IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received August 31, 2019, accepted September 16, 2019, date of publication September 19, 2019, date of current version October 7, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2942362

ezswap: Enhanced Compressed Swap Scheme for Mobile Devices

JONGSEOK KIM[1], CHEOLGI KIM[2], AND EUISEONG SEO[1]
[1]Department of Computer Science and Engineering, Sungkyunkwan University, Seoul 16419, South Korea
[2]Department of Software and Computer Engineering, Korea Aerospace University, Goyang 10540, South Korea

Corresponding author: Euiseong Seo (euiseong@skku.edu)

This work was supported in part by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea funded by the Ministry of Science, ICT (MSIT) under Grant 2017M3C4A7080245 and in part by the Institute for Information and Communications Technology Promotion Grant funded by the MSIT under Grant 2017-0-00068 (A Development of Driving Decision Engine for Autonomous Driving Using Driving Experience Information).

ABSTRACT The limited memory capacity of mobile devices leads to the popular use of compressed swap schemes, which reduce the I/O operations involving the swapping in and out of infrequently accessed pages. However, most of the current compressed swap schemes indiscriminately compress and store all swap-out pages. Considering that both energy and computing power are scarce resources in mobile devices, and modern applications frequently deal with already-compressed multimedia data, this blind approach may cause adverse impacts. In addition, they focus only on anonymous pages and not on file-mapped pages, because the latter are backed by on-disk files. However, our observations revealed that, in mobile devices, file-mapped pages consume significantly more memory than anonymous pages. Last but not least, most of the current compressed swap schemes blindly follow the least-recently-used (LRU) discipline when choosing the victim pages for replacement, not considering the compression ratio or data density of the cached pages. To overcome the aforementioned problems and maximize the memory efficiency, we propose a compressed swap scheme, called enhanced zswap (ezswap), for mobile devices. ezswap accommodates not only anonymous pages, but also clean file-mapped pages. It estimates the compression ratio of incoming pages with their information entropy, and selectively compresses and caches the pages only with beneficial compression ratios. In addition, its admission control and cache replacement algorithms are based on a cost-benefit model that considers not only the access recency of cached pages but also their information density and expected eviction cost. The proposed scheme was implemented in the Linux kernel for Android. Our evaluation with a series of commercial applications demonstrated that it reduced the amount of flash memory read by up to 55%, thereby improving the application launch time by up to 22% in comparison to the original zswap.

INDEX TERMS Compressed swap, paging, compression, replacement algorithm, memory management, mobile devices.

I. INTRODUCTION

The main memory capacity of mobile devices has been rapidly increasing due to the ever-increasing degree of multi-tasking and improving quality of multimedia data. For example, the random access memory (RAM) size of the reference Android smart phone was 4 GB in 2018, while it was only 512 MB in 2010, an eight-fold increase in 8 years. The demand for a larger memory capacity is still strong. However, increasing the memory capacity is a challenging issue because it results in higher manufacturing costs and poor energy efficiency [1], [2].

Conventional computing systems, such as personal computers and servers, have achieved larger main memory space than RAM capacity by swapping out infrequently accessed pages to secondary storage devices. However, because the access speed of swap storage devices is usually lower than that of the RAM by up to 100 times, the swap-to-secondary storage scheme is undesirable for consumer electronics, which requires an immediate response to user inputs. In addition, the structure of flash memory, which is being popularly used for storage device in mobile consumer electronics

The associate editor coordinating the review of this manuscript and approving it for publication was Victor Sanchez.

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

for their high speed, robustness, and small form-factors, is transitioning from single- (SLC) to multi-level cell (MLC), including a triple- and quadruple-level cell (TLC and QLC, respectively) technology [3]. Because the write endurance cycles of TLCs and QLCs are 10–100 times smaller than those of SLCs [4], the frequent small random writes caused from page swap-out adversely impact the life span of mobile devices.

In the last two decades, various in-memory compressed swap schemes have been proposed to accommodate more data than the RAM size and improve the system performance by reducing page swapping I/O operations [5]–[10]. As shown in Fig. 1, such compressed swap schemes compress cold pages, which are not expected to be accessed in the meantime, and store the compressed pages, called *zpages*, in a swap page in the compressed swap pool in the RAM. When a zpage stored in the pool is requested from the main memory, it is decompressed and moved back to the main memory. In turn, the decompressed zpage will be removed from its swap page.

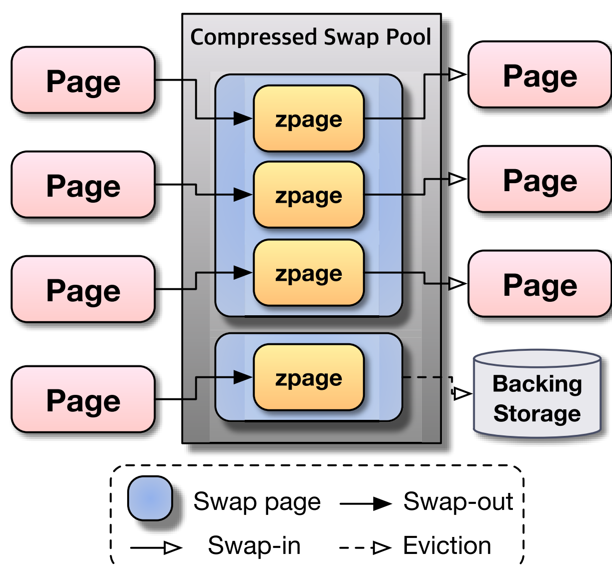


FIGURE 1. Design overview of compressed swap schemes.

As previously mentioned, the compressed swap schemes reduce the number of paging I/O operations, which are usually small random writes. Consequently, they improve the performance and, at the same time, the life span of the product. Therefore, to reduce the gap between memory demand and memory capacity limitation in mobile consumer electronics, compressed swap schemes are commonly used [11].

Currently, the Linux kernel provides two kinds of in-memory compressed swap schemes; *zram* [12] and *zswap* [13]. The Android OS have been using zram by default after its 4.4 release. This saves up to approximately a half of the memory usage [14]. zswap improved the critical drawback of zram that it cannot evict its zpages to the swap space in the secondary storage. Therefore, it is expected to perform better than zram under heavy memory capacity pressure. However, the current Linux compressed swap schemes in common do not consider the characteristics of memory contents and access patterns in mobile devices and hence, have significant room for improvement.

First, both zram and zswap blindly compress all pages evicted from the main memory and store them in the compressed swap pool. Because most of the applications that run in smart phones or hand-held devices deal with multimedia data in some way and the size of such multimedia contents is usually larger than that of text or numerical data, a large portion of the memory pages in such systems is expected to be already compressed. Compressing and storing such already-compressed pages will incapacitate the compressed swap schemes.

Second, most of the existing compressed swap schems, including zswap, evict the least-recently-used swap page to the swap file in the secondary storage when there are no more free space in the pool available for the newly incoming zpages. Depending on the compression ratios of the zpages stored in the victim swap page, the number of evicted zpages may differ when the swap page is evicted. However, because the existing compressed swap schemes do not consider the compression ratio or information density of swap pages when choosing the victim page, a swap page with many zpages is treated in the same way as a swap page with few zpages and their access recency will be the sole criterion in determining the victim. If the replacement algorithm considers the information density and access recency of swap pages at the same time, the compressed swap scheme will significantly improve the overall information density, and consequently the hit rate of the compressed swap pool.

Finally, both current zswap and zram target only anonymous pages, which are not backed by files and hence require backing up to the swap space when evicted. However, a significant portion of the main memory is dedicated to file-mapped pages, especially in mobile devices dealing with multimedia data. Moreover, clean (unmodified) file-mapped pages are evicted from the main memory before the anonymous pages because they can be reloaded from their corresponding files when they are necessary and thus, can be simply discarded without any storage writes. However, owing to their large size, their reloading may take a significantly long time. Therefore, a compressed swap scheme should accommodate the evicted file-mapped pages while considering the eviction cost difference between them and anonymous pages.

To overcome the aforementioned drawbacks and meet the demand of memory usage characteristics of mobile devices, in this study, we propose a compressed swap scheme optimized for mobile devices, called enhanced zswap (*ezswap*).

To increase the data density of the compressed swap pool, a compressed swap scheme should accommodate only

zpages with low compression ratios.[1] However, compressing all evicted pages to test their compression ratios is undesirable due to the limited performance and energy source of mobile devices. The selective admission (SA) scheme of ezswap enables rapid estimation of the compressibility of a swap-out page through entropy sampling, and therefore, can determine the pages that have low compression ratios and high-performance benefits to selectively store them with little overhead.

In addition, to improve the aforementioned naive least-recently-used (LRU) cache replacement algorithm, ezswap chooses the eviction victims based on a cost-benefit model that simultaneously considers the data density, access recency, and eviction cost. To determine a victim page at the constant time without costly sorting or linear searching, this compressibility-aware (CA) replacement algorithm adopts the lottery selection technique [15].

Last but not least, ezswap employs the all-page accommodation (APA) scheme, which enables the accommodation of both anonymous pages and file-mapped pages. For this, the cost-benefit model of the page replacement algorithm also incorporates the eviction cost difference between these two pages types.

The proposed scheme was implemented in the Linux kernel 3.10.9 for Android 7.1.2. The kernel was ported to an embedded board with 2 GB RAM, which is being used in commercial smart phones. The prototype system was evaluated with the commodity applications, which are publicly obtainable from the application stores.

The remainder of this paper is organized as follows. In Section II, we introduce the related work on memory overcommitting techniques, and explain the design of the current Linux compressed swap schemes and their limitations. After proposing ezswap in Section III, we evaluate its prototype implementation in Section IV. Finally, in Section V, we outline the conclusions drawn from our research.

II. BACKGROUND AND RELATED WORK

A. MEMORY OVERCOMMITMENT TECHNIQUES

Being a spatial sharing resource, the RAM has been the major limiting factor of the degree of multitasking of computing systems since their inception. Therefore, overcommitting memory management techniques that enable storage of more data than the given RAM capacity have been actively researched.

The delta encoding scheme, which groups similar data blocks together and stores only the base block and their differences from the base block, has been proposed to save disk space of file systems that have a large number of similar files [16], [17]. It is known to be unsuitable for the main memory because it rarely contains similar pages and it is frequently updated. However, its application to the main memory in virtualized servers showed promising results because, in them, two or more virtual machines (VMs) generally run the same OS and share the same software frameworks [18]. However, delta encoding requires a large amount of energy and computing resources for permutation operations between similar pages; this is unacceptable in mobile devices, which have strict limitation in energy source and computing performance. In addition, unlike virtualized server environments, mobile devices have few similar pages, which makes delta encoding ineffective.

The page sharing technique, which unifies and stores multiple pages with the same content into a single page, has been also widely used to reduce memory usage in systems with redundant data. The Linux kernel provides the kernel same-page merging (KSM) scheme [19]; the VMware ESX Server also provides an identical-page sharing feature [20]. The KSM scheme saves a significant amount of memory consumption in virtualized server systems [21]. However, in embedded systems, most of the benefit from applying KSM was gained from deduplicating zero pages [21], [22]. The overhead of page sharing is also inappropriate for mobile devices because it continually scans through the entire memory space and compares the fingerprint of each page to that of the others to find identical pages [23].

Three decades ago, the memory compression technique, which compresses and stores infrequently accessed data, was proposed for the Sprite OS [6]. However, it was known to have high computational overhead and cause long read latency with inconsistent amounts of memory savings. Wilson *et al.* analyzed the causes of these phenomena and proposed a compression algorithm suited to in-memory data compression. They also presented a dynamic capacity adjustment scheme of the compressed cache area that reacts to memory access patterns [5].

Memory compression can significantly reduce disk I/O for swap operations and thus reduce the execution time when there are sufficient computing power and energy. Therefore, it has been used in one way or another by most commercial OSs for personal computers [24], [25]. A few compressed swap schemes for server systems have also been proposed to meet the strong memory demand of highly consolidated VMs or memory-intensive workloads [26]–[28].

To reduce the compression overhead, a few hardware components that are dedicated to memory compression were proposed [29], [30]. These hardware components are commonly placed between the content-compressed main memory and the processor; they retrieve contents requested from the processor and decompress them to the processor cache. Although these dedicated units can effectively compress the main memory with negligible impact on performance, they are hardly applicable to mobile devices owing to their energy consumption and manufacturing cost.

B. SWAP SCHEMES FOR MOBILE DEVICES

The overhead from compression operations prevents the application of compressed swaps to mobile devices.

[1]The compression ratio in this paper is defined as (*compressed size*/*uncompressed size*) for the easiness of formulation and explanation because this keeps the ratio to stay in the range (0.0, 1.0].

In particular, as mentioned earlier, the blind compression of all pages may result in significant performance loss because modern consumer electronics systems frequently process audio and video data, which are already compressed.

CRAMES is a compressed swap scheme designed for embedded systems [9]. It reduces energy and computing resources consumption for compression by dynamically adjusting the compressed swap area. However, because it compresses all swap-out pages, it still incurs significant compression overhead.

Han *et al*. proposed a hybrid swap scheme that stores frequently accessed data in the in-memory compressed swap area and sends infrequently accessed data to the swap space in the secondary storage [14]. This scheme improved the hit rate of the compressed swap by accommodating only the pages with low compression ratios and access frequencies in the compressed swap.

The Cloudswap scheme evicts read-intensive pages to local storage and write-intensive pages to cloud storage to extend the life span of flash memory secondary storage [31]. Fundamentally, it is similarly to the hybrid swap approach. However, it focused on the read-latency optimization techniques, such as prefetching and read-ahead, to mitigate the long access latency of the cloud storage.

Song et al. reduced the wear-and-tear of flash memory caused by the page swap operations by proposing the *Enhanced Flash Swap* (EFS) file system that combines compression, deduplication, and flash-friendly writing methodologies [32]. When swapping out to the EFS file system, the degree of wear-out could be reduced by up to 138 times when compared to the conventional swap mechanism. The use of the ezswap on top of the EFS file system can further extend the life span of flash memory because the ezswap shrinks the number of pages flowing down to the swap file stored in the flash memory.

When numerous identical pages or zero pages are present in the compressed swap pool, the deduplication technique, which merges them into a single page, enables the pool to accommodate a considerably larger number of zpages. Desireddy et al. showed that the zswap with deduplication could store up to 20% more zpages [33]. The incorporation of the deduplication technique into the ezswap scheme may increase the effective pool size. However, deduplication requires additional processor cycles to compare the page fingerprints, and it degrades the cache hit ratio because of page traversing. We believe that an analysis of the trade-off between the overhead and benefit is beyond the scope of this paper.

A memory management scheme that temporarily stores swapped-out pages of a VM in the surplus memory space of the other VMs was proposed for virtualized embedded systems [27], [28]. This approach is effective only in the virtualized environments where the memory space is managed separately into multiple isolated entities.

Non-volatile RAM (NVRAM) can be equipped in mobile devices to overcome the limitation in DRAM capacity. In-memory file systems on NVRAM, which provide file storage, improve the performance through rapid file I/O operations. Choi *et al.* proposed an in-memory file system with efficient swap support for NVRAM that evicts a portion of its files to the flash memory storage [34]. Our approach can be applied to the main memory swap as well as to the swap module of the in-memory file systems.

The memory paging patterns in mobile devices have a close relationship with the application life cycles. For example, termination of a background task that has not been used for a long time for an out-of-memory exception results in a collective reclamation of the pages that were allocated for that task. However, the chances are high that the pages were evicted before the out-of-memory condition occurs because the task has not recently accessed its pages. In such cases, the I/O operations performed for evicting the pages become obsolete. Kim et al. proposed an application-aware swapping approach for mobile devices [35] that links the eviction victim selection policy to the application life cycle management. Their approach is orthogonal to the ezswap; thus, we expect that the combination of the application-aware eviction policy and the ezswap will bring about synergic effects in terms of the number of I/O operations and the quality of user experiences.

A selective compression scheme that only stores data with a low compression ratio to remove unnecessary decompression delay and data bloat from having metadata was also proposed for managing compressed entries [36], [37]. The existing selective compression schemes, unlike the one proposed in this paper, compress the incoming data to determine their compression ratio. This results in performance and energy losses, especially when the data mostly have a high compression ratio. To reduce the computing power required, the selective compression scheme proposed for ezswap uses a rapid lightweight filter to estimate the compression ratio of a given page accurately.

C. COMPRESSED SWAP IN LINUX

As previously mentioned, the Linux kernel also provides memory compression features. The users can choose between two compressed swap schemes, namely zram and zswap, which are integrated in the mainline kernel. zram and zswap commonly use a small portion of memory for storing compressed pages. However, zram provides the compressed swap area as a block device while zswap locates the compressed space between the main memory and the swap file. Therefore, zram functions as a conventional swap storage while zswap works as a swap cache.

Because of this difference in their design, zram has to bypass incoming pages to the swap file stored in the secondary storage when the compressed space is full. However, zswap evicts some of the cached swap pages to the secondary storage first and receives newly incoming pages when it is full. Therefore, when there are many long-living inactive tasks in a system, for example a smart phone, zram can be easily filled up with cold data and lose its effectiveness; on the

other hand, zswap continually replaces cold pages with hot pages, which are frequently accessed pages. Consequently, considering the memory usage patterns of the target systems, we chose zswap as the design basis in this research.

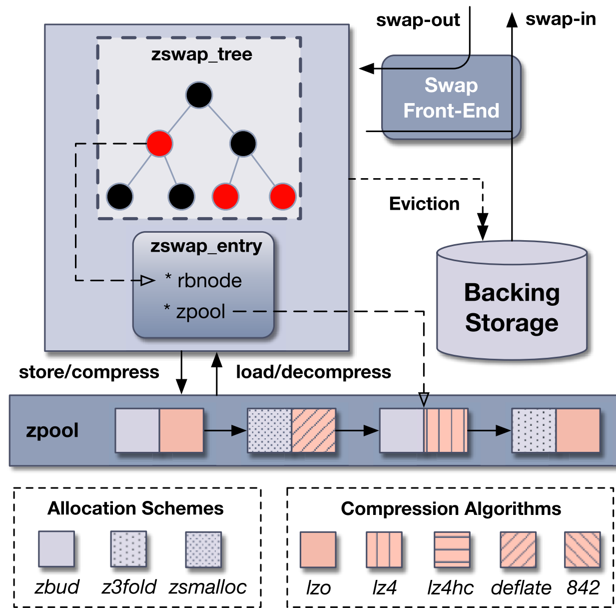


FIGURE 2. Interaction among swap front-end, zswap and zpool.

As shown in Fig. 2, zswap intercepts swap-out requests issued by the swap front-end to the block I/O layer, and forwards them to the compressed swap pool, which is called *zpool*, where they are stored. When a page swap-in request is triggered by the swap front-end, zswap intercepts it and determines whether it has the matching zpage in its zpool. If it is in the pool, the zswap will decompress the zpage and deliver the restored page to the swap front-end. Otherwise, the requested page will be loaded from the swap file directly by the swap front-end.

When the zpool has no more free space, a swap page, which is a unit of the zpool management and may hold multiple zpages, will be chosen as a victim, and all zpages stored in the victim swap page will be evicted to the swap file in the storage device.

The actual compressed swap pool management of zswap is performed by the zpool memory manager. In fact, zswap only intercepts the requests coming from the swap front-end and delivers them to the zpool manager. The zpool manager allocates an available swap page for an incoming swap-out page, compresses the incoming page to a zpage, and stores the zpage in the allocated swap page.

zswap keeps track of the location of each zpage in the zpool with a red-black tree. The key of the red-black tree is the offset of the target page in the original swap file. Even when a swapped-out page has been stored not in the swap file but in the zpool, the swap system allocates space for the page in the swap file. Thus, the zpage' s offset in the swap storage can be used as its identifier. With this red-black tree, zswap can easily and rapidly find the requested page in the zpool. The actual decompression and removal of the requested zpages are also performed by the zpool manager.

As shown in Fig. 2, the zpool manager provides multiple alternative swap page allocation schemes and compression algorithms. zswap is configured to use a specific combination of the compression algorithm and swap page allocation scheme in advance. The kernel currently provides the *lzo*, *lz4*, *lz4hc*, *deflate*, and *842* compression algorithms, with the default one being the lzo, which exhibits low overhead and high compression speed.

The zpool manager provides three swap page allocation schemes: the *zbud*, *z3fold*, and *zsmalloc*. The user chooses one of these three for the zpool management. Each zpage cannot be stored across the boundary between two swap pages for the ease of management and efficiency of the CPU cache. Because each zpage has a different compression ratio, it is difficult to predict the number of zpages that will be packed in a single swap page.

The zsmalloc allocator vigorously packs as many zpages as possible in a single swap page to maximize the space utilization. However, as zpages are repetitively removed and inserted, a large number of holes, or free spaces, will be generated with various sizes scattered over swap pages. This leads to an internal fragmentation that, although there is free space available, does not allow new zpages into the pool.

The zbud stores up to two zpages in a swap page even when there are sufficient space sections in the swap page to accommodate more zpages. Therefore, there can only be a free section in a swap page and it is always located at the front- or read-end of a swap page when it has only one or zero zpage in it. This approach solves the internal fragmentation issue. However, the space utilization under zbud will be significantly low, especially when the average compression ratio is below 0.3 because it cannot utilize the surplus free space in a swap page after storing two zpages.

z3fold was proposed to improve the inefficient space utilization of zbud. It aims at storing up to three zpages in a swap page. As illustrated in Fig. 3, z3fold splits and manages a swap page into 64 fixed-size chunks. Every swap page must belong to either a buddied or an unbuddied list. The buddied list is a list of pages that already have three zpages in them. The unbuddied list is a list of 64 swap page lists. Each node in the unbuddied list is the list head of a swap page list, of which pages have the same number of free chunks. For example, the 17th node in the unbuddied list points to the list that links all swap pages with 17 free chunks and with two or fewer zpages in them. The unbuddied list is used for chunk allocation for a new zpage. For instance, if a new zpage requires ($\frac{17}{64} \times$ page size) bytes, the first swap page of the 17th list of the unbuddied list will be removed from the list and be used for storing the zpage. If there are no swap pages in the list, then the next list will be explored. Every list follows the LRU discipline. The newest swap page in a list is placed at its head and the oldest one at its tail. Therefore, after strong

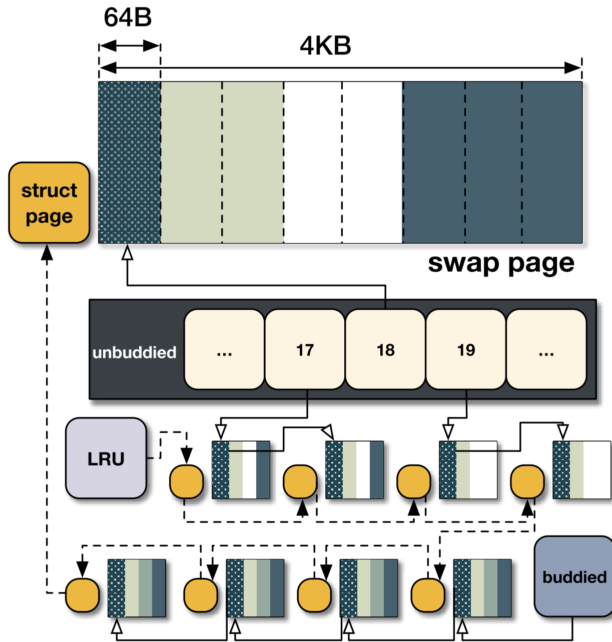


FIGURE 3. zpool management with the z3fold scheme.

the new zpage, the swap page will be placed at the head of the proper list according to its new state.

Because a swap page can have up to three zpages, fragmentation occurs when the first and last zpages are removed from a swap page filled with three zpages. Therefore, z3fold performs compaction when a zpage is removed so that all free chunks are placed consecutively at the rear-end of a swap page. With this design, z3fold can improve the space usage efficiency by up to 50% in comparison to zbud at the cost of marginally increased zpage removal overhead, while resolving the internal fragmentation issue of zsmalloc. Therefore, we chose z3fold as the swap page allocator for our research.

Separate from the above mentioned lists, the zpool manages another list that holds all swap pages in the compressed swap pool. This list is managed with the LRU policy. When a swap page stores a new zpage, it will be moved to the head of the list. Consequently, the swap page that has not stored a new zpage for the longest time is located at the tail of the list. When eviction of a swap page is necessary to create free space for incoming zpages, the zpool, regardless of the underlying allocation scheme, chooses the swap page at the tail as the eviction victim.

III. OUR APPROACH

A. ALL-PAGE ACCOMMODATION

When the free pages in the main memory become few, the kernel iterates the pages and frees them according to the page reclamation policy. The current Linux kernel evicts the clean file-mapped pages before dirty (modified) file-mapped pages and anonymous pages. The anonymous pages are kept last because their eviction causes unnecessary write operations to the swap file.

As stated earlier, the conventional compressed swap schemes do not store clean file-mapped pages because they can be simply re-read from files when necessary. However, swapping out a large number of clean file-mapped pages may considerably delay the system, especially when a significant portion of file-mapped pages will be reused because reading from the storage device takes a long time although it is faster than writing.

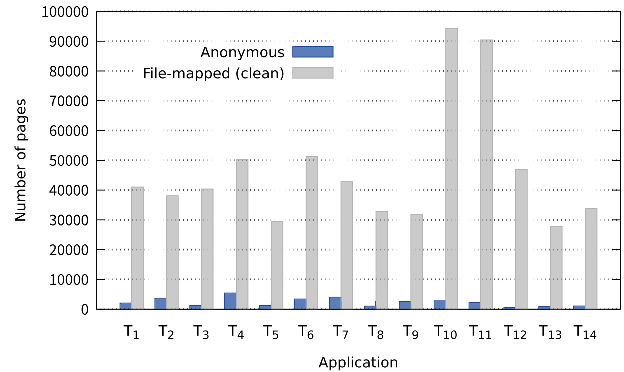


FIGURE 4. Number of file pages and anonymous pages evicted from the main memory during the execution of each application.

Fig. 4 shows the number of anonymous pages and clean file-mapped pages evicted from the main memory during the execution of a series of 14 applications listed in Table 2. Each bar represents results collected during the execution of each application. The details of the experimental set up are explained in Section IV.

The results showed that, for all applications, the number of file-mapped pages evicted was 20.3 times larger than that of anonymous pages. If the evicted file-mapped pages are to be never reused in the future, discarding them will not impact the performance. However, most of them are likely to be reloaded in the target environment because of the nature of mobile systems, i.e., a few applications are repetitively executed one at a time [38], [39]. Therefore, eviction of such a large number of file-mapped pages ends up with significant performance loss.

A dirty file-mapped page must be eventually written back to its originating file to permanently apply the changes to the file. Therefore, caching the dirty file-mapped pages in the compressed swap may produce inconsistency at the file system level. For example, a committed write operation may be lost when the system crashes while retaining the written page in the compressed swap pool. For this, the ezswap only considers clean file-mapped pages. The APA scheme proposed for the ezswap compresses file-mapped pages that are discarded from the main memory and stores them together with anonymous pages in the compressed swap pool.

When a read request to a file occurs, the Linux kernel checks whether the requested file block exists in the buffer cache. If it does not exist, the kernel allocates the buffer page and submits a read request to the block I/O layer to actually read the file blocks from the storage to the allocated

buffer pages. The ezswap intercepts every block I/O request and checks whether the requested file blocks are stored in the compressed swap pool. When all the required file blocks are stored in the zpool, the block I/O operation will be simply ignored and the matching zpages will be decompressed, forwarded to the buffer cache layer, and removed from the zpool. If only a part of the requested file blocks is held in the compressed swap pool, the ezswap will create new read requests to read the non-existing parts and submit them to the block I/O layer. When finishing the file read operations, the ezswap merges the read data with its zpage contents.

Because the number of evicted file-mapped pages is significantly larger than that of evicted anonymous pages, adoption of the APA scheme causes the following three issues.

First, the number of pages to be possibly accommodated in the swap pool increases dramatically. Consequently, a significantly larger number of anonymous page evictions may occur due to the excessively large number of file-mapped pages.

Second, when the zswap accommodates an anonymous page, it can additionally save a page write operation, unlike the case of storing a file-mapped page. Write operations to the flash memory are critically slower than read operations. In addition, they reduce the life span of the flash memory storage. Therefore, APA may reduce the life span and performance due to the increased write operations arising from storing the lower number of anonymous pages in the zpool.

Last but not least, a large portion of the file-mapped pages contains already compressed media data. Therefore, the APA crucially lowers the data density in the compressed swap pool.

To overcome these issues and the problems of the conventional compressed swap schemes mentioned previously, we propose selective admission and compressibility-aware replacement schemes for the ezswap.

B. SELECTIVE ADMISSION

Compressed swap schemes trade off memory consumption with compressing and decompressing overheads. However, when the page compression ratio is high, they waste computing resources, while only slightly reducing the memory usage. Because mobile devices frequently use media data, such as videos, photos, and audio, and ezswap accommodates file-mapped pages, the proportion of the incoming pages with extremely low compression ratio is expected to be a lot higher with ezswap than with conventional compressed swap schemes.

When the distribution of the page compression ratio is wide, it is possible to store more pages in the swap pool with the same capacity by selectively accommodating only pages with low compression ratios. For example, assume that the zpool stores 200 zpages on average with a half of the swap-out pages having compression ratios of 1.0 and the other half having compression ratios of 0.5. If the swap pool holds pages with the compression ratio of 0.5 only, the number of zpages stored in the swap pool will increase to 300. The selective accommodation of highly compressible pages would increase the data density of the swap pool, and thus improve the effectiveness of the compressed swap.

In addition, if an anonymous page and a file-mapped page have the same compression ratio and reuse probability, it is more profitable in terms of performance and flash memory wear to store the anonymous page in the compressed swap pool than the file-mapped counterpart, because it will reduce additional write operations.

We propose the SA technique to prevent waste of computing resources due to blind compression and to maximize the benefits of the ezswap. SA determines whether the ezswap accepts a swap-out page in the pool based on its estimated compression ratio, probability of reuse, and eviction cost. The ezswap admits a page only when the *Expected benefit* of keeping it in the zpool is greater than or equals to the *Average Benefit* of retaining the existing zpages in the zpool.

The *Expected Benefit* of a target page is determined by Equation (1). *Cost* is the sum of the cost of abandoning the target page and reloading it from the storage in the future. In other words, the larger this value, the more beneficial it is to keep the target page in the zpool. In general, a random write in a flash memory storage is two to four times slower than a random read [40]. Thus, we set *Cost* to 1 for file-mapped pages because they require one read operation, and to 4 for anonymous pages because abandoning an anonymous page requires a read operation and an additional write operation. *Compression Ratio* represents the expected compression ratio of the target page. The lower is this value, the higher are the benefits of the ezswap by retaining the target page.

$$Expected\ Benefit = \frac{Cost}{Compression\ Ratio} \tag{1}$$

Equation (2) calculates the average benefit of zpages in the zpool. *Avg Cost* and *Avg CompressionRatio* represent the average eviction cost and compression ratio of the zpages held in the zpool, respectively. *Hit Rate* represents the hit rate of recent page-in requests to the ezswap. To calculate this, the ezswap maintains a 256-bit map called the *hit window*. When a page-in request is delivered to the ezswap, it performs a bit-wise right shift operation over the hit window. Then, it sets the MSB of the hit window if the requested page exists in the zpool. Otherwise, it clears the MSB. With this bit map, the ezswap can count the number of hits occurred during the last 256 accesses. *Capacity* is the ratio of the number of used chunks to the number of total chunks in the zpool. This encourages the ezswap to more aggressively admit the incoming pages into the zpool by lowering *Average Benefit* as the zpool is less occupied.

$$AverageBenefit = Capacity \times \frac{Avg\ Cost \times Hit\ Rate}{Avg\ Compression\ Ratio} \tag{2}$$

If the ezswap actually compresses all the incoming pages to determine their *Compression Ratio*, a significant waste of computing resources will occur because a large number of incoming pages will be simply discarded after compression by the SA feature.

It is well known that information entropy has a strong positive correlation with the compression ratio [41]–[43]. To avoid meaningless compression, the ezswap uses the information entropy of a target page, which is calculated by Equation (3), to estimate its *Compression Ratio*. P_i is the probability that a randomly sampled byte has an integer value of i. To suppress the overhead, we used only the first 512 bytes of a page to approximate its P_i, and we verified that such approximation method produces sufficiently accurate estimation as shown in Fig. 6. The entropy ranges from 0, the perfectly regular data, to 8, the complete chaos. This entropy value is scaled and mapped to the estimated compression ratio, ranging from 0 to 1.

$$-\sum_{i=0}^{255} P_i log_2 P_i \tag{3}$$

Calculating entropy is a lot faster than compression owing to its simplicity. However, the calculation may still consume a significant amount of processor cycles when it is performed across all the evicted pages. The ezswap suppresses the overhead arising from entropy calculation by estimating the Shannon entropy of a page through calculation of the entropy of its leading 512 bytes. Fig. 5 shows the relationship between the estimated entropy based on the 512-byte sample of a target page and the actual entropy of the whole page. The Pearson correlation coefficient between the estimated entropy and actual value was 0.955. The coefficient was 0.938 when sampling only 256 bytes. Therefore, the ezswap secures sufficient accuracy while keeping low overhead by using the leading 512 bytes of an incoming page as the sample to estimate its entropy.

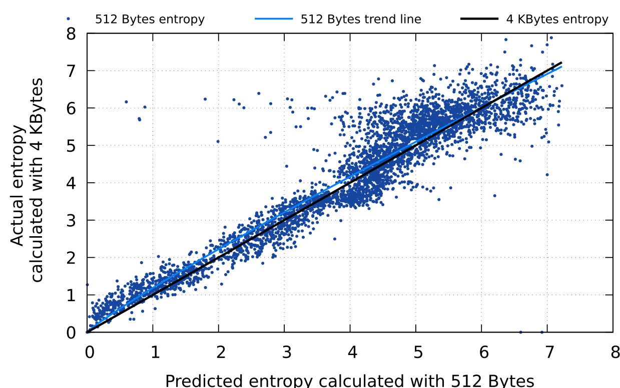


FIGURE 5. Relationship between entropy values obtained from leading 512 bytes of pages and that from whole bytes of them.

Fig. 6 shows the relationship between the information entropy of the leading 512 bytes of the pages that the ezswap receives and their actual compression ratio obtained from the lzo algorithm. The number of pages observed in this experiment was 200,000. The Pearson correlation coefficient between the entropy and compression ratio was 0.88, which indicates a strong positive correlation. The lzo algorithm is approximately 16 times faster than the widely-used zip

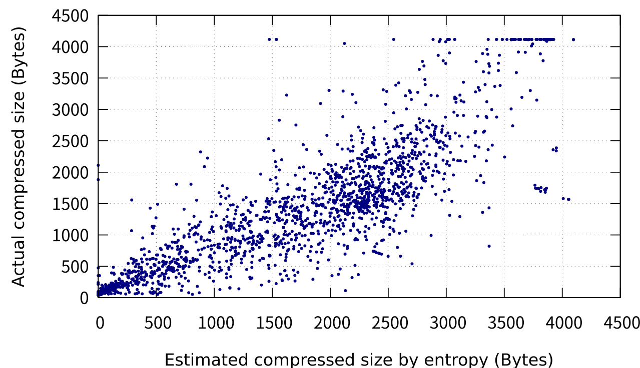


FIGURE 6. Relationship between estimated compression ratios of incoming pages based on their 512-byte entropy and actual compression ratios.

algorithm when using the system described in Table 1. The time taken to calculate the entropy was only 14% of that taken to compress the pages with the lzo.

C. COMPRESSIBILITY-AWARE PAGE REPLACEMENT

The ezswap chooses a victim page among the swap pages in use when there is no more free space in the pool for the new incoming zpages. Once a swap page is chosen as the victim, all zpages in the swap page are either evicted to the swap storage or simply discarded, depending on their types (anonymous pages or file-mapped pages) and, in turn, the swap page becomes a free page.

As stated earlier, the compressed swap pool management in the zswap depends entirely on the zpool module, and its swap page allocator is in charge of victim selection for eviction. All three swap page allocation schemes of the zpool commonly follow the LRU discipline for victim selection, which chooses the swap page unmodified for the longest time.

The zpages stored in the compressed swap pool usually have various compression ratios, and therefore, their information densities are also diverse. In other words, different zpages have different sizes. If the compression ratio is the only variable, it is desirable for the hit rate of the zpool to retain a zpage with low compression ratio rather than a poorly compressed zpage. Therefore, the swap page replacement policy should consider both the modification recency of a swap page and its data density.

Unlike conventional compressed swap schemes, ezswap holds both file-mapped pages and anonymous pages. As aforementioned, these two-page types have different eviction costs. Therefore, a swap page replacement algorithm should also consider the type of swap pages, which relates to their eviction cost, when choosing a victim.

As the swap page replacement policy of ezswap, we propose the CA replacement algorithm. This aims at maximizing both hit rate and performance gain from page hits. To achieve these goals, it simultaneously considers the age, data density, and eviction cost of swap pages with the following cost-benefit model.

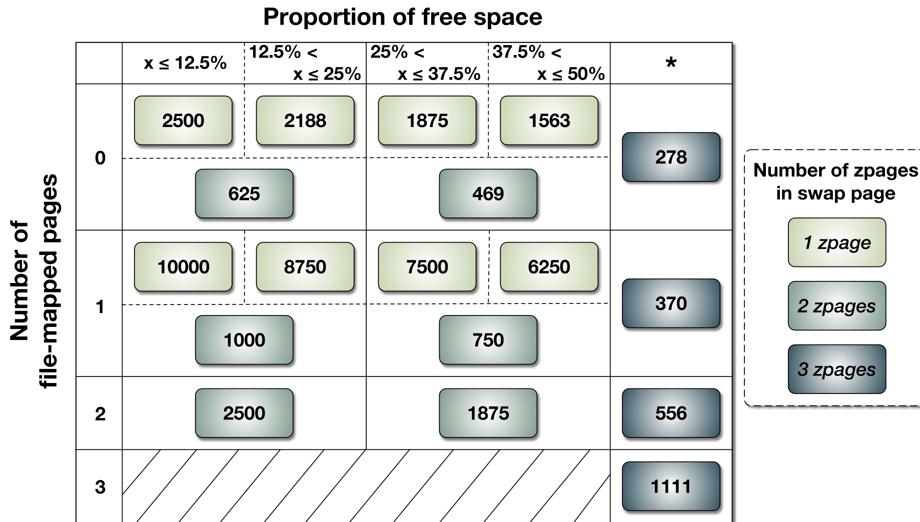


FIGURE 7. Categorization of swap pages and victim selection weight of each category.

In z3fold, each swap page can hold up to three zpages with different compression ratios. The CA algorithm uses the number of zpages and the number of remaining free chunks in a swap page combined to denote its data density. The fewer free chunks a swap page has and the fewer zpages the swap page stores, the more likely it is to be chosen as a victim. In addition, if every condition is the same, the more file-mapped pages a swap page stores, the more likely it is to be chosen as a victim, as this policy will lower the eviction cost.

The ezswap classifies its zpages into 18 categories, as shown in Fig. 7, depending on the number of its total zpages, the number of its file-mapped pages, and the amount of free space. It maintains a swap page list for each of these categories, and all non-free swap pages must belong to one of these lists.

Each list has its weight for victim selection that means the likelihood to be chosen as a victim. The weight values are determined by Equation (1). The weight should be inversely proportional to Equation (1) because the loss caused by the eviction of a swap page is the opposite of the benefit achieved from keeping it in the pool.

We use the lowest number in the free space range of a category to represent its average compression ratio or data density. For example, when a swap page holds one file-mapped page and one anonymous page, and 30% of its space is free, its compression ratio is approximately calculated to 0.75/2, and its eviction cost is five, one from the file-mapped page and four from the anonymous page. Therefore, its weight is $1/(5/0.375) = 0.075$. Because floating point calculations inside the kernel require saving and restoring of the floating point registers, floating point operations in the kernel context are avoided as much as possible to prevent performance degradation [44]. Therefore, the value, i.e., 0.075, is multiplied by 10,000 to convert it to an integer, i.e., 750.

A larger weight of a swap page means lower performance loss caused from evicting it. Therefore, evicting the swap page with the largest weight is expected to minimize the performance loss. However, if ezswap always evicts the swap pages with the largest weight, recently modified swap pages with only one or two zpages will always be expelled before full old pages. This will completely deprive recent and unfilled swap pages of maturing opportunities and ultimately lower the performance. To address this issue, ezswap operates a lottery to actually determine the victim.

Each swap page list is given tickets for a lottery. The number of tickets given to list i, denoted as T_i, is determined by Equation (4) and is proportional to the number of swap pages in the list, N_i, and the victim selection weight of the list, W_i.

$$T_i = N_i \times W_i \tag{4}$$

When a free swap page becomes necessary, the ezswap produces a random value ranging from 0 to $\sum T_i$. If the random value falls in the range of a list, that list will become the victim list. The swap pages in a swap page list are ordered with the approximated LRU discipline, with the oldest swap page being located at the tail. Therefore, when a victim list is selected by the lottery, the last swap page in that list will be finally selected as the victim swap page. After evicting or discarding the zpages in the victim page, it will be put back to the free page list of the pool.

When a zpage is removed from or inserted to a swap page, the swap page will be migrated to the swap page list that matches its changed state. To ensure the list complies with the approximated LRU order, the list insertion of a swap page is performed as follows.

ezswap maintains a global clock, which is a monotonically increasing integer value. When a new zpage enters the pool,

it receives a time stamp of the current global clock, and then the clock value is increased by 1. A swap page manages the sum of the time stamps of its zpages, by adding or deducting the time stamp of a zpage to or from the sum when the zpage enters or leaves the swap page, respectively. The time stamp sum is recorded in the swap page header.

When a swap page is inserted into a swap page list, ezswap compares the time stamp sum of the swap page with the average time stamp sum of the swap pages that currently belong to the list. If the time stamp sum of the to-be-inserted page is bigger than the list average, the swap page will be attached to the list head. Otherwise, it will be placed at the list tail. With this approach, the head side of each list will have the recently-modified swap pages, and the tail side will have the old swap pages.

The proposed replacement scheme reflects the cost-benefit model so that it resolves the performance loss from the naive LRU-based swap page replacement and from the blind eviction of the recent unfilled swap pages. In addition, by adopting the lottery selection approach, regardless of the zpool size, it decides the eviction victim within a constant amount of time.

TABLE 1. Evaluation system configurations.

Board	Odroid-XU4
SoC	Exynos 5422
CPU	4×Cortex-A15 @ 2.0GHz 4×Cortex-A7 @ 1.4GHz
Memory	2GB LPDDR3 RAM @ 933MHz
Storage	32GB eMMC 5.0 HS400 Flash Storage
OS	LineageOS based on Android 7.1.2 with Linux Kernel 3.10.9

IV. EVALUATION

A. EVALUATION ENVIRONMENT

To evaluate ezswap, we used the experimental environment described in Table 1. In order to obtain consistent results, we used only the A7 cores and disabled the A15 cores because the unpredictable task migration between these two heterogeneous core sets complicates the performance outcomes. We used the Linux kernel 3.10.9 ported to the target board. Because this version does not include the zswap and the z3fold allocator, we imported the zswap and zpool modules from the Linux kernel 4.7 to the experimental kernel. The compressed swap pool size was set to 100 MB, which is 5% of the total memory capacity and 10% of the free space after loading the OS and application frameworks. To reduce the variance in the results, we also disabled the low memory killer (LMK) daemon.

We used 14 applications listed in Table 2 to apply memory pressure. These were commodity applications obtainable from the Google Play Store. To simulate the smart phone users' usage pattern, we launched an application and let it run for 30 s, returned to the home screen, and repeated these steps for the next application in the list. An experiment run consisted of a five-time repetition of the launch-run-termination cycles of the 14 applications. The system was rebooted and the page cache was emptied through the *drop_caches* interface after every experiment run. The data was collected after finishing the first repetition so that the swap pool was warmed up with cached contents. This excludes the delays caused by the first access of each page from our analysis and simulates the real-world usage patterns of mobile devices, which usually run for days and weeks without rebooting. The data shown in this paper were average values obtained by performing the experiment run ten times.

TABLE 2. List of applications used for evaluation.

Category	Application	Category	Application
Utility	Play Store	Multimedia	YouTube
	Google Calendar		Twitch
	Gmail		Amazon kindle
	Wikipedia		TED
	Google Chrome	Game	Angry Birds 2
SNS	Facebook		Candy Crush Saga
	Twitter		PUBG Mobile

B. OVERALL PERFORMANCE

Fig. 8 shows the launch time, the amount of read and written data under varying configurations. To assess the effectiveness of each feature in ezswap, we also created configurations whereby SA, CA, and APA, combined and separated, are applied to the original *zswap*. The label *ezswap* on the graphs represents cases where all these three features are enabled, and *swap* indicates the configuration without any compressed swap schemes. The amounts of total read and written data were 6.9 GB and 375 MB without any compressed swap schemes, respectively.

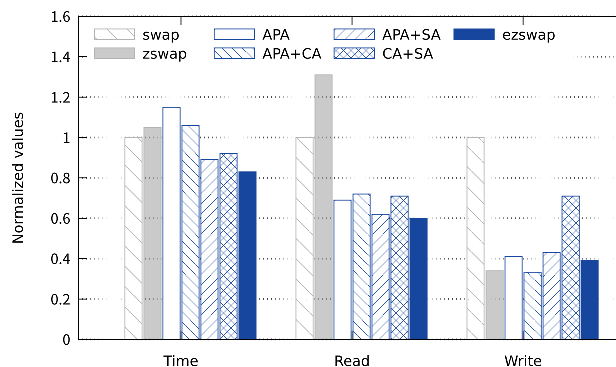


FIGURE 8. Launch time and number of read and written pages normalized to conventional swap.

As expected, the conventional zswap reduced the amount of written data by 66% through absorbing the write operations for swapping out anonymous pages. However, because 100 MB of the main memory was allocated for the compressed swap pool, the available space for the application was reduced and, in turn, more swap-out operations were triggered. This led to heavy loss of clean file-mapped pages from the main memory. Because of this, the evicted clean

file-mapped pages had to be brought back from the storage when the applications were launched again. As a result, the number of read pages increased by 31% and delayed the application launch time by 6% in comparison to the case without any compressed swap schemes.

When APA was applied, the amount of read data decreased by 31 and 47% in comparison to swap and zswap, respectively. On the contrary, APA increased the number of written pages by 19% in comparison to zswap but decreased it compared to the swap case. Therefore, APA alone slowed down the launch time by 9% because of its reckless storage of a large number of file-mapped pages.

APA+SA showed 11% less read operations than APA alone. This means that SA provides higher data density and hit rate through the proposed cost-benefit page selection model. For this, although the number of written pages was increased by 5% because the anonymous pages with high compression ratios were dropped, the combination of APA and SA improved the application launch time by 22 and 15% in comparison to APA and zswap, respectively.

Every zpage had the same eviction cost under CA+SA without APA, and thus the compression ratio of a zpage critically determines the admission decision. Therefore, lots of pages with high compression ratios were immediately abandoned and forwarded to the swap storage even when the zpool had plenty of free space. In the experiments, only 110,000 pages out of 340,000 swap-out pages were admitted to the compressed swap pool.

The number of written pages was slightly increased in the ezswap as well. However, due to the 55% reduction in read operations, it improved the application launch time by 22% compared to the zswap, the launch time of which was delayed.

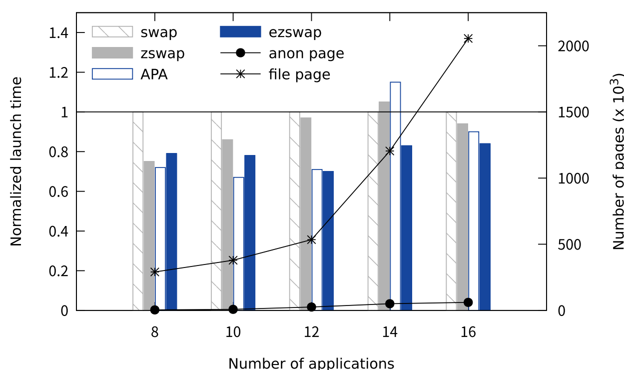


FIGURE 9. Normalized application launch time and number of file-mapped/anonymous pages evicted from the main memory while varying number of applications executed.

To evaluate the effectiveness of ezswap under varying memory pressure, we conducted the same experiments while varying the number of applications executed in a run, ranging from 8 to 16. Fig. 9 shows the observed application launch time and the number of swap-out pages categorized by their type. The applications used in these sets were sequentially selected from Table 2. The 16-application set additionally included Microsoft Excel (Utility) and 2048 (Game).

The benefit of the original zswap rapidly diminishing as the memory pressure increased. Under high memory pressure, zpages were evicted from the pool before they were reused because of its blind compression and cost-benefit-agnostic replacement policy. On the other hand, by surrendering valueless zpages, ezswap improved the performance by more than 16% in all cases.

The performance of APA alone was generally better than that of the original zswap. However, when applications with a large amount of file data were included in the experimental set, the performance diminished. The ezswap could eliminate most of APA's detrimental effects by preventing accommodation of fruitless zpages and keep the performance gain stable.

With 16 applications running, the APA and ezswap outperformed the zswap by 4 and 10%, respectively. This is because the number of pages that had to be loaded was excessively large relative to the total memory capacity. The conventional zswap had to reload all file-mapped pages, which were 34 times more than the anonymous pages, and thus many evictions occurred from the swap pool due to the indiscriminate admission. On the other hand, the APA saved 30% of read operations.

When the number of applications was 8 or 10, the number of pages evicted from the main memory was small so that the SA and CA schemes lost their effectiveness because there were fewer zpage evictions than the cases with a larger number of applications. The effectiveness of APA became larger in these circumstances because a notable portion of file-mapped pages could be accommodated in the pool and their impact to the anonymous pages was small. ezswap performed poorer than the zswap+APA configuration because ezswap's SA feature significantly slowed down the fill-up speed of the compression pool due to the low memory pressure.

ezswap performed best among all when 12 and 14 applications were used. The difference between APA and ezswap was especially substantial with 14 applications because the number of anonymous pages evicted from the main memory was 51,408, while it was 4,050 and 8,085 when 8 and 10 applications were running, respectively. Most of these evicted anonymous pages went directly to the swap storage under APA, because APA treats them in the same way as file-mapped pages although their eviction cost is a lot higher than that of file-mapped pages.

These results showed that the effectiveness of compressed swap schemes heavily depends on the memory pressure and the swap pool size. However, the performance of the ezswap remains stable under varying memory pressure and is significantly higher than that of conventional zswap especially under high memory pressure.

C. ANALYSIS OF PERFORMANCE GAIN

We measured the reduction in computation overhead obtained from the SA technique. Fig. 10 shows the number of actually compressed pages under APA and APA+SA, respectively.

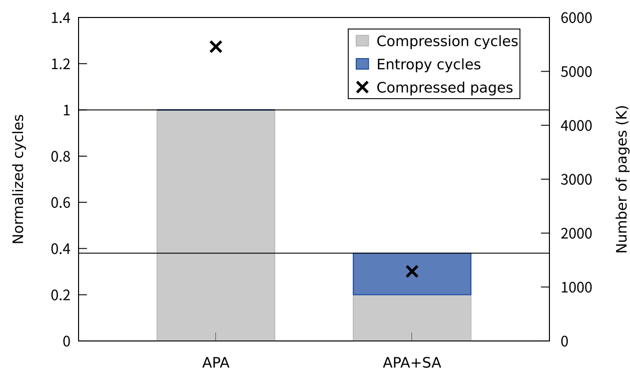


FIGURE 10. Number of actually-compressed pages and processor cycles consumed for compression.

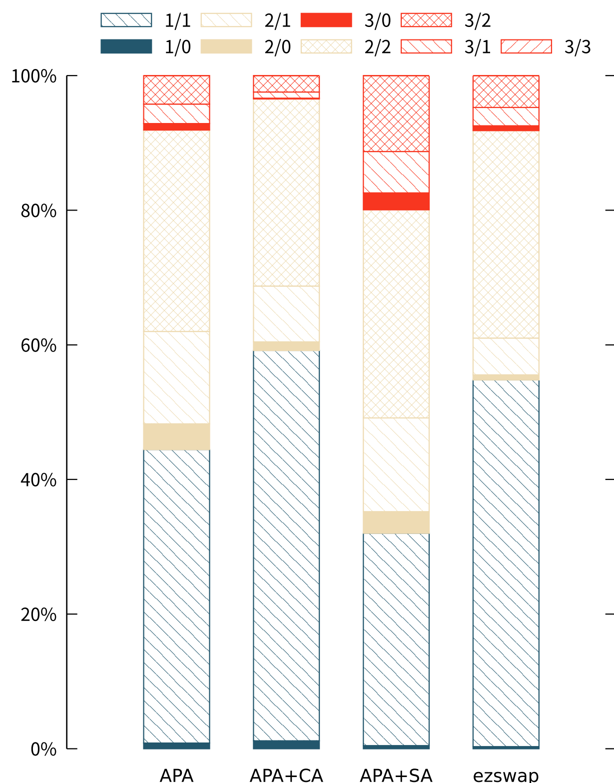


FIGURE 11. Constitution of swap pages evicted from zpool categorized by a combination of (total zpage / file-mapped zpage).

It also exhibits the normalized processor cycles consumed for the compressibility estimation and actual compression algorithm, respectively, under each configuration. The number of page compression instances was diminished by 76% with SA and thus, the processor cycles consumed for compression were accordingly reduced. Even when adding up the entropy calculation time to the whole, the SA reduced the number of processor cycles by 62%.

Fig. 11 shows the constitution of evicted swap pages categorized by the combination of stored total zpage count and stored clean file-mapped zpage count in a swap page under varying configurations. When the CA algorithm was not applied, the default LRU replacement policy was used instead.

With the CA policy, regardless of whether SA was applied, the proportion of evicted swap pages having only one file-mapped page increased and that of the other types of swap pages decreased. This is because ezswap prioritizes the eviction of file-mapped pages with high compression ratios. Accordingly, the data density of the zpool is expected to increase. In fact, as shown in Fig. 8, when CA was applied to APA and APA+SA, the number of write operations was decreased by 19 and 10%, respectively, and the launch time was shortened by 8 and 7%, respectively. The number of read operations under APA+CA was incremented only by 4% in comparison to APA. This demonstrates that the increased data density from CA considerably offsets the detrimental effects of prioritized eviction of file-mapped pages.

APA+SA and ezswap, both of which utilize the SA feature, commonly showed low 1/* portion and high 3/* proportion in comparison to their non-SA counterparts. These results are attributed to the fact that the number of 3/* swap pages in the zpool was increased while that of the 1/* swap pages was decreased by applying SA.

TABLE 3. Average number of zpages per megabyte of swap pool.

zswap	SA+CA	APA	APA+SA	APA+CA	ezswap
684	698	397	469	504	534

Table 3 shows the average number of zpages stored per one MB of zpool capacity measured after the execution of every application in the experiment. The average compression ratio of anonymous pages was significantly lower than that of file-mapped pages because a large portion of anonymous pages were zero pages, whose compression ratio is extremely low. On the contrary, as expected, the compression ratio of most file-mapped pages was poor because of their large portion of multimedia data. Because of these, APA showed less density than the original zswap.

In fact, ezswap without APA showed the highest density among all configurations. Both SA and CA significantly improved the data density, and thus ezswap could store 35% more pages than zswap with APA.

V. CONCLUSION

The compressed swap improves the execution time under high memory pressure by suppressing the number of pages to be swapped out to the secondary storage. In addition, it extends the life span of flash memory storage by reducing the number of write operations and thus, the wear of flash memory cells. Consequently, most commercial OSs are using compressed swap.

In this paper, we proposed ezswap, an enhanced compressed swap scheme for mobile devices. ezswap accommodates both anonymous pages and clean file-mapped pages. To increase the data density and thus, the hit rate, it selectively admits pages with beneficial eviction costs and compression ratios through entropy-based compression ratio estimation. In addition, it comprehensively considers multiple factors,

such as the compression ratio, hit rate, access recency, and page type, for replacement victim selection.

Through a series of evaluations with commercial applications on the Android OS, we verified that ezswap reduced the number of read operations by up to 55%, and increased the data density of the compressed swap pool by 35% in comparison to the original zswap of the Linux kernel. The computational overhead for compression was also repressed by 62% with entropy-based compression ratio estimation. Therefore, ezswap improved the application launch time by up to 22% while providing a reduction rate of write operations comparable to that of the conventional zswap. Consequently, we concluded that ezswap significantly improves the responsiveness of mobile devices while extending the life span of their storage.

As in most existing compressed swap schemes, the capacity of the compressed swap pool in ezswap is fixed to the predefined number. Therefore, it occupies the same amount of memory even when the memory pressure is low, and does not increase under heavy memory demand. We believe that the dynamic adjustment of the pool capacity will further improve the effectiveness of ezswap.

REFERENCES

[1] R. Duan, M. Bi, and C. Gniady, "Exploring memory energy optimizations in smartphones," in *Proc. Int. Green Comput. Conf. Workshops*, Jul. 2011, pp. 1–8.

[2] K. Zhong, D. Liu, L. Liang, X. Zhu, L. Long, Y. Wang, and E. H.-M. Sha, "Energy-efficient in-memory paging for smartphones," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 10, pp. 1577–1590, Oct. 2016.

[3] T. Coughlin. Getting solid at FMS. Forbes. Accessed: Aug. 16 2019. [Online]. Available: https://www.forbes.com/sites/tomcoughlin/2019/08/16/getting-solid-at-fms/

[4] T. Yang, H. Wu, and W. Sun, "GD-FTL: Improving the performance and lifetime of TLC SSD by downgrading worn-out blocks," in *Proc. IEEE 37th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Nov. 2018, pp. 1–8.

[5] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory systems," in *Proc. USENIX Annu. Tech. Conf.*, 1999, pp. 101–116.

[6] F. Douglis, "The compression cache: Using on-line compression to extend physical memory," in *Proc. USENIX Winter Tech. Conf.*, 1993, pp. 519–529.

[7] R. Cervera, T. Cortes, and Y. Becerra, "Improving application performance through swap compression," in *Proc. USENIX Annu. Tech. Conf.*, 1999, pp. 207–218.

[8] R. S. de Castro, A. P. D. Lago, and D. da Silva, "Adaptive compressed caching: Design and implementation," in *Proc. 15th Symp. Comput. Archit. High Perform. Comput.*, Nov. 2003, pp. 10–18.

[9] L. Yang, R. P. Dick, H. Lekatsas, and S. Chakradhar, "Online memory compression for embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 9, no. 3, Feb. 2010, Art. no. 27.

[10] I. C. Tuduce and T. Gross, "Adaptive main memory compression," in *Proc. USENIX Annu. Tech. Conf.*, 2005, pp. 237–250.

[11] B. Lee, S. M. Kim, E. Park, and D. Han, "MemScope: Analyzing memory duplication on Android systems," in *Proc. 6th Asia–Pacific Workshop Syst.*, 2015, Art. no. 19.

[12] N. Gupta, "zram: Compressed RAM based block devices," in *Proc. Linux Kernel User's Administrator's Guide*, 2014.

[13] S. Jennings. LWN.net Article. (Feb. 12, 2013). *The Zswap Compressed Swap Cache*. [Online]. Available: https://lwn.net/Articles/537422/

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

[15] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *Proc. 1st USENIX Conf. Oper. Syst. Design Implement.*, 1994, p. 1.

[16] U. Manber and S. Wu, "GLIMPSE: A tool to search through entire file systems," in *Proc. USENIX Winter Tech. Conf.*, Berkeley, CA, USA, 1994, pp. 23–32.

[17] A. Z. Broder, "Identifying and filtering near-duplicate documents," in *Proc. 11th Annu. Symp. Combinat. Pattern Matching*. Berlin, Germany: Springer-Verlag, 2000, pp. 1–10.

[18] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *Proc. 8th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2008, pp. 309–322.

[19] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Proc. Ottawa Linux Symp.*, 2009, pp. 19–28.

[20] C. A. Waldspurger, "Memory resource management in VMware ESX server," *ACM SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, 2002.

[21] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman, "An empirical study of memory sharing in virtual machines," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 273–284.

[22] S. Kim, H. Kim, J. Lee, and J. Jeong, "Group-based memory oversubscription for virtualized clouds," *J. Parallel Distrib. Comput.*, vol. 74, no. 4, pp. 2241–2256, 2014.

[23] S.-H. Kim, J. Jeong, and J. Lee, "Selective memory deduplication for cost efficiency in mobile smart devices," *IEEE Trans. Consum. Electron.*, vol. 60, no. 2, pp. 276–284, May 2014.

[24] "OS X mavericks core technologies overview," Apple, White Paper, Oct. 2013.

[25] G. Richard, III, and A. Case, "In lieu of swap: Analyzing compressed RAM in Mac OS X and Linux," *Digit. Invest.*, vol. 11, pp. S3–S12, Aug. 2014.

[26] V. Beltran, J. Torres, and E. Ayguadé, "Improving Web server performance through main memory compression," in *Proc. 14th IEEE Int. Conf. Parallel Distrib. Syst.*, Dec. 2008, pp. 303–310.

[27] J. Hwang, J. Jeong, H. Kim, J. Choi, and J. Lee, "Compressed memory swap for QoS of virtualized embedded systems," *IEEE Trans. Consum. Electron.*, vol. 58, no. 3, pp. 834–840, Aug. 2012.

[28] C. Lee, C. H. Hong, S. Yoo, and C. Yoo, "Compressed and shared swap to extend available memory in virtualized consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 60, no. 4, pp. 628–635, Nov. 2014.

[29] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland, "IBM memory expansion technology (MXT)," *IBM J. Res. Develop.*, vol. 45, no. 2, pp. 271–285, Mar. 2001.

[30] L. Benini, D. Bruni, A. Macii, and E. Macii, "Hardware-assisted data compression for energy minimization in systems with embedded processors," in *Proc. Conf. Design, Autom. Test Eur.*, 2002, p. 449.

[31] D. Chae, J. Kim, Y. Kim, J. Kim, K.-A. Chang, S.-B. Suh, and H. Lee, "CloudSwap: A cloud-assisted swap mechanism for mobile devices," in *Proc. 16th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGrid)*, May 2016, pp. 462–472.

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

[33] S. Desireddy and D. R. Pathireddy, "Optimize in-kernel swap memory by avoiding duplicate swap out pages," in *Proc. Int. Conf. Microelectron., Comput. Commun. (MicroCom)*, Jan. 2016, pp. 1–4.

[34] J. Choi, J. Ahn, J. Kim, S. Ryu, and H. Han, "In-memory file system with efficient swap support for mobile smart devices," *IEEE Trans. Consum. Electron.*, vol. 62, no. 3, pp. 275–282, Aug. 2016.

[35] S.-H. Kim, J. Jeong, and J.-S. Kim, "Application-aware swapping for mobile systems," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, Oct. 2017, Art. no. 182.

[36] J.-S. Lee, W.-K. Hong, and S.-D. Kim, "Design and evaluation of a selective compressed memory system," in *Proc. IEEE Int. Conf. Comput. Design*, Oct. 1999, pp. 184–191.

[37] S. Park, H. Lim, H. Chang, and W. Sung, "Compressed swapping for NAND flash memory based embedded systems," in *Proc. Int. Workshop Embedded Comput. Syst.*, 2005, pp. 314–323.

[38] C. Shin, J.-H. Hong, and A. K. Dey, "Understanding and prediction of mobile application usage for smart phones," in *Proc. ACM Conf. Ubiquitous Comput.*, 2012, pp. 173–182.

[39] Q. Xu, J. Erman, A. Gerber, Z. Mao, J. Pang, and S. Venkataraman, "Identifying diverse usage behaviors of smartphone apps," in *Proc. ACM SIGCOMM Conf. Internet Meas. Conf.*, 2011, pp. 329–344.

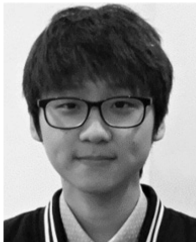
[40] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 57–70.

[41] G. Hansel, D. Perrin, and I. Simon, "Compression and entropy," in *Proc. Annu. Symp. Theor. Aspects Comput. Sci.*, 1992, pp. 513–528.

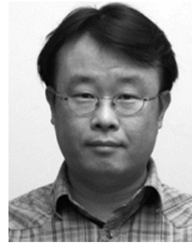
[42] S. Meguerdichian, H. Noshadi, F. Dabiri, and M. Potkonjak, "Semantic multimodal compression for wearable sensing systems," in *Proc. IEEE SENSORS*, Nov. 2010, pp. 1449–1453.

[43] C. Ji, L.-P. Chang, L. Shi, C. Gao, C. Wu, Y. Wang, and C. J. Xue, "Lightweight data compression for mobile flash storage," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, 2017, Art. no. 183.

[44] R. Love, *Linux Kernel Development*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2010, ch. 2, p. 20.



JONGSEOK KIM received the B.S. degree in computer science and engineering from Sungkyunkwan University, South Korea, in 2018, where he is currently pursuing the M.S. degree with the Department of Computer Science and Engineering. His research interests include memory management, non-volatile memory, flash SSDs, and file systems.



CHEOLGI KIM received the B.S. degree in computer science and the Ph.D. degree from the Korea Advance Institute of Science and Technology (KAIST), in 1996 and 2004, respectively. He is currently an Associate Professor with the Software and Computer Engineering Department, Korea Aerospace University, South Korea. He founded Ratio LLC., South Korea, an IoT device company, in 2017. His research interests include safety critical system software architecture, low-energy embedded systems, and real-time systems.

EUISEONG SEO received the B.S., M.S., and Ph.D. degrees in computer science from KAIST, in 2000, 2002, and 2007, respectively. He is currently an Associate Professor with the Department of Computer Science and Engineering, Sungkyunkwan University, South Korea. Before joining Sungkyunkwan University, in 2012, he was an Assistant Professor with the Ulsan National Institute of Science and Technology (UNIST), South Korea, from 2009 to 2012, and a Research Associate with Pennsylvania State University, from 2007 to 2009. His research interests include system software, embedded systems, and cloud computing.