*electronics*

MDPI

*Article*

# Storage Type and Hot Partition Aware Page Reclamation for NVM Swap in Smartphones

**Hyejung Yoon [1], Kyungwoon Cho [2] and Hyokyung Bahn [3,*]**

[1] Department of Artificial Intelligence and Software, Ewha University, Seoul 03760, Korea; dalliy0628@ewhain.net
[2] Embedded Software Research Center, Ewha University, Seoul 03760, Korea; cezanne@oslab.ewha.ac.kr
[3] Department of Computer Engineering, Ewha University, Seoul 03760, Korea
* Correspondence: bahn@ewha.ac.kr; Tel.: +82-2-3277-2368

**Abstract:** With the rapid advances in mobile app technologies, new activities using smartphones emerge every day including social network and location-based services. However, smartphones experience problems in handling high priority tasks, and often close apps without the user's agreement when there is no available memory space. To cope with this situation, supporting swap with fast NVM storage has been suggested. Although swap in smartphones incurs serious slowing-down problems in I/O operations during saving and restoring the context of apps, NVM has been shown to resolve this problem due to its fast I/O features. Unlike previous studies that only focused on the management of NVM swap itself, this article discusses how the memory management system of smartphones can be further improved with NVM swap. Specifically, we design a new page reclamation algorithm for smartphone memory systems, which considers the following: (1) storage types of each partition (i.e., file system for flash storage and swap for NVM), and (2) access hotness of each partition including operation types and workload characteristics. By considering asymmetric I/O cost and access density for each partition, our algorithm improves the I/O performance of smartphones significantly. Specifically, it improves the I/O time by 15.0% on average and by up to 35.1% compared to the well-known CLOCK algorithm.

**Keywords:** smartphone; page reclamation; memory swap; storage; NVM



**Citation:** Yoon, H.; Cho, K.; Bahn, H. Storage Type and Hot Partition Aware Page Reclamation for NVM Swap in Smartphones. *Electronics* **2022**, *11*, 386. https://doi.org/10.3390/electronics11030386

Academic Editor: Jordi Guitart

Received: 31 December 2021
Accepted: 24 January 2022
Published: 27 January 2022



## 1. Introduction

Due to the recent advances in mobile platform and application technologies, smartphones have become one of the essential consumer devices in our daily life [1]. Activities with smartphones are performed every day and people are increasingly connected to various social media and location-based services through their smartphones [2–4]. In fact, the hardware resources of contemporary smartphones are sufficient to support the concurrent executions of various apps, which was not possible in feature phones or early models of smartphones [1]. Specifically, the most recent Android reference phone, Google Pixel 6 Pro, consists of 1.8 to 2.8 GHz 8-core CPU, 20-core Mali-G78 MP20 GPU, 12 GB DRAM, and 512 GB UFS 3.1 storage, which is sufficient to perform multitasking [5].

A smartphone is not just a personal entertainment device, but facilitates some professional work such as video editing, personal broadcasting, and software development. Additionally, office resources such as word processing and spreadsheet, which have been main tasks performed using desktops, are increasingly compatible with smartphones by connecting external screen and keyboard. Nevertheless, smartphones still experience problems when handling high priority work, as it does not support the memory swap function. Specifically, the current smartphone platforms such as Android close apps without the user's agreement when there is no available memory space [6,7]. This was not a significant matter when smartphones were primarily an entertainment consumer device, but now it

is critical as smartphones are needed for official work. For example, terminating a music player is not a significant issue but killing a stock-trading app while ordering a stock selling may cause serious problems.

To handle this situation, smartphones should maintain the context of an app unless users explicitly terminate the app. This can be realized by memory swap, which makes use of secondary storage as an extension of the main memory to save the context of an app when free memory space is not sufficient [8]. Even if swap has been widely adopted in existing computer systems, it is not easy to support swap in smartphones [7,9]. In particular, a serious slowing-down problem during launching an app is observable with a smartphone that activates the swap function [7]. To cope with this situation, studies on smartphone swap suggested NVM (Non-Volatile Memory) as a fast swap device to accelerate the I/O performance of apps [7,8]. However, previous studies focused only on the swap architecture and the management of NVM itself rather than considering memory management issues with NVM swap.

In this article, we discuss how the memory management system of smartphones can be further improved when NVM is used as a swap device. Specifically, we design a new page-reclamation algorithm for smartphone memory systems, which considers heterogeneous storage media and data access characteristics in each partition. Page reclamation algorithms decide pages to be evicted from memory when there is no free memory space to accommodate new page requests. Unlike existing page reclamation algorithms, the proposed approach considers the following distinct characteristics of NVM swap.

First, by adopting NVM swap, the two storage partitions, i.e., file system and swap partitions, are located in different storage media, i.e., flash storage and NVM, respectively. Thus, the cost of accessing a page is not uniform depending on its storage location and the operation type for the page. When a page that resides in the file system partition is requested, data should be retrieved from flash storage. We call these type of pages file-mapped pages. In contrast, a page fetched from the NVM swap partition is called an anonymous page. As we use different storage media, retrieving a file-mapped page from the file system partition takes more time than retrieving an anonymous page from the swap partition. Thus, it is reasonable for page reclamation algorithms to give higher priorities to pages from the file system partition that is located at slow flash storage. Additionally, the I/O cost of read and write operations is asymmetric in these storage media. When a page that has been modified while resident in memory is selected as a reclamation target, it should be swapped out to NVM or flushed to flash storage before removal, which results in write I/O 2–8 times slower than read I/O [10,11]. Hence, an efficient reclamation algorithm should take into account these asymmetric I/O costs.

Another important issue to consider in the design of a page-reclamation algorithm is that access patterns for I/O partitions vary depending on workload characteristics. Specifically, our previous analysis showed that a storage partition that incurs the heaviest I/O traffic is not the same for all apps, but it is varied significantly for the categories of apps [12]. For example, swap I/O accounts for a large portion of I/O in memory-intensive workloads such as graph visualization as such apps have a large footprint for computation and visualization. As the memory capacity is not sufficient to accommodate the working-set of these apps, swap frequently occurs. In contrast, in case of file-intensive workloads such as a web browser and multimedia player, file system partitions account for a large portion of I/O. This is because such types of apps consistently access data files from the file system partition, and thus making a dominant portion of I/Os from this partition. Web browsers usually read data from web sites and stores them on the local file system, which incurs file I/O [13]. Multimedia player apps read video frames from the file system, decode them, and display on the screen, which also require consistent access to the file system partition [14].

Based on these observations, we propose a new page-reclamation algorithm for swap-supported smartphones that consider different operation cost of each page and access density for each partition. Our algorithm takes into account the cost of a page retrieved from the file system partition and the swap partition as well as nonuniform cost of read/write

operations. Specifically, the algorithm classifies page access into file-mapped/read, file-mapped/write, anonymous/read, and anonymous/write, relying on their operation costs. Then, the algorithm divides the memory space into four regions, namely read/file region, write/file region, read/anon region, and write/anon region. Each region is then resized according to the access densities and contribution of the region to performance improvement. In this process, shadow regions are added for each region to monitor the workload characteristics. For each region, the recency of page access is separately monitored by making use of a circular list widely used in the CLOCK algorithm. Simulations performed by replaying real memory access traces show that the proposed algorithm, namely SPO-CLOCK (Storage type, Partition hotness, and Operation cost aware CLOCK), improves the I/O performance of smartphones significantly. Specifically, it improves the I/O time by 15.0% on average and up to 35.1% compared to the widely acknowledged CLOCK algorithm.

The remainder of this article is organized as follows. Section 2 describes the workings of the proposed page reclamation algorithm called SPO-CLOCK in detail. In Section 3, we discuss performance evaluation results obtained through simulations. Section 4 briefly summarizes related works. Finally, we conclude this article in Section 5.

## 2. Storage Type and Hot Partition Aware Page Reclamation

This section describes an efficient page reclamation algorithm for smartphone's swap with NVM, which is called SPO-CLOCK. We first explain system architecture by adopting NVM as swap storage and discuss how such architecture can be managed efficiently in page reclamation.

### *2.1. System Architecture*

Figure 1 shows the system architecture for the proposed page reclamation algorithm in smartphone memory. In our architecture, NVM is used as storage in addition to flash storage to support virtual memory swap. Note that NVM here means the 3rd generation byte-addressable NVM such as PCM (Phase-Change Memory) or STT-MRAM (Spin-Transfer Torque Magnetic RAM) differentiated from flash storage. As NVM will not replace traditional storage such as HDD or flash due to cost per capacity, it is considered only as an add-on component to enhance performances such as our swap in smartphones [15,16]. In this architecture, the file system partition is maintained in NAND flash, whereas the swap partition is created on NVM. Due to the performance gap between NVM and flash storage, fetching a page from the file system partition takes more time than retrieving a page from the swap partition. Therefore, removing a page fetched from file system and re-fetching it is not a good choice. For this reason, an efficient reclamation algorithm should grant higher priorities to pages from the file system partition if all other situations are the same. In addition, pages that incur storage reading and writing have different I/O costs, which should also be considered in the design of a reclamation algorithm.

### *2.2. Allocation and Adjustment of Each Region*

Our algorithm manages memory spaces by logically separating them into four regions, namely, the read/file region, write/file region, read/anon region, and write/anon region. Each region maintains pages with the attribute of file-mapped/read, file-mapped/write, anonymous/read, and anonymous/write, respectively. All regions initially have the same size. Then, the size of each region is adjusted according to the change of access densities and the contribution of the region to memory performance. In this process, a small size of history buffer, which is called the shadow region, is adopted for each region to monitor the workload characteristics. A shadow region maintains the history of a page recently evicted from the real memory region without storing the contents of a page, and thus it is known to be lightweight [17,18].

In our algorithm, read/file and write/file regions manage recently read and written pages, respectively, retrieved from the file system partition. Similarly, read/anon and write/anon regions manage recently read and written pages, respectively, from the swap

partition. All the pages in write/file and write/anon regions are modified after entering memory, which need to be flushed to flash storage and swapped out to NVM, respectively, before their eviction from memory. Note that writing is 2–8 times slower than reading in flash and NVM storage, implying that discarding a page from write/anon (or write/file) region incurs more I/O cost than evicting a page from read/anon (or read/file) region. Additionally, retrieving a page from flash storage takes more time than retrieving a page from NVM. Thus, SPO-CLOCK gives higher priorities to write-accessed pages and file-mapped pages in proportion to their costs; but it also maintains read-accessed pages and anonymous pages if they are frequently accessed and hence their contribution to improving the memory performance is considerable.

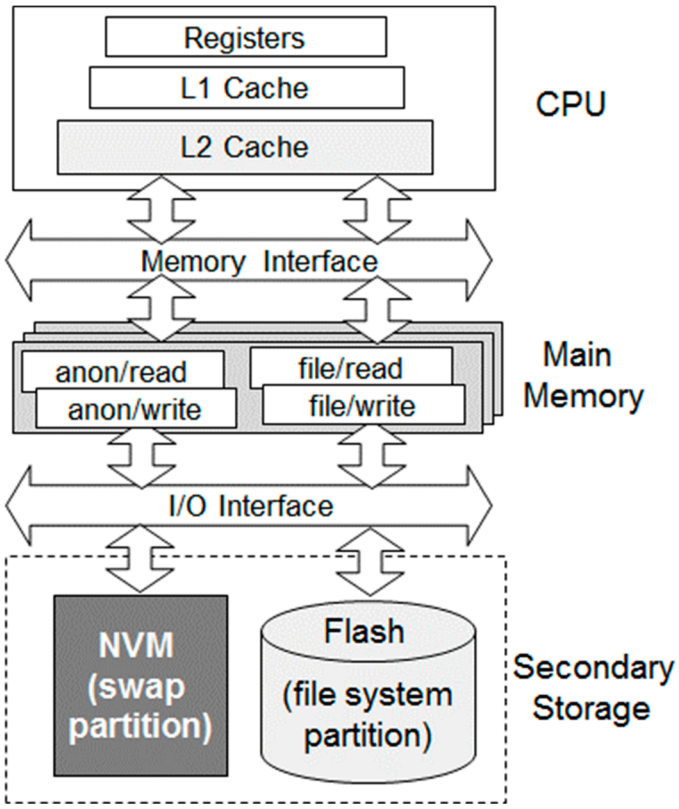


**Figure 1.** The system architecture that supports NVM-swap in Android.

As previously mentioned, SPO-CLOCK adopts shadow regions to monitor and adjust the size of the four memory regions as shown in Figure 2. Shadow regions maintain the descriptor of recently discarded pages without their real contents. By tracking accessed page in the shadow regions, SPO-CLOCK estimates the effect that extending each region would have on performance. If there are frequent accesses to pages in the read/anon shadow region, SPO-CLOCK extends the read/anon region to reduce the number of I/Os incurred by the pages in this region. Similarly, the size of other regions is also enlarged if there are frequent page accesses in the corresponding shadow regions. In addition to page accesses in shadow regions, different operation costs are considered in adjusting the size of each region. That is, SPO-CLOCK extends write regions more aggressively than read regions if all the other conditions are the same by considering asymmetric I/O cost in flash storage and NVM. Suppose that the I/O cost of a page in a write region is twice that of a read region. Then, whenever the number of hits in a shadow region reaches a certain threshold, SPO-CLOCK increases the size of the corresponding real memory region by 1. In this process, we set the threshold for a read region to twice that for a write region to account for cost differences.

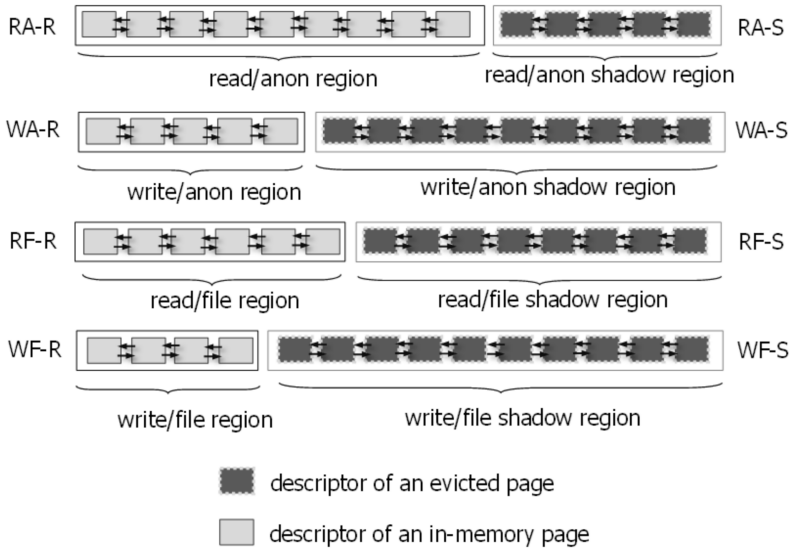


**Figure 2.** Memory regions and their shadow regions in SPO-CLOCK.

The optimal size of a shadow region varies depending on the system and workload characteristics, and thus it can be seen as a control parameter to be tuned. In this article, we set the number of page entries in a shadow region such that the total number of pages in a real memory region and its shadow region is equal to the memory capacity of the system. This is reasonable because an evicted page itself is not stored but its history is maintained to see whether it will be used again in case the size of a region is as large as the full memory capacity of the system. For example, if the size of the read/anon region is increased by 1, SPO-CLOCK decreases the size of the read/anon shadow region by 1. By doing so, the total number of hits with the entire memory capacity can be estimated as if shadow regions also maintain actual pages.

Figure 2 shows the size of regions and their corresponding shadow regions. Specifically, there are four regions, the read/anon region RA-R, the write/anon region WA-R, the read/file region RF-R, the write/file region WF-R, and their corresponding shadow regions, the read/anon shadow region RA-S, the write/anon shadow region WA-S, the read/file shadow region RF-S, and the write/file shadow region WF-S, respectively. In reality, it is possible that a page is read and then written, so a page descriptor can be linked to both read and write regions simultaneously. This allows for the accurate estimation of access characteristics in each region. In contrast, it is not possible that a page descriptor is linked to both file and anon regions as a page can be one of an anonymous page or a file-mapped page. Additionally, a real memory region and its shadow region do not overlap as real regions manage pages resident in the memory, whereas shadow regions maintain descriptors of pages evicted from memory. Since SPO-CLOCK allows pages to be linked to multiple regions at the same time, evicting the contents of a page from physical memory is performed when the page descriptor is not linked to any of real regions, RA-R, WA-R, RF-R, or WF-R.

### *2.3. Page Reclamation*

When there is not enough free memory in the system, the reclamation algorithm selects a certain number of pages, and evicts them from memory. Of all pages in memory, the reclamation algorithm usually selects pages not accessed recently as the target of eviction. Note that memory management systems cannot be aware of the exact time of every page access but recognize only the binary information of whether they have been recently accessed or not by making use of an access bit. Thus, page reclamation algorithms in memory systems typically adopt a circular list with a clock hand pointer for selecting

a victim by checking the access bit of pages to see whether a page is recently accessed or not [19].

Figure 3 depicts the circular list structure of SPO-CLOCK. Pages in main memory are managed by four memory regions, RA-R, WA-R, RF-R, and WF-R. For each region, the recency of page accesses is monitored by making use of separate circular lists. When a page is discarded from a real memory region, its descriptor is inserted to the corresponding shadow region, RA-S, WA-S, RF-S, or WF-S. Page reclamation for each region is also performed independently by using a circular list of the region. When SPO-CLOCK needs to select a page for eviction from read regions RA-R or RF-R, it traverses the circular list of that region by checking the read access bit of the page the clock hand pointer currently points to. If the bit is 1, it is cleared; otherwise, the page is discarded from that region. If SPO-CLOCK fails to find the page for eviction, the clock hand pointer moves to the next page of the circular list until identifying a page with its read access bit of 0. When SPO-CLOCK needs to evict a page from write regions, WA-R or WF-R, the write access bit is investigated instead of the read access bit.

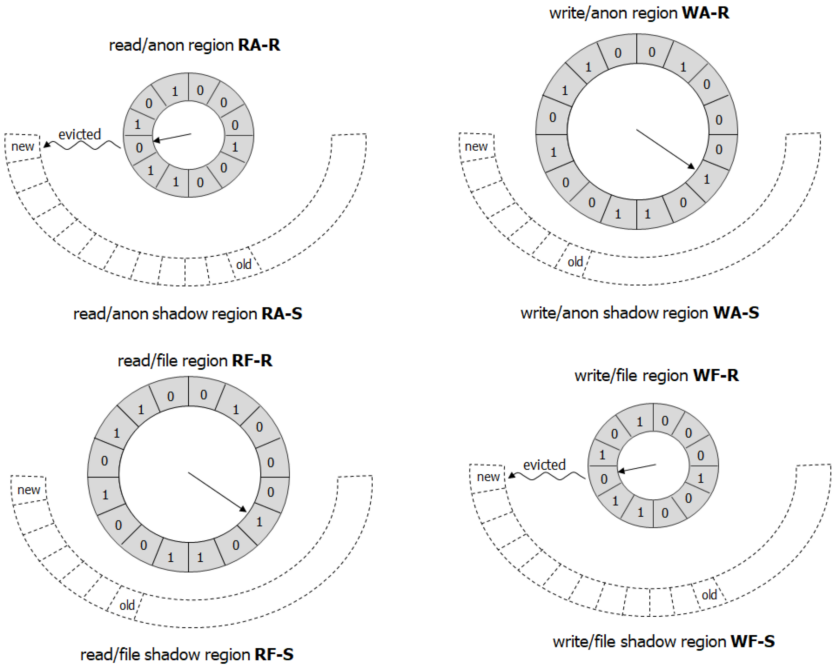


**Figure 3.** CLOCK lists used in SPO-CLOCK.

During the scanning of the circular list to find a victim page from a read region, if a page not in the write region is found with its write access bit of 1, SPO-CLOCK clears the write access bit and adds that page to the write region. Similarly, during the scanning of a write region, if a page not in the read region is found with its read access bit of 1, SPO-CLOCK clears the read access bit and adds the page to the read region. This process is necessary as list insertion is possible only when a page is being retrieved from storage. That is, if a read operation is performed on a page, which has been already retrieved from storage due to a write operation, it is not in the read region, but the read access bit has been set to 1 by the paging system hardware. The same may occur for a write operation.

When a reclamation is necessary in a shadow region, the oldest page is discarded first. As depicted in Figure 3, a newly added page is linked to the newest location in the shadow region, and the page in the oldest location is discarded.

Now, we will describe further details of our reclamation algorithm for different memory access scenarios. If a page already stored in the memory is accessed, the paging system hardware sets the access bit of the page to 1. The read access bit of a page is set upon a read operation while the write access bit is set upon a write. When a page not in the memory

is requested, SPO-CLOCK retrieves that page from the source partition it resides in (i.e., file system partition for a file-mapped page and swap partition for an anonymous page), and stores it in memory. If there is no free memory space to store the currently retrieved page, SPO-CLOCK evicts a page from the region corresponding to its source partition and the operation type to make free space. Then, SPO-CLOCK adds the page descriptor to that region. For example, if a page is retrieved from the file system partition by a read system call, it is inserted to the read/file region RF-R, and thus the victim page is also selected from RF-R.

If the history of the page exists in the corresponding shadow region, it is deleted from the shadow region and the hit count for this region increases. Note that each region has its corresponding hit count to monitor whether the size of that region needs to be increased. If the hit count reaches the threshold for the region, the size of the real memory region is increased. To increase the size of a region, however, the size of other regions should be decreased as the total memory capacity is fixed. In our algorithm, the victim region is selected by monitoring the appropriate size of each region evaluated through hits from shadow regions and the I/O cost involved in each region. Once the sizes of real memory regions are adjusted, the sizes of the shadow regions are then adjusted accordingly to preserve the balance between real and shadow regions. Algorithm 1 lists the pseudo-code upon a page fault of a page in SPO-CLOCK.

**Algorithm 1.** Workings of SPO-CLOCK.

```
procedure PAGE-FAULT (page Pg, operation Op, partition Pt)
    Reg ← target region of Pg based on Op and Pt;
    Reg′ ← shadow region of Reg;
    if no free space in Reg then
        RECLAIM (Reg);
    end if
    if no free space in Reg′ then
        remove the oldest page from Reg′;
    end if
    if Pg∈Reg′ then
        Remove Pg from Reg′;
        Reg.hit++;
        if Reg.hit = Reg.threshold then
            increase the size of Reg by 1;
            decrease the size of Reg′ by 1;
            adjust the size of other regions;
            Reg.hit ← 0;
        end if
    end if
    add Pg to Reg;
end procedure
procedure RECLAIM (region Reg)
    Pg ← page pointed by clock-hand of Reg;
    Reg_op ← region with the opposite operation of Reg;
    while access-bit (Pg, Reg) = 1 do
        access-bit (Pg, Reg) ← 0;
        if access-bit (Pg, Reg_op) = 1 & Pg ∉ Reg_op then
            insert Pg to Reg_op;
        end if
        advance clock-hand of Reg;
        Pg ← page pointed by clock-hand of Reg;
    end while
    delete Pg from Reg and add Pg to Reg′;
end procedure
```

### *2.4. Overhead of SPO-CLOCK*

Although monitoring and maintaining history information in the proposed algorithm seems to incur a large overhead, that is not the case in reality when compared to traditional memory management or caching algorithms. Specifically, the space overhead of maintaining our shadow regions is only 16 bytes per page as the page descriptor with a pointer link per page is needed. Thus, the total space required to manage 1 GB memory capacity, which consists of 262,144 4 KB pages, is only 1 page.

Additionally, the time overhead of SPO-CLOCK is relatively short compared to traditional caching algorithms such as LRU (Least Recently Used). In particular, upon every memory access, LRU requires the list manipulation of moving the currently accessed page to the end of the list to maintain all pages in the access time order. On the contrary, SPO-CLOCK is activated only when the requested page does not reside in memory. That is, SPO-CLOCK does not perform anything when a page already in the memory is accessed, but only the access bit is set by the paging system hardware. SPO-CLOCK performs list manipulations only when the page fault handler is invoked to access storage. As storage access is very slow, the software overhead of SPO-CLOCK while page fault handling is not large. Resizing of areas is also considered only when a page fault occurs, and it does not happen frequently as shown in Algorithm 1.

The time complexity of SPO-CLOCK is identical to that of the original CLOCK algorithm, in which the only non-constant part is involved in the clock-hand scanning process to find a victim page. Though the worst case time complexity may be $O(n)$, where $n$ is the number of pages, the scanning requires only a few movements of the clock-hand, implying that it has the constant time complexity in practical terms. In fact, a worst case analysis such as a time complexity analysis does not consider the practical situations of real system environments but just uses unrealistic conditions for worst cases to the algorithm. For example, LRU has the time complexity of $O(1)$ even though its overhead is much larger than that of CLOCK in real systems.

## 3. Experimental Results

In this section, we discuss the performance evaluation results of the SPO-CLOCK algorithm based on simulation experiments. For our simulation, memory access traces were collected during the execution of apps, and then trace-driven simulations were performed by replaying them. For collecting traces, we made use of the Cachegrind utility in the Valgrind toolset [20]. We collected memory access traces from five Android apps, namely the Angrybirds game, Youtube, which is a video streaming service, Chrome, which is a web browser, Facebook, which is a social network service, and Farmstory, which is a video game. The characteristics of memory access traces collected from these apps are listed in Table 1. Note that we opened our memory access traces on our GitHub page for other researchers to reproduce the results (https://github.com/oslab-ewha/mem-trace; accessed on 30 December 2021).

**Table 1.** Brief characteristics of memory access traces collected in Android apps.

| App | Workload Footprint (KB) | Ratio of Operations (Read: Write) | Memory Access Counts | | | |
|---|---|---|---|---|---|---|
| | | | Data Read | Data Write | Instruction Read | Total |
| Angrybirds | 78,782 | 3.50:1 | 13,387,756 | 3,822,479 | 980,312 | 18,201,717 |
| Youtube | 70,287 | 4.44:1 | 14,040,959 | 3,162,229 | 993,316 | 18,196,504 |
| Chrome | 266,092 | 4.11:1 | 15,272,935 | 4,104,436 | 1,622,628 | 20,999,999 |
| Facebook | 203,431 | 5.67:1 | 11,121,174 | 2,045,716 | 486,165 | 13,653,055 |
| Farmstory | 55,030 | 6.24:1 | 12,675,555 | 2,101,818 | 447,297 | 15,224,670 |

We developed a functional simulator that has the ability to evaluate the effectiveness of page reclamation algorithms with the exact I/O parameters of modern NVM and flash

storage. Specifically, the simulator replays memory access traces consisting of a series of logical page numbers and the access types. The access type is one of "instruction read," "data read," and "data write," which is the memory reference operation performed by CPU. While simulating the memory system with a given algorithm and the memory capacity, if the requested page is not in the memory, we simulated storage I/O activities based on the performance characteristics of each storage type. For NAND flash, we used the parameters of Samsung UFS 3.1, of which the read and write performances are 100,000 IOPS and 70,000 IOPS for 4 KB, respectively. For NVM, we used the parameters of Intel Optane M10, of which the read and write performances are 190,000 IOPS and 35,000 IOPS for 4 KB, respectively. The size of a page is set to 4 KB as is typically used in Android and Linux.

As we are interested in the memory management issue of smartphones, we did not simulate the detailed activities in internal storage layers. The size of swap is usually set to twice the size of physical memory in typical swap-capable systems such as Linux. In our case, as we allocate each application's memory size relative to its footprint, we set the swap size to the entire footprint size to accommodate the full memory image in the swap area.

We compared SPO-CLOCK with CLOCK [19], ARC [21,22], and LRU algorithms. The performance of these reclamation algorithms was compared by the overall I/O time while replaying the memory access traces for each app. Note that write I/Os caused by the write/file region were not measured in this experiment since it is not possible to determine whether "data write" captured in the Valgrind memory access trace is actually written to an anonymous page or a file-mapped page. Thus, we assume "data write" in our trace to be a write to an anonymous page. In reality, when a file-mapped page is written in memory, it is shortly diverted to storage by the journaling or flush daemon in order to resist a system crash situation even though the page is not evicted by the reclamation algorithm [15,23]. This is different from a write to an anonymous page, which does not need to be reflected to storage until it is evicted by the reclamation algorithm. This implies that write I/Os incurred by file-mapped pages are not related to reclamation algorithms, and thus it is an issue independent to our performance evaluation.

Figure 4 shows the I/O time of SPO-CLOCK, ARC, LRU, and CLOCK as the memory size is varied. Note that we plot the I/O time of the algorithms as a relative scale to the result of the CLOCK algorithm. In this experiment, the memory size varied from 5% to 25% of the entire workload footprint. That is, the memory size of 100% was able to accommodate the full workload footprint simultaneously, which is the condition identical to the infinite memory capacity that does not incur any page reclamation at all. As shown in the figure, SPO-CLOCK performed better than the other algorithms regardless of the memory size for all workload cases. The reason behind this improvement is the adaptive memory management of SPO-CLOCK in accordance with the hotness of each partition, I/O cost of each storage type, and the workload characteristics. Specifically, SPO-CLOCK improves the I/O time by 15.0% on average and up to 35.1% compared to CLOCK. When compared to ARC, the performance improvement of SPO-CLOCK is 12.0% on average and up to 35.2%. Since the operation of LRU is similar to the CLOCK algorithm, it can be seen that the performance results of LRU and CLOCK are very similar.

In the figure, we separately plotted the I/O time of swap read, swap write, and file read, and then accumulated them. As shown in the figure, in most cases, the SPO-CLOCK's effectiveness apparently can be observed when the memory size is relatively small. This is due to the characteristics of SPO-CLOCK that manages the limited memory capacity efficiently. When we quantify the I/O time in detail, SPO-CLOCK improved the performance by reducing swap writes compared to CLOCK though it slightly increased file reads. This is because the cost of a swap write is the highest among all I/O types, so SPO-CLOCK took this into consideration while managing memory space in a cost-effective manner.

Another observable trend in Figure 4 is that swap writes account for the dominant I/O time compared to swap reads regardless of the memory size, workload types, and reclamation algorithms. This implies that a large number of pages are evicted to the swap area in case free memory is exhausted, but only some of hot pages are used again.

Meanwhile, in Chrome and Facebook workloads, the ratio of swap I/O diminished rapidly as the memory size increased. The reason is that the locality of memory accesses in these apps is high, and thus when the memory size is large enough to accommodate some limited hot pages, swap does not occur any longer. Note that, in this case, the proportion of file reads increased significantly as the memory size increased, as shown in Figure 4, but this does not imply the actual increase in I/O time since we plot the I/O time in a relative scale.

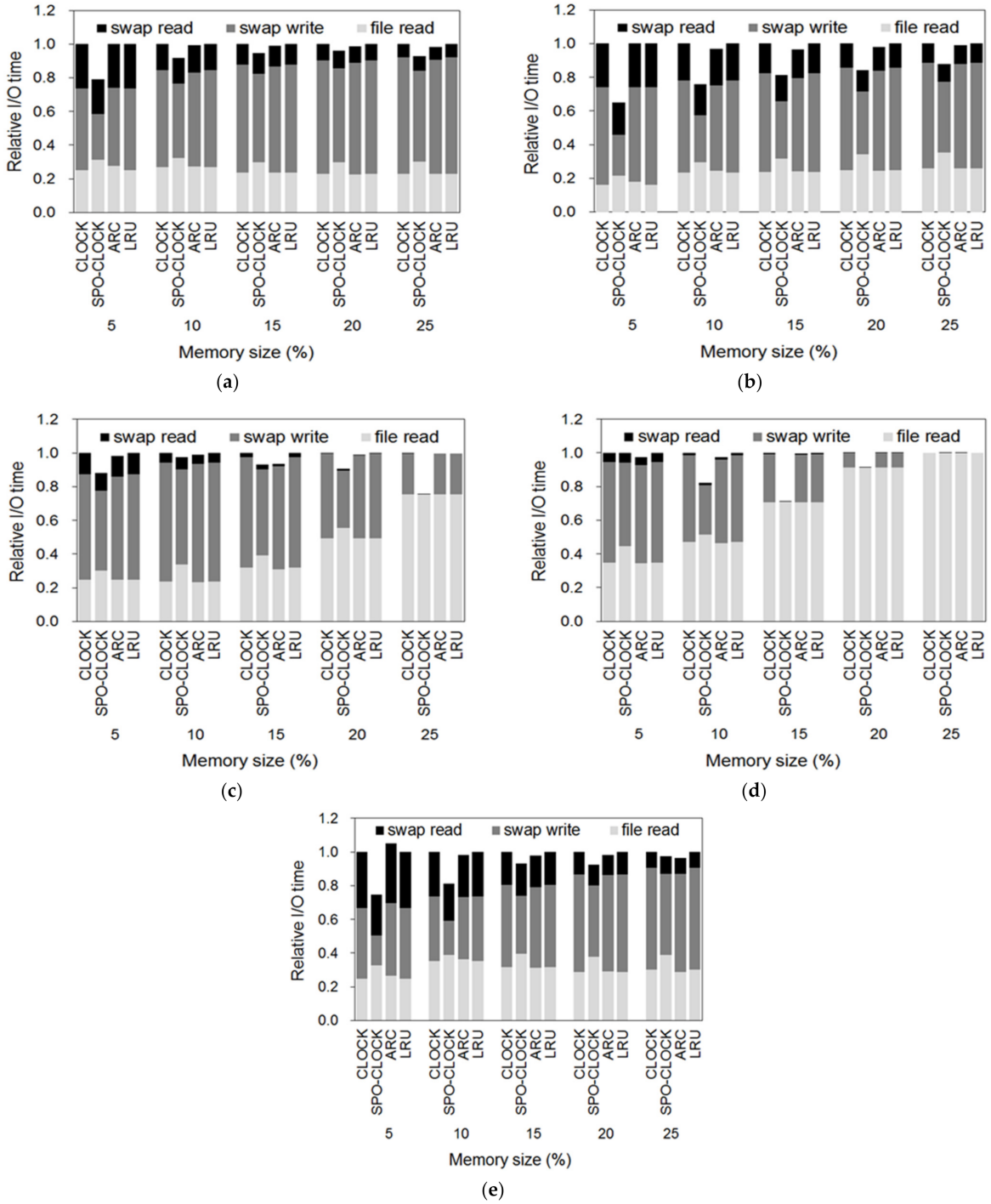


**Figure 4.** I/O time of CLOCK, ARC, LRU, and SPO-CLOCK as the memory size is varied. (**a**) Angrybirds, (**b**) Youtube, (**c**) Chrome, (**d**) Facebook, (**e**) Farmstory.

## 4. Related Works

To support swap in smartphones, much research has been conducted. zRAM-swap is supported in Android smartphones, which utilizes a certain part of DRAM as a swap partition that stores swapped pages in compressed form [24]. Han et al. suggest a hybrid swap policy to support storage-swap as well as zRAM-swap [25]. In particular, their policy first swaps pages of a process in zRAM-swap, and then the oldest pages are swapped to the storage-swap. Kim et al. suggested a selective swap scheme that limits the number of apps involved in swap by considering the context-saving characteristics of apps [9]. Specifically, if an app maintains its context by itself, their scheme does not support swap as is the case in conventional Android memory management. In contrast, if an app does not have the function of saving its context, their scheme supports swap, thereby saving the context in the swap area before removing its memory pages. Chae et al. proposed cloud-swap for smartphones by making use of cloud or server storage as a swap partition [26].

Some recent studies attempted to adopt NVM as the swap device of smartphones [7,8]. NVM is considered as a memory/storage medium that can accelerate storage system performance, but write operations on NVM are vulnerable in terms of endurance cycles and reliability. Thus, studies on NVM-swap have focused on resolving the weaknesses of NVM media. Liu et al. showed that flash-swap degrades the smartphone performance significantly, and suggested the adoption of NVM instead of flash as the swap device of smartphones [8]. Specifically, they focused on the management of NVM-swap such as wear leveling to distribute writes in NVM evenly to avoid a fast wear out of NVM cells.

Hadizadeh et al. proposed STAIR (STT-MRAM Aware Multi-Level I/O Cache Architecture) that adopts hierarchical storage architecture consisting of a first-level NVM cache and a second-level SSD cache [27]. For the NVM cache, they made use of STT-MRAM, and addressed reliability issues of read disturbance, write failure, and retention failure. Specifically, STAIR classifies the cached pages into clean and dirty, and distributes them to the two caches considering their vulnerability. To improve the reliability of storage further, STAIR dynamically generates additional ECCs (Error-Correction Codes) for dirty pages. Hadizadeh et al. also presented an NVM journal architecture for DRAM-based buffer in order to address the vulnerability of volatile DRAM [28]. For NVM, they made use of STT-MRAM as the persistent journal area, and presented a new buffer management scheme called CoPA (Cold Page Awakening). In order to reduce the retention failure of pages in STT-MRAM, CoPA periodically overwrites pages in the persistent journal area.

Our work also adopts NVM as the swap device of smartphones, similar to the aforementioned previous studies, but instead of focusing on NVM management, the main focus of our work is in the memory management issue of smartphones with NVM-swap. Thus, our work is orthogonal to previous studies on NVM-swap that focused on the management of the swap partition consisting of NVM. So, our work can be incorporated into previous works managing NVM-swap such as wear-leveling and failure resistance techniques.

## 5. Conclusions

As the domain of smartphone use expands from personal entertainment to various official tasks, supporting swap to preserve the context of apps is becoming increasingly important. However, the I/O overhead of swap degrades the performance of smartphones significantly as the number of apps increases. Previous studies showed that fast NVM storage can resolve the performance degradation problem of smartphone swap. In line with previous approaches that use NVM swap, we discussed how the memory management system of smartphones can be further improved with NVM swap. Specifically, we presented a new page-reclamation algorithm for smartphone memory that considers heterogeneous storage partitions of file system and swap, and their access characteristics. Our simulation experiments with real memory traces of Android apps showed that the proposed page reclamation algorithm improves the I/O performance of smartphones by 15.0% on average and by up to 35.1% compared to the CLOCK algorithm.

**Author Contributions:** H.Y. performed simulation experiments. K.C. performed the design of basic frameworks for experiments. H.B. designed the overall work and provided expertise. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported by the Ewha Womans University Research Grant of 1-2021-0473-001-1, and also by Basic Science Research Program through the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Education) (NRF-2020R1I1A1A01066121).

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Nayeem, I.; Want, R. Smartphones: Past, present, and future. *IEEE Pervasive Comput.* **2014**, *13*, 89–92.
2. Huang, F.; Li, X.; Zhang, S.; Zhang, J.; Chen, J.; Zhai, Z. Overlapping community detection for multimedia social networks. *IEEE Trans. Multimed.* **2017**, *19*, 1881–1893. [CrossRef]
3. Geng, J.; Xia, L.; Xia, J.; Li, Q.; Zhu, H.; Cai, Y. Smartphone-based pedestrian dead reckoning for 3D indoor positioning. *Sensors* **2021**, *21*, 8180. [CrossRef] [PubMed]
4. Lee, E.; Bahn, H. Electricity usage scheduling in smart building environments using smart devices. *Sci. World J.* **2013**, *2013*, 468097. [CrossRef] [PubMed]
5. Google Pixel 6 Pro. Available online: https://store.google.com/us/product/pixel_6_pro?hl=en-US (accessed on 30 December 2021).
6. Kim, S.; Jeong, J.; Kim, J.; Maeng, S. SmartLMK: A memory reclamation scheme for improving user-perceived app launch time. *ACM Trans. Embed. Comput. Syst.* **2016**, *15*, 1–25. [CrossRef]
7. Wang, K.Z.T.; Zhu, X.; Long, L.; Liu, D.; Liu, W.; Shao, Z.; Sha, E. Building high-performance smartphones via non-volatile memory: The swap approach. In Proceedings of the ACM EMSOFT Conference, New Delhi, India, 12–17 October 2014.
8. Liu, D.; Zhong, K.; Zhu, X.; Li, Y.; Long, L.; Shao, Z. Non-volatile memory based page swapping for building high-performance mobile devices. *IEEE Trans. Comput.* **2017**, *66*, 1918–1931. [CrossRef]
9. Kim, J.; Bahn, H. Maintaining application context of smartphones by selectively supporting swap and kill. *IEEE Access* **2020**, *8*, 85140–85153. [CrossRef]
10. Cho, M.; Kang, D. ML-CLOCK: Efficient page cache algorithm based on perceptron-based neural network. *Electronics* **2021**, *10*, 2503. [CrossRef]
11. Hyun, S.; Bahn, H.; Koh, K. LeCramFS: An Efficient Compressed File System for Flash-Based Portable Consumer Devices. *IEEE Trans. Consum. Electron.* **2007**, *53*, 481–488. [CrossRef]
12. Kim, J.; Bahn, H. Accelerating storage performance with NVRAM by considering application's I/O characteristics. In Proceedings of the IEEE International Conference on Big Data and Smart Computing (BigComp), Shanghai, China, 15–17 January 2018; pp. 383–389.
13. Bahn, H.; Noh, S.; Min, S.; Koh, K. Using full reference history for efficient document replacement in web caches. In Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems, Boulder, Colorado, 11–14 October 1999; pp. 187–196.
14. Kwon, O.; Bahn, H.; Koh, K. Popularity and prefix aware interval caching for multimedia streaming servers. In Proceedings of the 8th IEEE International Conference on Computer and Information Technology, Sydney, Australia, 8–11 July 2008; pp. 555–560.
15. Lee, E.; Kang, H.; Bahn, H.; Shin, K. Eliminating periodic flush overhead of file I/O with non-volatile buffer cache. *IEEE Trans. Comput.* **2016**, *65*, 1145–1157. [CrossRef]
16. Wang, H.; Zhao, Y.; Li, C.; Wang, Y.; Lin, Y. A new MRAM-based process in-memory accelerator for efficient neural network training with floating point precision. In Proceedings of the IEEE International Symposium on Circuits and Systems, Seville, Spain, 12–14 October 2020.
17. Bansal, S.; Modha, D.S. CAR: Clock with adaptive reclamation. In Proceedings of the 3rd USENIX Conference on File and Storage Technologies, San Francisco, CA, USA, 31 March 2004.
18. Johnson, T.; Shasha, D. 2Q: A low overhead high performance buffer management reclamation algorithm. In Proceedings of the VLDB Conference, Santiago, Chile, 12–15 September 1994.
19. Carr, R.; Hennessy, J. WSCLOCK—a simple and effective algorithm for virtual memory management. In Proceedings of the 8th ACM Symposium on Operating Systems Principles, Pacific Grove, CA, USA, 14–16 December 1981; pp. 87–95.
20. Nethercote, N.; Seward, J. Valgrind: A Program Supervision Framework. *Electron. Notes Theor. Comput. Sci.* **2003**, *89*, 44–66. [CrossRef]
21. Megiddo, N.; Modha, D.S. Outperforming LRU with an adaptive replacement cache algorithm. *IEEE Comput.* **2004**, *37*, 58–65. [CrossRef]
22. Megiddo, N.; Modha, D.S. ARC: A self-tuning, low overhead replacement cache. In Proceedings of the USENIX FAST Conference, San Francisco, CA, USA, 31 March–2 April 2003; pp. 115–130.
23. Kim, D.; Lee, E.; Ahn, S.; Bahn, H. Improving the storage performance of smartphones through journaling in non-volatile memory. *IEEE Trans. Consum. Electron.* **2013**, *59*, 556–561. [CrossRef]
24. Gupta, N. Compcache: Compressed Caching for Linux. 2010. Available online: http://code.google.com/p/compcache (accessed on 30 December 2021).

25. Han, J.; Kim, S.; Lee, S.; Lee, J.; Kim, S. A Hybrid Swapping Scheme Based on Per-Process Reclaim for Performance Improvement of Android Smartphones. *IEEE Access* **2018**, *6*, 56099–56108. [CrossRef]
26. Chae, D.; Kim, J.; Kim, Y.; Kim, J.; Chang, K.A.; Suh, S.B.; Lee, H. CloudSwap: A cloud-assisted swap mechanism for mobile devices. In Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud, Grid Computing, Cartagena, Colombia, 16–19 May 2016; pp. 462–472.
27. Hadizadeh, M.; Cheshmikhani, E.; Asadi, H. STAIR: High Reliable STT-MRAM Aware Multi-Level I/O Cache Architecture by Adaptive ECC Allocation. In Proceedings of the IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 9–13 March 2020; pp. 1484–1489.
28. Hadizadeh, M.; Cheshmikhani, E.; Rahmanpour, M.; Mutlu, O.; Asadi, H. CoPA: Cold Page Awakening to Overcome Retention Failures in STT-MRAM Based I/O Buffers. *IEEE Trans. Parallel Distrib. Syst.* **2021**. [CrossRef]