IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 31, 2019, accepted August 19, 2019, date of publication August 27, 2019, date of current version September 23, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2937852*

# Analysis of Smartphone I/O Characteristics – Toward Efficient Swap in a Smartphone

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

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

This work was supported in part by the ICT Research and Development Program of MSIP/IITP under Grant 2018-0-00549 (extremely scalable order preserving OS for many core and non-volatile memory) and under Grant 2019-0-00074 (developing system software technologies for emerging new memory that adaptively learn workload characteristics).

**ABSTRACT** Due to the recent advances in mobile platform technologies, people are increasingly working with their smartphones. For example, digital healthcare, automotive navigation, and stock trading are also performed by a smartphone as well as phone calls. However, there are some technical hurdles for executing reliable software in a smartphone. Specifically, current smartphones kill applications without using swap when free memory space is exhausted. Although supporting swap in a smartphone is not impossible, our observation shows that swap in Android increases storage accesses significantly, leading to thrashing conditions. To resolve this, we further analyze Android swap I/O traces and make two prominent observations. The first is the existence of hot 15% data, which account for 80% of total swap I/O, and the second is the existence of cold 50% data that are never used again after entering the swap area. Based on these observations, we present a new architecture that adopts non-volatile memory at the Android swap layer. Specifically, as Android swap has bimodal data access characteristics, we identify and manage hot and cold data efficiently by making use of precise admission control and replacement algorithms. This is possible as our swap architecture can access the full information of request time and frequency, which is different from the main memory layer with restricted information. Experimental results show that our architecture supports Android swap without performance degradations.

**INDEX TERMS** Android, mobile platform, smartphone, non-volatile memory, swap.

## I. INTRODUCTION

Due to the rapid proliferation of smartphone applications as well as the advances in mobile platform technologies, smartphone has now become one of the mainstream computing devices [1]. People are increasingly working with their smartphones or tablets, and a variety of new applications, including social network services, online multimedia games, and location-based services, emerge every day [2]–[4]. Actually, modern smartphone's hardware spec has already reached to that of a general purpose computing device like a desktop or a laptop [5]. For example, Google Nexus 6P, a reference phone of Android, consists of Qualcomm®SnapdragonTM 810, 2.0 GHz quad-core 64bit, Adreno 430, DDR4 3GB, and eMMC 128GB, which is sufficient to perform multitasking [6].

A smartphone is no longer a personal entertainment device but official works such as video editing, spreadsheet, e-banking, and software development are also supported by smartphones or tablets. Accordingly, desktop applications are increasingly compatible with smartphones by making use of external keyboard and/or screen devices. However, smartphone systems have critical weaknesses to be a general purpose multitasking computing device. Specifically, current smartphone platforms such as Android terminate processes without user's approval when free memory space is exhausted [7], [8]. This was not a serious issue when a smartphone was a personal entertainment device, but now it is critical to perform official works. For example, terminating a music player does not incur serious results but killing a video editor while editing a large movie file may cause significant problems.

To resolve this issue, smartphones should preserve processes unless users explicitly terminate applications. This can be realized by *swap*, which uses a certain portion of secondary storage as main memory's extension for saving application's memory data when free memory space is exhausted [9].

The associate editor coordinating the review of this article and approving it for publication was Yungang Zhu.

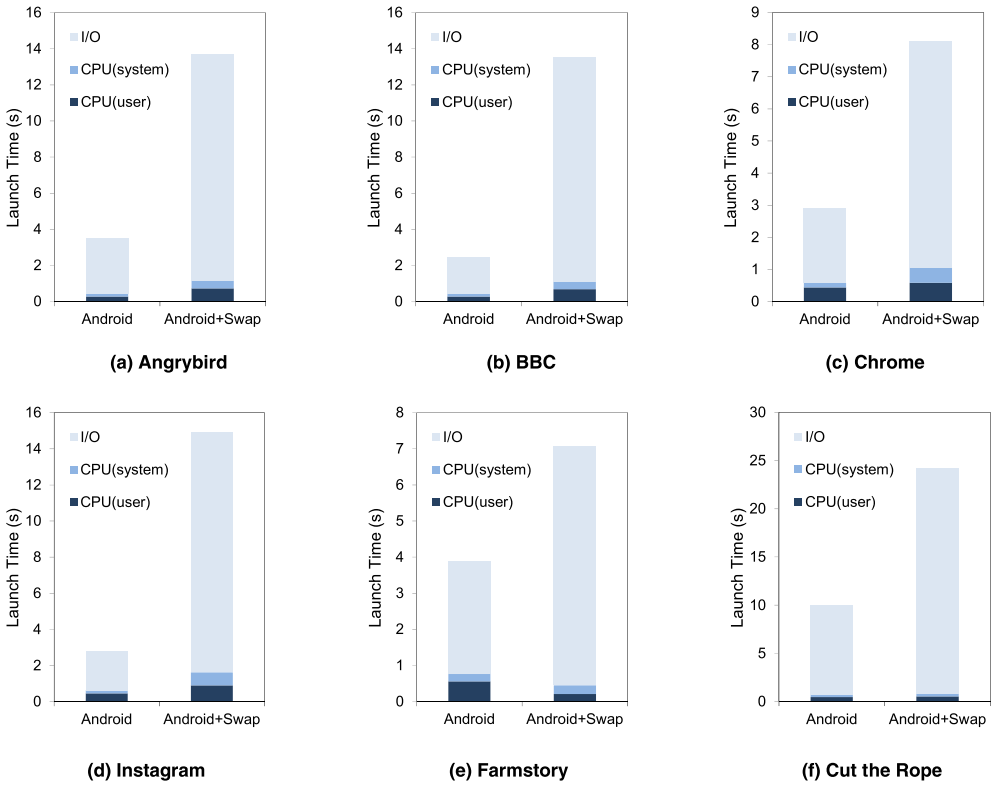


**FIGURE 1.** Comparison of application's launch time for original Android and swap-supported Android.

Although swap has been widely used in traditional computer systems, our analysis shows that supporting swap in a smartphone is not an easy matter. In particular, a serious system-wide thrashing happens in a swap-added Android device, slowing down the launch time of applications significantly. Nevertheless, we observe that such storage accesses are extremely skewed to 10-15% of hot data. Based on these observations, we present a new architecture utilizing non-volatile memory in order to eliminate a large portion of storage accesses in Android swap. In particular, we show that only a small size of non-volatile memory is sufficient to absorb hot data accesses by making use of efficient management techniques. Experimental results show that our architecture significantly reduces the number of storage accesses, thereby supporting swap without performance degradations.

### A. MOTIVATIONAL EXPERIMENTS

Our goal is to support swap in a smartphone, thereby making it a general purpose computing device like a desktop or a laptop. To do so, we reconfigure an Android smartphone and measure the performance of swap-supported Android in comparison with that of the original Android without swap. Fig. 1 depicts the launch time of various applications measured with original Android and swap-supported Android after their memory has been warmed up by a sufficient number of applications. (Details of experimental conditions will be explained later in Sections II and IV). As shown in the figure, supporting swap significantly degrades the smartphone performances. Specifically, application's launch time is increased by two to five times when swap is turned on.

This is not consistent with the results in some previous studies, in which swap only increases the application's launch time by 20-40% [26]. However, experiments in these studies were performed by executing a simple scenario with two applications. That is, a synthetic application with the given file size is executed first and then another application is performed to exhaust the remaining memory space; then the first application is executed again to measure the launch time. In reality, this is difficult to account for full phenomena that occur in a real Android device as we usually execute a large number of applications without restarting a phone for a long time.

Unlike previous studies, we observe the system state after sufficient time has passed since the smartphone started. That is, the number of applications executed increases as time progresses and we point out that a serious system-wide thrashing problem happens in an Android device with its swap function turned on.

The main source of this thrashing is the increased I/O time, which accounts for 80% of the total launch time as shown in Fig. 1. CPU time is also increased because increases in I/O time result in some CPU works such as I/O scheduling and memory reclamation for I/O requests. To support swap in a

smartphone, therefore, such negative effects in swap should be alleviated.

### B. CONTRIBUTIONS

The first contribution of this article is to analyze storage accesses of swap-supported Android generated by a variety of applications. Our analysis shows that swap-supported Android incurs 4-15 times more I/Os than original Android. 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. Another interesting observation is that storage accesses generated by swap-supported Android are extremely skewed. In particular, 10-15% of top ranking data account for about 80% of total storage accesses. This is different from original Android in that 50-60% of top ranking data account for 80% of total storage accesses.

Our second contribution is the design of a new architecture to eliminate a bunch of storage accesses generated by swap-supported Android. In particular, we adopt a small size of non-volatile memory to absorb hot data accesses that appear in swap-supported Android. The proposed architecture eliminates most of additional storage accesses by making use of the storage access characteristics observed in Android applications. Experimental results show that the proposed architecture reduces the number of storage accesses by 89% on average and up to 93%.

This article also quantifies the size of non-volatile memory required to support swap without performance degradations, and suggests appropriate policies for the non-volatile memory architecture. By identifying and maintaining top 10-15% data that account for most of storage accesses in swap-supported Android, we show that a small size of 256MB non-volatile memory is sufficient to eliminate additional storage accesses generated by swap. Specifically, we adopt an admission control policy that does not allow a large portion of cold data to be inserted into the non-volatile memory, thereby protecting expensive memory space from being polluted by non-profitable data. The saved space is utilized for maintaining hot data. Experimental results show that the application launch time of our swap-supported Android is improved by 82% on average, which is similar to the original Android.

The remainder of this article is organized as follows. Section II analyzes storage access characteristics extracted from smartphone applications. Section III presents the proposed architecture for improving swap performances and management algorithms used therein. The experiment results of the proposed architecture are given in Section IV. Finally, Section V concludes this article.

## II. ANALYZING STORAGE ACCESSES IN SWAP-SUPPORTED ANDROID

This section analyzes storage accesses of smartphone applications to investigate the overhead of swap in Android. To do this, we reconfigure the Android kernel to support virtual memory swap, and then collect storage access logs of swap-supported Android in comparison with those of original Android. We warm up memory by executing a variety of Android applications to make full use of available memory, and then induce swap situations.

**TABLE 1. Application scenarios used in the experiments.**

| Scenario | Applications |
|---|---|
| Scenario 1 | Angry Bird, Candy Crush, File Browser, Temple Run2, Balance 3D |
| Scenario 2 | BBC News, Chrome, NBC News, Weibo, Android Browser |
| Scenario 3 | Instagram, Twitter, Google Maps, Gmail, Candy Camera |
| Scenario 4 | Farm Story, PianoTile2, Cut the rope, Where is my water?, Block Puzzle |

Table 1 shows the scenarios we executed. Note that we run the applications in each scenario sequentially and then repeat them to see the effect of swap. As each scenario consists of a sufficient number of applications, applications are essential to be terminated and restart upon their next launch in original Android whereas swap-supported Android saves and restores application's memory data by making use of secondary storage. Our experimental setting consists of ODROID-Q with 1GB DDR2-DRAM memory and 2GB swap file on 16GB eMMC [10]. We set the swap file in the /mnt/sdcard/ partition. Table 2 shows the details of each partition we set.

**TABLE 2. Partition information of swap-supported Android.**

| Partition | Directory | Logical block address |
|---|---|---|
| /dev/block/mmcblk0p2 | /system | 0~1048576 |
| /dev/block/mmcblk0p3 | /data | ~3145728 |
| /dev/block/mmcblk0p4 | /cache | ~3276740 |
| /dev/block/void/179:1 | /mnt/sdcard | ~15257540 |

Fig. 2 shows the distribution of storage accesses when original Android and swap-supported Android are installed. In the figure, the $x$-axis represents the operation sequences and the $y$-axis shows the logical block numbers. The blue and red plots represent the read and write operations, respectively. As shown in the figures, the number of plots increases significantly when swap is turned on. This is because swap-supported Android repeatedly performs storage accesses for swap when free memory space is exhausted. Unlike swap-supported Android, original Android kills applications when there is not enough free memory. As it does not save application's process contexts or retrieve them from secondary storage, original Android incurs relatively small number of storage accesses.

Another important observation is that the distribution of swap-supported Android is skewed to a certain hot data blocks whereas that of the original Android is not so. Specifically, as available memory becomes insufficient, swap-supported Android swaps out anonymous pages in data,

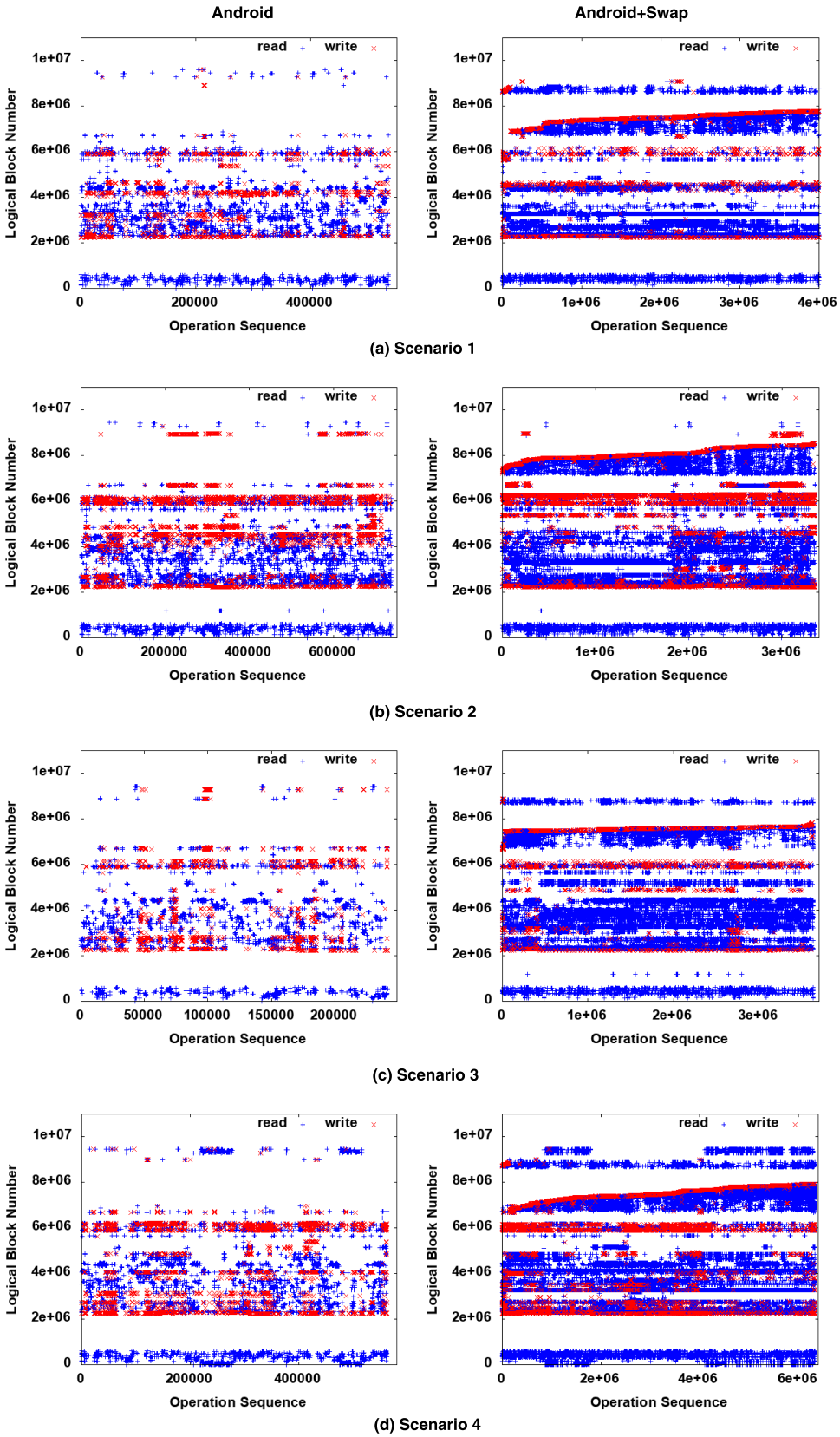


**FIGURE 2.** Distribution of storage accesses in original Android and swap-supported Android.

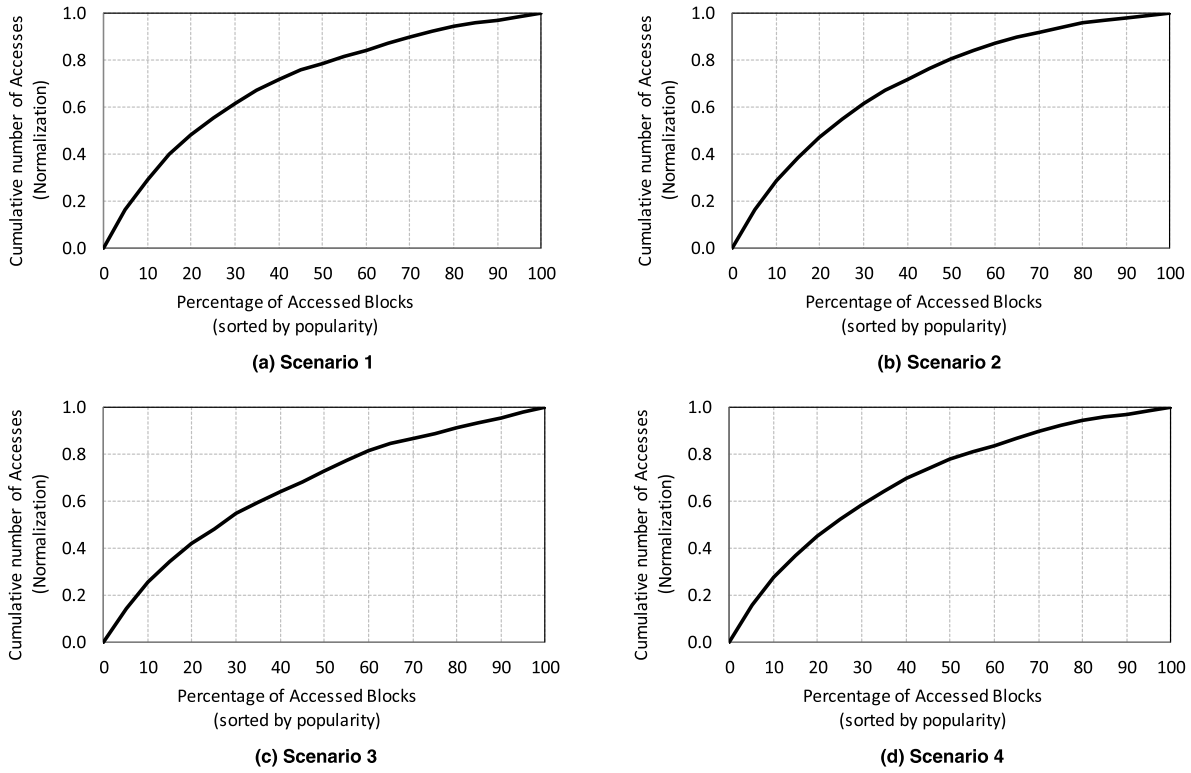


**FIGURE 3.** Cumulative distribution of storage accesses in original Android.

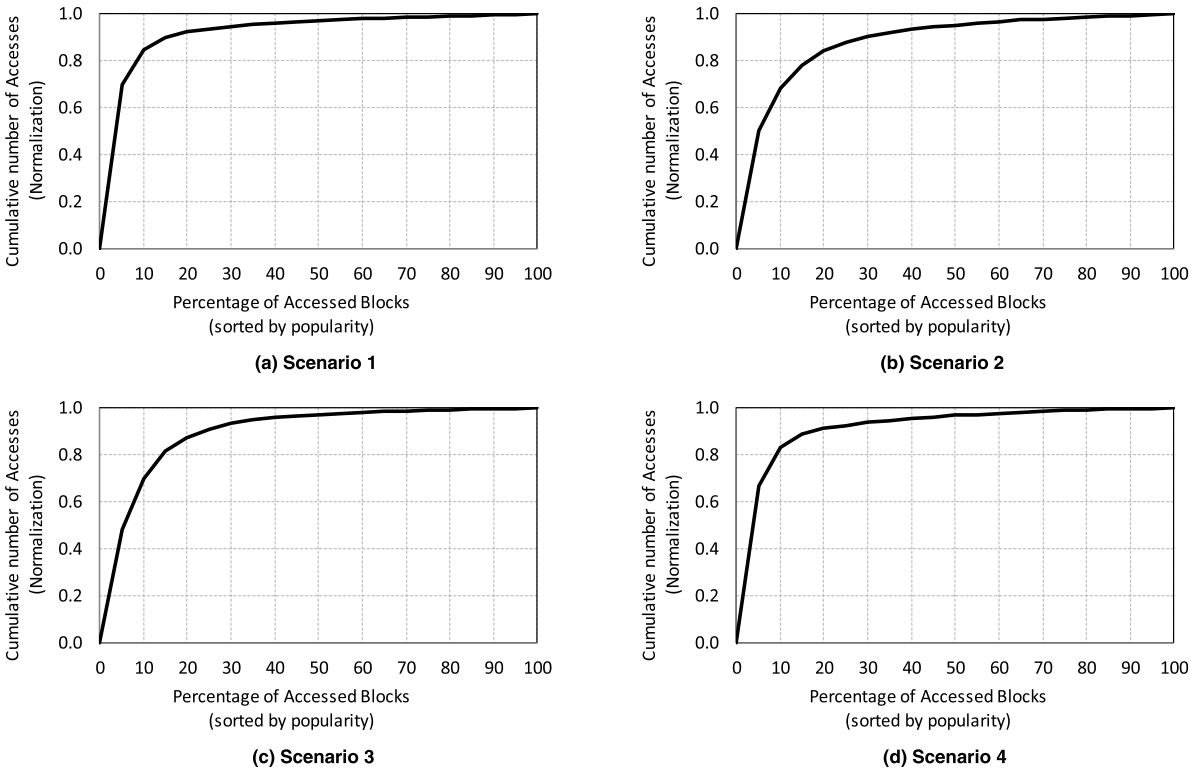


**FIGURE 4.** Cumulative distribution of storage accesses in swap-supported Android.

heap, and stack regions, and then reloads them into memory when the process is re-activated; this generates hot I/Os for the swap area.

Figs. 3 and 4, respectively, show the cumulative number of storage accesses for original Android and swap-supported Android. The $x$-axis represents the percentage of

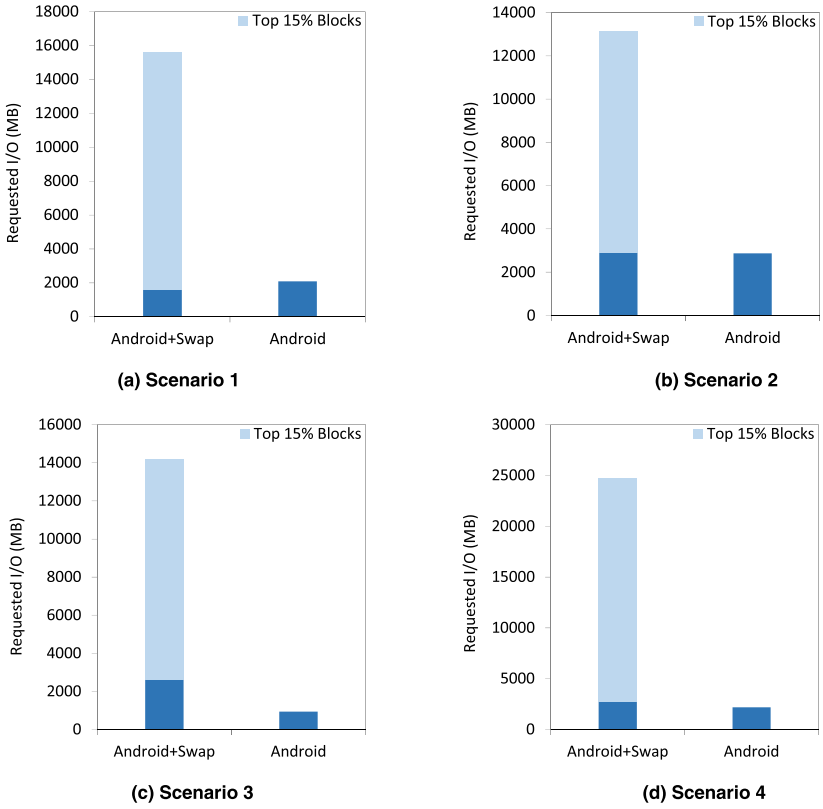


**FIGURE 5.** Comparison of I/O traffic for original Android and swap-supported Android.

accessed data sorted by their access frequency and the *y*-axis represents the ratio of accesses for the given fraction of data.

For example, 10% in the *x*-axis implies the top 10% data, and the corresponding point in the *y*-axis represents the ratio of storage accesses they made. As shown in Fig. 3, 50-60% of top ranking data account for 80% of total storage accesses in original Android. This implies that skewness of data accesses in original Android is relatively weak. In contrast, Fig. 4 shows that 10-15% of top ranking data account for 80% of total storage accesses in swap-supported Android, implying that storage accesses in swap-supported Android mostly result from some hot data.

Actually, these hot data should not appear in swap I/O traces, but be kept in memory, if the system is in its normal state. As the bottom core part of Android consists of the Linux kernel, which has a page reclamation module that only replaces inactive pages, existence of hot data in swap traces seems to be unnatural. We further analyzed the Android swap traces and found out that the hot data consist of some kind of essential Android services and shared libraries. As the number of concurrent applications increases, they are also evicted from memory to make free space and then reloaded soon because they are used again. This is different from the original Android without swap, in which applications are aggressively killed, making a certain level of free memory space although the number of concurrent applications increases.

Fig. 5 compares the total number of storage accesses in original Android and swap-supported Android. As shown in the figure, swap-supported Android incurs 9 times more storage accesses than original Android on average. In case of swap-supported Android, we separately show the storage accesses caused by top 15% of data and those by the others. As we see, top 15% of data account for 84% of total storage accesses; this implies that if we eliminate storage accesses caused by top 15% data, the total number of storage accesses in swap-supported Android will approach that of the original Android.

Our conclusion is that swap itself is not a problem but a lot of concurrent applications may cause serious thrashing if we activate swap in Android. Once thrashing happens, memory is exhausted and some essential parts of Android are also evicted from memory and then reloaded repeatedly. Furthermore, we observed that OOM (out of memory killer) is also activated when free memory space is almost exhausted in swap-supported Android. Note that OOM works when the page reclamation module fails to free page frames, thereby making the system difficult to normally operate. In that case, processes are killed based on their memory occupation and nice values until minimum free memory space is made.

By considering this, swap-supported Android needs a mechanism that does not incur such phenomena.

So far, we made an important observation that the top 15% of Android swap data account for 80% of total swap I/O, which is the main reason of thrashing. To resolve this thrashing problem, we will adopt a small size of non-volatile memory (NVM) as a fast swap device and aim at placing hot data on this NVM-swap.

A major role of our NVM-swap is to hold hot data, but it is also important to exclude cold data from entering NVM-swap. This is because hot data in NVM-swap may be pushed out by cold data as the capacity of NVM-swap is limited.

To see the effect of such characteristics, we classify storage accesses of swap-supported Android into single-access and multiple-accesses. As shown in Fig. 6, the ratio of single-access is over 50% for all scenarios we considered. Note that these large portions of single-access data will not be used again although we maintain them in fast NVM-swap.

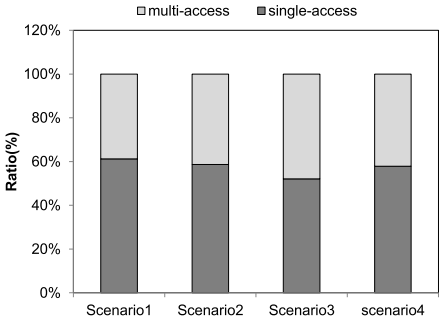


**FIGURE 6.** Ratio of single-access and multi-accesses in swap-supported Android.

Thus, identifying cold data and prohibiting them from entering NVM-swap can maximize the benefit of NVM as it prevents the pollution of NVM-swap from a large portion of single-access data that we observed in Android swap traces. Our observations indicate that detecting cold data that are unlikely to be re-referenced and preventing them from being loaded into NVM-swap is necessary. Unlike main memory systems, we can efficiently manage such cold data at the NVM-swap layer by injecting the admission control policy, which we will discuss in the next section.

## III. SUPPORTING SWAP IN SMARTPHONES

This section describes an efficient swap supporting mechanism for Android smartphones. We first present a new architecture by adopting non-volatile memory and show how such an architecture can be implemented efficiently.

### A. THE NEW SWAP ARCHITECTURE

Based on the observations in Section II, we present a new architecture to support virtual memory swap in Android without performance degradations. The proposed architecture exploits a small size of non-volatile memory residing between DRAM memory and secondary storage as shown in Fig. 7. The key idea of the proposed architecture is to keep hot data evicted from DRAM memory in non-volatile memory such as PCM (phase-change memory) or STT-MRAM (spin torque transfer magnetic RAM) [12], [13]. Non-volatile memory technologies provide persistent storage at low latency; their energy consumption is also very low compared to DRAM as they do not need refresh operations in idle states [14]–[16]. Patents published by Intel describe a detailed micro-architecture to support non-volatile memory in the storage hierarchy of computer systems, implying that the era of non-volatile memory is imminent [24], [25]. However, as non-volatile memory will not replace DRAM due to cost and/or performance, it is considered only as an add-on component to enhance performances [17], [18], [27], [28].

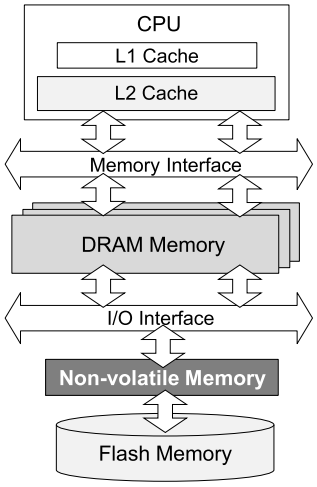


**FIGURE 7.** The proposed architecture to support swapping in Android.

In this article, we show that only a small size of non-volatile memory suffices to eliminate most storage accesses that additionally occur in swap-supported Android by adopting efficient management techniques. When there is not enough free memory in the system, swap-supported Android selects a certain number of memory data not used recently, and evicts them from DRAM memory. If the evicted data was modified while resident in DRAM memory, the data is first written to secondary storage before its eviction. In our architecture, instead of secondary storage, non-volatile memory temporarily holds data discarded from DRAM memory. Thus, whenever storage accesses are requested, non-volatile memory is checked before finding the data in secondary storage. If the requested data is found in non-volatile memory, it is loaded into DRAM memory without storage accesses.

### B. OPTIMIZED ADOPTION OF NON-VOLATILE MEMORY

For now, as the capacity of non-volatile memory is limited, an efficient management technique is needed. When free space is necessary in non-volatile memory, a replacement policy should select data to be evicted and flush it to secondary storage. Unlike DRAM memory cases, we can adopt more complicated algorithms in non-volatile memory. That is, main memory systems cannot be aware of the exact time

of each memory access but knows only limited information (e.g., binary information of recently accessed or not), and thus replacement policies used in DRAM memory are usually simple. For example, the popular CLOCK algorithm uses one bit for each data to identify whether it is recently accessed or not, and evicts data with its access-bit 0 [19]. In contrast, as our non-volatile memory handles requests only when the data in DRAM memory is evicted, it knows full information of the requests such as requested time and frequency. Thus, we can adopt more complicated algorithms such as evicting the least recently used or the least frequently used data.

The primary issue of non-volatile memory management is to identify hot data that account for a majority of storage accesses and absorb them via non-volatile memory. During this process, identifying and discriminating cold data is also important as cold data may push out hot data under the limited non-volatile memory capacity. Specifically, 10-15% of hot data account for 80% of total storage accesses as shown in Fig. 4, which implies that 85-90% of cold data account for only 20% of total accesses. In addition, we showed that more than 50% of single-access data exist in Android swap I/Os as shown in Fig. 6. This large portion of cold data is not helpful for the performance improvement as they will not be re-used at all before evicted from the non-volatile memory. This happens as non-volatile memory is a second level memory, which receives requests only when DRAM memory evicts some data. In particular, when the working-set of a system is beyond the capacity of non-volatile memory, thrashing happens, which incurs excessively frequent replacement in non-volatile memory. Usually, we can expect the performance gain by maintaining all requested data in memory although we do not know whether the data will be subsequently requested or not. However, as mentioned in Section II, this is not the case for our non-volatile memory, which receives a large portion of cold data but the size of the non-volatile memory is limited.

Based on these observations, we propose an admission control (AC) policy that estimates data unlikely to be re-used and prohibits them from entering non-volatile memory. In particular, we do not insert data into non-volatile memory when it is firstly evicted from DRAM memory, but insert it into the non-volatile memory only after its second eviction from DRAM occurs within a certain time window. This allows the filtering of data disruptive to non-volatile memory, thereby eliminating pollution and improving performances.

To maintain the time window, we use a small size of history buffer that does not store the contents of actual data, but maintains the information that the data has been evicted from DRAM recently. The optimal size of the history buffer varies depending not only on the workload characteristics but also on the actual non-volatile memory size, and thus it can be a control parameter to be tuned. As a basic configuration, we set the number of history buffer entries to the number of actual data in non-volatile memory. This is reasonable because a bypassed data itself is not stored but its history is maintained to see whether it will be used again within a certain time window determined by the actual non-volatile memory size. Note that maintaining this size of history buffer has very low overhead because it only contains a small size of metadata (less than 20 bytes for each data) whereas an actual data consists of 4KB [20]–[23].

Now, let us return to the explanation of the admission control policy. The motivation of this policy is the existence of cold data that account for less than 20% of total storage accesses but their volume approaches 85-90%. Thus, the second eviction within a short time duration is a good indicator of whether the data will be effective or not in the near future. Therefore, bypassing non-volatile memory on the first eviction is effective in discriminating non-profitable data. The benefit of our admission control policy is that it can protect expensive non-volatile memory space from being polluted by non-profitable data when the non-volatile memory capacity is relatively smaller than the current working set size. The saved space can be utilized for maintaining hot data longer, thereby preventing from frequent replacement.

## IV. PERFORMANCE EVAUATIONS

In this section, we present the performance evaluation results to assess the effectiveness of the proposed swap-supported Android architecture by making use of non-volatile memory.

We perform our experiments with an Android reference device, ODROID-Q, which consists of 1GB DDR2-DRAM memory and 2GB swap file on 16GB eMMC. We install Google Android 6.0.1 and Linux 3.4.0, and reconfigure the Android kernel to support virtual memory swap. In our experiments, measurements were performed to see the launch time of applications, but simulations were also performed to investigate the effectiveness of algorithms such as replacement and admission control. The reason we use simulations is that simulation with actual traces can repeat the real workloads with identical conditions for each application run, providing more fair comparison than the direct execution of real workloads each time. This is because workloads cannot be executed with the same user interaction and/or system status for each workload run, making fair comparison difficult, but simulations can repeat identical conditions. In the measurement study, we repeat the executions of each scenario 10 times, and report the average launch time of each run. In the simulation study, I/O traces were extracted during the execution of each scenario, and then trace-driven simulations were performed by replaying them.

### A. SENSITIVITY ANALYSIS ON THE NON-VOLATILE MEMORY SIZE

We collect storage I/O traces of swap-supported Android and then perform trace-driven simulations. Storage I/O traces used in our simulations were extracted by the ftrace utility in Android kernel [11]. In our experiments, the size of a block is set to 4KB, which is common to most operating systems [9]. The traces used in our experiments were collected while executing 20 Android applications. There are 4 scenarios and each scenario consists of 5 applications as shown in Table 1.

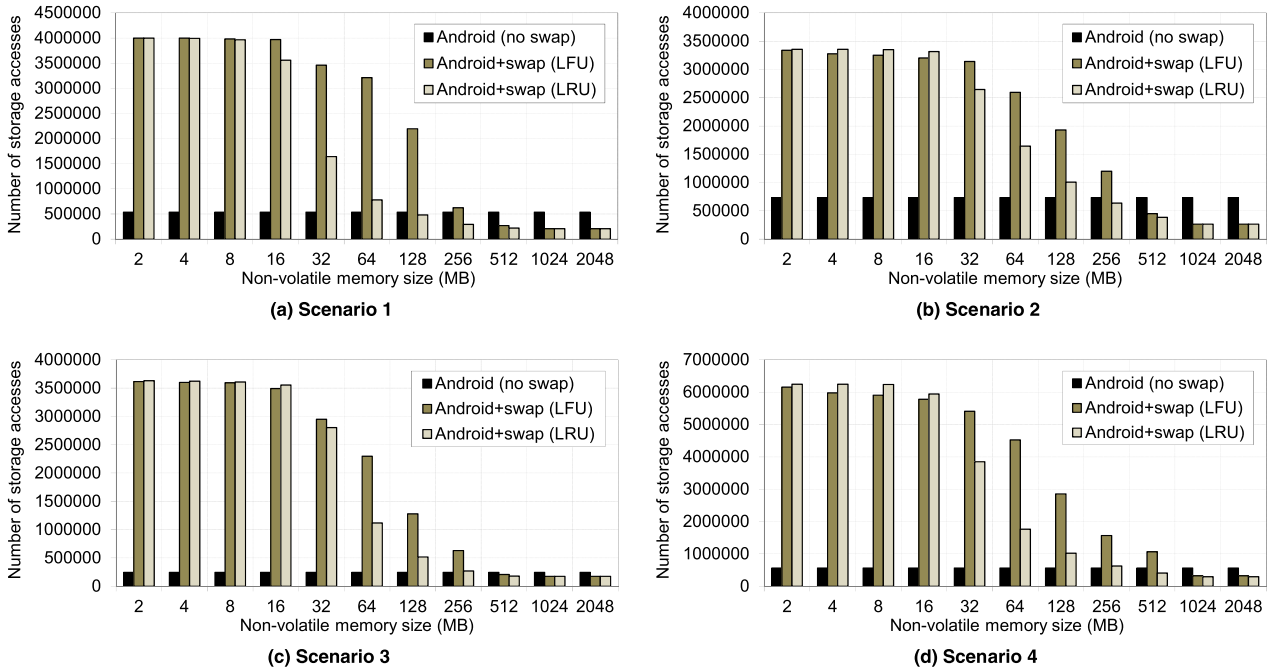


**FIGURE 8.** Number of storage accesses for the proposed swap-supported Android in comparison with original Android.

We execute the applications in each scenario sequentially and then repeat them to see the effect of swap.

Fig. 8 shows the simulation results for the proposed architecture as the size of the non-volatile memory is varied from 2MB to 2048MB. When we need to insert data into non-volatile memory but there is no available space, we evict the least recently used (LRU) data from non-volatile memory, which is the most commonly used policy in cache systems [9]. For comparison purpose, we simulate another policy that evicts the least frequently used (LFU) data when there is no non-volatile memory space. The result of original Android is also evaluated and compared with these two policies. As shown in the figure, the number of storage accesses in swap-supported Android is excessively large when the size of non-volatile memory is less than 32MB regardless of eviction policies for all scenarios. However, as the non-volatile memory size becomes larger than 32MB, the performance gap between the two policies clearly appears. Specifically, LRU outperforms LFU when the non-volatile memory size is in the range of 32MB to 256MB irrespective of the scenarios. Finally, when the non-volatile memory size becomes 1024MB, the two policies merge and perform even better than the original Android. This is because the capacity of non-volatile memory is large enough to maintain all requested data simultaneously. Since our goal is to use the size of non-volatile memory as small as possible, we can use 128-256MB non-volatile memory by adopting the LRU policy. Then, the expected number of storage accesses will be similar to that of the original Android.

### B. EFFECTIVENESS OF ADMISSION CONTROL

Now, let us examine the effectiveness of our admission control policy described in Section III. Fig. 9 shows the number of storage accesses for the admission control policy in comparison with the conventional policy that does not use it. In this experiment, we commonly adopt the LRU eviction policy as it performs better than LFU. The size of the non-volatile memory is varied from 16MB to 2048MB. As shown in the figure, the number of storage accesses is reduced significantly as our admission control policy is adopted. Specifically, when the size of non-volatile memory is in the range of 128MB to 256MB, the effectiveness of the admission control policy clearly appears.

When the size of non-volatile memory is large enough, the effectiveness of the admission control policy reduces significantly. This is because the size of non-volatile memory is large enough to accommodate cold data as well as hot data in such cases. Performance gaps are also narrow when the size of non-volatile memory becomes extremely small because we can make data in non-volatile memory useful only for small inter-access time in this case. That is, as the time duration of residing in non-volatile memory is short, data are more likely to be evicted from the non-volatile memory before re-used.

Except for these two extreme cases, our admission control policy performs well by filtering the large portion of cold data. In addition, the admission control policy has an effect of increasing the effective non-volatile memory space to consistently keep hot data. As the proposed admission control policy does not insert data upon the first eviction from DRAM, it incurs an additional storage access when the data is

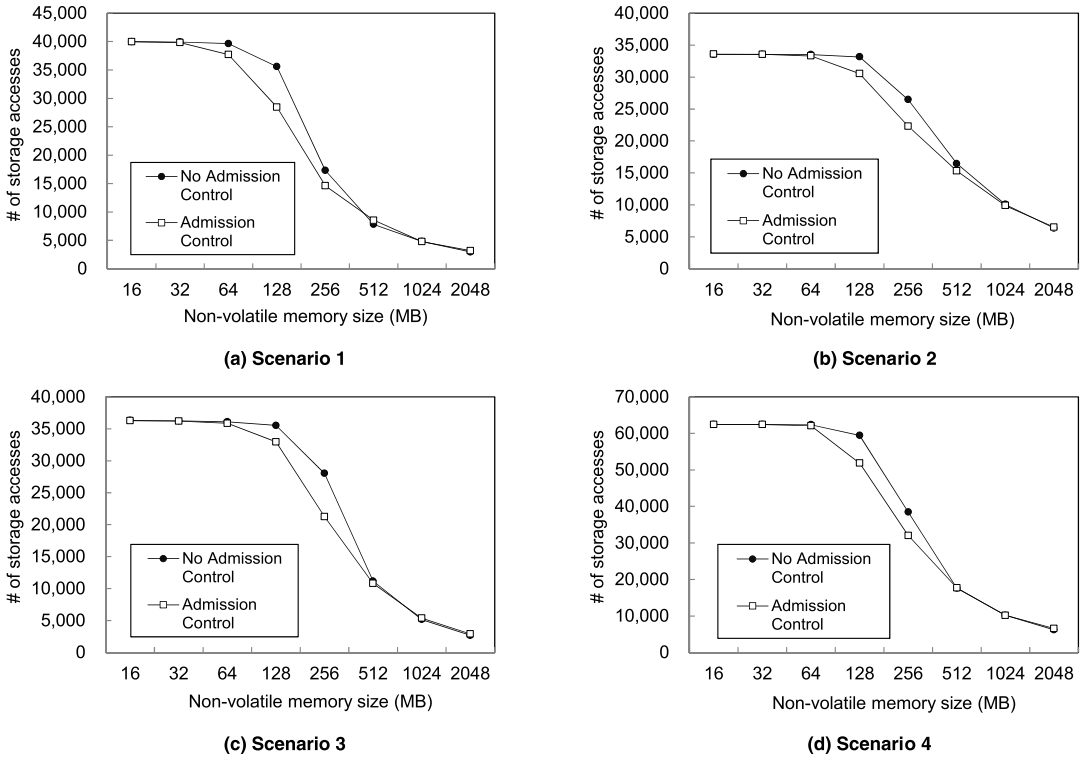


**FIGURE 9.** Number of storage accesses for the proposed swap-supported Android with/without admission control.

accessed again. However, our results showed that admission control performs well when the size of non-volatile memory is not large enough, which implies that filtering out cold data is effective in non-volatile memory management.

### C. COMPARISON OF THE LAUNCH TIME

In order to validate the proposed architecture further, we run a series of applications introduced in Section I, i.e., Angrybird, BBC, Chrome, Instagram, Farmstory, and Cut the Rope, and compare the launch time of original Android, swap-supported Android, and our architecture. Note that our architecture adopts the non-volatile memory size of 256MB with admission control. For each experiment, the application is first launched, and then a sufficient number of other applications are executed to fully fill DRAM memory; then, the target application is executed again.

Fig. 10 shows the application launch time of the proposed architecture in comparison with the original Android and the swap-supported Android without our architecture. As shown in the figure, swap-supported Android significantly degrades the application's launch time, but our architecture exhibits similar results with the original Android although it supports swap. In some cases, our architecture even performs better than original Android. Specifically, the performance improvement against original Android is in the range of 17-40% in case of Chrome, Instagram, Farmstory, and Cut the Rope. In comparison with the swap-supported Android, our architecture reduces the application launch time by 77% on average.

### D. DISCUSSIONS

In this section, we summarize our observations and briefly discuss the evaluation results. We observed a serious thrashing phenomenon (i.e., performance degradation of 2x to 5x) in swap-supported Android as the number of applications in execution increases. To quantify this, we analyzed Android swap I/O traces and made two prominent observations that can be exploited in supporting Android swap efficiently. The first is the existence of hot 15% data, which account for 80% of total swap I/O, and the second is the existence of cold 50% data that are never used again after entering the swap area. Based on these two observations, we analyzed how to efficiently manage NVM-swap.

As Android swap has bimodal reference characteristics, we need to identify and manage hot and cold data efficiently. Instead of managing them at the main memory layer, we adopt NVM at the swap I/O layer and use more precise algorithms consisting of admission and replacement. By so doing, we showed that Android swap can be efficiently supported without performance degradations.

Note that the overhead of fully associative data placement in NVM is not heavy in our case as replacement policies in the swap I/O layer can be implemented by software, which is different from the hardware implementation of replacement policies in on-chip caches.

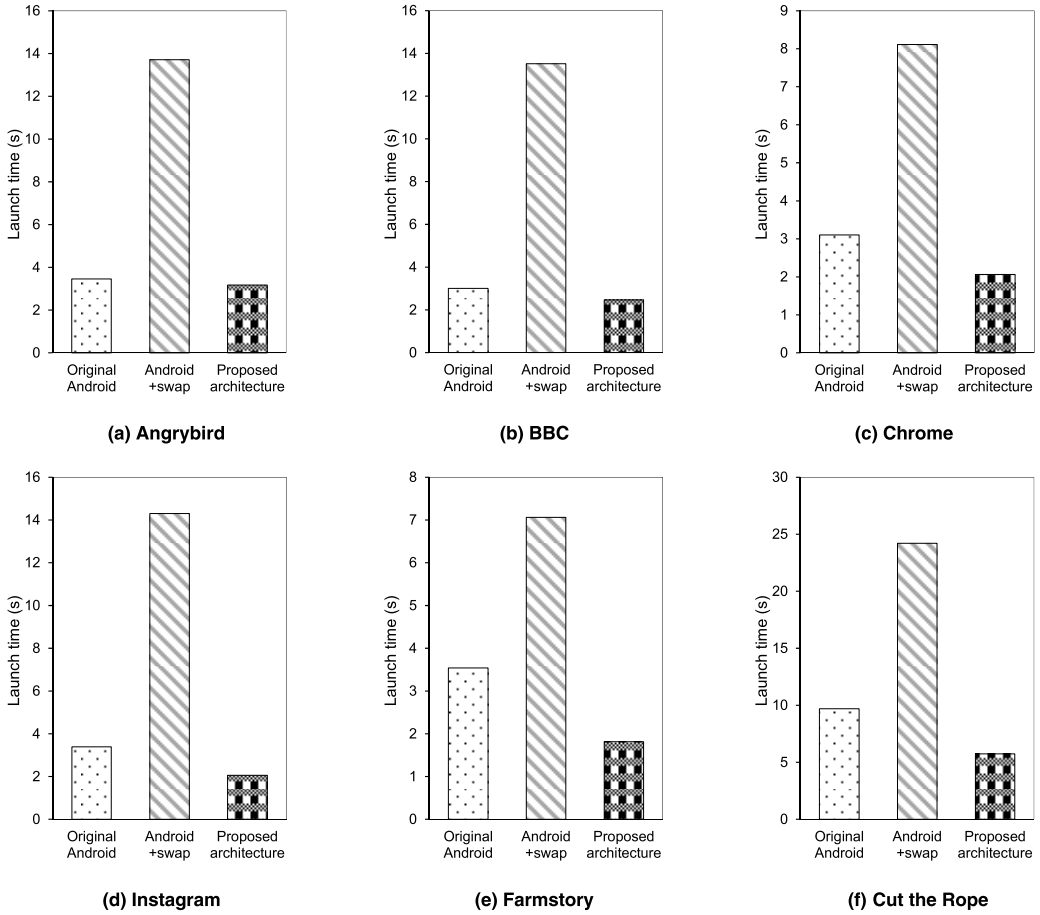


**FIGURE 10.** Application's launch time for original Android, swap-supported Android, and the proposed architecture.

This article did not consider the endurance problem of NVM as swap accesses are not so frequent compared to main memory or on-chip cache accesses. Note that the endurance cycles of PCM and STT-MRAM, respectively, are about 3-4 and 7-8 orders of magnitude more than that of flash memory. Endurance problems of NVM can be further referenced in other studies [27], [29], [30].

## V. CONCLUSION

In this article, we discussed issues for smartphone to be a general purpose computing device like desktop PCs. We argued that one of the significant problems in current smartphone systems is the termination of applications without user's approval, thereby losing the context of the processes. To resolve this issue, we presented a virtual memory swap architecture that can be adopted efficiently in current smartphone systems. We analyzed the characteristics of storage accesses in original Android and swap-supported Android, and found out two important phenomena. First, swap-supported Android incurs 4-15 times more storage accesses than original Android. Second, storage accesses in swap-supported Android are extremely skewed to 10-15% of hot data.

Based on these observations, we presented a new architecture to eliminate a bunch of storage accesses in swap-supported Android. In particular, we adopt a small size of non-volatile memory to absorb hot data that appear in swap-supported Android by making use of efficient management policies. We showed that the required non-volatile memory size is only 256MB but it reduces the number of storage accesses and the application launch time by up to 93% and 82%, respectively.

## REFERENCES

[1] S. Bae, H. Song, C. Min, J. Kim, and Y. I. Eom, "EIMOS: Enhancing interactivity in mobile operating systems," in *Computational Science and Its Applications—ICCSA 2012* (Lecture Notes in Computer Science), vol. 7335. Cham, Switzerland: Springer, 2012, pp. 238–247.

[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] K.-T. Chen, Y.-C. Chang, H.-J. Hsu, D.-Y. Chen, C.-Y. Huang, and C.-H. Hsu, "On the quality of service of cloud gaming systems," *IEEE Trans. Multimedia*, vol. 16, no. 2, pp. 480–495, Feb. 2014.

[5] N. Islam and R. Want, "Smartphones: Past, present, and future," *IEEE Pervas. Comput.*, vol. 13, no. 4, pp. 89–92, Oct. 2014.

[6] *Google Nexus 6P Product*. Accessed: Apr. 1, 2019. [Online]. Available: https://www.google.com/intl/en_us/nexus/6p/
[7] S. 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, p. 47, 2016, Art. no. 25.
[8] R. Prodduturi, "Effective handling of low memory scenarios in Android using logs," Ph.D. dissertation, Dept. Comput. Sci. Eng., Indian Inst. Technol., Mumbai, India, 2013.
[9] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. New York, NY, USA: Wiley, 2013.
[10] *Odroid-Q*. Accessed: Apr. 1, 2019. [Online]. Available: http://www.hardkernel.com/
[11] S. Rostedt, "Ftrace Linux kernel tracing," in *Proc. Linux Conf. Jpn.*, 2010, pp. 1–50.
[12] H.-S. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proc. IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010.
[13] 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, 2013, Art. no. 074001.
[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.*, 2011, pp. 1–5.
[15] C. D. Wright, M. M. Aziz, M. Armand, S. Senkader, and W. Yu, "Can we reach Tbit/sq.in. storage densities with phase-change media?" in *Proc. Eur. Phase Change Ovonics Symp.*, 2004, pp. 1–8.
[16] E. Lee and H. Bahn, "Caching strategies for high-performance storage media," *ACM Trans. Storage*, vol. 10, no. 3, p. 11, 2014.
[17] E. Lee, H. Kang, H. Bahn, and K. G. Shin, "Eliminating periodic flush overhead of file I/O with non-volatile buffer cache," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1145–1157, Apr. 2016.
[18] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 73–80.
[19] 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.
[20] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. 20th Int. Conf. Very Large Data Bases*, 1994, pp. 439–450.
[21] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 187–200.
[22] Y. Zhou, J. F. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proc. USENIX Annu. Tech. Conf.*, 2001, pp. 91–104.
[23] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conf. File Storage Technol.*, 2003, pp. 1–17.
[24] B. Nale, R. K. Ramanujan, M. P. Swaminathan, and T. Thomas, "Memory channel that supports near memory and far memory access," U.S. Patent 2011 054 421, Sep. 30, 2013.
[25] R. K. Ramanujan, R. Agarwal, and G. J. Hinton, "Apparatus and method for implementing a multi-level memory hierarchy having different operating modes," U.S. Patent 20 130 268 728 A1, Oct. 10, 2013.
[26] 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. ACM EMSOFT*, 2014, p. 30.
[27] 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.
[28] E. Cheshmikhani, H. Farbeh, S. Miremadi, and H. Asadi, "TA-LRW: A replacement policy for error rate reduction in STT-MRAM caches," *IEEE Trans. Comput.*, vol. 68, no. 3, pp. 455–470, Mar. 2019.
[29] H. Farbeh, A. M. H. Monazzah, E. Aliagha, and E. Cheshmikhani, "A-CACHE: Alternating cache allocation to conduct higher endurance in NVM-based caches," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 66, no. 7, pp. 1237–1241, Jul. 2019.
[30] E. Lee, S. H. Yoo, and H. Bahn, "Design and implementation of a journaling file system for phase-change memory," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1349–1360, May 2015.

**JISUN KIM** received the B.S. degree in computer science and engineering from Hanshin University, in 2011. She is currently pursuing the Ph.D. degree in computer science and engineering with Ewha 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** (M'02) 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 University, Seoul, South Korea. He has published over 70 papers in leading conferences and journals in his research 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.

Prof. Bahn received the Best Paper Award from the USENIX Conference on file and storage technologies, in 2013.