Application-Aware Swapping for Mobile Systems

SANG-HOON KIM, Virginia Polytechnic Institute and State University
JINKYU JEONG and JIN-SOO KIM, Sungkyunkwan University, Republic of Korea

There has been a constant demand for memory in modern mobile systems to provide users with better experience. Swapping is one of the cost-effective software solutions to provide extra usable memory by reclaiming inactive pages and improving memory utilization. However, swapping has not been actively adopted to mobile systems since it incurs a significant amount of I/O, which in fact impairs system performance as well as user experience.

In this paper, we propose a novel scheme to properly harness the swapping to mobile systems. We identify that a vast amount of I/O for swapping comes from the conflict of the traditional page-level approach of the swapping and the process-level memory management scheme tailored to mobile systems. Moreover, we find out that the current victim page selection policy is not effective due to the process-level policy. To address these problems, we revise the victim selection policy to resolve the conflict and to selectively perform swapping according to the efficacy of swapping. Evaluation using a running prototype with realistic workloads indicates that the propose scheme effectively reduces the paging traffic, thereby improving user experience as well as energy consumption.

CCS Concepts: • **Software and its engineering → Allocation/deallocation strategies**; **Embedded software**;

Additional Key Words and Phrases: Memory management, mobile systems, swapping

ACM Reference format:
Sang-Hoon Kim, Jinkyu Jeong, and Jin-Soo Kim. 2017. Application-Aware Swapping for Mobile Systems. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 182 (September 2017), 19 pages.
https://doi.org/10.1145/3126509

1 INTRODUCTION

There has been a constant demand for more memory during the evolution of the mobile systems. The high memory demand in mobile systems is mainly due to the unique application (or *app* shortly) life cycle management policy, which is commonly found in many modern mobile systems [1, 5, 35]. These mobile systems attempt to cache apps in memory so that they can quickly respond to a user's request for accessing recently used apps, thereby improving overall user experience (UX). This *app caching policy* mandates a large amount of memory for caching enough apps.

This article was presented in the International Conference on Embedded Software 2017 and appears as part of the ESWEEK-TECS special issue.
This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 2016R1A2A1A05005494).
Authors' addresses: S.-H. Kim, The Bradley Department of Electrical and Computer Engineering, Virginia Polytechnic Institute and State University, 250 Perry St, Blacksburg, VA 24061, USA; J. Jeong and J.-S. Kim, College of Information and Communication Engineering, Sungkyunkwan University, 2066 Seobu-ro, Jangan-gu, Suwon 16419, South Korea.

https://doi.org/10.1145/3126509

Meanwhile, the memory footprint of mobile apps has been increased quickly to support the increased size and resolution of display. For instance, Motorola Milestone XT720 released in July 2010 is equipped with a 3.7-inch FWVGA (854 × 480 pixels) display [14] whereas Samsung GALAXY S7 released in March 2016 is equipped with a 5.1-inch WQHD (2560 × 1440 pixels) display [33]. Samsung GALAXY S7 has 9× more pixels on the screen, which roughly implies 9× memory footprint for frame buffers and textures.

However, satisfying the demand for memory is getting more difficult than ever. To keep up with the ever-increasing memory demand, device manufacturers have equipped devices with more and larger memory chips, doubling memory capacity in every one to two years [21]. In general, memory is made up with DRAM, which continuously dissipates energy to refresh memory cells even when it is not accessed. As a result, memory modules now consume approximately 5 to 30% of the total energy in modern mobile devices [8, 9, 12, 25]. The fraction can be further increased by installing more memory, which can be critical for mobile devices running with highly energy-constrained batteries. In addition, equipping more and larger memory modules increase the manufacturing cost of devices. It is known that memory components account for 5 to 30% of total manufacturing cost of modern mobile devices [24], and the cost will be increased to equip more memory on devices. This concerns device manufacturers since the market becomes extremely price sensitive and sales are shifted from premium to cheaper models these days [18], resulting in the industry trend towards low-end devices with system software supports [27].

In this sense, it is required to manage and utilize memory more efficiently, and swapping is considered as a cost-effective software solution for dealing with the requirement. Swapping reclaims inactive pages by writing them to secondary storage, improving the memory utilization and increasing the effective size of memory. Traditionally, swapping has not been considered in mobile system domains since traditional mobile systems could not afford the storage for the evicted inactive pages. However, contemporary mobile systems are usually equipped with NAND flash-based storage such as eMMC (embedded MultiMedia Card) and UFS (Unified Flash Storage), which provides a large capacity at a low latency. Also, compressed caching techniques such as ZRAM and Zswap [15, 19] allow swapping to be performed without secondary storage by compressing inactive pages. Thus, swapping becomes feasible in mobile systems, and manufacturers attempted to incorporate swapping into their commercial products to improving overall user experience by utilizing the additional memory provisioned by swapping to cache more apps [4, 31, 32].

Unfortunately, it is turned out that adopting swapping to mobile systems is not obvious due to the app life cycle management policy tailored to mobile system environments. Specifically, the app caching policy which reclaims memory at a process-level complicates the page-level approach of swapping. The app caching policy attempts to keep as many apps as possible, which incurs high memory pressure. At the same time, swapping attempts to lessen the memory pressure in order to keep around enough free memory. These contradictory goals of the memory management policies trigger a large amount of I/O, which can impair user experience and offset the benefit of swapping. Moreover, an app which is cached but not used for a long time has many cold inactive pages. These inactive pages are likely to be evicted to secondary storage by the victim page selection policy of swapping. The eviction of these pages is a waste of the I/O bandwidth and lifetime of the storage as the pages can be reclaimed by terminating the app without incurring the I/O and impairing user experience much. However, the current victim selection policy is oblivious of the process-level memory reclamation, thereby missing the opportunities to improve the efficiency of swapping. In addition, previous work only focuses on the process of the swap-out traffic instead of improving the traffic in the first place.

In this paper, we present *A2S*, which stands for "Application-Aware Swapping" for mobile systems. First, we propose a simple yet effective way to assess the efficacy of swapping so that the

system can avoid worthless effort for securing more memory via swapping when the efficacy is low. In addition, we attempted to make the swapping consider the life cycle of apps in mobile systems and cooperate with the process-level memory management policy. To this end, we revise the victim selection policy to avoid the pages that are likely to be reclaimed soon via the process termination.

We implemented the proposed scheme on a commercial Android-based smartphone, and evaluated it with the workloads obtained from real smartphone usage traces. The evaluation indicates the vanilla swap (i.e., just turning on swapping) is not effective in mobile systems since the heavy I/O for swapping offsets the benefits come from improving the memory utilization. Meanwhile, the proposed scheme reduces the amount of I/O by up to 39% for swap-in and by up to 59% for swap-out whereas it still improves the memory utilization as much as the vanilla swap does. As a result, we can improve overall user experience as well as energy consumption, which are of the utmost importance for mobile systems.

The remainder of the paper is organized as follows. Section 2 describes the background and related work in memory management for mobile devices. Section 3 describes the problems that motivated us to address them, and Section 4 explains our approach to resolve the problems. Section 5 describes the implementation of the proposed scheme on Android and evaluates the proposed scheme. Finally, Section 6 discusses the future work and concludes this paper.

2 BACKGROUND AND RELATED WORK

2.1 Memory Management in Page Granularity

Swapping is one of the traditional and fundamental memory management techniques commonly used by modern operating systems. Swapping provides the system with extra memory by migrating data in main memory to secondary storage. First, the operating system identifies the pages to evict from main memory based on its *victim selection policy*. Then, the operating system reclaims these so-called *victim pages*. The operating system locates spaces for the victim pages in the secondary storage called *swap device* or a file in the storage called *swap file*. The contents in the victim pages are copied to the spaces. The page table entries mapped to the victim pages are updated to indicate that these pages are available at the particular locations on the swap device. And then, the victim pages are unmapped from the page tables and converted to free pages. Later, the free pages can be allocated to processes or the operating system. This procedure is usually called *swap out*.

A swapped-out page is brought back to main memory when a process references to the page. Referencing to a swapped-out page raises a memory fault which is usually called *page fault*, and a routine so-called *page fault handler* is invoked to deal with the fault. The page fault handler resolves the situation by locating the corresponding page in the swap device, reading the page to a free page, and updating the corresponding page table entry to the newly recovered page. And then, the program execution is resumed at the location where the fault occurred. This procedure is called *swap in*.

It is crucial to identify proper victim pages during swapping, otherwise the system will suffer from frequent swapping. In general, the victim page selection for swapping is similar to the victim selection for a cache if we consider main memory as a cache of pages stored in the secondary storage, and there have been a vast amount of studies for the purpose [7, 22, 23, 26, 28]. However, the victim page selection in the operating system is not identical to that of cache. It is infeasible for operating systems to track every memory access whereas many cache-domain work assumes each access can be examined. Instead, operating systems can track page accesses in a limited manner by checking the reference bit in the page table asynchronously or through the costly page fault

handler. Thus, many of the studies in operating systems focus on identifying *inactive pages* which are not included in the current working set of memory relying on the limited features [20]. In practice, the Linux kernel employs a simplified version of LRU 2Q [20] for identifying victim pages; it maintains two lists—active list and inactive list—that are supposed to link active or inactive pages. The kernel scans these lists and migrates the pages among the lists according to the references to the pages. And if necessary, the kernel reclaims inactive pages by swapping them out while scanning the inactive list. The scanning and reclamation is triggered when the number of free pages is lower than a threshold or when a high-order page allocation is failed, each of which are called as *background reclamation* and *direct reclamation*, respectively.

2.2 Memory Management in Mobile Systems

Android is an open-source software platform designed for mobile devices [1]. Android is comprised of the kernel and the Android framework. The kernel is a customized version of the Linux kernel implementing Android-specific functionalities such as Binder, memory management modules, etc. Its primary role is to manage global resources such as CPU and memory, and controls the accesses to hardware devices. The Android framework runs on top of the kernel, and provides Android apps with a runtime environment. The framework also provides apps with system-level libraries such SQLite, WebKit, SSL, and so forth.

Android employs a unique life cycle for apps [3, 16], which is tailored to the user behaviors to mobile devices. It is known that users tend to install tens to hundreds of apps on their devices and to interact with many apps briefly one at a time [11, 30, 34]. It is also known that there is locality among the apps, making some apps be used frequently whereas the others do not [11, 38]. Android makes use of these characteristics; if an app is not terminated but paused in memory after a user's access to the app, the subsequent access to the app can be served fast by resuming the app rather than starting the app from scratch. We refer to the former type of the app access as *app resume* where as the latter one as *app start*. Generally, app resume requires less time and system resources to re-access the app, thereby providing users with better experience and system performance. Thus, the framework attempts to keep around a certain number of apps in memory (e.g., 8 apps in Kitkat and 16 in Lollipop) as long as the amount of free memory is sufficient.

When the system runs out of free memory and the free memory gets below a threshold, Android reclaims memory at a process granularity. A memory management module called *Low Memory Killer (LMK)* picks up a victim app among the cached apps and reclaims the memory belonging to the app by terminating the app. The reclamation can occur whenever the system is low on free memory, and app developers are advised to maintain the app status and information required to restore the app in persistent secondary storage [3].

The victim app for the process-level memory reclamation is selected based on the importance of apps. While providing the runtime environment to apps, the framework classifies apps into several categories according to their importance. In particular, system processes such as telephony and system server are considered very important and they are not terminated for the memory reclamation. Other apps are classified into classes according to their status from the user's perspective and their objective. Among those apps, regular apps cached in the background are further refined according to the recency of the access to the app. Based on the classification, LMK kills the least important app. In practice, the victim app is usually the one that is the least recently used.

3 MOTIVATION

Considering the current market trends towards cheaper devices [18, 27], it makes sense to consider swapping in mobile systems to increase the effective size of memory without increasing the manufacturing cost. Incorporating swapping in working mobile systems, however, introduces new

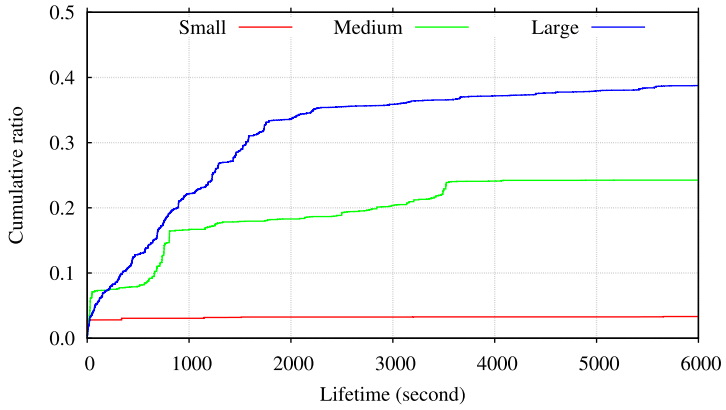


Fig. 1. Lifetime of swap entries on various workloads.

challenges, which is not obvious to deal with. As a pilot evaluation, we set up a Google Galaxy Nexus smartphone to log system performance metrics. We turned on swapping in the stock Android firmware and used the smartphone as usual. From the evaluation, we observed approximately 1 GB of swap-out traffic over 3 days' use, and the traffic accounts for approximately 80% of the entire write traffic of the system. Such a large *quantity* of swapping traffic might mask the benefit of swapping, and even result in a performance degradation in terms of user experience, energy efficiency, and the reliability of the NAND flash-based storage (see Section 5 for the details of the adverse effects of the massive I/O traffic for swapping).

An in-depth analysis revealed that the problematic I/O traffic for swapping is caused by the conflict of the goals of the process-level (i.e., LMK) and page-level (i.e., swapping) memory reclamation schemes. Due to the app caching policy, Android wants to keep as many apps in memory as possible. This policy fills up the memory with cached apps and keeps generating memory pressure to the memory subsystem. At the same time, the memory subsystem attempts to keep around enough free memory in response to the memory pressure. To this end, the memory subsystem identifies inactive pages and reclaims them via swapping. The reclaimed pages are, however, utilized for caching more apps, and the memory pressure is increased again. This situation is similar to the thrashing where the system keeps swapping to deal with memory references whose working set is larger than the installed memory. At a glance, it seems to be sufficient to adjusting the thresholds for the memory reclamation policy. However, such a naive approach can lead the system to an out-of-memory situation or memory under-utilization, both of which are not desirable.

In addition to the heavy swapping traffic, we found that the pages comprising the I/O are not effectively selected. We can define the lifetime of a swap entry as the period during which the entry is valid; a swap entry becomes valid when a page is written to the swap entry, and it gets invalid when all processes owning the page are terminated. Writing pages to swap entries spends system resources, drains battery, incurs overheads, and wears out the flash-based secondary storage. In this sense, the lifetime of swap entries implies the *efficacy* of the swap traffic given that the swap entry having long lifetime provides the extra memory for a long time and the overhead associated with the entry is amortized over the lifetime. Figure 1 shows the cumulative distribution of the lifetime of swap entries on three workloads, whose name implies the memory working set size of the workload. A point at (x, y) means that y% of swap entries become invalid within x seconds since they are written to the swap device. See Section 5.4 for more detail of the analysis. As seen in Figure 1, a large portion of the swap entries are short-lived. Specifically, 16.7% and 22.2% of entries

last less than 1,000 seconds on the Medium and Large workload, respectively, implying that the swapped pages are not effectively selected.

We found that the poor *quality* of the swapping traffic is caused by the conflict between the process-level and the page-level reclamation policies as well. The victim selection policy for swapping tries to identify inactive pages which are not referenced for a while. A cold background app, which are not accessed by a user for a long time, is paused in the background without incurring memory accesses. Thus, many pages owned by the app are inactive, and they are likely to be swapped out to the swap device. Meanwhile, such app is likely to be selected as the victim process by LMK since the app is not accessed for a long time. Thus, LMK picks up the cold background app as the victim of the process-level memory reclamation. As a result, in many cases, the writes of inactive pages to swap devices are shortly followed by the termination of the owner app of the pages, which only makes the effort to swap out vain.

From these observations, we conclude that if the process-level and page-level policies were aware of each other, they can make a better decision. This gap is originated from the separated integration of the memory management policies into the mobile system without reciprocal consideration. In particular, the process-level management scheme drastically changes the way of using memory, and it necessitates to revise the existing memory management schemes accordingly. However, to the authors' best knowledge, they have not scrutinized at all, and none of previous work questions the quality of swap traffic in the first place but only processes the swap-out traffic as it is. The following section describes our approach to revise the existing schemes to close the gap and solve the quality problem fundamentally.

4 IMPROVING SWAPPING FOR MOBILE SYSTEMS

Our approach is comprised of two parts; to mitigate the amount of I/O and to improve the quality of the I/O traffic. To this end, we first improve the memory management policy to selectively apply swapping considering the efficacy of swapping at a given moment. And then we move to improve the victim page selection so that the page-level swapping does not conflict with the process-level memory reclamation policy but select victim pages considering the contexts inherent in mobile systems.

4.1 Efficacy-Based Selective Swapping (ESS)

As we explained in Section 2.1, modern operating systems attempt to keep around a number of free pages by reclaiming inactive pages when the number of free pages gets below a threshold. However, this approach might not be the most effective way to reclaim memory in mobile systems due to the app caching policy.

Let us assume a mobile system that caches apps in memory. Due to the locality among app uses [11, 34, 38], the cached apps have different probabilities of being used again. In general, recently used apps are more likely to be used again than the other ones that are not recently used. This is analogous to the LRU cache in the memory hierarchy if we consider the main memory as the cache of installed apps, and the use of app as the memory access. Thus, we can estimate the probability of an app use by using an LRU list of apps and measuring the distance of the app from the MRU (Most Recently Used) position in the list. If the main memory is sufficiently large, the system can cache many apps. In this case, the distance from the MRU position is large for the least recently used app, and the (expected) probability of using the app becomes small. Thus, killing the app would not be noticeable by users. In other words, if a system caches sufficiently many apps, the system can reclaim many pages without impairing user experience much by terminating the app rather than keeping the app and swapping out its pages. On the other hand, if the system caches only a few apps, even the least recently used app has a short distance from the MRU position and

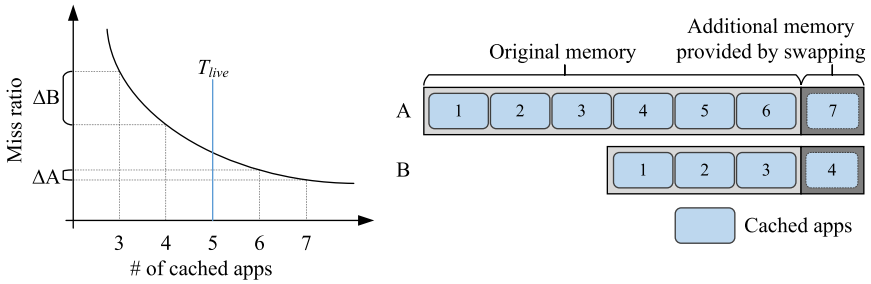


Fig. 2. Changes of the miss ratio on different memory sizes.

the probability of using the app again is not negligible. Thus, if the system terminates the least recently used app, it is likely for users to notice the termination, thereby impairing user experience. In this case, reclaiming memory via swapping is beneficial so that the system can cache more apps using the reclaimed memory.

This argument is similar to the discussion presented in [40]. We can consider the miss ratio curve (MRC) of app use with respect to the number of cached apps by assuming the app resume as the cache hit and the app start as the cache miss. Figure 2 illustrates a typical miss ratio curve with respect to the number of cached apps and two memory situations. Let us consider a system that caches 6 apps as the case A in Figure 2 shows. Then, even if one additional app (i.e., 7th app) can be cached with the additional memory provided by swapping, the miss ratio is not decreased much as shown as ΔA. In this case, the overhead for swapping might offset the benefit of the decreased miss ratio. On the contrary, if the number of cached apps is small as illustrated as the case B, the miss ratio will be decreased considerably so that the benefit of the decreased miss ratio outbalances the overhead for swapping.

From the observation, we propose the *Efficacy-based Selective Swapping (ESS)* policy. The key idea is to select the better memory reclamation scheme based on the current efficacy of the swapping. Considering the current app cache policy based on LRU, we find the number of cached apps implies the efficacy of swapping. Specifically, when the number of cached apps N exceeds a threshold T_{live}, it indicates that the efficacy of swapping is low and the process-level memory reclamation is more suitable than performing swapping. In this case, ESS disallows the memory reclamation via swapping. Otherwise, ESS allows swapping so that the system can benefit from swapping.

T_{live} is a tunable parameter that determines the preference for swapping or terminating processes. The larger the value of T_{live} is, the more the system prefers to perform swapping instead of terminating apps. It is similar to the `swappiness` parameter in the Linux kernel, which is a tunable parameter that controls the tendency for reclaiming anonymous pages or page cache pages. Thus, T_{live} has to be set carefully considering the memory configurations and the acceptable amount of I/O for swapping.

4.2 Lifetime-Aware Victim Page Selection (LVSP)

As discussed in Section 3, many swap entries are shortly lived due to the conflict of the process-level and the page-level policies. Specifically, the victim page selection policy for swapping works at the page-level and focuses on finding out system-wide inactive pages. At the same time, LMK works at the process-level and terminates cold cached apps in the background. This leads to many inactive pages discarded shortly after being written to the swap device.

Our approach to resolve the conflict is making the victim page selection policy take into account the future lifespan of pages so that the pages that are likely to be freed soon by the process

termination are not selected as victim pages. The key challenge of this approach is to predict the lifespan of a page. In general, it is not obvious to predict the moment when a page will be freed since pages are frequently allocated and deallocated in the course of running programs. However, we argue that the lifetime of inactive pages can be predicted by referring to the lifetime of the process that owns the pages. Inactive pages are not referenced nor freed for a long time, and are released when the owner process dies. Thus, it is reasonable to assume that the lifetime of inactive pages coincides with the lifetime of their owner process.

However, predicting the lifetime of an app is not obvious either. In a naive approach, tracking apps with an LRU list seems to be reasonable and sufficient since the most recently used app will have the longest remaining lifetime. This approach, however, has a critical shortcoming. LRU cannot distinguish frequently used apps and rarely used apps when an app comes to the MRU position. LRU considers an app to outlive other apps when the app comes to the MRU position, even though the app is not accessed again. This is originated from the shortcoming of LRU that tells an app is unlikely to be used again only when the app gets close to the LRU position.

For this reason, we need to proactively predict the future of an app. We utilize the reuse distance of an app, which indicates the number of other app launches between two consecutive launches of the app. Let us assume an app launch as the independent event of selecting the app among the installed apps having different probabilities of preference. Then, the selection satisfies the requirement for the Poisson process [17], and the set of reuse distances of an app follows the exponential distribution. Specifically, consider an app a_i and its average reuse distance r_i. As the exponential distribution is memoryless, we can predict, $P(a_i, x)$, the probability that the app a_i is used again within next x app launches as follows independent of the recency of use;

$$P(a_i, x) = 1 - e^{-x/r_i} \tag{1}$$

For example, when the average reuse distance is 4.33 for an app, the probability of using the app within next 5 launches is 90%. If the average reuse distance is small for an app, the app will be frequently accessed again and the chance to be terminated by LMK is small. Thus, we can expect the app has a long lifetime. Contrarily, the app having a large average reuse distance has a short lifetime as the app will be rarely accessed again and eventually killed by LMK. In this way, we can use the average reuse distance of an app as the metric to estimate the remaining lifetime of the app regardless of the recency of the app usage.

From the observation, we present *LVSP*, which stands for the Lifetime-aware Victim Selection Policy. When a page is considered inactive by the original victim page selection policy (i.e., has not referenced for a long time), LVSP additionally estimates the lifetime of the page. Currently, we estimate the lifetime from the reuse distance of app. Specifically, if the average reuse distance is smaller than the threshold T_{reuse}, LVSP considers the page will live long and allows to swap out the page. Otherwise, LVSP disallows swapping the page since the page will be reclaimed soon via the process termination.

In many mobile systems, apps are isolated by processes, and a process hosts an app [1, 35]. A page can be shared by multiple processes and released to the system when all owner processes are terminated. Among these owner processes, the process hosting the app having the smallest average reuse distance will outlive other processes. In this regard, LVSP estimates the lifetime of a page using the smallest average reuse distance of the apps owning the page.

We found that updating the reuse distance of an app only when the app is launched is not effective if the app is not used for a long time after a burst of launches. To deal with such a case, LVSP predicts the current average reuse distance by assuming the app is launched at this moment. For example, assume that an app is launched 3 times and their average reuse distance is 4. If there have been 10 launches since the last launch of the app, the current average reuse distance of the

ALGORITHM 1: Determine whether the system allows to swap out an inactive page or not

Input: p: An inactive page chosen by the original victim selection policy
Data: N_{live}: The number of live apps
Data: T_{live}: The threshold for ESS
Data: T_{reuse}: The threshold for LVSP
Result: true if eligible for swapping. false otherwise

```
if N_live ≥ T_live then
    return false;                          /* ESS: Do not swap out if enough apps are cached */
end
foreach app a in the owner apps of p do
    if the number of launch of a < T_launched then
        continue;
    end
    D = calculate the current average reuse distance of a;
    if D < T_reuse then
        return true;                                  /* LVSP: Eligible for swapping */
    end
end
return false;
```

app is predicted as $(3 \times 4 + 10)/(3 + 1) = 5.5$, not 4. Also, the confidence of the average of reuse distances is low if the number of samples (i.e., the number of app launches) is too small. To avoid this, LVSP qualifies the average reuse distance of an app when the app is launched more than $T_{launched}$ times. Otherwise, the average reuse distance of the app is considered to be infinite.

Besides typical interactive apps that are launched and accessed by users, there exist service apps and system processes such as Zygote, Media Server, System Server, etc. We cannot estimate their reuse distance as they are not explicitly launched by users. However, they have higher priorities than the normal apps and are likely to outlive normal apps. In this sense, the average reuse distance of these apps are considered to be 0 so that their inactive pages are always allowed for swapping. For the same reason, the apps whose reuse distance is not assigned by the framework is also considered to have 0 as their average reuse distance.

We particularly used the reuse distance in this paper since it is simple yet effective to predict the lifetime and easy to explain. More sophisticated models [10, 29, 38, 39] can replace the current model to improve the accuracy of the prediction. In this case, the effectiveness of LVPS would be improved as well.

Algorithm 1 outlines the proposed scheme to determine whether the system allows to swap out an inactive page or not. Note that the decision is made for the pages that are determined as inactive by the original victim selection policy.

5 EVALUATION

5.1 Methodology

We developed a prototype of the proposed scheme on a Google Nexus 5 smartphone, which is based on a Qualcomm MSM8974 Snapdragon 800 system-on-a-chip (SoC). The SoC includes a quad-core 2.26 GHz ARM CPU and 2 GB of RAM. We implemented the proposed scheme on the Android framework 5.1.1-r9 Lollipop with the Linux kernel 3.4.0 [13]. When an app is launched or terminated, the event is hooked to a framework module that calculates the number of live apps

Table 1. Workloads Used for Evaluation

Workload	User	# of apps	Avg. reuse distance	Apps
Small	B10	12	2.23	Hangout, Gmail, Phone, Facebook, Gallery, Setting, Evernote, Browser, Youtube, Doodle Jump, Feedly, Subway Surfer
Medium	A04	23	4.56	Hangout, Timely, Phone, Gmail, Wunderlist, Calendar, Estrong Explorer, Play Store, Weather, Browser, Youtube, Google Map, Setting, Evernote, Google Now, Calculator, Kindle, Camera, Angry Birds, Facebook, Candy Crush Saga, IMDB
Large	B07	36	6.10	Hangout, Phone, Calendar, Wunderlist, Gmail, Youtube, Timely, Candy Crush Saga, Facebook, Calculator, Google Map, Browser, Play Store, Weather, Google Now, Angry Birds, Camera, Setting, Kindle, Gallery, Fruit Ninja, Game 2048, Timesheet, Adobe Reader, eBay, JuiceSSH, Subway Surfer, IMDB, EStrong Explorer, Google Keep, Doodle Jump, Amazon, Doodle God, Aliexpress, Word Search, Barcode Scanner

and the average reuse distance of apps. This information is delivered to the kernel via `procfs` and is utilized in determining victim pages as we described in Algorithm 1.

As the proposed scheme utilizes the user's app usage behaviors, it is critical to evaluate on a realistic workload. We evaluated the proposed scheme with the traces provided by the Livelab research [30, 34]. Due to the difference of the software platforms between our prototype (Android) and the Livelab research (iPhone), we cannot directly apply the traces to our evaluation. Instead, we converted the original traces to equivalent traces as follows.

First, we removed the launches of SpringBoard, which is the home launcher app for iOS, and merged consecutive app launches to one app launch. And then, we extracted a sequence of 256 app launches at the middle of each user's trace and counted the number of unique apps appeared in the sequence. We sampled the sequence at the middle of trace to avoid any bias. The count varies from 12 to 36 apps. We can assume that the count for a trace implies the memory working set size of the trace since if a trace includes more apps, the system should cache more apps to provide a certain number of app resumes from a given number of app launches. We picked up three traces from the user B10, A04, and B07, each of which represents the workload requiring small, medium, and large memory working set size, respectively. And then, we mapped each iOS app appeared in the workloads to an equivalent Android app. We picked the app from the Google Play Store if the same app exists. Otherwise, we picked a popular app from the category that is equivalent to the original category in the Apple App Store. Table 1 summarizes these apps and the usage pattern of the workloads. The apps are listed in the decreasing order of the number of launches in the workload. It is noteworthy that the average reuse distance is increased along with the working set size. The workloads cover 85 to 261 hours' run time so that they include the diurnal usage pattern which is one of the most frequently observed and important app usage patterns discovered by many independent research [11, 37].

The workloads are replayed using a benchmark that launches apps listed in a workload file. To equalize the running environment across runs, the benchmark first reboots the device, and drops the pages in the page cache upon the completion of the reboot. And then, the benchmark idles for

three minutes so that the background jobs triggered by the system reboot are subsided. After the idle time, the benchmark launches the first app in the workload, waits for 20 seconds so that the app is completely launched, and launches the home launcher, which is followed by 3 seconds idle time. The app launch is done by injecting the corresponding launcher intent of the app to the system using the command-line activity management tool called `am`. And then, the benchmark launches the next app in the workload. This procedure is repeated for each app listed in the workload. The benchmark also collects the traces required for the evaluation while running the workload, such as the kernel messages, the framework logs, memory usage, CPU time, etc.

The device is connected to a host via USB and the host controls the device with ADB (Android Debug Bridge) tool [2]. Also, the device is connected to the Internet over 802.11ac WiFi connection. In order to clarify the effect of swap, we reduced the system memory size to 1.5 GB by setting the booting parameter. To enable swapping, we created a 1 GB swap file in the system partition. Other parameters are set to default values unless we mention.

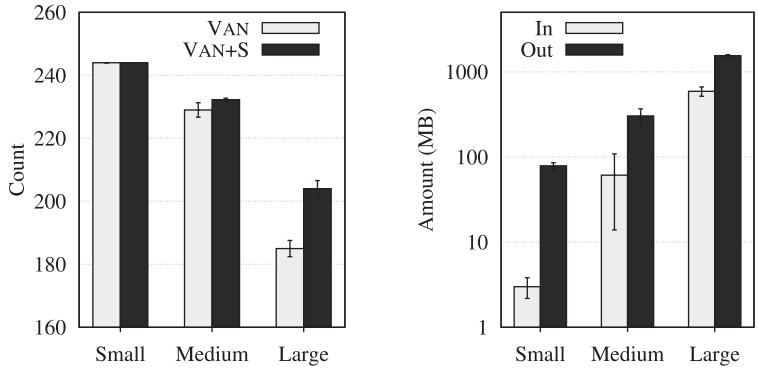
It is noteworthy that we do not present the performance comparison to the compression-based approaches [6, 15, 19] since our approach is orthogonal to them. Basically, these compression-based approaches reduce the memory footprint by compressing rarely used pages. This implies they should identify these rarely used pages first, and practically they piggy-back on the victim page selection of the swap mechanism. Our scheme works on that point and improves the victim page selection process. From this standpoint, our scheme and the compression-based approaches are orthogonal, and we focus on the primary contribution of our approach (i.e., improving the victim page identification) in this paper. Of course, we can easily combine these approaches together, and it will improve the system efficiency further by eliminating unnecessary compression and decompression.

5.2 Effect of Swapping in a Mobile System

First, we evaluate the effect of swapping in the mobile system. We collected system performance metrics while running the benchmark. As the baseline, we used the firmware built with the default configurations. We will refer to this vanilla configuration as Van. The kernel built with the default configuration is not configured for swapping. We rebuilt the firmware to perform swapping by configuring the kernel option and the booting script. We will refer to this configuration as Van+S. Note that the victim selection policy of Van and Van+S is same.

To evaluate the influence of swapping on the memory utilization, we analyzed the number of app launches served by resuming apps (i.e., app resume) while running the benchmark. The higher app resume count is, the better the system utilizes the given amount of memory so that more apps can be cached. Figure 3(a) summarizes the number of app resumes among 256 app launches. Remaining launches are served by starting the apps from scratch. We present the average of four runs with the bar and their standard deviation with the error bar at the end of the bar. As can be seen in Figure 3, the number of app resumes for the Small workload are same for Van and Van+S since the sum of the memory footprints of the apps in the workload are so small that all apps can be cached in the memory without swapping. In this case, all app launches are served by resuming the app except for the cases when the app is launched for the first time. Thus, improving memory utilization is not effective in the Small workload. As the working set size grows, more launches are served by resuming apps in Van+S. Specifically, the app resume count is increased by 3.25 and by 19.0 for the Medium and Large workload, respectively. This implies that swapping is helpful in improving the memory utilization for the system with limited memory.

To evaluate the cost of swapping, we measured the amount of I/O traffic incurred by swapping, and Figure 3(b) summarizes the result. The bars represent the averages of four runs, and the error bars the standard deviations of the runs. Note that the y-axis is in a log-scale. We can observe a



(a) App resume count (b) The amount of I/O for swapping

Fig. 3. Baseline performance on various workloads.

considerable amount of I/O traffic is generated even in the Small workload where swapping is not effective at all. As shown in Figure 3(a), swapping is not effective for the Small workload. We note that approximately 3.0 MB of swap-in and 78.8 MB of swap-out traffic are incurred for the Small workload. These I/Os are just wasting the I/O bandwidth since the system already has enough memory. The I/O traffic increase along with the working set size of the workloads, and it becomes 591.5 MB for swap-in and 1,549.0 MB for swap-out on the Large workload.

From Figure 3, we can confirm that adopting swapping to mobile systems is not obvious, and merely turning on the swapping may only incur overhead without improving the system performance much. Also, even though the improved memory utilization can help the system to cache more apps, each app launch can be slowed down by the enormous swapping traffic. We will further analyze the actual influence of swapping on the user experience in Section 5.4. To the rest of the evaluation, we present the results obtained from the Large workload since the effectiveness of swapping can be shown most clearly in the workload.

5.3 Parameters for ESS and LVPS

We evaluate the influence of the threshold T_{live} and T_{reuse}, for ESS and LVPS respectively.

We first evaluate the effect of T_{live} for ESS. We measured the number of app resume counts while varying the value for T_{live}. Meanwhile, T_{reuse} is set to `INT_MAX` so that LVSP evicts inactive pages regardless of the average reuse distance of apps. Figure 4 shows the app resume counts normalized to that of VAN. Each bar represents the average of four runs and the error bar at the end of the bar denotes the standard deviation of the runs. When the value for T_{live} is small, the app resume count is not as much increased as VAN+S since ESS allows swapping only for a limited period, and has less chances to get improvement from swapping. However, in spite of the limited swapping, ESS increases the app resume count by 7.4% compared to VAN. As T_{live} is increased, ESS allows more swapping and further increases the number of app resumes. The change in the app resume counts is slowed down when T_{live} is equal to or greater than 18, and eventually, the app resume count becomes comparable to that of VAN+S.

Figure 5 summarizes the amount of I/Os for swapping on various T_{live} values. The trend is similar to that of the app resume counts. The traffic is small when T_{live} is small. The traffic increases quickly along with the value of T_{live}, and the increase stops when T_{live} becomes 22. This trend comes from the number of apps the system can hold. An analysis identified the system holds up to 22 live apps, which are comprised of 1 foreground app, 1 previous foreground app, 1 home launcher

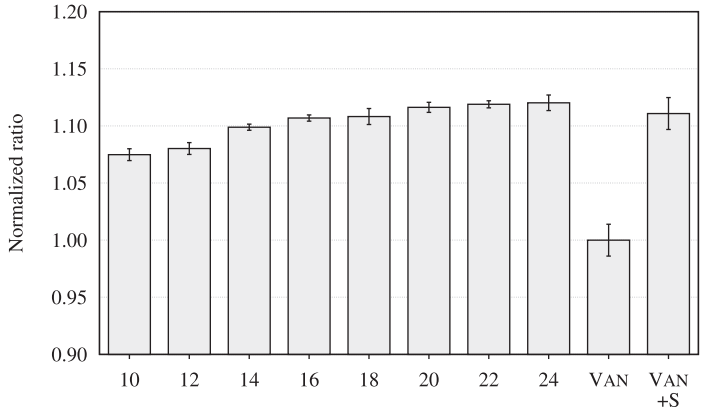
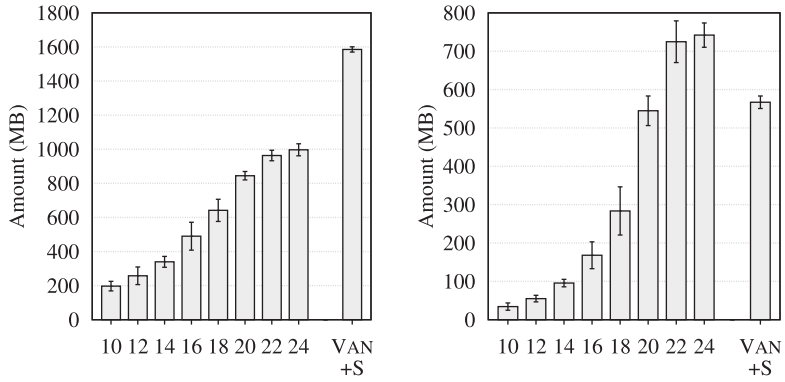


Fig. 4. Normalized app resume counts on various T_{live}.



(a) Swap out (b) Swap in

Fig. 5. Amount of I/Os for swapping on various T_{live}.

app, 16 normal background apps, and 3 background apps with service. Thus, the value for T_{live} greater than 22 effectively disable ESS.

The app resume count represents the benefit of swapping whereas the I/O traffic implies the cost of swapping. The app resume count is not improved much when T_{live} is equal to or greater than 18 whereas the I/O traffic increases further beyond the value. Thus, it is reasonable to use 18 for T_{live} with which ESS performs cost-effectively.

We evaluate T_{reuse} for LVPS similar to T_{live} for ESS. While evaluating T_{reuse}, we set T_{live} to `INT_MAX` so that ESS is effectively disabled. Figure 6 summarizes the number of app resumes normalized to that of VAN. The overall trend is similar to that of T_{live} for ESS. For small values of T_{reuse}, the app resume count is not much increased since only a few frequently used apps and system apps are eligible for swapping. As T_{reuse} is increased, more apps are projected to live long, and more pages from the apps are reclaimed via swapping. This provides the system with more extra memory, which can be used to cache more apps, and increases the app resume count. The number of app resumes becomes comparable to that of VAN+S when T_{reuse} is 64, and does not increase much beyond the point.

Figure 7 shows the amount of I/Os for swapping on various T_{reuse} values. The result is also similar to that of T_{live}. The I/O traffic is small when T_{reuse} is small, and is increased along with the value of T_{reuse}. The increase is slowed down when T_{reuse} is equal to or greater than 64. Considering

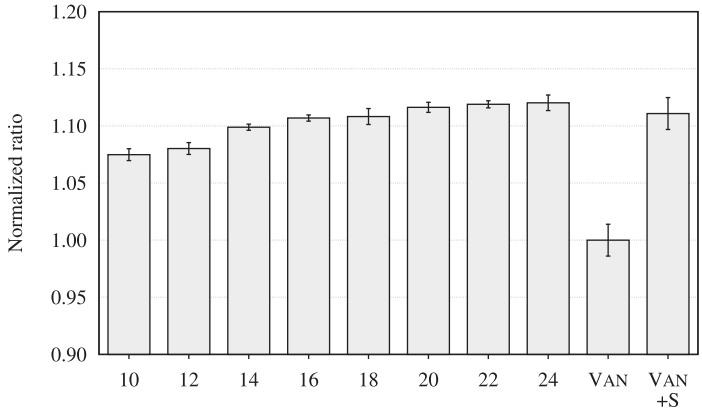


Fig. 6. Normalized app resume count on various T_{reuse}.

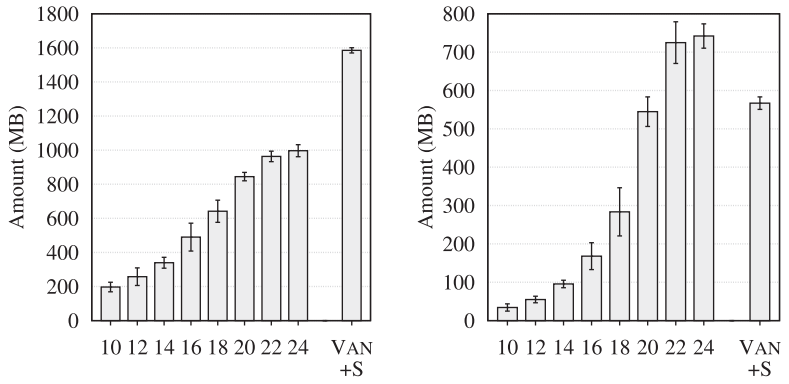


Fig. 7. Amount of I/Os for swapping on various T_{reuse}.

the trend in the app resume count, it makes sense to use 32 or 64 as the value for T_{reuse} on this system.

We can observe that the swap-out traffic is not increased as much as that of VAN+S whereas the swap-in traffic exceeds that. This trend is also observable from Figure 5. We found that the reduced swap-out traffic is caused by $T_{launched}$, which is the minimum number of app launches required for an app to be considered to live long, as described in Section 4.2. In the Large workload, 21 out of 36 apps are launched less than three times. After the launches, these apps are shifted to the background and paused for a long time without incurring any activity. Thus, many inactive pages belonging to these apps are considered for swapping on VAN+S, producing a large amount of swapping traffic. On the other hand, LVSP does not allow these inactive pages to be swapped out since the owner apps are not eligible for swapping. Instead, these inactive pages are reclaimed via the process termination without incurring the swap traffic.

The increased swap-in traffic is caused by the increased lifetime of swap entries. As we will discuss in the following section, many pages are invalidated soon after being evicted to the swap device on the original policy. Thus, the swap entries are unlikely to be referenced during the short lifetime. Contrarily, LVSP selectively swaps out the pages that are likely to be valid for a long time.

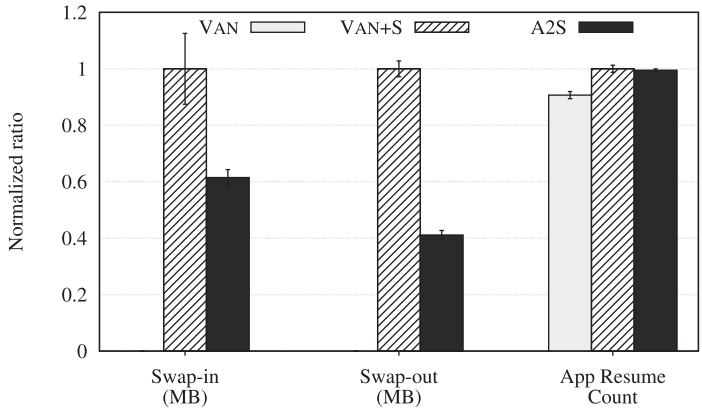


Fig. 8. Comparison of fundamental performance metrics.

Therefore, the swap entries have a larger chance to be swapped-in than the original policy, thereby increasing the swap-in traffic.

5.4 Performance of the Proposed Scheme

Now we evaluate the effectiveness of the proposed scheme using the parameter values found in the previous subsections. From the previous analysis, we concluded that using $T_{live} = 18$ and $T_{reuse} = 64$ is cost-effective for ESS and LVSP, respectively. Henceforth, we will refer to the combined configuration of ESS and LVSP using these parameter values as A2S, which is short for "Application-aware Swapping".

We collected various performance metrics while running the benchmark with the Large workload. Figure 8 compares the several important performance metrics with the ones obtained from VAN and VAN+S. The metrics are normalized to that of VAN+S. The bars denote the average of four runs, and the error bars represent the standard deviation of the runs.

As shown in Figure 8, both swap-in and swap-out traffic are considerably decreased for A2S. Specifically, the swap-in traffic is decreased by 38.5% from 591.5 MB to 363.8 MB, and the swap-out traffic is reduced by 58.9% from 1,549.0 MB to 637.3 MB. In spite of the reduced swap traffic, the app resume count is similar to that of VAN+S, which is 9.72% larger than that of VAN. This comparison indicates that A2S improves the memory utilization by the similar degree to what VAN+S does, whereas the overhead is greatly reduced.

To analyze the actual effect of swapping on user experience, we analyze the time to launch apps. We collected the app launch times provided by the Android framework while running the benchmark with the Large workload four times (i.e., $256 \times 4 = 1{,}024$ samples). And then we classified the samples according to the quantification method provided by Tolia et al. [36]. Figure 9 summarizes the classification result. Note that users experience positive to neutral feeling from Crisp to Noticeable whereas they experience negative feeling from Annoying to Unusable.

Compared VAN+S to VAN, the the portion for Crisp is decreased from 23.6% to 21.2% whereas that for Noticeable is increased from 61.0% to 62.4%. Also, the portion for negative feeling is increased from 15.4% to 16.4%. In particular, 1.0% of the launches from VAN+S are classified to the Unusable class. This change implies that app launch times are so much increased that user experience is actually degraded in VAN+S. We attribute the cause of the extension to the excessive I/O traffic for swapping that offsets the benefit from swapping (i.e., converting costly app starts to fast app resumes). On the contrary, the portion for negative feeling is decreased to 13.8% for A2S, and the Unusable portion is reduced to less than 0.1%. This indicates that the application-aware swapping

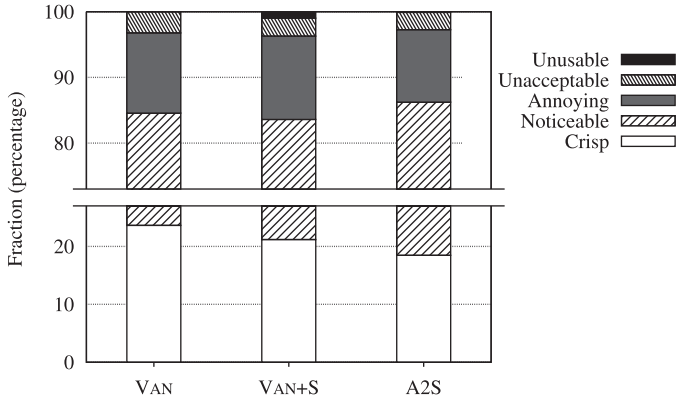


Fig. 9. Classification of launch time.

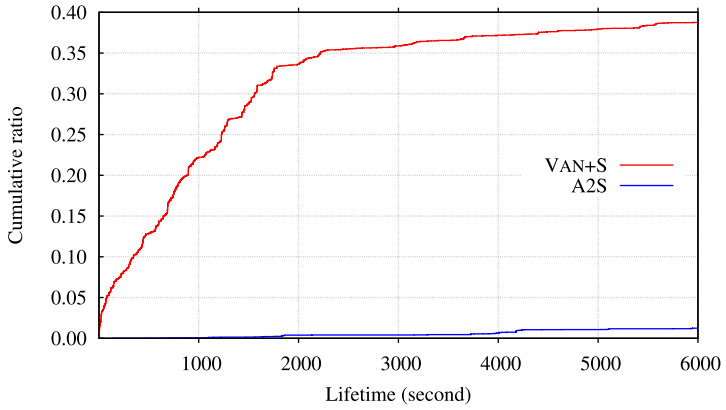


Fig. 10. Comparison of the lifetime of swap entries.

actually improves user experience by handling the excessive I/O effectively, thereby realizing the benefits of swapping.

To verify the effect of the proposed scheme to the quality of the swap traffic, we compare the lifetime of swap entries. We collected the lifetime of swap entries while running the benchmark with the Large workload. We obtained the lifetime of each swap entry by subtracting the time when a swap entry is written to the swap file from the time when all owner processes of the entry are terminated. Figure 10 presents the cumulative ratio of the lifetime of the swap entries. We show the x-axis up to 6,000 seconds among 6,500 seconds, and the omitted entries are not invalidated during the benchmark.

As we explained in Section 3, many swap entries live shortly on VAN+S since the cold background apps owning many inactive pages are terminated via the process-level memory reclamation shortly after the pages are evicted from the memory and written to the swap file. Specifically, 4.1% of entries live less than 60 seconds, 22.2% do 1,000 seconds, and 33.6% survive less than 2,000 seconds. On the other hand, such short-lived swap entries are eliminated when the proposed scheme is adopted; less than 1% of entries live shorter than 4,000 seconds, and approximately 98% of entries are not invalidated throughout the benchmark. This indicates that the proposed scheme effectively eliminates unnecessary swapping from cold background apps.

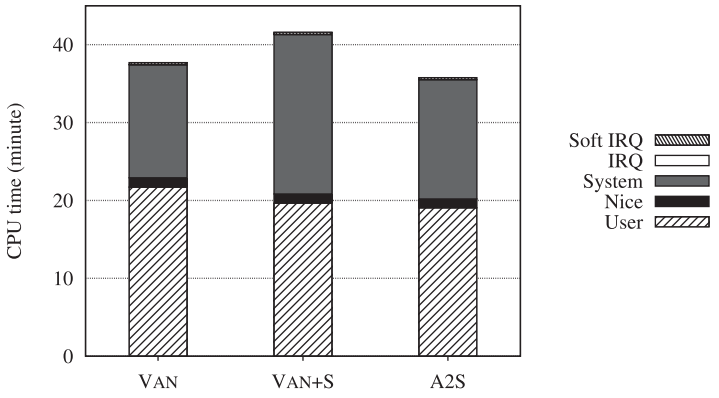


Fig. 11. Breakdown of the CPU time.

Finally, we break down the CPU time to understand the implication of the proposed scheme on the CPU energy consumption. We measure the time spent in each mode through `/proc/stat`, and Figure 11 summarizes the analysis result. On VAN, CPUs spend more time in the user mode than the system mode. As can be seen in VAN+S, swapping reduces the user time whereas the system time is significantly increased. The reduction of the user time comes from the increased app resume count since resuming apps requires less CPU time than restarting the apps in general. On the contrary, the system time is increased by 41.0% for handling swap entries in the kernel mode. With the proposed scheme, the user time is comparable to that of VAN+S since their app resume counts are similar. Thus, the user time of A2S is reduced by 12.2% than that of VAN. The system time is, however, considerably decreased compared to VAN+S as the swapping overhead is greatly reduced with the proposed scheme. As a result, the total CPU time for A2S is 5.1% less than that of VAN. From this result, we can conclude that the proposed scheme reduces CPU energy performance, which is one of the most important factors for mobile systems.

6 CONCLUSION

In this paper, we identified that the page-level approach of swapping and the process-level approach of the app caching policy conflict each other, causing a significant amount of ineffective I/O for swapping. To address the problem, we revised the victim selection policy to consider the efficacy of swapping at a given moment and the remaining lifetime of pages. Our comprehensive evaluation on a commercial smartphone with realistic workloads confirms that the proposed scheme effectively eliminates unnecessary I/Os for swapping, reducing the swap-in and swap-out by up to 39% and 59%, respectively. The reduced I/O decreases the overhead for swapping, thereby improving user experience and energy consumption.

Currently, we are working on an adaptive way for T_{live} and $T_{launched}$. These values are set empirically and statically, which hinders the adaptability and robustness of our schemes for ever-changing workload characteristics. We are attempting to characterize the workload characteristics and to adjust these parameter values accordingly. Also, we are working on building a mathematical model of the memory reclamation in the mobile systems. The process-level memory reclamation brings about a new aspect to be considered to pick the victim page. We are also working on a model to provide the theoretical optimality of memory reclamation schemes.

We believe the proposed scheme and the on-going work identify the great opportunities to improve the memory utilization in mobile systems, and provides an advancement toward a better mobile computing environment.

REFERENCES

[1] Android. 2013. http://www.android.com. (August 2013).
[2] Android. 2013. Android Debug Bridge (ADB). http://developer.android.com/tools/help/adb.html. (September 2013).
[3] Android. 2013. Managing the Activity Lifecycle. http://developer.android.com/training/basics/activity-lifecycle/index.html. (October 2013).
[4] Android. 2014. Low RAM. https://source.android.com/devices/tech/low-ram.html. (2014).
[5] Apple, Inc. 2013. Apple iOS 7. http://www.apple.com/ios. (2013).
[6] Anderson Farias Briglia, Allan Bezerra, Leonid Moiseichuk, and Nitin Gupta. 2007. Evaluating effects of cache memory compression on embedded systems. In *Proceedings of the 2007 Ottawa Linux Symposium (OLS'07)*. 53–64.
[7] Richard W. Carr. 1984. *Virtual Memory Management*. UMI Research Press.
[8] Aaron Carroll and Gernot Heiser. 2010. An Analysis of Power Consumption in a Smartphone. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*. 21–35.
[9] Aaron Carroll and Gernot Heiser. 2013. The Systems Hacker's Guide to the Galaxy: Energy Usage in a Modern Smartphone. In *Proceedings of the 4th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'13)*.
[10] David Chu, Aman Kansal, Jie Liu, and Feng Zhao. 2011. Mobile apps: it's time to move up to CondOS. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'11)*. 16–20.
[11] Hossein Falaki, Ratul Mahajan, Srikanth Kandula, Dimitrios Lymberopoulos, Ramesh Govindan, and Deborah Estrin. 2010. Diversity in Smartphone Usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys'10)*. 179–194.
[12] Jason Flinn, Keith I. Farkas, and Jennifer Anderson. 2000. *Power and Energy Characterization of the Itsy Pocket Computer (Version 1.5)*. Technical Report HP Labs Technical Reports WRL-TN-56. Western Research Laboratory.
[13] Google, Inc. 2015. Nexus 5. http://www.google.com/nexus/5/. (March 2015).
[14] GSMArena.com. 2010. Motorola XT720 MOTOROI - Full phone specifications. http://www.gsmarena.com/motorola_xt720_motoroi-3090.php. (2010).
[15] Nitin Gupta. 2010. compcache: Compressed Caching for Linux. http://code.google.com/p/compcache. (2010).
[16] Dianne Hackborn. 2010. Multitasking the Android Way. http://android-developers.blogspot.kr/2010/04/multitasking-android-way.html. (2010).
[17] Anthony Hayter. 2012. *Probability and Statistics for Engineers and Scientists*. Brooks/Cole, Cengage Learning.
[18] International Data Corporation (IDC). 2015. Worldwide Quarterly Mobile Phone Tracker. http://www.idc.com/tracker/showtrackerhome.jsp. (Aug. 2015).
[19] Seth Jennings. 2013. Zswap: Compressed Swap Caching. http://lwn.net/Articles/528817. (2013).
[20] Theodore Johnson and Dennis Shasha. 1994. 2Q: A low overhead high performance buffer management replacement algoritm. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB'94)*. 439–450.
[21] Sang-Hoon Kim, Jinkyu Jeong, Jin-Soo Kim, and Seungryoul Maeng. 2016. SmartLMK: A memory reclamation scheme for improving user-perceived app launch time. *ACM Transactions on Embedded Computing Systems* 15, 47 (May 2016).
[22] An-Chow Lai, Cem Fide, and Babak Falsafi. 2001. Dead-block Prediction & Dead-block Correlating Prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*. 144–154.
[23] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S. H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.* 50, 12 (Dec. 2001), 1352–1361.
[24] Sarah Lewin. 2014. Apple's and Samsung's changing smartphone recipes. *IEEE Spectrum*. (Nov. 2014).
[25] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: Saving DRAM Refresh-power through Critical Data Partitioning. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. 213–224.
[26] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. 115–130.
[27] Arpit Midha and Patrik Torstensson. 2017. Android Go. https://events.google.com/io/schedule/?section=may-18. (May 2017).
[28] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*. 297–306.
[29] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. 2013. Practical Prediction and Prefetch for Faster Access to Applications on Mobile Phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp'13)*. 275–284.
[30] Ahmad Rahmati, Clayton Shepard, Chad Tossell, Mian Dong, Zhen Wang, Lin Zhong, and Phillip Kortum. 2011. Tales of 34 iPhone Users: How They Change and Why They Are Different. *Technical Report TR-2011-0624* (2011).
[31] Samsung Electronics Co., Ltd. 2014. Samsung GALAXY S3. http://www.samsung.com/global/galaxys3. (2014).

[32] Samsung Electronics Co., Ltd. 2014. Samsung GALAXY S4 - Life companion. http://www.samsung.com/global/microsite/galaxys4. (2014).
[33] Samsung Electronics, Co., Ltd. 2016. Galaxy S7 and Galaxy S7 edge. (2016).
[34] Clayton Shepard, Ahmad Rahmati, Chad Tossell, Lin Zhong, and Phillip Kortum. 2011. LiveLab: measuring wireless networks and smartphone users in the field. *Performance Evaluation Review* 38, 3 (January 2011), 15–20.
[35] The Linux Foundation. 2013. Tizen: An open source, standards-based software platform for multiple device categories. http://www.tizen.org. (2013).
[36] Niraj Tolia, David G. Andersen, and M. Satyanarayanan. 2006. Quantifying interactive user experience on thin clients. *IEEE Computer* 39 (March 2006), 46–52.
[37] Ionut Trestian, Supranamaya Ranjan, Aleksandar Kuzmanovic, and Antonio Nucci. 2009. Measuring Serendipity: Connecting People, Locations, and Interests in a Mobile 3G Network. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference (IMC'09)*. 267–279.
[38] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast App Launching for Mobile Devices Using Predictive User Context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications and Services (MobiSys'12)*.
[39] Chunhui Zhang, Xiang Ding, Guanling Chen, Ke Huang, Xiaoxiao Ma, and Bo Yan. 2013. Nihao: A Predictive Smartphone Application Launcher. 110 (2013), 294–313.
[40] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. 177–188.

Received April 2017; revised June 2017; accepted June 2017