SEAL: User Experience-Aware Two-Level Swap for Mobile Devices

Changlong Li, *Member, IEEE*, Liang Shi, Yu Liang, and Chun Jason Xue

***Abstract*—App caching is important for mobile devices, which enables fast switching and state restoration of apps by caching all the pages in memory. Memory swapping can improve app caching capability by evicting pages to the secondary storage. However, enabling memory swapping could induce jitters in interactions, which significantly degrades the user experience. As a result, storage-based swapping is disabled by default in most mobile devices. This article proposes a novel swap framework, SEAL, a user experience-aware two-level swapping, which maximizes the benefits of memory swapping and minimizes the negative impact on user experience in interactions. Inspired by a study on the access characteristics of a set of popular apps on mobile devices, the framework adopts compressed memory as the first swap level (SL1) and secondary storage as the second swap level (SL2). To optimize user experience comprehensively, three schemes are proposed. First, a novel page identification scheme is proposed to guide the page placement between these two levels. Second, a hidden page loading (HPL) scheme is proposed to load pages from SL2 to SL1 for optimized user experience during app execution. Finally, an app-granularity swapping scheme is proposed to swap data in the unit of apps. Experiments on real devices show that app caching capability is improved by 2.43× on average when enabling SEAL while minimizing the negative impact on user experience.**

***Index Terms*—App caching, memory management, mobile system, swap, user experience.**

I. Introduction

MOBILE devices play an important role in everyone's daily life. To achieve fast switching and state restoration, apps are cached by maintaining all of their pages in the main memory. App caching requires a significant amount of memory resources. However, memory capacity is often constrained in mobile devices, as it results in high costs and poor energy efficiency [1]. Memory swapping could be an effective approach to resolve the conflict between app caching and limited memory capacity [2]. By swapping out infrequently accessed pages to secondary storage, memory pressure is effectively alleviated.

Several works target swap mechanisms on mobile devices. For instance, MARS [3] is designed to speed up app launching through flash-aware swapping. It isolates garbage collection (GC) from page swapping and employs several flash-aware techniques to speed up app launching. SmartSwap [4] presents a prediction-based process-level swap mechanism. In their design, victim processes are swapped to Flash ahead-of-time, which significantly improves efficiency. However, when swapping is enabled, the user experience during app execution will be seriously impacted, which is overlooked by existing works. Specifically, many perceptible display jitters are detected when an app is used. As revealed in ProfileDroid [5], the interaction-intensive apps may generate more than 20 input-events per second during execution. It results in more than 30% time for human-screen interactions when using apps, such as browsing and gaming.

Frame rate plays an important role in the user-device interaction experience. In general, drawing 60 frames per second is required to fulfill a smooth experience based on human sensitivity [6]. An important metric, "interaction alert" has been used to measure user experience based on frame rate detection. An interaction alert is defined as an event that one frame rendering takes more than 16.6 ms [7]. Maintaining a low interaction alert rate is critical to enhancing the user experience. The study in this article finds that the number of interaction alert increases significantly when memory swapping is enabled. This is because the swap induced I/O pressure could lead to task suspensions. As the commonly deployed storage medium, the I/O speed of Flash, including UFS and eMMC, is still the performance bottleneck in most mobile systems [8]. Due to these issues, enabling swapping without affecting user experience is a challenge for mobile devices.

This article proposes a novel framework, SEAL, a user experience-aware two-level swap for mobile devices. The design of the framework is inspired by our study on the access characteristics of a set of popular apps. The study shows an interesting observation: only a subset of memory pages is required during app launching, while more pages are needed during the app execution phase. Based on this observation, the framework reorganizes the conventional swap partition (SP) into two levels: 1) SL1 and 2) SL2. SL1 is designed with the main memory based on compression, and SL2 is designed with the secondary storage. Data accessed during app launching is placed in the first level, and data accessed during app execution

Manuscript received April 17, 2020; revised June 17, 2020; accepted July 6, 2020. Date of publication October 2, 2020; date of current version October 27, 2020. This work was supported in part by the grant from the NSFC under Grant 61772092, and in part by the Research Grants Council of the Hong Kong Special Administrative Region, China under Project CityU 11219319. This article was presented in the International Conference on Embedded Software 2020 and appears as part of the ESWEEK-TCAD special issue. *(Corresponding author: Liang Shi.)*

Changlong Li is with the School of Computer Science and Technology, East China Normal University, Shanghai 200241, China, and also with the Department of Computer Science, City University of Hong Kong, Hong Kong.

Liang Shi is with the School of Computer Science and Technology, East China Normal University, Shanghai 200241, China (e-mail: shi.liang.hk@gmail.com).

Yu Liang and Chun Jason Xue are with the Department of Computer Science, City University of Hong Kong, Hong Kong.

Digital Object Identifier 10.1109/TCAD.2020.3012316

is placed in the second level. Since the first level is fast enough without I/O, it can be accessed very fast. However, to access the second level, the user experience impact will need to be considered. To ensure smooth user experience, several challenges are addressed in the design of this framework: first, the data placed in SL1 should be identified ahead of time. Second, if data is read from SL2, its access should be well controlled to reduce interaction alerts. Finally, the performance of data migration between SL1 and SL2 should be optimized. Solving the above challenges, this article makes the following contributions.

1) The user experience of mobile devices is analyzed with and without swapping. A set of popular apps is studied with their access characteristics on the SP.
2) A novel data identification scheme is proposed to determine the placement of data between two levels of the SP. By identifying and placing the data requested during app launching at SL1, the launch time user experience is optimized.
3) An HPL scheme is proposed to minimize the impact of I/O requests from SL2. By overlapping the app launching latency with data loading from SL2, the execution time user experience is optimized.
4) An app-granularity swapping (AGS) scheme is proposed to control the swap operations. Based on this technique, pages belonging to an inactive app can be swapped in batch, which effectively improves the I/O throughput between SL1 and SL2.
5) Experiments on real devices show that app caching capability is improved by 2.43× on average when enabling SEAL. Meanwhile, the interaction alerts are reduced by 76% compared with the state-of-the-art.

The remainder of this article is organized as follows. Section II presents the background and user experience analysis. Section III presents SEAL overview. The design and implementation details are presented in Section IV. Experiments are presented in Sections V and VI. Related works are presented in Section VII. Finally, this article is concluded in Section VIII.

II. Background and Problem Statement

This section first introduces the background of app caching and swap mechanisms in mobile devices. Then the user experience with and without memory swapping is analyzed, respectively. Finally, the technical challenges are discussed.

A. Background

1) App Caching and Warm Launching: Mobile platforms prefer to cache apps in memory when switching them to the background, rather than release their pages straightforward. This app caching feature makes it possible to *warm launch* instead of *cold launch* an app. When launching an app after a boot-up, the mobile platform first creates a process for the app, then launches UI, which is known as main activity. Such launching style is referred to as cold launch. With app caching, a number of apps can reside in memory after execution. A cached app can be switched to the foreground when it is launched again. Such launching style is called warm launch. Warm launching is more friendly to users, as it avoids having to repeat object initialization, layout inflation, and rendering. How many apps could be warm launched from the background in maximum is always adopted to assess app caching capability of a mobile device.

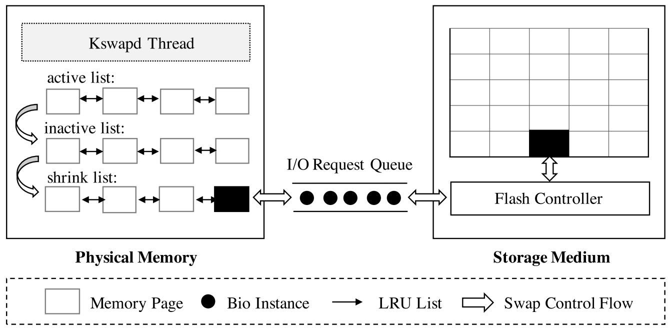


Fig. 1. Memory swapping on the mobile platform.

App caching and warm launching bring benefits from three aspects. First, the launching speed of an app is significantly improved. More importantly, app caching makes it possible to maintain app states, such as play progress of YouTube or navigation record of Google Maps. Mobile systems, such as Android manages the life cycle of every app and provides APIs to save the app-specific state. These states can be used to restore the app during warm launching, as corresponding pages are not released from memory. In addition, the warm launching is more energy efficiency. As a result, the app caching capability has become an important metric of mobile devices.

2) Mobile Memory Swapping: Memory swapping extends the volume of memory by allocating an additional partition at secondary storage. The architecture is as shown in Fig. 1. Based on this technique, the mobile system can evict the pages of background apps from physical memory to the SP, or bring the requested pages back. The process of page evicting from memory to partition is called *swapout*, and the process to bring pages back is named *swapin*. The swapout operation starts when memory is under pressure. Specifically, a kernel thread *kswapd* is wakened up periodically or by page allocation to monitor the memory watermark [9]. It swaps pages out when the size of available memory is lower than the watermark threshold. On the other hand, swapin is triggered when swapped pages are requested again.

There are many strategies to select victim pages for swapping. For instance, the least-recently used (LRU) policy in Linux kernel defines LRU pages as inactive, and moves them to the shrink-list as candidates. When swapping occurs, these candidate pages will be evicted to the SP in priority. On the other hand, the management of SP determines where candidate pages swap to. In mobile devices, Flash is the first choice to build SP, since it is commonly deployed and its capacity is much larger than the memory. To realize data migration between the main memory and the secondary storage, swapped pages are converted to `bio` instance, which represents an inflight block I/O request in the kernel [10]. These requests are transferred to the queue in the block layer and finally submitted to the storage.

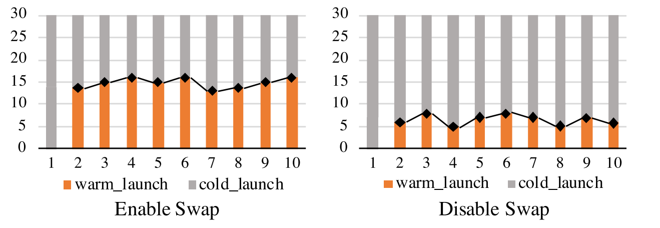


Fig. 2. App caching capability when enabling and disabling the swap mechanism. Seven apps are warm launched without swapping. x-axis represents the ten rounds testing, and y-axis represents the number of launched app.

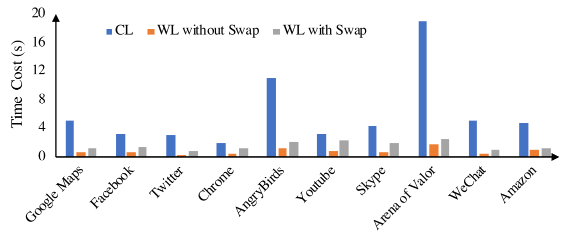


Fig. 3. App launch time statistics. The warm launching speed slows down, compared with the swap disabled case. Even though it is still faster than the cold launching.

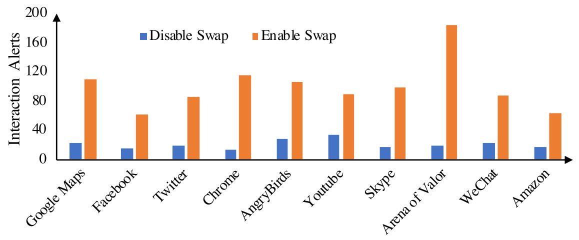


Fig. 4. Interaction alert statistic during app execution.

Memory swapping is a critical approach to enhance the app caching capability. In original, some of the cached apps will be killed to release space when the available memory is exhausted. The killed apps can only be cold launched when switched to the foreground again. Thanks to swapping, most of app killing behaviors are avoided since the data can be evicted out before memory exhausting. As a result, app caching capability could be enhanced.

B. *User Experience Analysis*

Maintaining good user experience is often a challenge for modern mobile devices, no matter the swap mechanism is enabled or not. When swapping disabled, more cold launched are needed. When swapping enabled, the warm launching rate is increased. However, the increased I/O pressure between memory and storage brings two additional impacts on user experience. First, the warm launching speed is slower. What is worse, even when an app is warm launched successfully, it may also suffer interaction alert in the following execution. The smooth of app usage seriously deteriorated. To understand how severe the user experience problem is, this article conducts evaluations on real-world mobile devices.

User Experience When Swap Is Disabled: This article first evaluates whether the user experience is good enough without memory swapping. In the evaluation, 30 popular apps, including Facebook, Twitter, and Chrome are preinstalled and repeatably launched for ten rounds. In the first round, all apps are cold launched. In the following nine rounds, several apps are successfully cached in memory. How many apps are cached in each round is recorded. Both swapping enabled and disabled cases are tested for comparison. A Linux swap file (2 GB) is created as a partition used for swapping in the swap enabled case. Fig. 2 shows the collected results. It shows that when swap is enabled, around 15 apps can be cached and warm launched. However, for the disabled case, only 7 out of the 30 apps are warm launched from the memory. As investigated, people often use 13 apps or more on a daily basis [11]. It means almost half of earlier started apps are cold launched when restarting again, which leads to higher launch latency and history-state loss. According to the above analysis, the app caching capability without memory swapping cannot support the best user experience. In conclusion, to enable swapping on mobile systems is important in terms of user experience.

User Experience When Swap Is Enabled: This article further explores user experience study when enabling swapping. 30 apps are repeatedly launched for experiments. Results show that most apps' state is maintained by the system since the app caching capability is significantly improved. However, the introduced swap mechanism brings two additional problems. First, the speed of warm launching slows down. This is because a large number of pages are read from the storage. We measured the amount of time needed to warm launch an app when enabling swapping. Fig. 3 shows the results for the latency on cold launching, warm launching with and without swap. The results show that it takes $2-6\times$ longer to warm launch an app when swap enabled, even though it is still much faster than the cold launching.

What is worse, the user experience could be seriously deteriorated during app execution. Many interaction operations, such as screen scrolling, become jerky and slow. As we know, drawing screen frames with a consistent rate is essential for a good experience. In general, drawing 60 frames per second (60 fps) is required by many commercial smartphones [6]. It is based on the human eye sensitivity. If an app drops out of this rate, the display can become jerky or slow from the users' perception. To quantitatively analyze the user experience during app execution, Systrace [12] is deployed to detect the interaction alerts. Interaction alerts happen when the time spent rendering a frame exceeds 16.6-ms time limit required to maintain a stable 60 fps [7]. Since the space is limited, Fig. 4 only shows the interaction alert statistic of ten sampled apps, including different types of apps. It shows that the number of interaction alerts when enabling swap is around six times more than the swap disabled case. Logs are traced to assess where each interaction alert comes from. For example, one frame spent 19.8 ms to display on the screen. It is found that CPU takes only 4.127 ms, while more than 15 ms is spent in the queue waiting for I/O. The time consumption to draw a frame is significantly increased when a request I/O cannot respond on time. To the best of our knowledge, the swap induced interaction alert issue has not been noticed and effectively addressed yet.

In conclusion, swapping is necessary for mobile devices since it can significantly improves app caching capability. However, it brings two additional problems: high latency of app launching and interaction alerts during app execution. This article tries to enable swap and address the above issues so that the user experience can be improved comprehensively.

C. Technical Challenges

There are two challenges to minimize user experience deterioration when enabling swap.

1) High-volume I/O Induced High Latency When Launching App: I/O is still the performance bottleneck in current mobile devices [13]–[16]. This article finds that the increased warm launching latency when enabling swapping is due to the increase of I/O pressure. A large number of earlier evicted pages are requested to swapin when an app is switched to the foreground again. Existing solutions, such as LRU or least-frequently used (LFU), select the least recently (frequently) used pages to swap based on the usage history. Unfortunately, there are still launch-required pages swapped out, which leads to the I/O congestion. It is because many launching required pages are not accessed again in the following app usage phase. These pages are more likely to be identified as 'inactive' and be evicted to the storage. According to the above analysis, modern solutions are not friendly to the launch time user experience.

2) Small I/O Induced Interaction Alerts When Interacting With App: As illustrated in Fig. 4, a large number of interaction alerts are detected during app execution. This is because many tasks are suspended due to swap induced I/O requests. It is too late to swapin a page when page fault has already been triggered. It is hard to predict exactly when and which pages will be swapped in during interaction. And rarely used page does not mean that the page will never be accessed again. Most requested pages are small. As detected, even a small I/O request (4 kB) may lead to a severe user experience deterioration. This is because existing systems cannot control the response latency of a request: too many factors interfere with the I/O performance, such as queue in the block layer and GC inside the storage medium. Thus, how to minimize interaction alerts when enabling swapping is the other challenge to overcome.

III. Two-Level Swap Framework

In this section, the observation is first presented to show the page access characteristic of each app. Then, an overview of the two-level swap framework is described.

A. Observation

The page access characteristic of each app is observed at first. 30 popular apps are studied on a mobile device. As shown in Fig. 5, three types of page accesses are identified, including launch requested, execution requested, and not requested. Launch requested represents pages required during launch, execution requested represents pages only required during execution, and not requested represents the pages not accessed at all. The figure shows the statistics for the page-access ratios

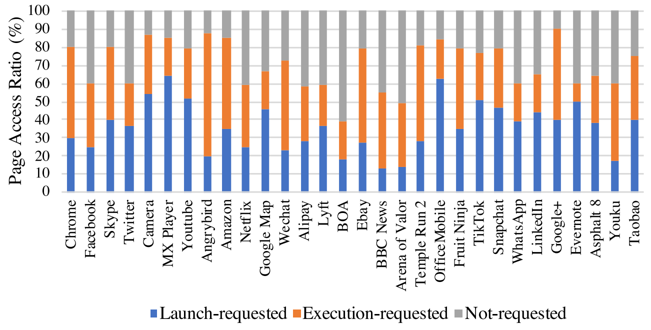


Fig. 5. Ratio of launch-requested, execution-requested, and not-requested pages.

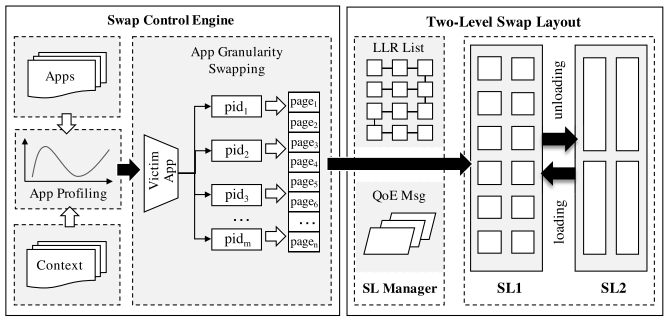


Fig. 6. SEAL system overview.

among these three types. To ensure that all pages swapped in are required by the app, the page prefetching mechanism in the system is disabled. The results show an interesting observation: only a small part of pages are required during app launching. As shown in the figure, the ratio of accessed pages during launch is less than 40% for most apps. Considering some requested pages may not be used, the ratio is even less. More than 60% of pages are requested in the following execution phase, or even not accessed. This article further found that launch-requested pages mostly stay the same during each round evaluation, while execution-requested pages change a lot. This is because the code path of app launching is often stable, while the app behavior in execution is usually dynamic. Inspired by the above observations, this section presents a two-level swap framework to address the interaction deterioration issue during app execution and while enable fast warm launching.

B. SEAL Overview

Fig. 6 shows the overview of SEAL. In the design, the SP is organized with two levels: 1) SL1 and 2) SL2. SL1 is designed based on memory compression to store the data requested during app launching, while SL2 is designed based on secondary storage to store the other data, which may be requested during app execution. During *swapout*, all data pages at these two levels are first compressed and transferred to SL1. Then part of them is converted to I/O requests and submitted to the secondary storage, similar to the conventional I/O management as introduced in Section II-A. During *swapin*, there are two stages: 1) for app launching, data is accessed from SL1 and 2) for app execution, data from SL2 is loaded to SL1 for accessing.

To comprehensively improve user experience, several critical issues are further addressed in the framework design. (C1) Since SP is organized into two levels, identifying and placing data belonging to the corresponding level is critical. (C2) It is too late to load pages in SL2 back when they have already been requested, since even one small I/O can significantly affect the frame rate. (C3) The performance of data migration between the two levels should be improved. To address the above issues, three schemes are proposed correspondingly. First (C1), a least-launch-required (LLR) scheme is designed to identify and place pages for SL1 and SL2. Different from original policies like LRU or LFU, SEAL separates the pages of each app by identifying whether it is launch-required. The launch-required pages are locked in SL1, while other pages are unloaded to SL2. Second (C2), an HPL scheme is proposed to load data in advance. The timeline of page loading and app launching is overlapped. Hidden loaded pages are not required in the launch phase, so their impact on launch speed is marginal. Third (C3), an AGS scheme is proposed to reduce migration cost. Pages belonging to the same app are batch swapped instead of one by one. In this way, the I/O throughput is efficiently improved.

IV. Design and Implementation of SEAL

In this section, three proposed schemes are presented. Then, the implementation details of SEAL is discussed.

A. LLR-Based Data Identification and Placement

To determine the data identification and placement between SL1 and SL2, the LLR-based scheme is proposed. The basic idea of LLR is to differentiate the pages used during app launching and execution. It takes app usage patterns into consideration. Once identified, pages required during launch are placed in SL1 and pages required during execution are placed in SL2, separately.

To differentiate data pages from the launch and execution stage, LLR is designed with list pairs. Each list pair consists of two lists: 1) sl1-list and 2) sl2-list. SEAL creates one list pair for each app. It is initialized instantly when an app is successfully installed. Pages accessed during app launching (including both cold and warm launching) are appended to the sl1-list, while others are appended to the sl2-list. During an app's life cycle, the list pair is updated when app switching happens between the foreground and background.

Algorithm 1 shows how the list pair works. If a page in the sl2-list is required in the launch phase, it will be migrated to the sl1-list, and its weight is set as k (lines 9–11). On the other hand, if a page in the sl1-list is not requested, its weight is decremented by 1 (line 13). Only when the weight is reduced to 0, will the page be migrated to the sl2-list (lines 14 and 15). LLR gives "k-chances" to the page in the sl1-list. The condition of migration from sl1-list to sl2-list is: one page is not requested during launch for k times consecutively. The k-chances strategy has two benefits. First, it prevents frequent page migration between the two lists. Second, it ensures that launch-required pages have a higher chance to stay in sl1-list. SEAL prefers to lock pages in SL1, rather than unload pages

Algorithm 1 K-Chances Algorithm

```
Parameter:
$llr\_sl1\_list$: list of launch-required pages;
$llr\_sl2\_list$: list of lest-launch-required pages;
Procedure:
1: /* LLR lists initialization */
2: for all $page[i]$ belonging to target app do
3:     if $page[i].flag != 0$ then
4:         $llr\_sl1\_list.append(page[i])$;
5:     else
6:         $llr\_sl2\_list.append(page[i])$;
7: /* LLR list-pair updating */
8: if launchRequired($page[i]$) then
9:     $page[i].flag \leftarrow k$;
10:    if $page[i] \in llr\_sl2\_list$ then
11:        migrating $page[i]$ to $llr\_sl1\_list$;
12: else
13:    $page[i].flag \leftarrow page[i].flag - 1$;
14:    if $page[i].flag == 0$ then
15:        migrating $page[i]$ to $llr\_sl2\_list$;
16: return ;
```

to SL2, as the latter has a higher cost on user experience. The migration between these two lists is not frequent since most warm launch-required pages have already been indexed during cold launching, only a small set of pages, such as the app state will be newly requested.

SEAL creates one list pair for each app, instead of using one list pair to index all apps' pages due to two reasons. First, if all pages are indexed by one list in the design, app information should be recorded on the list. It is complex and inefficient to maintain. On the contrary, the overhead of multiple list pairs without app information is low. Second, existing features in kernel, such as `cgroup`, is more friendly to our approach. The list pairs can be implemented with minimum kernel modifications. Based on LLR, the user experience during app launching can be significantly improved.

B. Hidden Page Loading From SL2 to SL1

When launching an app, all required pages can be quickly accessed from SL1, while many pages need to be loaded back from SL2 during the following interaction. As introduced above, I/O induced by swap can significantly increase the interaction alert. It will be ideal that all pages will be ready in SL1 when the app starts execution. This inspires the design of HPL from SL2 to SL1. The main idea of HPL is to load all the pages back when an app is switched to the foreground. The page loading overlaps with the time window of app launching, so the loading process is user imperceptible. The time window is defined as the time interval from the start to the end of an app launching (T_1 to T_3 in Fig. 7). The process of HPL is as follows. First, an app is launched at T_1. During this stage, the data pages are read from SL1 to launch the app quickly. Second, once the app is activated to launch, the execution-required data placed in SL2 is ready and read to SL1 at T_2. There is a time gap between T_2 and T_1 for the activation of page loading. A worker thread is created to load the pages in parallel. Third, before the app is launched at T_3, the data loading from SL2 to SL1 is finished and ready for

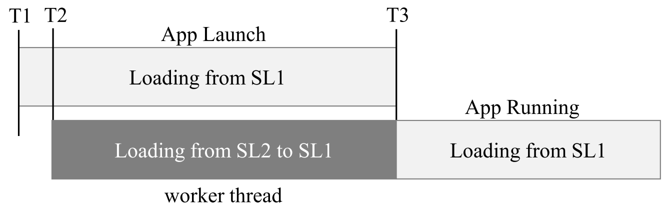


Fig. 7. HPL from SL2 to SL1. Pages belonging to an app are prefetched, by overlapping with the time window of app launching. The loading process will be completed before the end of app launching.

following execution. All launch-required pages are placed in SL1 and do not need to be loaded by HPL, and thus HPL has a small impact on launch speed.

This technique is not simple since there are still two challenges to overcome. First, the time cost of page loading should no bigger than that of app launching, as presented at T_3 in Fig. 7. Supposing it takes 600 ms to launch an app, but loading all its pages back needs 1 s. In this case, the user experience after launching will be dramatically deteriorated. To avoid such case, the number of pages placed in SL2 for each app should be precisely calculated. Second, SEAL induced I/O requests could compete with other I/O requests. To avoid I/O competition, the I/O flow between SL1 and SL2 should be carefully managed. To solve the above challenges, two techniques are presented as follows.

1) Time Cost Model: To overcome the first challenge, a time cost model is provided. This model is responsible to calculate the peak data size for SL2 for each app. It estimates the peak size as a function of launching time-window, I/O throughput (Kbps), and coefficient μ. Assuming the time window of app$_i$ is T_{wi}, and I/O throughput is detected as TH. The peak size (PS$_i$) allowed to swap to SL2 is

$$\mathrm{PS}_i = \left\lfloor \frac{\mu \times T_{wi} \times \mathrm{TH}}{4\mathrm{KB}} \right\rfloor. \tag{1}$$

This model is calibrated dynamically. The value of T_{wi} is calculated based on app$_i$'s launching latency in history. Another important value, I/O throughput TH, is measured by the workloads with various request sizes. Since the throughput is impacted by many factors, such as the frequency of GC, page erasure event, and queue length in the block layer, this value is detected in real-time. Next, μ is a configurable coefficient from 0 to 1 to tolerance the bias of throughput and time-window prediction. The motivation of μ is to provide a safe margin. If an app spent 1 s to startup, it is safer to complete hidden loading within 0.8 s instead of 0.99 s. The above design is conservative in determining the size of data placed in SL2. With this model, the data in SL2 can be loaded to SL1 on time during app launching.

2) I/O Flow Control: The SEAL induced I/O may compete with other I/O. The competition leads to two problems. First, the page loading operation may be interrupted when I/O is congested. As a result, the efficiency of page loading cannot be ensured well. Second, the page unloading operation may interrupt other I/O requests, which is unfriendly to the user experience either. To avoid these problems, the I/O flow between SL1 and SL2, including page loading and unloading is controlled. Specifically, page loading (SL2 to SL1) induced I/O request has a high priority and thus can be quickly loaded back. To achieve this goal, requests generated by SEAL are marked. Note that the data loaded back to SL1 is still in a compressed state and will be decompressed when it is requested. If no available space in SL1 to support page loading, or the total amount of launch-required pages of all the victim apps are more than the size of SL1, the app killing mechanism will wake up to release space. Page unloading (SL1 to SL2) is executed in the background. Moreover, to avoid its impact on the others, especially the foreground apps, page unloading induced I/O request has a low priority. Thus, different from page loading, the unloading operation will suspend as soon as other requests are submitted.

In summary, the HPL and earlier presented LLR can improve the user experience during app launching and execution, respectively. LLR is designed to support the fast app launching, and HPL is used to maintain a high frame rate during app execution. However, with the above two schemes, the app caching capability is sacrificed. The reasons come from two aspects: first, the data amount allowed to unload to SL2 is limited by the peak size presented in (1). If this is violated, the loading latency will exceed the app's time window. Second, to support fast app launching, the launch-required pages are not allowed to be unloaded to SL2 based on the design of LLR. Due to the above two reasons, the total amount of data that can be swapped to the secondary storage is limited. Thus, the app caching capability will be sacrificed. To compensate for the app caching degradation caused by the above two techniques, this article further proposes an AGS scheme.

C. App-Granularity Swapping

The basic idea of AGS is to swap pages belonging to the same app together. Under AGS, candidate pages are also compressed and transferred to SL1 at first, then the launch unrequired pages are further evicted to SL2. AGS improves app caching capability from two aspects. First, since pages belonging to an app are handled together, it is possible to unload them from SL1 to SL2 in batch, which significantly improves the I/O throughput. As a result, more pages are allowed to place to the storage, as presented in (1). Second, AGS is more aggressive than original swapping: it swaps all pages of a selected app out without constrained by the memory status. Specifically, original swapping operation can be prevented when the released memory exceeds a threshold (as mentioned in Section II-A). On the contrary, AGS will not stop until all pages of an app are evicted. In the following, this article presents how an app's pages are tracked and batch swapped and which app should be selected as a victim.

1) App-Granularity Batch Swapping: The first challenge to realize app-granularity batch swapping is page tracking for each app. Fig. 8 illustrates how pages of an app are tracked. First, each app is uniquely identified by its package name (PKN), which is unique among all apps installed on the mobile platform at a given time. So PKN is used to identify app. During the life cycle of an app, many processes may be generated. The second step is to obtain the process IDs (PIDs) of

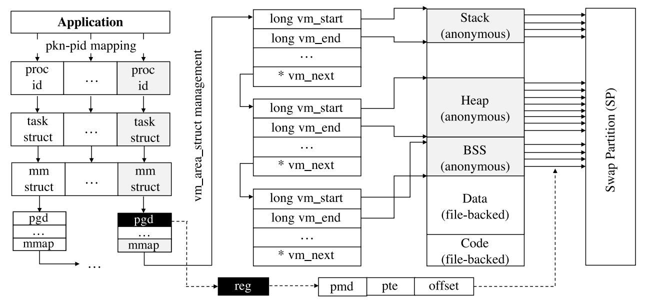


Fig. 8. AGS. Pages belonging to an app are tracked and swapped together.

a given app. Then, pages belonging to each process are further traced. The mapping between app and processes is maintained in PKN-PID mapping. Third, each process is represented by `task_structure`. It further points to `mm_struct`, which describes the memory context of the process. In this structure, many virtual memory areas (VMAs) are defined. With the help of the VMA information, corresponding page addresses can be obtained. Thus, all pages belonging to a given app are tracked.

The tracked pages are compressed and placed at SL1. When unloading the LLR pages to SL2, they are handled in the storage stack in batch. Specifically, swap induced I/O requests are merged to a bigger request when they are waiting in the block layer. The maximum amount of data carried by a merged request is set to 1MB. This size aligns with the maximum `bio` size supported by Linux kernel. In this way, most overhead in storage stack is avoided and the throughput is significantly improved. Note that the above scheme also works for other operating systems. This is because the process and memory management schemes are similar.

The other challenge of app-granularity batch swapping is to break the boundary of the kernel and user space. Taking Android as an example. Processes are managed as the basic unit in kernel, while apps run in the user space (managed by the Android framework). This article addresses this issue by adopting sysfs. The sysfs file system is mounted on/sys. Sysfs provides functionality similar to the sysctl mechanism, where the former is implemented as a virtual file system. SEAL supports app information transferring across space by writing protocol string to/sys files. For example, the time window of app launching can be transferred to kernel by "$echo 600 ms >/sys/seal/tw". In the design, information is transferred from user space to kernel, instead of the opposite direction. Meanwhile, all features in the user space are encapsulated as system components, and no interface is provided to the end user. Thus, the security of SEAL is well protected.

2) Victim App Selection: Now we discuss how to select the victim app. Unlike most previous work which predicts the most *likely* to be used apps. In our design, AGS needs to select victim apps to swap out. Thus, we prefer to predict the most *rarely* used apps. This article exploits the process-priority management in the native kernel to realize victim app selection. We first describe the design detail, then explain the advantages of the design.

There are many processes managed by the system. Each process is assigned with a priority by the system to determine

Algorithm 2 Victim App Selection Algorithm

```
Input: $pkn\_array$: package name array of cached apps;
       $pp\_map$: the mapping between pkn and pid;
Output: $pkn[index]$: pkn of the selected victim app;
Procedure:
 1: Initialize max-priority ($mp$) and average-priority ($ap$);
 2: for $pkn[i]$ in $pkn\_array$ do
 3:     $pkn[i].mp \leftarrow High_{j=0}^{m} pkn[i].get\_prio(pid[j])$;
 4:     $pkn[i].ap \leftarrow \sum_{j=0}^{m} pkn[i].get\_prio(pid[j])$;
 5:     /* Select the pkn with lowest mp as victim */
 6:     if $pkn[i].mp > LP$ then
 7:         $LP \leftarrow pkn[i].mp$;
 8:         $index \leftarrow i$;
 9:     /* Select the pkn with lowest ap as victim */
10:     else if $pkn[i].mp == LP$ then
11:         if $pkn[i].ap > pkn[index].ap$ then
12:             $index \leftarrow i$;
13:         else if $pkn[i].ap == pkn[index].ap$ then
14:             $index \leftarrow Random(index, i)$;
15: /* Return victim pkn index from the function */
16: return index;
```

how much CPU or processor time is allocated. The priority of each process is managed by the native system. Since apps are made up of processes, it is possible to represent an app's activity by process priority. In the design, the activity of an app is defined based on the priority of its processes. As shown in Algorithm 2, a victim app is selected based on max-priority (MP) and average priority (AP). The MP is set as the highest priority of its processes (line 3). When swapping is initiated, the app with the lowest MP is selected as the victim. In case that some apps may have the same MP, AP is defined. The AP of an app stands for the average value of its process priorities (line 4). If more than one app have the lowest MP, SEAL further compares their AP. The app with lower AP will be evicted (lines 5–12).

The process priority is utilized to select the victim app for two reasons. First, by applying the original priority management in the native system, the overhead of AGS is significantly reduced. Second, many existing features, such as low memory killer (LMK [17]) and out of memory killer (OOMK [18]) are also designed based on the process priority management. Sharing the same priority system in memory management instead of building a new one can avoid potential conflicts.

D. Implementation

To deploy SEAL on off-the-shelf devices, there are a couple of implementation details to be addressed.

1) SEAL Swap Engine: A swap engine is implemented to determine when should pages be swapped out and when should pages be swapped back. To achieve this goal, the contextual information is taken into account, including launching latency and I/O traffic. The condition to trigger *swapin* and *swapout* is different. *Swapin:* The process of page swapin is split into two phases: 1) page loading from SL2 to SL1 and 2) page decompression from SL1 to the main memory. Page loading occurs when app launching is detected. This is a pro-active action. All pages are loaded back, no matter whether they are needed right away or not. On the contrary, the page decompression is a passive action. It will not happen until the page in SL1 is indeed requested. Only the requested page will be decompressed to the main memory.

Swapout: The process of page swapout is more complicated. It is split into two phases also: page compression from the main memory to SL1, and page unloading from SL1 to SL2. When the watermark threshold is exceeded, as introduced in Section II-A, a victim app will be selected and compressed to SL1. SEAL prefers to unload pages from SL1 to SL2 as soon as possible, rather than wait for the SL1 space to fill out. To avoid I/O congestion, the I/O traffic is monitored. If I/O is busy (IOPS higher than a predefined threshold), the unloading operation will suspend. When the I/O pressure reduced (e.g., the screen turned off), the unloading procedure will continue. Pages in SL1 will not be released until page unloading completes. If an app is switched to the foreground at the time of page unloading, the unloading operation of this app will be canceled.

2) Two-Level SP Management: Implementation of the two-level SP management is presented in this section. SL1 is deployed in physical memory as a dedicated RAM disk. In this approach, the compression technique, `lzo`, is deployed. Candidate pages are compressed to this mounted RAM disk. SL2 is generated by creating a new partition on the Flash. Specifically, the source code of *ptable.img* is modified. After porting the recompiled image to the mobile device and rebooting, a new virtual block device can be seen in the directory "/dev/block". SEAL unloads the compressed pages from SL1 to SL2 by identifying this newly generated virtual block device.

Page migration between the two levels takes place frequently, so effectively page indexing is the other challenge of SP management. In the implementation, flags L1 and L2 are maintained to index pages located in SL1 or SL2, respectively. Page table entry (PTE) records the information of swapped pages. When a page is compressed to SL1 or unloaded to SL2, the content of PTE is updated to reflect the change. Original swap related code and corresponding mapping table are modified to support the two-level page indexing. Specifically, when a page locates in SL1, the address bit of PTE is set as L1. An array, `table[index].handle`, is maintained to record the start address of the page locate in the RAM disk and the offset. The start address and the offset in storage are recorded in the mapping table. For example, to index a page during unloading, four steps are performed.

1) Encapsulating candidate pages to I/O requests and merging them in the queue.
2) Calling `write` interface to submit the I/O requests.
3) Changing the address pointer from SL1 to SL2 when the request completed successfully.
4) Updating PTE and releasing the space in SL1.

The address information is maintained throughout the whole process.

SEAL is implemented on real-world mobile devices, with both user-space and kernel-space modifications. The modification consists of 2864 lines of code (1136 lines of C/C++ and 1728 lines of Java), excluding the libraries.

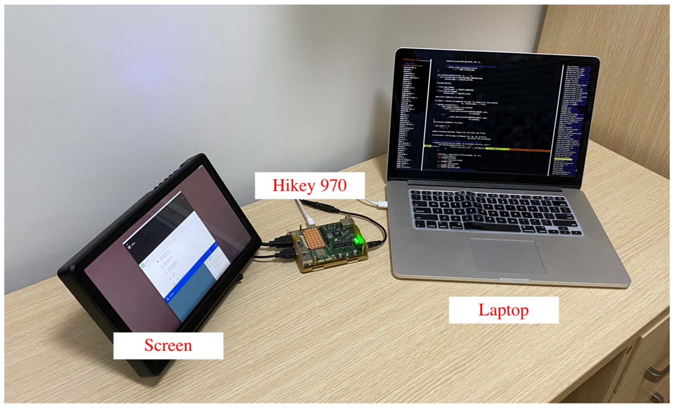


Fig. 9. Evaluation platform. SEAL is deployed on Hikey 970 platform. The laptop controls app behavior. The app launching and execution process is displayed on the screen.

V. Experiment Setup

A. Evaluation Platforms

The experiments are performed on Hikey 970 [19] platform, which is equipped with HiSilicon Kirin 970 Core, 6-GB DDR4 RAM, and 64-GB UFS2.1 Flash. Android 9.0 and Linux kernel 4.9 are deployed on the device. Fig. 9 shows the experiment environment. During the evaluation, two tools are adopted: 1) Android Debug Bridge (Adb [20]) and 2) UI/Application Exerciser Monkey (Monkey [21]). Adb is a versatile command-line toolkit, which is used to startup apps, identify their launch styles, and record the launch latency. For example, an app can be startup by command "$adb shell am start -W." Monkey is adopted to simulate app execution. It generates pseudo-random streams of user events, such as clicks, touches, or gestures, as well as a number of system-level events. To run this tool, the app's PKN and the total number of user events that we want to generate are provided. Then random events on UI elements will be generated automatically. Based on Adb and Monkey, all app behaviors, including both launch and execution, can be effectively controlled.

B. Evaluated Workloads

There are 30 applications deployed on the platform. As shown in Table I, these apps covered multiple categories, including social network, multimedia, game, electronic commerce, and utility. The characteristics of workloads are different, such as memory usage size, the ratio of launch -(un)required pages, and the sensitivity to launch latency as well as interaction alert. For example, 87 MB is needed to launch Ebay as measured, but it takes more than 200 MB to launch "Arena of Valor," a popular mobile game. Since almost 3-GB space is occupied by the mobile system, only 3-GB DRAM is left to the workload apps. Thus, the workload is big enough to perform the memory pressure tests.

C. Evaluated Schemes

Five schemes implemented and measured to show the effectiveness of SEAL.

TABLE I
WORKLOAD APPLICATIONS

Category	Application
Social Network	Facebook, Skype, Twitter, WeChat, WhatsApp LinkedIn, TikTok, Google+
Multimedia	MXPlayer, Youtube, Netflix, Snapchat, Youku
Game	Angry Birds, Arena of Valor, Temple Run 2 Fruit Ninja, Asphalt 8
Commerce	Amazon, AliPay, BOA, Ebay, Taobao
Utility	Chrome Browser, Camera, Google Map Lyft, BBC News, OfficeMobile, Evernote

1) *No-SWAP:* This is the baseline of the evaluation. All swap features in the system are disabled. Including both storage-based-swap [3] and compression-based swap [22]. All cached pages are placed in the physical memory in this case. When the memory is under pressure, available memory will be released by killing some apps instead of evicting some pages to an SP.
2) *ZSWAP [23]:* The basic architecture of this scheme is close to SEAL. It deploys "two-level" SP: a dedicated RAM disk in the physical memory as the first level and generates a swap file on storage as the second level. When a swap operation is performed, all pages are first compressed to the first level. Then, rarely accessed pages in the RAM disk are transferred to the swap file in the background.
3) *LLR:* This scheme represents the proposed LLR page identification scheme. The basic idea of this scheme is to differentiate the pages used during app launching and execution. It redefines the placement of swapped pages.
4) *LLR+HPL:* This scheme is to combine LLR and HPL, which represents the HPL scheme in Section IV-B. HPL loads all pages back when an app switched to the foreground. The page loading overlaps with the time window of app launching, so that this prefetch process is user imperceptible. Combining LLR and HPL, both the launch latency and execution time user experience will be improved.
5) *SEAL:* This is the proposed framework, which combines the LLR, HPL, and AGS schemes. Among them, AGS performs memory swapping in app granularity. All pages belonging to the same app are swapped together, instead of page by page. In this way, SEAL is able to improve the app caching capability.

VI. Experiment Results and Analysis

A. User Experience Improvement

Three metrics are analyzed for the user experience evaluation: 1) warm launching latency; 2) interaction alerts; and 3) app caching capability. Thus, the user experience improvement of SEAL can be verified in three aspects. First, warm launching latency is evaluated to show the advantages of LLR, as LLR places launch-related data in SL1 to enable a fast app launching. Second, interaction alert is evaluated to measure

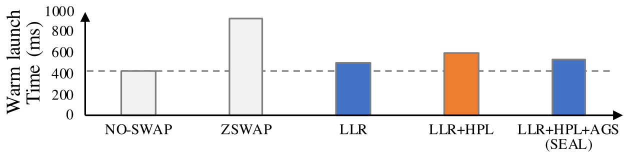


Fig. 10. Warm launching latency.

the benefit of HPL. As HPL loads the app execution needed data during app launching time, interaction alerts should be significantly reduced. Finally, the app caching capability is evaluated to verify the benefits of AGS. AGS will handle swaps in batches and optimize data transfer between SL1 and SL2. In this case, more data can be swapped to SL2, and the app caching capability can be optimized.

1) Warm Launching Latency: To collect warm launching latency, warm launching is executed for ten times for each scheme and the average is recorded. Fig. 10 shows the warm launching latency of the five evaluated schemes. The baseline scheme NO-SWAP achieves the best launch speed with 526 ms, on average. This is because NO-SWAP launches the app from the main memory. The state-of-the-art scheme ZSWAP has the worst launch speed with 1443 ms. This is because ZSWAP does not differentiate whether the data is launch-required or execution-required. It needs to read data from the swap area in the secondary storage to launch the app.

The proposed LLR is able to significantly reduce the launch latency compared to ZSWAP. This is because LLR is designed to identify the launch-related data and place them in the memory-based swap. In this case, during app launching, the app will be launched from the main memory. Hence, the warm launching related user experience is significantly improved. However, the results also show that the warm launching latency of LLR is higher than that of NO-SWAP by around 17%. The reason for the increase is from the decompression induced cost. In the proposed scheme, the data placed in SL1 is compressed for memory efficiency. By adding HPL, the warm launching latency is further increased. This is because HPL is designed to load the execution time data from SL2 to SL1 during app launching. As discussed in Section IV-B, with this approach, the data loading from SL2 to SL1 during app launching may interference with the process of warm launching from SL1. From the results, the warm launching latency is marginally increased by around 10%. Compared with ZSWAP, the warm launching latency is still significantly improved. Finally, by adding AGS, the warm launching latency is improved compared with HPL. The reason comes from that AGS is designed to batch process the swap. Then, the interference can be well controlled. The results show that the proposed SEAL has similar warm launching latency to that of LLR.

2) Interaction Alert: The interaction alerts are measured to verify whether user experience is improved during app execution. In the evaluation, Android Monkey is utilized to simulate interaction. In the simulation, various interaction events are generated. The impact on user experience during app execution is detected by recording the number of interaction alerts.

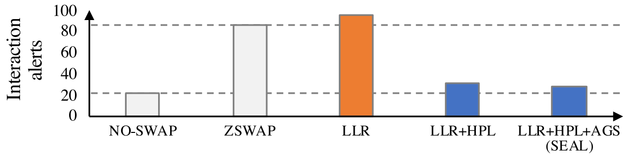


Fig. 11. Interaction alerts.

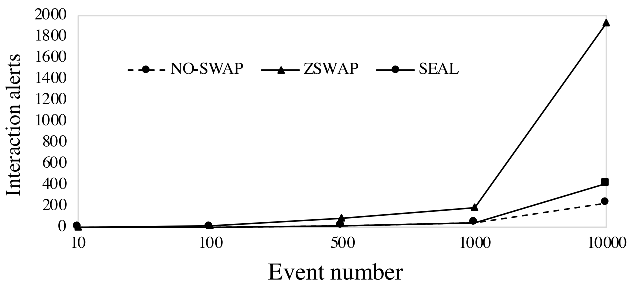


Fig. 12. Interaction alerts number of tested three cases. The interaction alerts of SEAL are effectively controlled.

One event generated by the Android Monkey contains an interaction operation, including scrolling the screen, clicking a button, or opening a new window. To evaluate the interaction alert induced user experience impact, 500 events are created when running the Android Monkey. Systrace is utilized to record the interaction alerts number during the above events.

Fig. 11 shows the collected results. There are several observations are uncovered based on the results. First, the number of interaction alerts for NO-SWAP is very small. This is because the execution time accessed data locates in the main memory. Second, for ZSWAP, its interaction alerts are significantly increased by more than 4 times. This is because it needs to access a lot of data during execution from the secondary storage. Third, similar to ZSWAP, LLR also has a significant amount of interaction alerts. The reason is the same as the ZSWAP. More importantly, LLR is designed to place the data used during execution at SL2. In this case, the user experience during execution is significantly degraded. Forth, by adding HPL, the interaction alerts are significantly reduced compared with ZSWAP and LLR. This is because HPL is designed to preload data from SL2 during app launching. Finally, SEAL maintains similar interaction alerts as that of LLR+PHL. In conclusion, the evaluation results show that the number of interaction alerts of SEAL is effectively controlled. It is close to the NO-SWAP case, which is only 14% increase, and reduced by 76% compared with ZSWAP.

To show more details and the robustness of the proposed scheme, we further analyzed the interaction alerts with different numbers of events, including 10, 100, 500, 1000, and 10000, respectively. Fig. 12 shows the results. The results show that the proposed scheme is able to significantly reduce the number of interaction alerts. More importantly, with the increases in interaction numbers, the alerts are slightly increased and approach the No-SWAP.

3) App Caching Capability: Fig. 13 shows the app caching capability among the evaluated schemes. Only 7 of 30 apps are warm launched with NO-SWAP due to the limited main

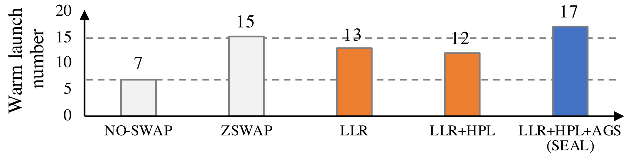


Fig. 13. App caching capability.

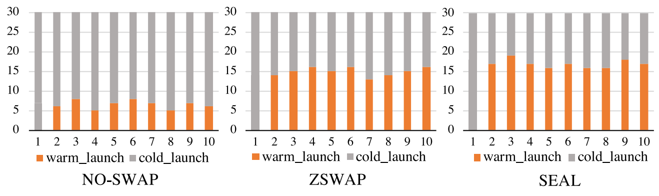


Fig. 14. App caching benefit. x-axis represents the ten rounds test, and y-axis represents the ratio of cold and warm launching.

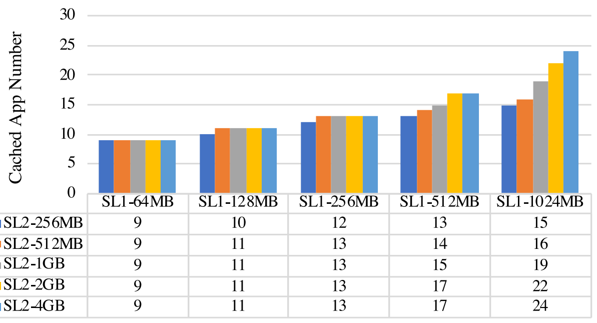
memory size. As a comparison, more apps can be cached when enabling ZSWAP. Specifically, 15 apps are warm launched on average. Then the proposed three schemes are tested. LLR is able to improve the app caching capability compared with NO-SWAP due to that it proposes to store some data to SL2. However, the capability of LLR+HPL is slightly degraded compared with LLR. This is because HPL limits the size of data stored in the SL2. Finally, by adding AGS, the capability is significantly improved and better than that of the NO-SWAP (2.43×) and ZSAWP (1.13×) case. Log analysis illustrates that it is because of the efficiency of AGS. Pages are swapped out on time. On the contrary, several apps are killed before completing the swap operation in the ZSWAP case. Fig. 14 shows detail results on the app caching capability among the ten arounds. The results confirm that SEAL consistently improve the app caching capability.

B. Sensitive Studies

To further understand SEAL, several sensitive studies are conducted in this section, including the size of SP, the k-chance algorithm, and the coefficient μ in (1).

1) Swap Partition Size: The user experience benefit of SEAL is impacted by the SP size. In this evaluation, we dynamically changes the size of SL1 (from 64 MB to 1 GB) and SL2 (from 256 MB to 4 GB). Fig. 15 shows the results. Two conclusions are obtained. First, the app caching capability increased with the expansion of SL1 and SL2. However, it is not increased linearly. When the size of SL2 raised to a peak size, the number of cached app will not increase again. This is because no enough pages are allowed to swap to SL2. Second, the size ratio of SL1 and SL2 have an impact on the app caching capability. This is due to the relatively stable ratio of app launch-required and not required pages. For example, in the (SL1-256MB, SL2-512MB) case, around six apps are swapped to the partition. However, only two apps are swapped in the (SL1-64MB, SL2-4GB) case, as SL1 becomes the bottleneck.

2) K-Chances Impact: The k-chances algorithm of LLR determines the page migration between sl1-list and sl2-list. In this evaluation, we configure the value of k from 1 to 6 to



	SL1-64MB	SL1-128MB	SL1-256MB	SL1-512MB	SL1-1024MB
SL2-256MB	9	10	12	13	15
SL2-512MB	9	11	13	14	16
SL2-1GB	9	11	13	15	19
SL2-2GB	9	11	13	17	22
SL2-4GB	9	11	13	17	24

Fig. 15. App caching capability improvement when SP size is varied.

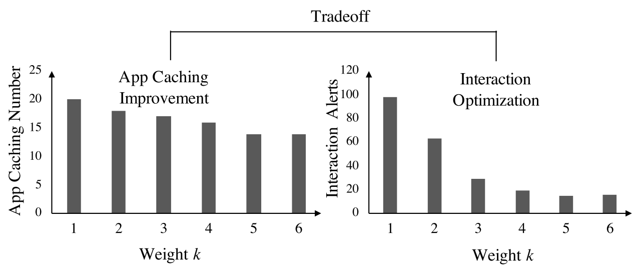


Fig. 16. Relationship between app caching and interaction alert by varying the weight k in LLR.

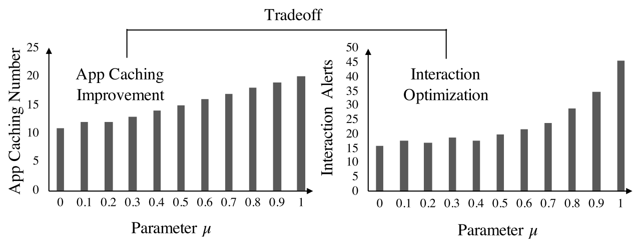


Fig. 17. Relationship between app caching and interaction alert by varying the coefficient μ.

illustrate the impact. Fig. 16 shows that along with the increase of k, the number of cached app decreases. This is because the condition of page unloading becomes stricter. More pages are locked in SL1. On the other hand, the interaction alerts decreased. Based on the evaluation, we find that both of these two metrics perform well when $k = 3$.

3) Coefficient μ Impact: The coefficient μ is used to control the ratio between SL1 and SL2. Both app caching capability and interaction alerts are collected. Fig. 17 shows the results by varying μ from 0 to 1. The app caching capability is increased and the interaction alerts are increased with the increases of μ. This is because with the increases of μ, more data will be placed at SL2 based on (1). In this case, data from SL2 may not be able to preload to the SL1 during app launching. Then, the possibility of interaction alert will increase. To balance the tradeoff, μ is set to 0.8 in the experiments of this article.

VII. Related Work

Even though there are many valuable efforts on memory management optimization [24], [25], memory swapping is still promising as memory is scarce in mobile devices. There are several recent works on swapping. Zhong *et al.* [26] built the SP with nonvolatile memory (NVM), to improve the performance of smartphones. Kim and Bahn [27] analyzed the I/O characteristics of swapping operation in smartphones and presented a new architecture that adopts NVM at the swap layer. In their work, new generation storage mediums are deployed as SP to improve efficiency. However, Flash, including eMMC and UFS, is still the commonly deployed storage medium in mobile devices. the assumption of close-to-memory I/O throughput and low latency is not always available in a real-world environment.

A memory compression-based swap is an alternative approach since the speed of page (de)compression is much faster than I/O. Take Android as an example, memory compression techniques are deployed and enabled in the system. It allows "swap to ZRAM (Compcache)" [22], which compresses pages and stores them in a dedicated RAM disk to save memory space. Several compressed swap schemes for server systems have also been proposed to meet the memory demand of highly consolidated SPs or memory-intensive workloads [28]. Kim *et al.* [29] further proposed a compressed swap scheme for mobile devices, named ezswap. It accommodates not only anonymous pages but also clean file-backed pages. The design of SL1 of SEAL is inspired by the memory compression technique. However, the effectiveness of compression-based-swap is limited since compressed pages still occupy memory.

Different from the expensive and scarce memory resource, storage resource is abundant. With the growing capacity and wear-leveling strategies, the wear-out risk of Flash becomes controllable. Existing works, such as [30] have discussed how to optimize lifetime when enabling swap mechanism. Several works target secondary storage-based swap so that Flash resource can be fully used. MARS [3] is designed to speed up app launching through flash-aware swapping. It isolates GC from page swapping for compatibility and employs several flash-aware techniques to speed up app launching. SmartSwap [4] presented a predictive process-level swap mechanism. In the design, victim processes are swapped to Flash ahead-of-time, which significantly improves efficiency. FlashVM [31] focuses on changes to the virtual memory system to make effective use of available fast storage devices for swapping.

To take advantage of both memory compression and secondary storage-based swap, several works utilized compressed cache between main memory and secondary storage. ZSWAP [23] improves the drawback of ZRAM, which cannot evict compressed pages to the SP in secondary storage. In the design of ZSWAP, pages are moved to the compressed cache first. After that, it evicts some of the cached pages to the secondary storage and then receives newly incoming pages. Han *et al.* [32] proposed a hybrid swap scheme that stores frequently accessed data in the in-memory compressed swap area and sends infrequently accessed data to the swap space in the secondary storage. This scheme improved the hit ratio of the compressed swap by accommodating only the pages with low compression ratios and access frequencies in the compressed swap.

VIII. Conclusion

This article proposes a user experience-aware two-level swapping, which maximizes the benefits of app caching capability and minimizes the impact on user experience when enabling swapping. It is the first work that focuses on addressing the swap induced user experience degradation. Both launch time and execution time user experience are effectively optimized. Three novel schemes, LLR, HPL, and AGS are proposed. Experiments on real devices show that app caching capability is improved by 2.43× on average when enabling SEAL. Meanwhile, the interaction alerts reduced by 76% compared with the state-of-the-art technique.

References

[1] K. Zhong *et al.*, "Energy-efficient in-memory paging for smartphones," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1577–1590, Oct. 2016.

[2] J. Lee, S. Park, M. Ryu, and S. Kang, "Performance evaluation of the SSD-based swap system for big data processing," in *Proc. IEEE 13th Int. Conf. Trust Security Privacy Comput. Commun.*, 2014, pp. 673–680.

[3] W. Guo, K. Chen, H. Feng, Y. Wu, R. Zhang, and W. Zheng, "*mars*: Mobile application relaunching speed-up through flash-aware page swapping," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 916–928, Mar. 2016.

[4] X. Zhu, D. Liu, K. Zhong, J. Ren, and T. Li, "SmartSwap: High-performance and user experience friendly swapping in mobile systems," in *Proc. 54th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, 2017, pp. 1–6.

[5] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, "ProfileDroid: Multi-layer profiling of Android applications," in *Proc. MobiCom*, Aug. 2012, pp. 137–148.

[6] T. Masashi and U. Takeshi, "Smartphone user interface," *Fujitsu Sci. Tech. J.*, vol. 49, pp. 227–230, Mar. 2013.

[7] Android Developers. *Analyzing UI Performance With Systrace*. Accessed: 2020. [Online]. Available: https://android.magicer.xyz/tools/debugging/systrace.html

[8] D. E. Porter, T. Zhang, A. Zuck, and D. Tsafrir, "Apps can quickly destroy your mobile's flash: Why they don't, and how to keep it that way," in *Proc. MobiSys*, 2019, pp. 207–221.

[9] M. Ju, H. Kim, M. Kang, and S. Kim, "Efficient memory reclaiming for mitigating sluggish response in mobile devices," in *Proc. IEEE 5th Int. Conf. Consum. Electron. Berlin (ICCE-Berlin)*, 2015, pp. 232–236.

[10] J. Suse, "Linux block IO—Present and future," in *Proc. Ottawa Linux Symp.*, Jan. 2004, pp. 51–61.

[11] Mark Raymond. (2019). *Mobile App Usage Report*. [Online]. Available: https://www.goodfirms.co /resources/app-download-usage-statistics-to-know

[12] Android Developers. (2020). *Systrace*. [Online]. Available: https://developer.android.com/topic/perfor mance/tracing

[13] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O stack optimization for smartphones," in *Proc. USENIX Annu. Techn. Conf. (ATC)*, 2013, pp. 309–320.

[14] C. Ji, L. P. Chang, C. Wu, L. Shi, and C. J. Xue, "An I/O scheduling strategy for embedded flash storage devices with mapping cache," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 4, pp. 756–769, Apr. 2018.

[15] Y. Liang *et al.*, "Read-ahead efficiency on mobile devices: Observation, characterization, and optimization," *IEEE Trans. Comput.*, early access, Apr. 2, 2020, doi: 10.1109/TC.2020.2984755.

[16] C. Wu *et al.*, "Maximizing I/O throughput and minimizing performance variation via reinforcement learning based I/O merging for SSDs," *IEEE Trans. Comput.*, vol. 69, no. 1, pp. 72–86, Jan. 2020.

[17] R. Prodduturi and D. B. Phatak, "Effective handling of low memory scenarios in android using logs," M.S thesis, Dept. Comput. Sci. Eng., Indian Inst. Technol., New Delhi, India, 2013.

[18] J. Kook, S. Hong, W. Lee, E. Jae, and J. Kim, "Optimization of out of memory killer for embedded Linux environments," in *Proc. ACM Symp. Appl. Comput.*, 2011, pp. 633–634.

[19] Engineers. (2020). *Hikey970 Platform*. [Online]. Available: https://www.96boards.org/product/hikey970/

[20] Engineers. (2019). *Android Debug Bridge (ADB) Tool*. [Online]. Available: https://androidmtk.com/download-minimal-adb-and-fastboot-tool

[21] Android Developers. *Android Monkey*. Accessed: 2020. [Online]. Available: https://developer.android.com/studio/test/monkey

[22] N. Gupta. *ZRAM Project. Linux Foundation, San Francisco, CA, USA*. Accessed: 2020. [Online]. Available: https://www.kernel.org/doc/Documentation/blockdev/zram.txt

[23] S.Jennings. *Zswap Project. Linux Foundation, San Francisco, CA, USA*. [Online]. Available: https://www.kernel.org/doc/Documentation/vm /zswap.txt

[24] L. Liu, Y. Li, C. Ding, H. Yang, and C. Wu, "Rethinking memory management in modern operating system: Horizontal, vertical or random?" *IEEE Trans. Comput.*, vol. 65, no. 6, pp. 1921–1935, Jun. 2016.

[25] L. Liu, S. Yang, L. Peng, and X. Li, "Hierarchical hybrid memory management in os for tiered memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2223–2236, Oct. 2019.

[26] K. Zhong *et al.*, "Building high-performance smartphones via non-volatile memory: The swap approach," in *Proc. Int. Conf. Embedded Softw. (EMSOFT)*, 2014, pp. 1–10.

[27] J. Kim and H. Bahn, "Analysis of smartphone I/O characteristics—Toward efficient swap in a smartphone," *IEEE Access*, vol. 7, pp. 129930–129941, 2019.

[28] L. Yang, H. Lekatsas, and R. P. Dick, "High-performance operating system controlled memory compression," in *Proc. 43rd ACM/IEEE Design Autom. Conf.*, 2006, pp. 701–704.

[29] J. Kim, C. Kim, and E. Seo, "*ezswap*: Enhanced compressed swap scheme for mobile devices," *IEEE Access*, vol. 7, pp. 139678–139691, 2019.

[30] T. Song, G. Lee, and Y. Kim, "Enhanced flash swap: Using NAND flash as a swap device with lifetime control," in *Proc. IEEE Int. Conf. Consum. Electron. (ICCE)*, 2019, pp. 1–5.

[31] S. Mohit and S. M. Michael, "FlashVM: Virtual memory management on flash," in *Proc. USENIX Annu. Tech. Conf.*, 2010, p. 14.

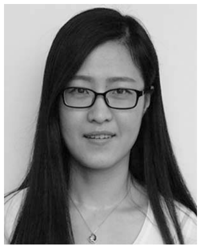
[32] J. Han, S. Kim, S. Lee, J. Lee, and S. J. Kim, "A hybrid swapping scheme based on per-process reclaim for performance improvement of android smartphones (August 2018)," *IEEE Access*, vol. 6, pp. 56099–56108, 2018.

Changlong Li (Member, IEEE) received the B.S. and Ph.D. degrees in computer science from the University of Science and Technology of China, Hefei, China, in 2012 and 2018, respectively.

He was a Visiting Scholar with the University of California at Los Angeles, Los Angeles, CA, USA, from 2015 to 2016. He is currently an Postdoctoral Researcher with the Department of Computer Science, City University of Hong Kong, Hong Kong. His research interests include memory management, storage, mobile devices, and distributed systems.

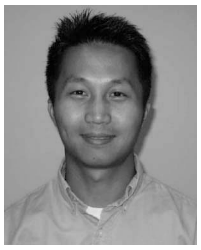
Liang Shi received the B.S. degree in computer science from the Xi'an University of Post and Telecommunication, Xi'an, China, in 2008, and the Ph.D. degree from the University of Science and Technology of China, Hefei, China, in 2013.

He is currently an Professor with the Department of Computer Science and Technology, East China Normal University, Shanghai, China. His current research interests include flash memory, embedded systems, and emerging nonvolatile memory technology.



Yu Liang received the B.E. and M.E. degrees from the Department of Computer Science and Technology, Shandong University, Jinan, China, in 2010 and 2013, respectively. She is currently pursuing the Ph.D. degree with the Department of Computer Science, City University of Hong Kong, Hong Kong.

Her research interests include file systems and memory management.



Chun Jason Xue received the B.S. degree in computer science and engineering from the University of Texas at Arlington, Arlington, TX, USA, in May 1997, and the M.S. and Ph.D. degrees in computer science from the University of Texas at Dallas, Richardson, TX, USA, in December 2002 and May 2007, respectively.

He is currently an Associate Professor with the Department of Computer Science, City University of Hong Kong, Hong Kong. His research interests include memory and parallelism optimization for embedded systems, software/hardware co-design, real-time systems, and computer security.