] Marvin code, 2019. https://github.com/UWSysLab/marvin-code/ [Accessed: 20-Nov-2023].

[4] Oppo introduces new memory expansion technology for its reno5 series, a94 and a74 series smartphones, 2021. https://www.oppo.com/sg/newsroom/press/oppo-introduces-new-memory-expansion-technology/ [Accessed: 20-Nov-2023].

[5] How to use xiaomi virtual ram to speed up your device?, 2022. https://xiaomiui.net/how-to-use-xiaomi-virtual-ram-to-speed-up-your-device-31416/ [Accessed: 20-Nov-2023].

[6] What is ram plus and how to use it?, 2022. https://www.samsung.com/sg/support/mobile-devices/what-is-ram-plus-and-how-to-use-it/ [Accessed: 20-Nov-2023].

[7] Android debug bridge (adb) tool, 2023. https://developer.android.com/studio/command-line/adb [Accessed: 20-Nov-2023].

[8] App startup time | app quality | android developers, 2023. https://developer.android.com/topic/performance/vitals/launch-time [Accessed: 20-Nov-2023].

[9] Capture a system trace on a device, 2023. https://developer.android.com/topic/performance/tracing/on-device/ [Accessed: 20-Nov-2023].

[10] Debugging art garbage collection, 2023. https://source.android.com/docs/core/runtime/gc-debug/ [Accessed: 20-Nov-2023].

[11] High voltage power monitor | monsoon solutions | bellevue, 2023. https://www.msoon.com/high-voltage-power-monitor/ [Accessed: 20-Nov-2023].

[12] Memory allocation among processes, 2023. https://developer.android.com/topic/performance/memory-management/ [Accessed: 20-Nov-2023].

[13] Memory swapping: What it is and how it works, 2023. https://forum.huawei.com/enterprise/en/memory-swapping-what-it-is-and-how-it-works/thread/696666286131658752-667213860488228864/ [Accessed: 20-Nov-2023].

[14] Perfetto - system profiling, app tracing and trace analysis, 2023. https://perfetto.dev/ [Accessed: 20-Nov-2023].

[15] Ui jank detection, 2023. https://developer.android.com/studio/profile/jank-detection/ [Accessed: 20-Nov-2023].

[16] ssvb/tinymembench: Simple benchmark for memory throughput and latency, 2024. https://github.com/ssvb/tinymembench [Accessed: 10-Mar-2024].

[17] Welcome to fio's documentation!, 2024. https://fio.readthedocs.io/en/latest/index.html [Accessed: 10-Mar-2024].

[18] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. Write-rationing garbage collection for hybrid memories. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 62–77. ACM, 2018.

[19] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In Mamdouh Ibrahim and Satoshi Matsuoka, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, pages 1–12. ACM, 2002.

[20] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 22–32. ACM, 2008.

[21] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In Gail E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 109–126. ACM, 2008.

[22] Stuart K Card, George G Robertson, and Jock D Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI Conference on Human factors in computing systems*, pages 181–186, 1991.

[23] Tao Deng, Shaheen Kanthawala, Jingbo Meng, Wei Peng, Anastasia Kononova, Qi Hao, Qinhao Zhang, and Prabu David. Measuring smartphone usage and task switching with log tracking and self-reports. *Mobile Media & Communication*, 7(1):3–23, 2019.

[24] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI), Monterey, California, USA, November 14-17, 1994*, pages 229–241. USENIX Association, 1994.

[25] Weichao Guo, Kang Chen, Huan Feng, Yongwei Wu, Rui Zhang, and Weimin Zheng. *MARS*: Mobile application relaunching speed-up through flash-aware page swapping. *IEEE Trans. Computers*, 65(3):916–928, 2016.

[26] Sangwook Shane Hahn, Sungjin Lee, Inhyuk Yee, Donguk Ryu, and Jihong Kim. {FastTrack}: Foreground {App-Aware} {I/O} management for improving user experience of android smartphones. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 15–28, 2018.

[27] Matthew Hertz, Yi Feng, and Emery D Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 143–153, 2005.

[28] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J. Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 69–80. ACM, 2004.

[29] Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. Application-aware swapping for mobile systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

[30] Iacovos G Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S Zakkak, Polyvios Pratikakis, and Angelos Bilas. Teraheap: Reducing memory pressure in managed big data frameworks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 694–709, 2023.

[31] Sejun Kwon, Sang-Hoon Kim, Jin-Soo Kim, and Jinkyu Jeong. Managing gpu buffers for caching more apps in mobile systems. In *2015 International Conference on Embedded Software (EMSOFT)*, pages 207–216. IEEE, 2015.

[32] Niel Lebeck, Arvind Krishnamurthy, Henry M Levy, and Irene Zhang. End the senseless killing: Improving memory management for mobile operating systems. In *2020 {USENIX} Annual Technical Conference ({USENIX} {ATC} 20)*, pages 873–887, 2020.

[33] Changlong Li, Liang Shi, Yu Liang, and Chun Jason Xue. SEAL: user experience-aware two-level swap for mobile devices. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 39(11):4102–4114, 2020.

[34] Yu Liang, Jinheng Li, Rachata Ausavarungnirun, Riwei Pan, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Acclaim: Adaptive memory reclaim to improve user experience in android systems. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, pages 897–910, 2020.

[35] Xinze Liu, Yubo Zhang, Ziqian Yan, and Yan Ge. Defining 'seamlessly connected': user perceptions of operation latency in cross-device interaction. *International Journal of Human-Computer Studies*, 177:103068, 2023.

[36] Martin Maas, Krste Asanovic, Tim Harris, and John Kubiatowicz. Taurus: A holistic language runtime system for coordinating distributed managed-language applications. In Tom Conte and Yuanyuan Zhou, editors, *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2016, Atlanta, GA, USA, April 2-6, 2016*, pages 457–471. ACM, 2016.

[37] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. Yak: A high-performance big-data-friendly garbage collector. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 349–365, 2016.

[38] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, Jianfei Hu, and Guoqing Xu. FACADE: A compiler and runtime for (almost) object-bounded big data applications. In Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas, editors, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2015, Istanbul, Turkey, March 14-18, 2015*, pages 675–690. ACM, 2015.

[39] Vijay Janapa Reddi, Hongil Yoon, and Allan Knies. 2 billion devices and counting: An industry perspective on the state of mobile computer architecture. *IEEE Micro*, 38:6–21, 2018.

[40] Thomas Shull, Jian Huang, and Josep Torrellas. Autopersist: an easy-to-use java NVM framework based on reachability. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 316–332.

ACM, 2019.

[41] Sam Son, Seung Yul Lee, Yunho Jin, Jonghyun Bae, Jinkyu Jeong, Tae Jun Ham, Jae W Lee, and Hongil Yoon. {ASAP}: Fast mobile application switch via adaptive prepaging. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 365–380, 2021.

[42] Darko Stefanovic, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In Brent Hailpern, Linda M. Northrop, and A. Michael Berman, editors, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1999, Denver, Colorado, USA, November 1-5, 1999*, pages 370–381. ACM, 1999.

[43] Yan Tang, Qi Gao, and Feng Qin. Leaksurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In Rebecca Isaacs and Yuanyuan Zhou, editors, *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, pages 307–320. USENIX Association, 2008.

[44] Niraj Tolia, David G Andersen, and Mahadev Satyanarayanan. Quantifying interactive user experience on thin clients. *Computer*, 39(3):46–52, 2006.

[45] Po-Hsien Tseng, Pi-Cheng Hsiu, Chin-Chiang Pan, and Tei-Wei Kuo. User-centric energy-efficient scheduling on multi-core mobile devices. In *The 51st Annual Design Automation Conference 2014, DAC '14, San Francisco, CA, USA, June 1-5, 2014*, pages 85:1–85:6. ACM, 2014.

[46] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A memory-disaggregated managed runtime. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 261–280, 2020.

[47] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. Memliner: Lining up tracing and application for a far-memory-friendly runtime. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 35–53. USENIX Association, 2022.

[48] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: transparent memory offloading in datacenters. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 609–621. ACM, 2022.

[49] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 70–83, 2018.

[50] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. Improving program locality in the GC using hotness. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 301–313. ACM, 2020.

[51] Ting Yang, Emery D Berger, Scott F Kaplan, and J Eliot B Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 103–116, 2006.

[52] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. Low-latency, high-throughput garbage collection. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 76–91. ACM, 2022.

[53] Kan Zhong, Duo Liu, Liang Liang, Xiao Zhu, Linbo Long, Yi Wang, and Edwin Hsing-Mean Sha. Energy-efficient in-memory paging for smartphones. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1577–1590, 2015.

[54] Xiao Zhu, Duo Liu, Kan Zhong, Jinting Ren, and Tao Li. Smartswap: High-performance and user experience friendly swapping in mobile systems. In *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, pages 22:1–22:6. ACM, 2017.