Efficient Memory Reclaiming for Mitigating Sluggish Response in Mobile Devices

Minho Ju*†, Hyeonggyu Kim*, Mincheol Kang*, and Soontae Kim*
*School of Computing, KAIST, Daejeon, Korea
†Samsung Electronics Co., Ltd., Suwon, Korea
*{minhoju, hyeonggyu, mincheolkang, kims}@kaist.ac.kr, †minho7.ju@samsung.com

***Abstract*—Mobile devices based on flash memory have unique hardware characteristics. They have different memory management mechanisms such as no memory swapping and app cache. Android platform adopts new modules such as low memory killer (LMK), activity manager service (AMS) besides kswapd and out of memory killer (OOMK). However, these modules generate many Kernel function calls that incur sluggish responses and inefficient app cache utilization, which reduces the chances of fast app re-launching. Thus, we propose an efficient memory management module called memory orchestration system (MOS) using dynamic page cache size and killed application counts. According to the experiments with 26 applications, it reduced the number of function calls by 84%. In addition, MOS keeps efficiently the app cache compared to the original Android system by 34.1%.**

I. Introduction

Mobile devices have the different mechanism for RAM management from that of PCs, because they are based on MTD (Memory Technology Device) that interacts with flash memory. For memory management, mobile devices do not perform memory swapping, and do not kill the applications that users send to the background. Instead, they put the applications into a specific area of RAM for fast re-launching, which is called *app cache*. On Android platform, the memory management is based on Linux Kernel such as `kswapd` module. They also have other memory modules: `AMS` (Activity Manager Service), `LMK` (Low Memory Killer), and modified `OOMK` [1], [2]. AMS and LMK kill applications when the number of background applications is more than 24, and the free memory is under 120 MB, respectively.

TABLE I shows the list of widely used recent applications with the max PSS (Proportional Set Size) which are ranked within the top 100 in Google Play [3]. The max PSS indirectly indicates the maximum memory size the application demands. For example, the game applications require 230 MB to 400 MB. Executing applications involves memory reclaiming operations to secure memory. We analyzed this through full system simulation [4]. We executed the applications which need 200 MB and 300 MB on two memory situations; the first is to execute them right after booting on the system and the second is to execute them after various 26 applications ran, which makes the free memory of about 100 MB. As a result, we observe that launch time is extended by 130 and 150 milliseconds, respectively, while 96 billion instructions are additionally executed on both cases as shown in Fig. 1. Furthermore, we found that more Kernel functions are called in the latter case due to frequent memory reclaiming. This eventually makes the sluggish response, which degrades the performance and makes worse the user experience.

M. Ju, H. Kim, and M. Kang equally contributed to this work.

TABLE I. The maximum memory (PSS) of famous applications of Google Play on nexus 5

Name	Category	Max PSS	Name	Category	Max PSS
Game of War	Game	401 MB	Cooking Fever	Game	237 MB
Facebook	Social	329 MB	The Weather Channel	Weather	198 MB
Candy Crush Saga	Game	328 MB	Waze	Travel and Location	193 MB
Clash of Clans	Game	314 MB	Kik	Communication	165 MB
Pinterest	Social	276 MB	SoundCloud	Music and Audio	153 MB
Instagram	Social	248 MB	Skype	Communication	129 MB
Empire War: Age of Hero	Game	248 MB	eBay	Shopping	135 MB
Dubsmash	Media and Movie	246 MB	Amazon Shopping	Shopping	124 MB
Angry Bird	Game	240 MB	Twiter	Social	122 MB

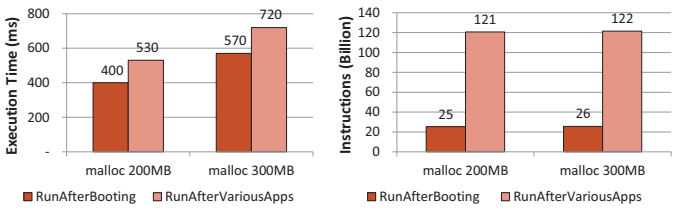

Fig. 1. The impacts on execution time (left) and instructions (right) depending on memory reclaiming.

In this paper, we introduce the problems caused by memory reclaiming in the Android platform, and propose the method that can efficiently reclaim the memory by avoiding the extensive Kernel function calls that make the sluggish response. To this end, we design a Kernel module called *MOS (Memory Orchestration System)* that has an interface to the user space using *Sysfs*.

The rest of this paper is organized as follows. In Section II, we show the problems of the conventional mechanisms with the background of them. Section III introduces some previous works and Section IV describes our efficient memory reclaiming method. Lastly, Section V analyzes the experimental results before concluding this paper in Section VI.

II. Background and Motivation

In Android system, the modules related to memory reclaiming include AMS in framework layer and kswapd, LMK, and OOMK in Kernel layer.

978-1-4799-8748-1/15/$31.00©2015IEEE

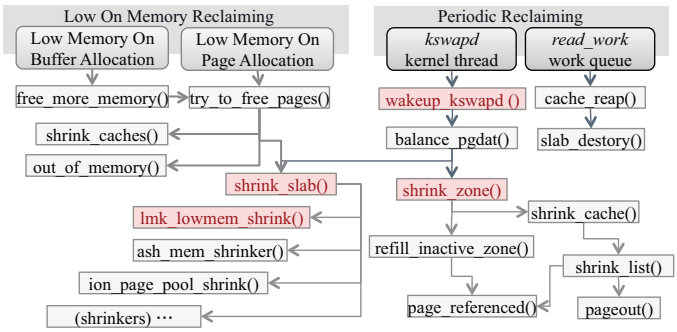

Fig. 2. Kernel functions related with memory reclaiming [5].

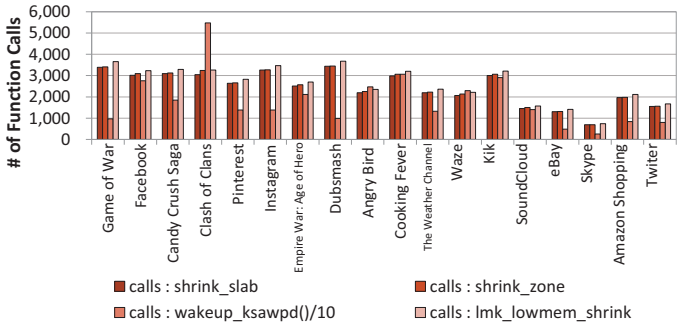

Fig. 3. The number of memory reclaiming-related Kernel function calls in the famous applications.

A. Android Memory Reclaiming Policies

Activity Manager Service (AMS). Each Android application has priority depending on the application state. This priority value is determined by oom_adj. For example, foreground application which is currently activated has high priority (low oom_adj value) than cached applications that are not visible applications. Thus, user event can change the priority of application. AMS manages application priority and also checks the number of cached applications, whether it is over the fixed threshold at 24. When the number of cached applications is higher than the AMS's threshold, AMS kills the cached application by LRU policy. Note that AMS does not consider the memory conditions.

kswapd. In Android Kernel, memory reclaiming policies can be operated on two conditions. One is that memory allocation is failed because free memory is not enough for allocating required memory. The other is the kswapd module which periodically checks free memory and is activated when free memory is less than the watermark value (3 MB to 5 MB in the default Android Kernel). As shown in Fig. 2, eventually, `shrink_zone` and `shrink_slab` are called together to gain free memory from `try_to_free_pages` and `wakeup_kswapd`. The `shrink_zone` scans each zone to find page caches that have to be reclaimed by LRU lists. However, the number of reclaimed pages is fixed at 32 (128 KB). Thus, if an application requires free memory of larger than 128 KB, `shrink_zone` has to be called repeatedly to meet the required free memory of the application. For example, if a user launches the application that requires 50 MB free memory, `shrink_zone` has to be called four hundred times.

Low Memory Killer (LMK). Android added LMK module for memory reclaiming which kills an application so that app cache memory reduces. If `shrink_slab` is called, LMK kills applications depending on the priorities received by the AMS. LMK explores all application to choose a victim application, but kills only one application. Thus, if the killed application capacity is lower than the required free memory, LMK has to be called two or more times to make free memory. For example, if an application requires 100 MB and current free memory is 50 MB but LMK selects a victim application that uses 30 MB, LMK must be called repeatedly. Even though AMS can kill only cached applications, LMK can kill higher priority applications such as previously launched applications and foreground applications.

Out Of Memory Killer (OOMK). If Android Kernel functions fail to reclaim memory, `out_of_memory` is called. This function is the last bastion to make the free memory by killing an application based on the OOM scores, which are calculated using several parameters such as process memory, CPU time, run time, and oom_adj. On the other hand, if the others Kernel functions related to memory reclaiming are called and make the free memory successfully, OOMK dose not need to call.

B. Inefficient Memory Reclaiming

We analyzed Android Kernel and modules related to memory reclaiming with Android applications which require large memory over 100 MB to 300 MB. Thus, we experimented using `Nexus 5` with Android Kitkat 4.4.4 r1.0.1 as a target device, and execute 18 applications in TABLE I. They are executed when free memory is under 50 MB in order to call Kernel functions related to memory reclaiming. Also, we traced Kernel functions using `FTrace`.

In order to estimate how many times those functions related to memory reclaiming are called, we count the numbers of `shrink_zone`, `shrink_slab`, `lmk_lowmem_shrink`, and `wakeup_kswapd` function calls. Since `shrink_zone` and `shrink_slab` can be called when reclaiming memory from the page cache and app cache, the number of `lmk_lowmem_shrink` calls can show how many times Android tries to kill applications. Then, the number of `wakeup_kswapd` calls indicates how many times kswapd is activated.

Sluggish Response. Our observation results are shown in Fig. 3. The number of total function calls such as `shrink_zone`, `shrink_slab`, `lmk_lowmem_shrink`, and `wakeup_ kswapd` is 25,718 on average. Moreover, the number of `do_shrinker_shrink`, which is called in `shrink_slab` to call shrinkers, is 70,800 on average. Also, these functions are called simultaneously like fine-grained strategy. Thus, this many function calls can extend application launch time.

Also, we experiment using the same device and execute 26 benchmarks for observing and evaluating the proposed scheme with representing various application categories, instead of TABLE I applications. The reason why we choose 26 applications is to call those functions by reducing free memory gradually and show killing application scenario by AMS. We use `MonkeyRunner` to run the applications automatically in the order named in TABLE. II. There are heavy applications (App

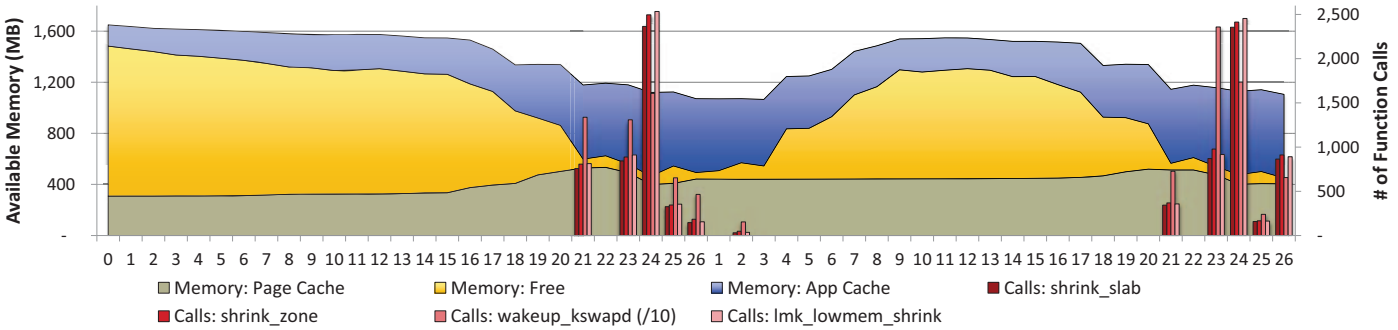

Fig. 4. Memory usage trends with the number of function calls related with memory reclaiming in the original Android.

TABLE II. THE APPLICATION LIST WITH THE MAXIMUM MEMORY (PSS) ON NEXUS 5

ID	Name	Max PSS	ID	Name	Max PSS	ID	Name	Max PSS
0	(boot)	-	9	Messaging	22 MB	18	Camera	110 MB
1	Calculator	28 MB	10	Video Editor	29 MB	19	Slideshow	144 MB
2	Calendar	30 MB	11	Music	8 MB	20	Talking Toms	96 MB
3	Clock	45 MB	12	People	29 MB	21	Asphalts	286 MB
4	DevTools	28 MB	13	Phone	38 MB	22	Minecraft	43 MB
5	Download	29 MB	14	Unity3D	53 MB	23	Boeing	124 MB
6	Email	40 MB	15	Settings	33 MB	24	Turbo	262 MB
7	Gallery	37 MB	16	Browser	171 MB	25	Angry Bird	240 MB
8	Adobe Reader	22 MB	17	Movie	144 MB	26	Temple Run	137 MB

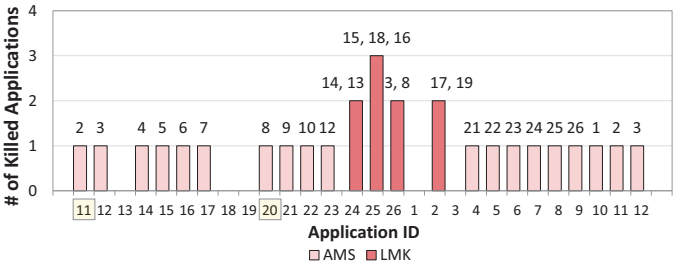

Fig. 5. Killer and killed applications on the Android platform. The numbers at the top of each bar indicates the killed applications and X-axis indicates killer applications.

ID: 16, 17, 19, 21, 23, 23, 24, and 26) among the benchmarks. Each PSS of them is over LMK threshold (120MB). Most of them are almost game application, movie or photography, which are visual applications.

Inefficient App Cache. Fig. 4 shows memory usage and the number of Kernel functions calls by executing applications twice in TABLE. II. As free memory is decreased, eventually, Android memory reclaiming functions are called to make free memory from the app 21 to 26. After then, free memory is increased because AMS kills heavy applications by LRU order when we re-launch the app 4 to 9 in Fig. 5. However, AMS has two problems for app cache. First, AMS does not consider the memory situation of the system and the used memory of the application to kill when it kills an application. Although there is enough free memory on the system, AMS kills an application based on the fixed number of background applications. Second, we can find that AMS cannot cover all cached applications, for example AMS does not kill applications 11 and 20 as shown in Fig. 5.

III. RELATED WORK

In [6] and [7], the authors attempted to improve performance by modifying AMS. Because the AMS uses a fixed value which is the maximum number of cached applications, [6] analyzed the reuse patterns of smartphone applications, and dynamically adjusted the limited number of background processes based on app-usage patterns. [7] changed the LRU-based replacement policy in the AMS. They suggested a pattern-based replacement algorithm using a Markov Decision Process (MDP) model. However, as we explained in Section II, AMS cannot cover all the applications, which means that it only kills background applications and cannot reclaim page cache to make free memory.

IV. EFFICIENT MEMORY RECLAIMING BASED ON MOS (MEMORY ORCHESTRATION SYSTEM)

We propose a method that adjusts the page cache size dynamically and kills the applications at a time as needed by utilizing the threshold of LMK and the maximum memory of each application from AMS. The difference of architecture is shown in Fig. 6. In Android platform, AMS changes the application priority using oom_adj parameter and kills the cached application when the number of applications is over a threshold value. However, AMS does not request directly to LMK for killing the application. In MOS, in contrast, the modified AMS can request the required memory of the application via MOS module in Kernel. We modified the AMS not to kill the background applications. The MOS module is the loadable Kernel module that provides the Sysfs interface to the user space. It can call directly the page allocator and memory shrinker in Kernel. Therefore, the MOS can calculate the memory to reclaim from the application cache and page cache at once so that LMK and kswapd can run efficiently.

The MOS module in Kernel provides the Sysfs interfaces to provide the reclaiming commands and tuning parameters as shown in TABLE III with the /sys/Kernel/memorchestra/ file path. The key interface to run efficient memory reclaiming is `run_free_memory` file. When AMS writes the value with 1 to the file, the MOS module in Kernel reclaims the memory as much as the LMK threshold, for example 120 MB on Nexus 5. When AMS writes the value with over 1 to the file, the MOS module considers that the input value is the memory to reclaim quickly. So, the MOS module directly kills the background applications as much memory as AMS needs. The target applications to kill are same as LMK because the criterion to choose applications is based on the oom_adj parameter of each process. However, MOS differs from LMK in the aspect of the number of killing applications at once. The MOS module can kill a number of applications as needed

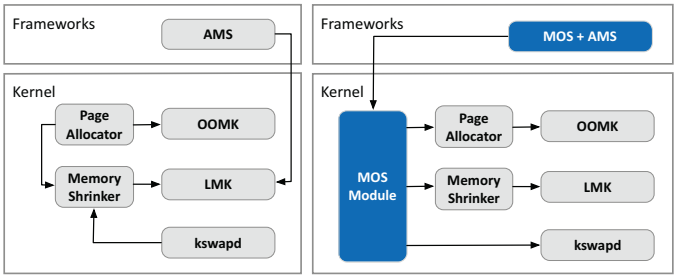

Fig. 6. Memory management systems of Android (left) and MOS (right).

TABLE III. MOS SYSFS INTERFACES TO PROVIDE THE RECLAIMING COMMANDS AND TUNING PARAMETERS.

Category	Sysfs File Name	Description
Command	run_free_memory	Reclaim the page cache and application cache.
	run_shrink_slab	Call the shrink_slab(). Reclaim slab objects and call the shrinkers.
	run_shirink_zone	Reclaim the page cache and normal and high zone.
Tuning Parameter	nr_pages_scanned	How many slab objects shrinkers() should scan and try to reclaim.
	nr_to_reclaim	Target number of pages to be reclaimed.
	priority	Priority of the scanning, ranging between 12 and 0. Lower priority implies scanning more pages.

to reduce the computation for selecting targets. Moreover, we provide interfaces to control the memory-reclaiming module such as shrinkers and kswapd with its tuning parameters.

Our key approach for reducing sluggish response is to reclaim memory as needed just before each application launches. To this end, MOS reclaims the memory when home (launcher) application runs (*MOS Phase 1*) and when each application is executed (*MOS Phase 2*) for faster application launch as shown in Fig. 7. To operate this in Android, we modified `ActivityStackSupervisor` class in AMS so that MOS catches when the applications including home launch or move to foreground application from the background. In MOS Phase 1, the MOS module reclaims the memory of each cached application and page cache until the free memory is the LMK threshold (120 MB in Nexus 5) that covers the normal applications such as Clock and Contacts. In MOS Phase 2 that is the time to run just before the application, AMS confirms the used maximum memory of the application and if the maximum memory is over the LMK threshold, AMS requests the memory reclaiming to MOS module in Kernel with the maximum memory. After Android 4.4 version, AMS keeps the average and maximum PSS memory in AMS and Android platform provides the tool named `procstats` to developer or users for confirming the memory information. Therefore, it does not need to periodically calculate the maximum memory of each application additionally.

The detailed mechanism of reclaiming memory in MOS module is shown in Algorithm 1. Depending on the writing of the aforementioned `run_free_memory` Sysfs file, MOS module executes the `doMosPhase1` or `doMosPhase2` function. In `doMosPhase1` function, it calculates the reclaiming memory size of application cache and page cache and calls `reclaimPageCache` and `reclaimAppCache` functions. In `reclaimPageCache` function, it calls the original Kernel memory reclaiming functions. However, MOS module controls dynamically the reclaiming page count to reduce the number

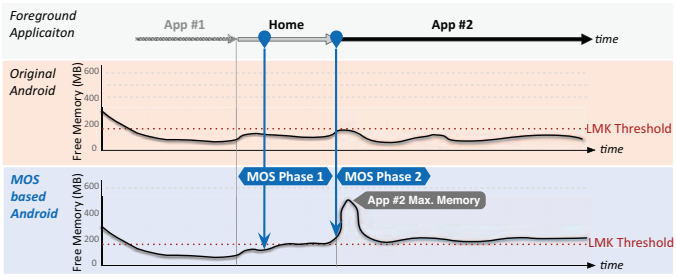

Fig. 7. The behavior example of the MOS reclaiming algorithm using Home application and maximum application memory.

Algorithm 1 Pseudo code for the *MOS Reclaiming* algorithm.

```
1:  function VOID DOMOSPHASE1
2:      memToReclaim ← tLmk - getFreeMem();
3:      recPageCache ← memToReclaim - tPageCache;
4:      recAppCache ← recToReclaim - recPageChage;
5:      reclaimPageCache(recPageCache);
6:      reclaimAppCache(recAppCache);
7:  end function
8:  function VOID DOMOSPHASE2(maxAppMem)
9:      recAppCache ← maxAppMem - getFreeMem();
10:     killAppCache(recAppCache);
11: end function
12: function VOID RECLAIMPAGECACHE(recPageCache)
13:     shrinkCache(recPageCache);
14:     shrinkSlab(recPageCache);
15:     thresReclaimPageCount ← ×2;
16:     doMosPhase1();
17: end function
18: function VOID RECLAIMAPPCACHE(recAppCache)
19:     while (memTatalReclaim ≥ recAppCache) do
20:         pid ← maxOomAdjustment();
21:         memTatalReclaim ← killapp(pid);
22:     end while
23: end function
```

of Kernel function calls. In `reclaimAppCache` function, it kills the applications that have the maximum OOM (Out Of Memory) adjustment until the total reclaimed memory is over the requested memory to reclaim. Because the `doMosPhase2` function only calls the `reclaimAppCache` function, our MOS system can reclaim memory rapidly when the application launches.

Our main algorithm is implemented in MOS loadable module in Kernel that has only one file with 932 lines. We added two files to Kernel that are `mm/vmscan.c` and `include/linux/swap.h` for MOS to call directly the functions related to memory reclaiming. Each added line is 26 lines and 2 lines, respectively. Moreover, we do not change the original memory reclaiming mechanism for the other system for the portability and compatibility. Because the MOS module in Kernel is executed by the Sysfs interface from the user space, even our MOS can be applied to another system using the Linux Kernel in addition to Android.

V. EVALUATION

Fig. 4 and Fig. 8 show memory usage trends of page cache, free memory, and the app cache with the number of

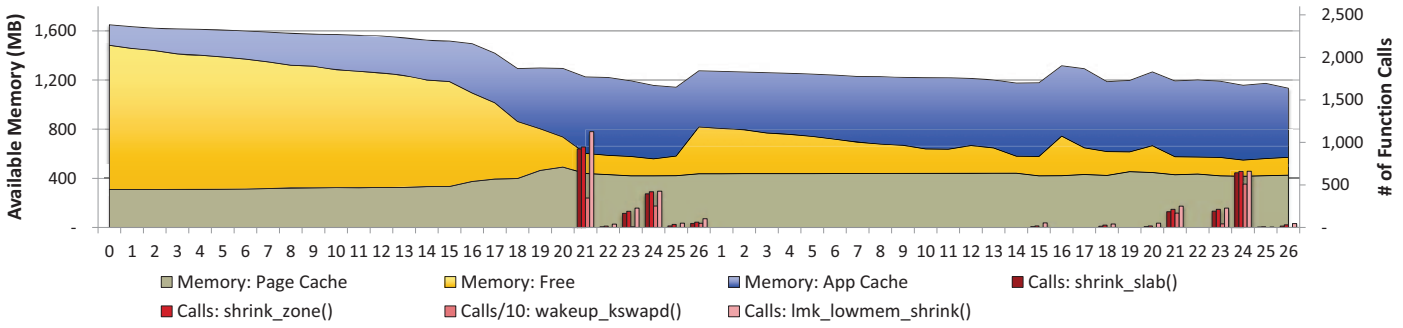

Fig. 8. Memory usage trends with the number of function calls related with memory reclaiming in the MOS system. Note that the utilization of app cache in the MOS is 34.1% better than Android.

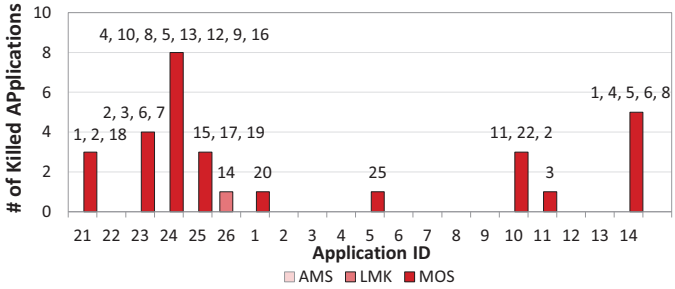

Fig. 9. Killer and killed applications in MOS system.

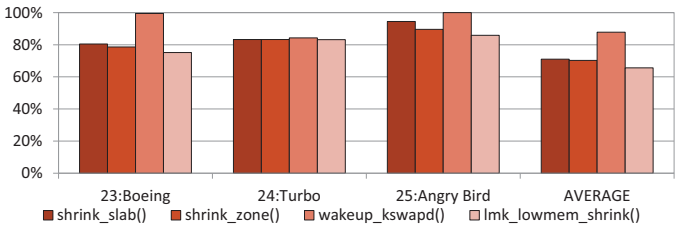

Fig. 10. The ratios of reduced function calls.

memory reclaiming-related function calls. For the page cache, both MOS and conventional Android platform manage stably as amount as almost 400 MB. However, free memory and app cache are fluctuated when applications are launched and re-executed in the existing Android platform. In other words, app cache rapidly diminishes in a moment and the free memory is increased.

The figure indicates the two better points compared to Fig. 4. First, the free memory is not unnecessarily reclaimed so that app cache space can be stably maintained – *MOS shows 34.1% better efficiency*, which is measured by comparing the relative app cache size in the re-execution phase. It extends applications' lifetime so that it can guarantee the faster re-execution of applications because it does not need to reload them from the flash memory. Second, it dramatically reduces the number of main Kernel function calls. It means that MOS prepares sufficient memory in advance for the next applications. As shown in Fig. 9, unlike the existing Android platform, one application is killed by LMK in our MOS when executing the 16th app because it suffers from the memory shortage in that stage.

The important functions on the Android platform, such as `shrink_slab`, `shrink_zone`, `wakeup_kswapd`, and `lmk_lowmem_shirnk` functions occur 140,902 times in total, while they occur only 22,567 times in MOS system. Fig. 10 shows the ratio of reduced function calls of three heavy applications as an example. Each reduced rate of above functions is 71.0%, 70.3%, 87.8%, and 65.8%, and the rate of total number of calls is 84.0%. Thanks to the efficient memory reclaiming, we have sufficient memory. Thus, it does not need to call those functions when launching applications.

VI. Conclusion

Modern mobile devices have relatively small RAM size, but the required memory grows. In addition, applications are not intelligently killed due to the discord among memory managing modules, so that memory reclaiming can frequently occur when launching an application. Those memory reclaiming may incur sluggish and harm the user experience.

To address this, we proposed an efficient memory reclaiming method for mitigating sluggish response based on the MOS Kernel module. The MOS secures the required memory in advance. And, it efficiently calls the memory reclaiming-related functions with dynamic page cache size and killed application counts, so that it prevents the extensive function calls that make the sluggish response when launching applications. According to our experiments, it reduced the main Kernel function calls by 84.0%. Moreover, it can maintain the app cache 34.1% more efficiently than Android.

References

[1] D. Rientjes, "OOM Killer Rewrite; When the Kernel Runs Out of Memory," in *Proceedings of LinuxCon Boston*, August 2010.

[2] D. Kayande and U. Shrawankar, "Performance Analysis for Improved RAM Utilization for Android Applications," in *Proceedings of the International Conference on Software Engineering (CONSEG)*, 2012.

[3] Google Play. [Online]. Available: https://play.google.com/store [Accessed May 2, 2015].

[4] A. Gutierrez *et al.*, "Sources of Error in Full-System Simulation," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[5] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*. O'Reilly Media, Inc., 2005.

[6] K. Baik and J. Huh, "Balanced memory management for smartphones based on adaptive background app management," in *Proceedings of the International Symposium on Consumer Electronics (ISCE)*, 2014.

[7] C.-Z. Yang and B.-S. Chi, "Design of an Intelligent Memory Reclamation Service on Android," in *Proceedings of the Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, 2013.