IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received April 8, 2020, accepted April 30, 2020, date of publication May 4, 2020, date of current version May 19, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2992072

Maintaining Application Context of Smartphones by Selectively Supporting Swap and Kill

JISUN KIM AND HYOKYUNG BAHN, (Member, IEEE)
Department of Computer Engineering, Ewha Womans University, Seoul 120-750, South Korea

Corresponding author: Hyokyung Bahn (bahn@ewha.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIP) under Grant 2019R1A2C1009275, and in part by the ICT Research and Development Program of MSIP/IITP (Developing System Software Technologies for Emerging New Memory that Adaptively Learn Workload Characteristics) under Grant 2019-0-00074.

ABSTRACT With the rapid diffusion of smartphones as well as the technical advances in mobile platform technologies, application domains supported by smartphones have dramatically increased. Unlike traditional computer systems, however, the context of applications may be lost in a smartphone as mobile OS usually kills applications without user's approval when free memory space is exhausted. This was not a serious issue when a smartphone was a personal entertainment device, but it is now significant as a lot of official works are also performed by a smartphone. Instead of killing a process, the context of an application can be backed up to swap storage, but smartphones do not accept it as swap incurs excessively heavy I/O traffic. However, our findings show that the overhead of swap is not serious in recent smartphone devices and it can also be eliminated by judicious software management. This article proposes a selective swap scheme that classifies applications based on their context-saving characteristics, and controls the number of processes involved in swap by monitoring system situations and application characteristics. That is, we maintain the context of applications selectively by swap or application's own state-saving. We implement and measure the effectiveness of our scheme on a real Android device, showing that it provides the context-savings of applications without additional overhead.

INDEX TERMS Android, mobile platform, application, process context, smartphone, swap.

I. INTRODUCTION

In traditional computer systems, the context of an application is maintained by the operating system until the process is terminated. Unless an exceptional case such as a system crash occurs, a process is not killed without user's acknowledgment. Thus, users do not need to be concerned about whether the context of an application will be lost or not. However, this is no longer the case for smartphone systems.

With the rapid diffusion of smartphone devices as well as the advances in mobile platform technologies, desktop applications increasingly switch their execution platform to smart devices [1], [2]. New smartphone applications using camera, GPS (global positioning systems), and sensors also emerge every day [3]–[6]. However, smartphone systems do not guarantee the keeping of application contexts due to the philosophy of mobile platforms. In particular, smartphone platforms such as Android lose the context of an application by killing it when the free memory space in the system becomes lower than a certain threshold [7], [8]. As this is performed without user's approval, applications usually restart without their previous context. This was not a serious issue when a smartphone was a personal entertainment device, but now it is significant as the domain of smartphone applications becomes wider. For example, terminating a music player does not incur serious results but killing a stock trading application while changing the price of some stock orders may cause serious problems. To resolve this issue, some applications preserve their context by themselves before termination and restore it when restarting. However, as this is not performed by the operating system, the saved context is limited and lots of applications do not even provide such functions

Instead of killing processes, traditional computer systems support *swap*, which uses a certain portion of secondary storage as main memory's extension for saving application's memory context [9]. As smartphones now act as general

The associate editor coordinating the review of this manuscript and approving it for publication was Tomás F. Pena.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

purpose multi-tasking systems, the necessity of swap is becoming increasingly important. However, it is reported that the I/O traffic of smartphones increases significantly when the swap function is provided, which also slows down the launch time of applications [10]. One way of coping with this situation is to use additional hardware components. For example, non-volatile memory can be adopted to absorb I/O traffic generated by swap, thereby alleviating the slow launch time of smartphone applications [11]–[13], [21], [23]. However, adding a hardware component incurs additional cost and is not effective unless smartphone vendors accept it. Furthermore, non-volatile memory such as PCM (phase-change memory) or STT-MRAM has not yet opened its complete market for smartphone systems [14]–[16].

Our aim is not to use additional hardware components, but to utilize only judicious software management for maintaining application contexts. To do so, we perform some preliminary experiments with an Android reference device and find out that recent smartphone devices do not incur excessively heavy I/O traffic although swap is supported. However, we also observe that Android swap still suffers from heavy I/O traffic as the number of applications in execution becomes excessively large. This is different from the original Android (i.e., swap-disabled) case, where the performance is not degraded seriously even though the number of applications increases. This is because low-memory-killer (LMK) controls the number of processes in the system by killing and restarting applications without incurring heavy I/O traffic. This implies that terminating and restarting an application incurs less overhead than swap-out and swap-in as killing an application removes most of process's memory address space (i.e., code, data, stack, heap) without saving to storage. If it restarts, the memory address spaces such as data, stack, and heap are created in memory rather than loading from storage. However, original Android (i.e. swap-disabled) has the problem of losing application's context if the application does not save its context by itself.

By considering this, we propose a selective swap scheme that controls the number of processes to be swapped by monitoring the system situation and application characteristics. Instead of equally maintaining all process contexts, our scheme maintains the context of processes selectively by the operating system's swap or the application's own state-saving. To determine whether to support swap or to kill an application, our scheme classifies applications based on their context-saving characteristics, and selectively support swap for applications that do not save context by themselves.

To assess the effectiveness of the proposed scheme, we implement our scheme on a real Android platform and perform measurement studies to compare the launch time of applications in original Android, swap-supported Android, and our scheme. Performance evaluation results show that Android with our selective swap scheme does not incur additional overhead in application's launch time. Furthermore, the variation of the launch time is significantly reduced compared to the swap-supported Android. Specifically, our scheme improves the average launch time of applications by 19-83% in comparison with the swap-supported Android, and also reduces the standard deviation of launch time by 20-96%. Our new findings and contributions can be summarized as follows.

- Supporting swap in a smartphone incurs excessively heavy I/O traffic in early days, but it is not the case for recent smartphone devices.
- However, when compared with the swap-disabled kill-based systems, swap-based systems still incur heavy I/O traffic as the number of applications in execution becomes very large.
- We categorize smartphone applications, and find out that some applications do not need the assistance of swap as they save application contexts by themselves.
- We propose a selective swap scheme that controls the number of processes involved in swap by monitoring system status and application characteristics.
- Although there is no hardware assistance, our scheme maintains the context of applications without performance degradations in terms of application's launch time and their variations.

The remainder of this article is organized as follows. Section II briefly explains free memory management schemes in Linux and Android. In Section III, we quantify the overhead of swap in Android. Section IV describes the proposed selective swap scheme for smartphones. Performance evaluations based on measurement studies to assess the effectiveness of the proposed scheme are depicted in Section V. Section VI briefly summarizes the related works of this study. Finally, Section VII concludes this article.

II. FREE MEMORY MANAGEMENT IN LINUX AND ANDROID

As the size of main memory in a system is limited, the available memory space is getting smaller and is finally exhausted as the number of processes in the system increases. When there is not enough free memory, swap is triggered in Linux. Specifically, `kswapd` is activated to make free memory space by saving some memory context of processes to secondary storage [17]. As Android also adopts Linux kernel in its bottom architecture, `kswapd` can be activated. However, the current Android architectures adopt LMK (low memory killer), which is firstly triggered when free memory space becomes low [10], [18]. Specifically, LMK kills low-priority processes without user's agreement when free memory space is below a certain threshold. Thus, the context of an application killed by LMK disappears in Android unless the application itself saves it.

There is another software module, OOM (out of memory killer), which is activated when the free memory space is almost exhausted. Specifically, OOM works when it fails to free page frames, thereby making the system difficult to normally operate. Actually, this is an emergency situation that does not happen in usual cases as LMK or `kswapd` works

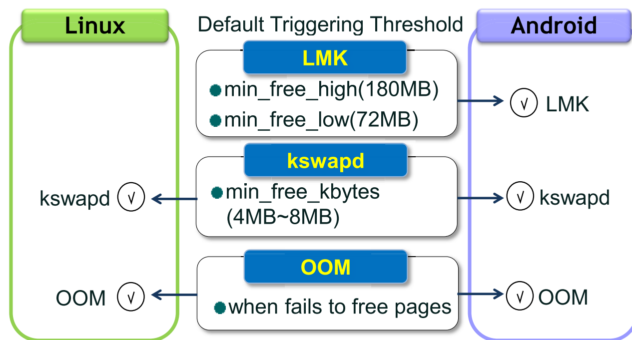

FIGURE 1. Triggering conditions of LMK, kswapd, and OOM in Android and Linux.

before OOM is activated. Once OOM starts its work, processes are killed based on their scores evaluated by memory occupation and nice values until minimum free memory space is made.

Figure 1 depicts the triggering conditions of LMK, `kswapd`, and OOM in Android and Linux. As shown in the figure, Android's LMK works more aggressively than `kswapd` because Android does not support swap. Although this improves the response time of applications by sufficiently keeping free memory space, it fails to guarantee the reliability of program executions.

III. QUANTIFYING THE OVERHEAD OF SWAP IN ANDROID

This article focuses on the Android smartphone environment that is reconfigured to support virtual memory swap. Although supporting swap in Android is not impossible, previous studies have shown that it incurs heavy I/O traffic, thereby degrading the smartphone system performances significantly [10], [23].

To quantify the overhead of swap, we also perform experiments by reconfiguring an Android smartphone to support swap and collect storage I/O traces while executing various kinds of applications. Storage I/O traces in our experiments were extracted by `ftrace` in Android kernel [19]. We warm up memory by executing a variety of Android applications to make full use of available memory, and then induce swap situations.

We sequentially execute a series of applications and then repeat them to see the effect of swap. (Detailed characteristics of execution scenarios will be explained later in Section V.) As each sequence consists of a sufficient number of applications to fill up memory, applications are essential to be terminated and restart upon their next launch in original Android whereas swap-supported Android saves and restores applications' memory data by making use of swap.

We use two kinds of Android reference devices to perform our experiments. The first is ODROID-Q, which consists of 1GB DDR2-DRAM memory and 1GB swap file on 16GB eMMC [20]. Actually, this is an outdated hardware spec but we use this to set up similar conditions with previous studies [21], [23]. Figure 2(a) shows the storage I/O traffic of original Android and swap-supported Android on ODROID-Q. The graph shows the I/O traffic normalized to the original Android case for each experiment. Similar to previous studies, our analysis shows that swap-supported Android incurs about 8 times more I/O traffic than original Android [23]. This is because supporting swap requires additional storage accesses for saving and retrieving application's memory address space, whereas killing and restarting an application without swap perform most of their works in memory.

Recently, Android versions and the hardware spec of smartphones are advancing rapidly. To see the effect of such improvements, we perform our second experiments with another Android reference device, Nexus 5, which consists of 2GB LPDDR3-SDRAM memory and 2GB swap file on 16GB eMMC [22]. Interestingly, our experimental results show that the I/O traffic problem of Android swap becomes weak or almost disappears. As shown in Figure 2(b), swap does not increase the I/O traffic at all in Scenario A. Note that Scenario A executes 12 concurrent applications repeatedly within 20 minutes. Although this is sufficient number of applications for common smartphone users, we raise the workload intensity and see the effect of swap in excessively heavy workload conditions. Figure 2(c) shows the I/O traffic of original Android and the swap-supported Android as we repeatedly execute 24 concurrent applications. As shown in the figure, swap increases I/O traffic but is not as serious as the situation in ODROID-Q.

This implies that swap does not incur serious performance problems in the current Android devices. We only need to see the effects of swap upon a thrashing condition where the number of applications involved in swap becomes more than a certain threshold, thereby exhausting the available memory space. In that case, some kinds of essential Android services and shared libraries are evicted from memory and then reloaded repeatedly. Thus, Android swap needs a mechanism that does not incur such phenomena. One way of coping with this situation is to adjust the number of processes involved in swap not to incur heavy I/O traffic, which will be discussed in the next section.

Now, let us briefly discuss the storage issue of swap. Flash memory is used as the storage device of smartphones, but it was difficult to use flash as the swap device not only due to the heavy I/O traffic but also the performance and endurance issues of flash memory devices. Early flash products suffered from the freezing phenomenon in which the performances of storage I/Os are degraded seriously when the Garbage Collection starts [34]. There were also performance fluctuations in flash storage as the internal state of the storage device changes over time. However, such problems have been improved significantly by adopting internal buffer and/or cache in recent flash storage products. Also, recent flash storage devices adopt wear leveling techniques in their FTL (Flash Translation Layers) and it is known that such techniques can redistribute the write traffic to storage

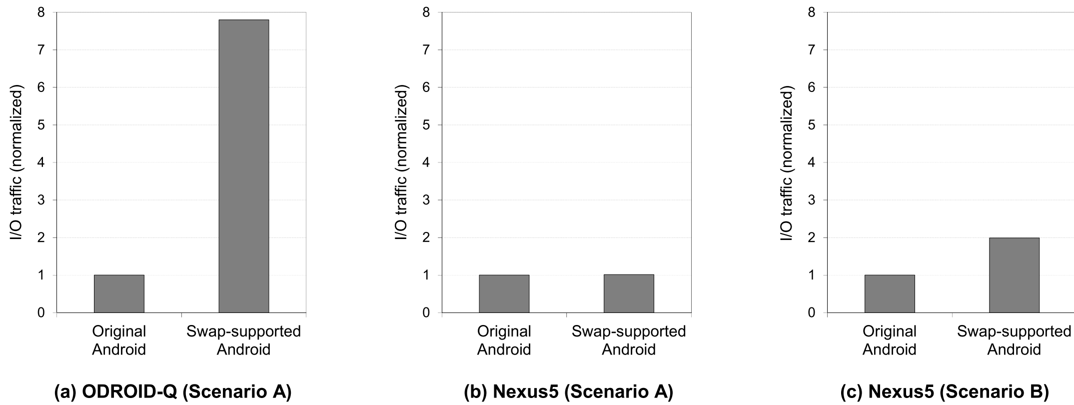

FIGURE 2. Comparison of I/O traffic for original Android and swap-supported Android.

well [34]. Thus, the lifespan of storage mainly depends on the write I/O traffic to storage. Unlike early Android devices that incur excessively heavy I/O traffic when swap is enabled, this article argues that swap does not incur heavy I/O traffic in recent smartphone devices. Specifically, if we control the number of applications involved in swap, the endurance of flash memory is not a problem any longer.

IV. THE SELECTIVE SWAP SCHEME IN SMARTPHONES

Our aim is to maintain the context of smartphone applications, thereby making smartphones more reliable computing devices. Nevertheless, we do not want either performance degradation or addition of hardware components. This implies that supporting such functions should be done only by judicious software management.

To do so, we utilize the following two characteristics that appear in smartphone applications. First, some smartphone applications are developed to maintain their context by themselves, not needing the assistance of swap. For example, the current version of the Android video player restarts from the previously watched position, although it is killed by LMK. We exclude such applications from the targets of swap. Second, the number of applications executed by a single user is limited, and thus situations that we need to control the number of processes to be swapped rarely happen. It is reported that a common smartphone user runs less than 10 applications within a day [24]. The problem situation happens only when the number of concurrent applications that need the assistance of swap becomes excessively large. That is, when the working-set of a system is beyond the capacity of main memory, thrashing happens, which incurs excessively heavy I/O traffic. We need to control the number of processes to be swapped in this case.

A. APPLICATION CLASSIFICATIONS

To determine whether to support swap or to kill an application, our selective swap scheme classifies applications based on their context-saving characteristics and priorities. Then it selectively supports swap for applications that do not save context by themselves.

We classify smartphone applications into two categories: the swap-target applications of which operating systems need to maintain their contexts, and the kill-target applications that internally save contexts by applications themselves or no contexts to be saved. However, if the number of processes needs to be controlled, some swap-target applications can be shift to kill-target applications based on the priority of applications.

Algorithm 1

```
// Low-Memory-Killer (LMK)
if free memory falls below kill_threshold then
    Select the oldest app A in kill_target_app_list;
    Terminate A through sending the kill signal;
end if
// Kernel-Swap-Daemon (kswapd)
if free memory falls below swap_threshold then
    Select victim page p not used recently from
    inactive_list;
    if p belongs to swap_target_apps then
        Evict p from memory;
    end if
    if the size of inactive_ list is below inactive_low then
        Refill inactive_ list with old pages in active_ list;
    end if
end if
if # of apps involved in swap exceeds swap_max then
    Select the lowest priority app p in swap_target_app_list;
    Delete p from swap_target_app_list;
    Insert p to kill_target_app_list;
end if
// Out-Of-Memory-Killer(OOM)
if kswapd fails to free pages then
    Select the lowest priority app A in the system;
    Terminate A through sending the kill signal;
end if
```

TABLE 1. Classification of applications and their examples.

Priority	Classification	Description	Examples	
			Context saved by applications	Context needs to be saved by swap
High	Office app, Financial app, Location tracing app	Apps for office works such as word processing, spreadsheet, note, editors and design paint; Financial apps such as e-banking, stock trading; Location tracing apps such as car navigation.	Gmail, sketch, Naver memo	Google maps, Naver map
Medium	Multimedia app, Social networks, Messenger	Apps for multimedia play such as video player, music player, game; Social network service apps, online video telephones.	MX Player, tenten, Youtube Music, Whatsapp, Messenger	Io.slither, templerun2, piano tile2, subwaysurf, candycrushsaga, twitter, trafficrider, Youtube, eightballpool, TED, Facebook, instagram
Low	Information provision, Camera app, Advertisement	Web-based apps such as news, sports, travel, weather; Basic camera in a smartphone, selfi camera.	Browser, Candy Camera, Android Camera	BBC, NBC

This article suggests the basic classification principles, but does not focus on who has the responsibility of this classification. However, it would be difficult to ask the application developers or end users to make such decisions. Thus, we suggest a simple way of classifying applications with respect to the context-saving characteristics. That is, Android supports a certain form of context-saving files for application developers whenever the activity state changes. Thus, in order to distinguish whether it is a state-saving application or not, our scheme makes use of the existence of such files created or saved when the application is killed.

We also categorize applications by their priorities as shown in Table 1. The priorities here are defined based on the application's functionalities similar to previous studies [4].

The high priority class includes office applications, financial applications, and location tracing applications. For example, car navigation, stock trading, and spreadsheet are classified into this class. Social networks and multimedia applications are categorized as the medium priority class. Most applications in this class need personalized services, but they are usually network-based applications that frequently update contexts to provide their information. Games, which require long loading time, are also classified into this class. The low priority class includes information provision, advertisement, and camera applications. We classify camera applications into low priority as taking a picture does not need context-saving because context in camera is valid only when the application is running as a foreground process. Note that the foreground process is not selected as the target of kill in any kinds of operating systems including Android.

B. TRIGGERING KSWAPD AND LMK

Now, let us explain the details of the proposed selective swap scheme. The key idea of the proposed scheme is to control the number of applications involved in swap. To do so, we classify applications' priorities and context-saving characteristics; then we maintain the context of applications selectively by the operating system's virtual memory swap or the application's own context-saving. In case of the latter, our scheme does not perform swap but kills and restarts the application as is done in original Android. Applications that do not perform self-context-savings are the targets of swap. However, if the number of applications involved in swap exceeds a certain threshold, thrashing may happen and thus we need to limit the number of applications to be swapped. In this case, the priority of applications classified in the previous section is considered, and the application with the lowest priority is excluded from the targets of swap.

When the available memory space becomes below a certain threshold, our selective swap scheme triggers both `kswapd` and LMK to make free memory. The default triggering conditions of `kswapd` and LMK, which we call swap-threshold and kill-threshold, are set to 4MB and 180MB, respectively. Note that these are the default settings of Linux and Android, respectively. When we apply these settings, LMK is triggered earlier than `kswapd`. This is a reasonable configuration as killing and restarting an application requires less cost than swap I/O. However, for a comparison purpose, we set up another version of the selective swap scheme that triggers swap earlier than kill. We call these two variants of the selective swap scheme, the kill-first and the swap-first schemes. Table 3 depicts the default thresholds of these two schemes used in our experiments.

TABLE 2. Application scenarios used in the experiments

Scenario A: normal workload (A1:sequential, A2: random)				
io.slither	templerun2	tenten	whatsapp	browser
gmail	subwaysurf	NBC	google maps	eightballpool
youtube	Naver map			

Scenario B: heavy workload (B1: sequential, B2: random)				
MX player	instagram	io.slither	BBC	templerun2
tenten	facebook	whatsapp	browser	candycamera
piano tile2	gmail	subwaysurf	NBC	candycrushsaga
Naver memo	trafficrider	google camera	google maps	eightballpool
TED	youtube	sketch	naver map	

Scenario C: realistic workload				
finger balance	google play store	google maps	contacts	google music
gmail	phone	chrome	camera	google photos
messages	clock	settings	youtube	chase mobile
hear radio app	looper	one more line	facebook	google search
genetics	floodit	adobe photoshop	narrator's voice	fingerprint
molecule 3D	echoes	navy federal credit	magicapp	echofon for twitter

If the free memory space is below the kill-threshold, our scheme selects the least recently used application among the kill-target applications and terminates it through sending the kill signal. If the free memory space becomes below the swap-threshold, memory pages are reclaimed by `kswapd`.

Pages that belong to swap-target applications are the candidates of reclamation. We use the default reclamation algorithm in Linux, which maintains two page lists, the active list and the inactive list, and evicts pages not used recently in the inactive list [25], [26].

Note that the granularity of LMK is a process whereas the granularity of `kswapd` is a page. Figure 3 overviews the proposed scheme and Algorithm 1 describes the pseudocode of our scheme.

V. MEASUREMENT STUDIES

In this section, we present the performance evaluation results to assess the effectiveness of the proposed selective swap scheme.

TABLE 3. Threshold parameter setting.

	Kill threshold	Swap threshold
Kill-first	Default (180MB)	Default (4MB)
Swap-first	100MB	180MB

A. EXPERIMENTAL SETUP

Our experimental setup consists of Nexus 5 with 2GB LPDDR3-SDRAM memory and 2GB swap file on 16GB eMMC. The swap file is created on /storage/self/primary. We install Google Android 6.0.1 and Linux 3.4.0. We reconfigure the Android kernel to support virtual memory swap and also implement our selective swap scheme on it.

Table 2 shows the scenarios we executed. In this table, applications in gray-cells are kill-target applications, whereas those in white-cells are swap-target applications. In the case

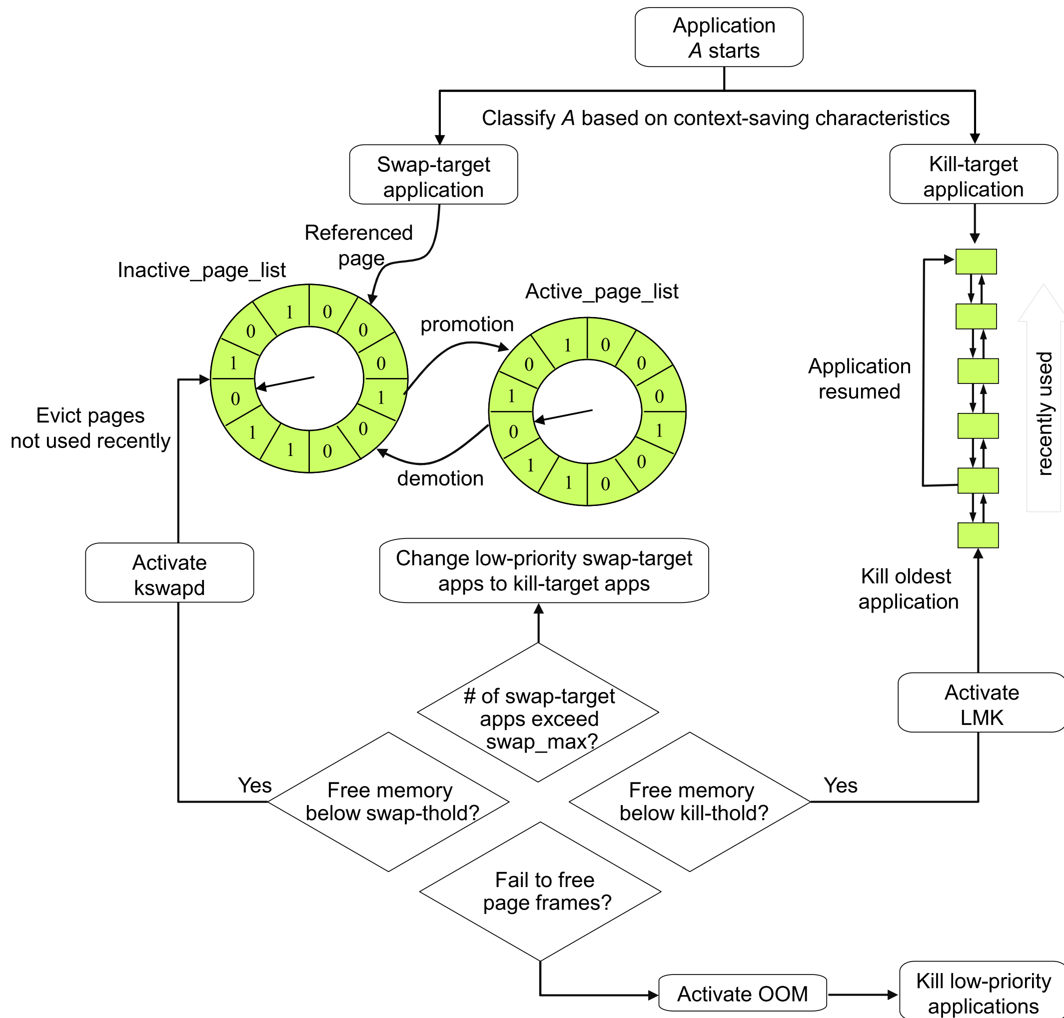

FIGURE 3. Overview of the proposed scheme.

of Scenarios A and B, we execute all applications in the table sequentially and repeat their executions 10 times to see the effect of swap. Scenarios A1 and A2 consist of 12 applications, which represents normal situations as the number of applications executed by a smartphone user is usually less than 10 per day [24]. Scenario A1 sequentially executes 12 applications and repeats by the same order whereas Scenario A2 randomly rearranges their execution order at each round. Scenarios B1 and B2 consists of 24 applications, which represents heavy workload situations. Similar to Scenarios A1 and A2, Scenarios B1 and B2 execute by the sequential and random orders, respectively, in each round.

We also make use of more realistic scenarios that reflect the access locality of smartphone's application use. Specifically, we used the workload scenario similar to Kim *et al.* [32] by making use of the smartphone usage traces in the Livelab project of Rice University [33]. As the traces in the Livelab project consist of the application usage of iPhone users, we picked the same or a similar Android application from the Google Play Store. Among the long traces of many users, we randomly selected a certain time window for a single user, and extracted the sequence of application launches within the trace. This is Scenario C in Table 2, which consists of 30 applications.

We implemented our scheme by modifying the LMK of Android not to kill swap-target applications. We also modified the reclamation module of `kswapd` to prevent the swap-out of pages belonging to kill-target applications. In order to measure the application's launch time, we first made the launcher script that contains the sequence of application's launch command for each scenario, and then executed the launcher script in original Android, swap-supported Android, and the two versions of our schemes. Specifically, we used the "`am`" command to launch each application and measured the launch time through the "`time`" command. After that, we inserted the "`sleep`" command for 30 seconds and then launched the next application. Figure 4 shows an example of our application launcher script.

```
time am start -W com.android.contacts/.activities.PeopleActivity
sleep 30
time am start -W com.android.launcher3/.Launcher
sleep 30
time am start -W com.facebook.katana/.LoginActivity
sleep 30
.....
```

FIGURE 4. An example of the launcher script for the measurement of the scenarios.

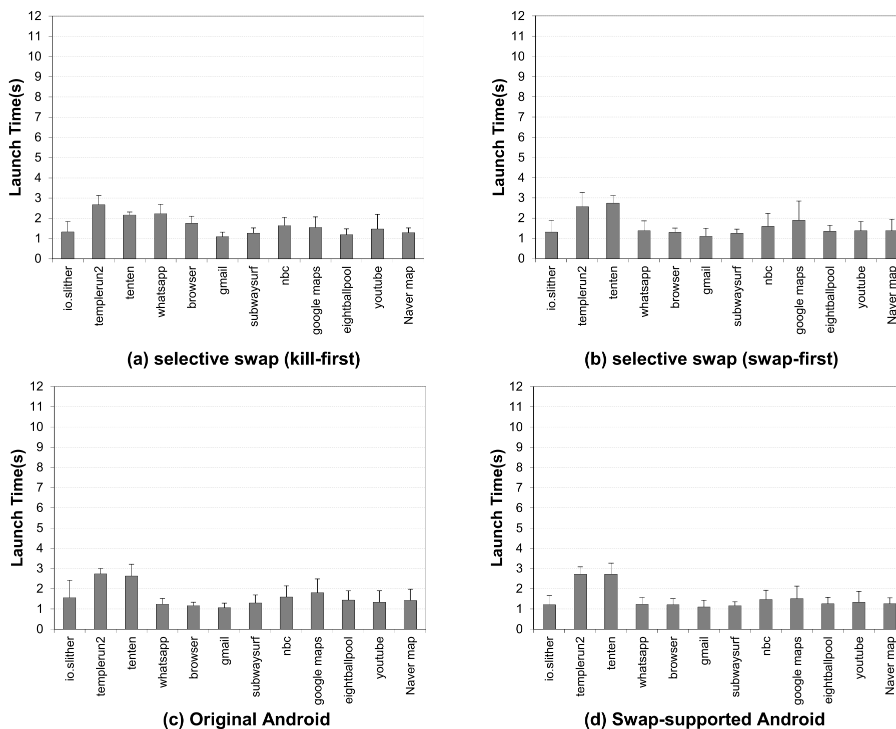

FIGURE 5. Launch time of applications in Scenario A1.

Although it would be more meaningful to measure the time taken for the full features of an application to be activated, this requires some interactions with the user to proceed to the next step, making fair comparison difficult due to the variations caused by humans. As the scenarios cannot be executed with the same user interaction for each application run, we use the launcher script that can repeat the scenarios with identical conditions and measure the launch time by the system.

B. EXPERIMENTAL RESULTS

We compare our scheme, called the selective swap scheme, with the original Android and the reconfigured Android that supports swap for all applications, which we call swap-supported Android. As we mentioned in the previous section, we use the two variants of our selective swap scheme: the kill-first scheme that uses the default threshold of LMK and `kswapd` thereby triggering LMK earlier than `kswapd`, and the swap-first scheme that triggers `kswapd` earlier than LMK.

Figures 5 to 9 show the average launch time and the standard deviation of the launch time of applications for each scenario. We compare the original Android, the swap-supported Android, and the two versions of our selective swap scheme. As shown in Figures 5 and 6, the launch time of the swap-supported Android and the proposed scheme is as good as that of the original Android when the number of concurrent applications is 12. This is because the capacity of memory is enough for executing these applications and thus the overhead of swap is negligible.

However, the results are contrasted under heavy workload conditions. Specifically, as the number of applications becomes 24, the launch time of the swap-supported Android is degraded significantly as shown in Figures 7 and 8. In comparison with the original Android, the performance of the swap-supported Android is degraded by 2x to 5x.

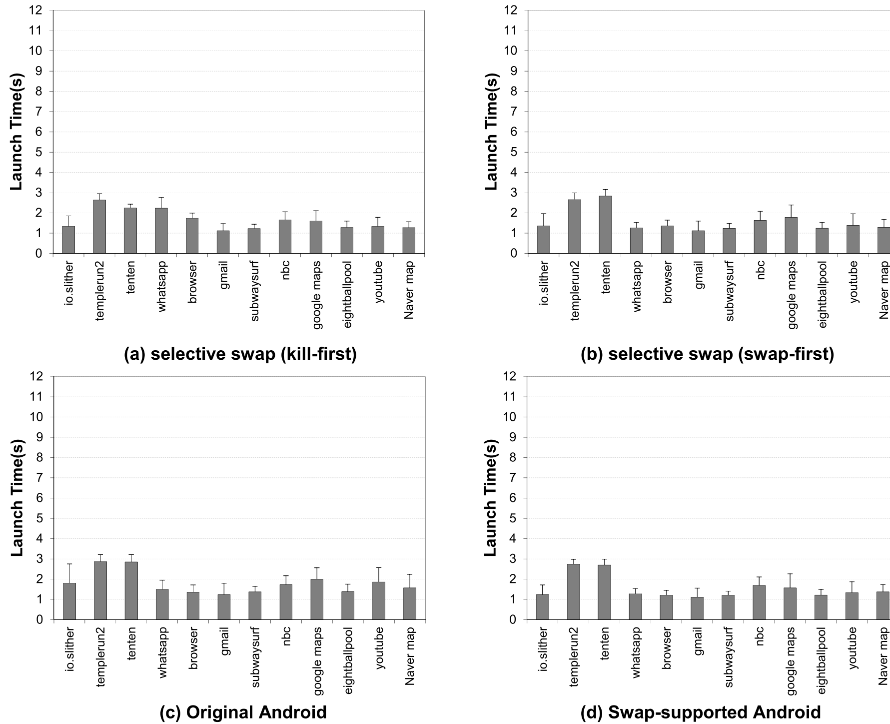

FIGURE 6. Launch time of applications in Scenario A2.

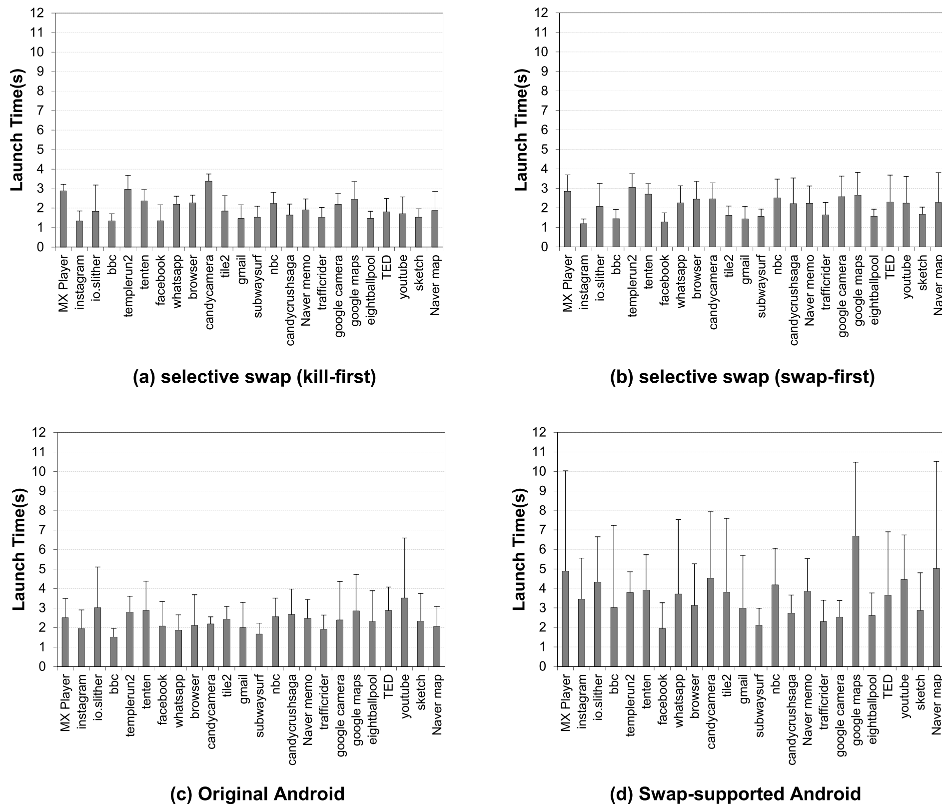

FIGURE 7. Launch time of applications in Scenario B1.

This indicates that supporting swap may still be the performance bottleneck in smartphones as the number of concurrent applications becomes large. Unlike the swap-supported Android, the performance of the proposed scheme is not degraded even when we compare it with the original Android. When compared with the swap-supported

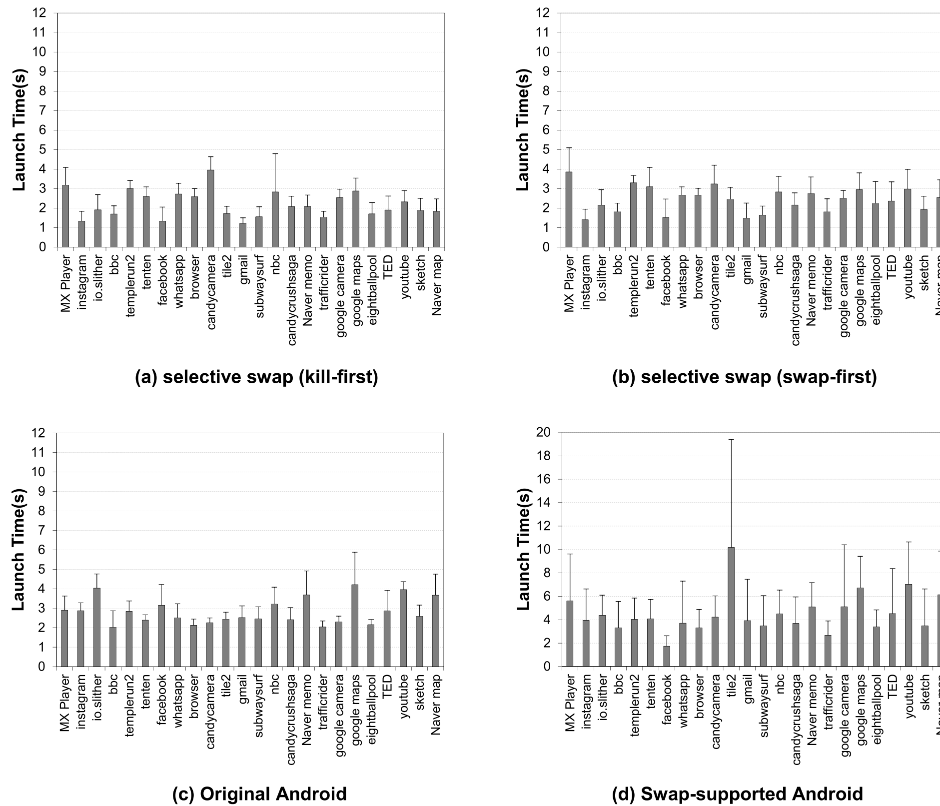

FIGURE 8. Launch time of applications in Scenario B2.

Android, our scheme with kill-first reduces the application's launch time by 51% on average. Our scheme also improves the standard deviation of application's launch time in comparison with the swap-supported Android. Specifically, the improvement is 78% on average. This is a significant result as users expect uniform latency for the same application launching.

In case of some swap-target applications like Instagram, Facebook, and Youtube, the launch time of the proposed scheme with swap-first is even shorter than that of the original Android. This is because our scheme sets the swap-threshold smaller than the kill-threshold, leading to less possibility of evicting swap-target applications from memory. When comparing swap-first and kill-first, kill-first performs slightly better than swap-first in terms of the average launch time. Since kill-first preserves free memory space by terminating kill-target applications first, it reduces the overhead of swap I/O even more. Although available memory becomes insufficient, kill-first still maintains hot pages of essential Android services and shared libraries in memory as it makes free memory by LMK first and then evicts cold pages not used recently. When we compare kill-first with the original Android and the swap-supported Android, the improvement of the average launch time is 22% and 51%, respectively.

Figure 9 shows the average and the standard deviation of each application's launch time as we performed Scenario C. Note that the standard deviation bar does not exist if an application is executed only once in the scenario. The result shows that the overhead of swap is similar to Scenario B although Scenario C has more applications. This is because Scenario B executes applications by sequential or random orders, but Scenario C considers the access locality of smartphone's application use. In this situation, frequently used applications are highly likely to restart in memory without killing or swapping, leading to improved performances. When comparing the four schemes, our two schemes perform similar to the original Android throughout all application launches with respect to the launch time and its variation, whereas the swap-supported Android do not exhibit stable performances in some cases.

Figure 10 shows the average launch time of applications when the original Android, the swap-supported Android, and the two versions of our scheme are adopted. We show the average launch time of swap-target and kill-target applications separately, and then show the average of all applications. As can be seen in Figures 10(a) and 10(b), the average launch time of swap-target applications in swap-supported Android performs the best. This is because the number of applications executed is not large enough to incur swap situations and thus most of application's data still remain in memory although swap is turned on. This happens due to the default gap of kill-threshold and swap-threshold.

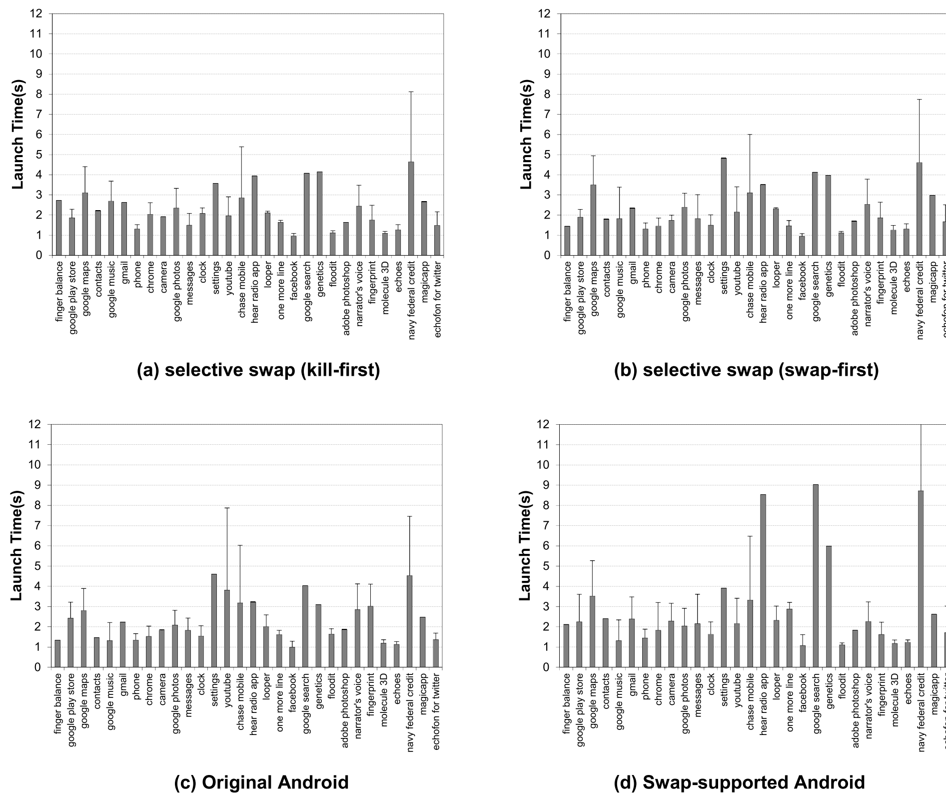

FIGURE 9. Launch time of applications in Scenario C.

Figures 10(a) and 10(b) also show that the performances of our kill-first and swap-first schemes are slightly degraded in kill-target and swap-target applications, respectively. This is due to the priority of applications the two types of our schemes give, but the overall performance is not degraded when considering the total applications as shown in the figure. Also, we do not take this seriously as it is within a certain short time range of 1.x seconds for a person to feel interactive. In reality, the more important thing for humans to feel comfortable is small variations in each launch case. In this aspect, our conclusion is that the proposed scheme does not have problems in light-weight scenarios considering the results.

Now let us see the results for heavy-weight scenarios. As shown in Figures 10(c) and 10(d), the proposed scheme with kill-first performs the best when the workload condition becomes excessively heavy. Since the proposed scheme preserves free memory space by terminating kill-target applications first, it reduces the overhead of storage I/O generated by swap. Although the available memory space becomes insufficient, the contexts of swap-target applications are still in memory, leading to better results. When we compare our scheme (kill-first) with the original Android and the swap-supported Android, performance improvement is 22% and 51%, respectively, in terms of the average launch time.

Finally, Figure 10(e) shows the performance of the four schemes when Scenario C is executed. Although the performance gap becomes narrow compared to the result of Scenario B, our schemes perform better than the swap-supported Android, and almost similar to the original Android.

Before concluding this section, we briefly discuss the I/O traffic of Android swap with respect to the read/write ratio and the percentage of I/O for swap versus non-swap traffic. We have already seen the heavy I/O traffic of Android swap in early days, but now the overhead of swap becomes small and we focus on the analysis of I/O traffic under realistic workload situations.

Figure 11(a) shows the read/write traffic of the original Android and the swap-supported Android while executing Scenario C. As shown in the figure, the two systems incur similar read traffic but the write traffic of swap-supported Android is heavier than that of the original Android. However, the increased I/O traffic is not serious when considering the total I/O traffic. Figure 11(b) separates the same I/O traffic into swap I/O and non-swap I/O. As we see, the original Android only incurs non-swap I/O, but most of these I/O traffics change to swap I/O as the swap function is turned on. Instead, non-swap I/O becomes very small in swap-supported Android.

VI. RELATED WORKS

A lot of studies have been performed to efficiently support swap in smart devices. Android supports the zRAM swap

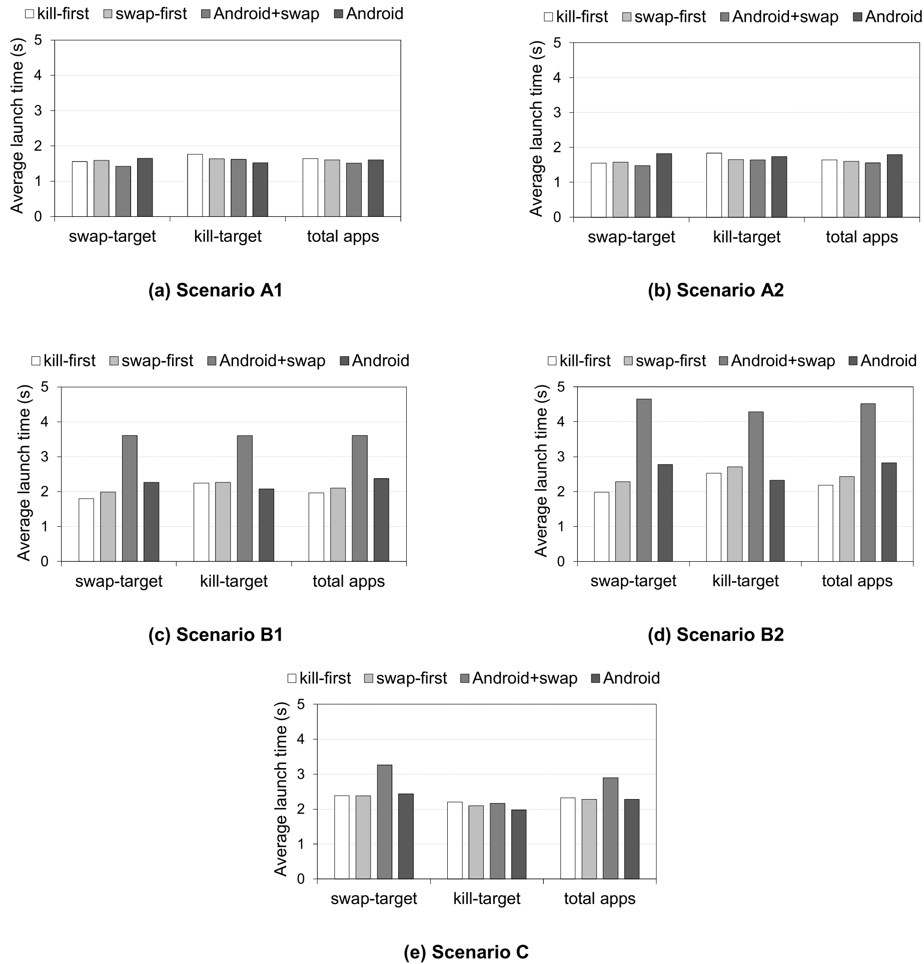

FIGURE 10. Average launch time of swap-target, kill-target, and total applications when selective swap (kill-first), selective swap (swap-first), original Android, and swap-supported Android are adopted.

FIGURE 11. Comparison of I/O traffic in original Android and swap-supported Android.

scheme since version 4.4, which makes use of a certain area of DRAM memory as ramdisk and performs swapping to this area in a compressed form [27]. Han *et al.* present a hybrid swapping scheme that supports both secondary-storage swap and zRAM swap [28]. They attempt to swap out all pages in the working set of a process to the zRAM swap space rather than killing the process selected by the low-memory killer, and swap out the least recently used pages into the secondary storage swap space. Chae *et al.* suggest CloudSwap as a swap mechanism for mobile devices by utilizing remote storage as a swap area [29]. Zhu *et al.* present the SwapBench to evaluate various swapping schemes, specifically focusing on the two performance measures, the application launch time and the application switch time on Android smartphones [30].

Some studies aim to use NVM (non-volatile memory) as the swap area of smartphones [11], [12]. Kim *et al.* adopt

a small size of NVM to absorb hot data that appear in swap-supported Android by making use of efficient management policies including admission-control [23]. Zhong *et al.* also use NVM as the swap area of smartphones and focus on the limited endurance problem of NVM [10], [11]. Kim *et al.* propose an NVM-based swap scheme for smartphone systems, called CAUSE (Critical Application Usage-aware Swap Enhanced memory system) [31]. CAUSE tries to distinguish critical pages by considering application usage patterns, and manages these pages so as to reduce the probability of being swapped-out.

If we do not use LMK and activate `kswapd` like traditional computer systems, the context of applications can be fully preserved. However, studies on smartphone swap usually focus on using both `kswapd` and LMK, and then try to improve the performance by harmonizing the two independent layers. For example, Kim *et al.* observe that the targets of `kswapd` and LMK can be overlapped as `kswapd` evicts old pages (i.e. pages not used recently) and LMK also kills old applications (i.e. applications not used recently) [32]. In this situation, swapped pages will become useless if the owner application of that pages is killed. To reduce such useless swap I/O traffic, Kim *et al.* do not allow the swap-out of pages that belong to old applications as they will be killed soon by LMK [32]. In other words, their scheme tries not to overlap the targets of kill and swap in the ''performance'' aspect not in the ''context-saving'' aspect. Thus, applications may be killed without preserving their contexts.

In contrast, our scheme separates the targets of kill and swap by the context-saving characteristics, i.e., kills for context-saving applications and swaps for non-saving applications. Thus, in our scheme, the situation that Kim *et al.* claimed, i.e., unnecessary I/O traffic due to the kill of swapped applications, does not happen.

Note that the motivation of our study is the ''reliability'' issue of smartphone's program execution rather than the ''performance'' issue. That is, we focus on the problem of losing application's context as smartphones kill applications without user's agreement when free memory is exhausted. Thus, to make smartphone a more reliable computing device, we maintain the context of applications by either swap or application's own state-saving. In particular, applications are selectively managed by LMK (for context-saving applications) or `kswapd` (for non-saving applications), thereby guaranteeing the context-savings for all types of applications. Unlike our study, existing studies have focused on the ''performance'' issue rather than the ''reliability'' issue.

VII. CONCLUSION

In this article, we presented the selective swap scheme for maintaining the context of smartphone applications. Our scheme classifies applications based on their context-saving characteristics, and selectively supports swap for applications that do not save context by themselves. We showed that the heavy I/O traffic problem of smartphone's swap does not occur in recent Android devices unless the number of concurrent applications becomes excessively large.

To avoid such thrashing situations, we make use of system status and application characteristics in controlling the number of applications involved in swap. Measurement studies under real Android reference devices showed that our scheme maintains the full context of applications without performance degradations in terms of application's launch time and their variations.

In the future, we will implement our scheme on a more recent Android reference device and perform measurement studies to see the effectiveness of the proposed scheme according as the hardware spec of a smartphone is improved.

REFERENCES

[1] S. Bae, H. Song, C. Min, J. Kim, and Y. Eom, ''EIMOS: Enhancing interactivity in mobile operating systems,'' *Lect. Notes Comput. Sci.*, vol. 7335, pp. 238–247, 2012.
[2] I. Bisio, F. Lavagetto, M. Marchese, and A. Sciarrone, ''GPS/HPS-and Wi-Fi fingerprint-based location recognition for check-in applications over smartphones in cloud-based LBSs,'' *IEEE Trans. Multimedia*, vol. 15, no. 4, pp. 858–869, Jun. 2013.
[3] F. Huang, X. Li, S. Zhang, J. Zhang, J. Chen, and Z. Zhai, ''Overlapping community detection for multimedia social networks,'' *IEEE Trans. Multimedia*, vol. 19, no. 8, pp. 1881–1893, Aug. 2017.
[4] D. Kim, S. Lee, and H. Bahn, ''An adaptive location detection scheme for energy-efficiency of smartphones,'' *Pervas. Mobile Comput.*, vol. 31, pp. 67–78, Sep. 2016.
[5] N. Islam and R. Want, ''Smartphones: Past, present, and future,'' *IEEE Pervas. Comput.*, vol. 13, no. 4, pp. 89–92, Oct. 2014.
[6] G. Sun, Y. Xie, D. Liao, H. Yu, and V. Chang, ''User-defined privacy location-sharing system in mobile online social networks,'' *J. Netw. Comput. Appl.*, vol. 86, pp. 34–45, May 2017.
[7] S.-H. Kim, J. Jeong, J.-S. Kim, and S. Maeng, ''SmartLMK: A memory reclamation scheme for improving user-perceived app launch time,'' *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 3, Jul. 2016, Art. no. 25.
[8] R. Prodduturi, ''Effective Handling of Low Memory Scenarios in Android Using Logs,'' Ph.D. dissertation, Indian Inst. Technol., Mumbai, India, 2013.
[9] A. Silberschats, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. New York, NY, USA: Wiley, 2013.
[10] K. Zhong, T. Wang, X. Zhu, L. Long, D. Liu, W. Liu, Z. Shao, and E. H.-M. Sha, ''Building high-performance smartphones via non-volatile memory: The swap approach,'' in *Proc. 14th Int. Conf. Embedded Softw. (EMSOFT)*, 2014, pp. 1–10.
[11] D. Liu, K. Zhong, X. Zhu, Y. Li, L. Long, and Z. Shao, ''Non-volatile memory based page swapping for building high-performance mobile devices,'' *IEEE Trans. Comput.*, vol. 66, no. 11, pp. 1918–1931, Nov. 2017.
[12] K. Zhong, D. Liu, L. Long, J. Ren, Y. Li, and E. H.-M. Sha, ''Building NVRAM-aware swapping through code migration in mobile devices,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 11, pp. 3089–3099, Nov. 2017.
[13] Y. Park and H. Bahn, ''Challenges in memory subsystem design for future smartphone systems,'' in *Proc. IEEE Int. Conf. Big Data Smart Comput. (BigComp)*, Feb. 2017, pp. 255–260.
[14] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson, ''Onyx: A prototype phase change memory storage array,'' in *Proc. USENIX Conf. Hot Topics Storage File Syst. (HotStorage)*, 2011, p. 1.
[15] H. Wong, S. Raoux, S. Kim, J. Liang, J. Reifenberg, B. Rajendran, M. Asheghi, and K. Goodson, ''Phase change memory,'' *Proc. IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.
[16] A. V. Khvalkovskiy, D. Apalkov, S. Watts, R. Chepulskii, R. S. Beach, A. Ong, X. Tang, A. Driskill-Smith, W. H. Butler, P. B. Visscher, D. Lottis, E. Chen, V. Nikitin, and M. Krounbi, ''Basic principles of STT-MRAM cell operation in memory arrays,'' *J. Phys. D, Appl. Phys.*, vol. 46, no. 7, Feb. 2013, Art. no. 074001.
[17] J. Huang, M. Qureshi, and K. Schwan, ''An evolutionary study of Linux memory management for fun and profit,'' in *Proc. USENIX Annu. Tech. Conf.*, 2016, pp. 465–478.

[18] J. Lee, K. Lee, E. Jeong, J. Jo, and N. B. Shroff, ''CAS: Context-aware background application scheduling in interactive mobile systems,'' *IEEE J. Sel. Areas Commun.*, vol. 35, no. 5, pp. 1013–1029, May 2017.
[19] S. Rostedt, ''Ftrace linux Kernel tracing,'' in *Proc. Linux Conf. Japan*, 2010, pp. 1–50.
[20] *Odroid-Q*. Accessed: May 1, 2019. [Online]. Available: http://www.hardkernel.com/
[21] J. Kim and H. Bahn, ''Comparison of hybrid and hierarchical swap architectures in Android by using NVM,'' *J. Semicond. Technol. Sci.*, vol. 18, no. 6, pp. 651–657, Dec. 2018.
[22] L. La. (2013). *Google Nexus 5 Review: A Nexus with Power, Potential, and the Right Price*. [Online]. Available: https://www.cnet.com/reviews/google-nexus-5-review
[23] 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.
[24] S. Perez, ''Report: Smartphone owners are using 9 apps per day, 30 per month,'' TechCrunch, San Francisco, CA, USA, Tech. Rep. 20170504, 2017. [Online]. Available: http://goo.gl/gnrvsF
[25] S. Lee, H. Bahn, and S. H. Noh, ''CLOCK-DWF: A Write-History-Aware page replacement algorithm for hybrid PCM and DRAM memory architectures,'' *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2187–2200, Sep. 2014.
[26] *The Linux Kernel Archives*. Accessed: May 1, 2019. [Online]. Available: https://www.kernel.org
[27] N. Gupta. *Compcache: Compressed Caching for Linux*. Accessed: May 1, 2019. [Online]. Available: http://code.google.com/p/compcache, 2010.
[28] 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,'' *IEEE Access*, vol. 6, pp. 56099–56108, 2018.
[29] D. Chae, J. Kim, Y. Kim, J. Kim, K.-A. Chang, S.-B. Suh, and H. Lee, ''CloudSwap: A cloud-assisted swap mechanism for mobile devices,'' in *Proc. 16th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGrid)*, May 2016, pp. 462–472.
[30] X. Zhu, D. Liu, L. Liang, K. Zhong, M. Qiu, and E. H.-M. Sha, ''Swap-Bench: The easy way to demystify swapping in mobile systems,'' in *Proc. IEEE 17th Int. Conf. High Perform. Comput. Commun., 7th Int. Symp. Cyberspace Saf. Secur., IEEE 12th Int. Conf. Embedded Softw. Syst.*, Aug. 2015, pp. 497–502.
[31] Y. Kim, M. Imani, S. Patil, and T. S. Rosing, ''CAUSE: Critical application usage-aware memory system using non-volatile memory for mobile devices,'' in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, Nov. 2015, pp. 690–696.
[32] S.-H. Kim, J. Jeong, and J. Kim, ''Application-aware swapping for mobile systems,'' in *Proc. ACM EMSOFT Conf.*, 2017, pp. 1–9.
[33] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, ''LiveLab: Measuring wireless networks and smartphone users in the field,'' *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 3, pp. 15–20, Jan. 2011. [Online]. Available: http://livelab.recg.rice.edu
[34] O. Kwon, K. Koh, J. Lee, and H. Bahn, ''FeGC: An efficient garbage collection scheme for flash memory based storage systems,'' *J. Syst. Softw.*, vol. 84, no. 9, pp. 1507–1523, Sep. 2011.

JISUN KIM received the B.S. degree in the computer science and engineering from Hanshin University, in 2011. She is currently pursuing the Ph.D. degree in computer science and engineering with Ewha Womans University, Seoul, South Korea.

Her research interests include operating systems, storage systems, caching algorithms, system optimizations, mobile systems, software platform technologies, block chain technologies, and embedded systems.

HYOKYUNG BAHN (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science and engineering from Seoul National University, in 1997, 1999, and 2002, respectively.

He is currently a Full Professor of computer science and engineering with Ewha Womans University, Seoul, South Korea. He has published more than 70 articles in leading conferences and journals in these fields, including USENIX FAST, the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, and *ACM Transactions on Storage*. His research interests include operating systems, caching algorithms, storage systems, embedded systems, system optimizations, and real-time systems. He also received the Best Paper Awards at the USENIX Conference on File and Storage Technologies, in 2013.