Selective Memory Deduplication for Cost Efficiency in Mobile Smart Devices

Sung-hun Kim, Jinkyu Jeong, and Joonwon Lee

Abstract — *Memory deduplication, which can remedy memory scarcity in mobile systems, can hardly be used due to its high computation cost. This paper proposes a computation efficient memory deduplication scheme that avoids unnecessary computations for memory deduplication. To make it efficient, the proposed scheme gets rid of pages, which are unlikely to be deduplicated, from the target of memory deduplication. This exclusion is performed in time-domain and in space-domain by exploiting the characteristics of mobile applications. The prototype implementation shows significant computation cost reduction while providing the same memory savings as in previous approaches. The saved memory is used for improving application launch time as well.*[1]

Index Terms — Smartphone, Memory deduplication, Energy efficiency, Memory management

I. Introduction

In these days, mobile smart devices, such as smartphones and smart tablets, are replacing other single purpose devices such as MP3 player, portable media player (PMP), and sometimes laptop PCs. Since a smart device is a general-purposed mobile device, it can perform multiple functions which multiple single-purposed devices have provided. A mobile smart device, however, has restricted resources such as CPU, memory, and storage because of its limited energy source (small-sized battery) and form-factor. Thus, efficient use of the limited hardware resources is important [1].

Although the computing power of a smart device is limited due to the constrained hardware resources, customers want to have better user experience. One of the important metrics that evaluate user experience is *application-launching time* [2]. In order to reduce the effective launch time of applications, many mobile application frameworks support application caching [2], [7]. Once an application is executed, it is cached in memory so that a user can quickly reuse the application. Accordingly, the more applications a system caches, the faster launch time it can provide. A drawback is that since smartphones have limited main memory capacity, the number of cacheable applications is also limited.

One of ways for increasing the number of cached applications within a fixed physical memory size is to increase memory usage density in a system. Memory deduplication is a software-based approach to increasing memory density [4], [5]. By merging identical pages within a system, this approach can secure more free memory. The saved memory pages can be used for caching additional applications. This approach, however, incurs significant computation overhead to a system because inspecting identical pages is highly CPU-intensive workload [6]. Although this CPU overhead is not a significant problem in server systems because those systems have unlimited power source, mobile systems cannot adopt the memory deduplication approach due to power concern.

This paper proposes an efficient memory deduplication scheme that reduces the CPU overhead accompanied by the memory deduplication. To reduce the CPU overhead, this scheme only scans memory pages that are likely to be deduplicated. The hints about the likelihood are derived from the characteristics of mobile application framework and the memory deduplication pattern in mobile systems. Briefly, in time-domain, cached application's pages are only scanned once and never scanned again until the application becomes active state because cached applications do not change their memory states. In space-domain, only small part of virtual pages of a process is the target of memory deduplication because the other part of pages has very low probability of being deduplicated. By filtering out such pages, the proposed scheme can greatly reduce the memory inspection overhead while providing mostly the same amount of memory savings a conventional memory deduplication approach can provide.

The proposed scheme is implemented on a well-known mobile smartphone and evaluated with real-world mobile applications. The evaluation results show that the deduplication cost of the proposed scheme is only 2% and 19% of the previous approaches. Due to the minimized deduplication cost and secured additional memory, the proposed scheme improves application launch time by 10% compared to the case without memory deduplication.

The rest of this paper is organized as follows. The following section describes the background associated with the proposed scheme and reviews the related work. Section III explains the idea of the proposed scheme and Section IV shows the implementation issues. Section V evaluates the proposed scheme. The last section concludes this paper.

II. Background and Related Work

A. Background

Many mobile application frameworks support various features to improve user experience. Among them, several features, such as *application caching* and *fork-dlopen execution model*, can

[1] This work was partly supported by the IT R&D program of MKE/KEIT. [10041244, SmartTV 2.0 Software Platform] and by the Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(2013R1A6A3A01023894).

Sung-hun Kim, Jinkyu Jeong, and Joonwon Lee are with the College of Information and Communication Engineering, SungKyunKwan University, Suwon, Rep. of Korea (e-mail: shkim@csl.skku.edu, jinkyu@skku.edu joonwon@skku.edu).

Contributed Paper
Manuscript received 04/01/14
Current version published 06/23/14
Electronic version published 06/23/14.

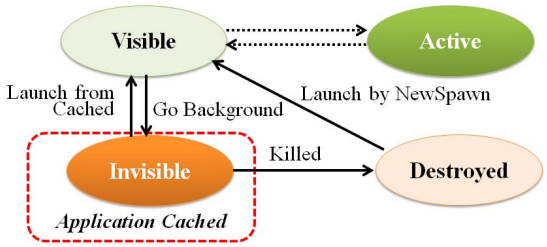

Fig. 1. State diagram of mobile applications

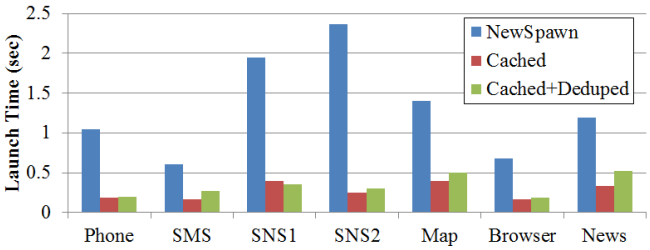

Fig. 2. Launch time of applications in three cases: spawning a new process, resuming a cached application, and resuming a cached one whose memory pages are deduplicated so copy-on-write-protected.

provide hints for reducing the memory deduplication overhead. This subsection illustrates the background of those features and hints associated with them.

1) Application Caching

Application caching is an approach to improving user experience by caching applications when a user finishes interaction with them. Since mobile devices are general-purposed systems, these can run multiple applications. This multi-programming environment, however, does not indicate simultaneous execution of multiple applications. Because mobile devices usually have a small screen, only one or a few applications run at a time. Except for a foreground application, other applications no longer interact with a user until those become active again. Application caching is that instead of killing processes hosting the finished applications, the processes are kept live in memory as depicted in Fig. 1. When a user wants to launch an application hosted by a cached process, its launch time can be greatly reduced as compared to spawning a new process. Fig. 2 shows the launch time comparison between spawning a new process and reusing a cached process. As shown in the figure, the application caching shows 3.5 - 10 times faster launch time than spawning a new process.

The other benefit of application caching is that when a user visits a cached application again, the user can spontaneously resume the interaction with the application. Because a cached application keep the last state a user have interacted (e.g., the last stage played in a game), a user do not need to input additional command to make the application move to the last state. This benefit also greatly reduces time to resume an application by avoiding the additional steps to recover the last state.

The drawback of application caching is memory space cost for caching each application process. Table I shows the memory cost for caching well known mobile applications. The memory cost varies across applications and the average is 20 MBs. Since the application caching can boost the launch time of applications in a system, the more applications a system caches, the better user experience the system can provide. A mobile smart device, however, has limited memory capacity due to many factors, such as unit cost, power consumption and small form factor. Accordingly, the number of applications a system can cache is limited. When a system has memory shortage, an operating system reclaims memory pages of a cached application by killing the application (the *Killed* arrow in Fig. 1). This memory reclamation scheme in process granularity is denoted as low memory killer (LMK) [3], [7].

TABLE I
MEMORY COSTS FOR CACHING APPLICATIONS

Application	Memory Cost	Application	Memory Cost
Phone	14.3 MB	SNS2	18.6 MB
SMS	5.7 MB	Map	32.4 MB
SNS1	24.5 MB	Browser	65.1 MB
News	8.0 MB	Email	6.7 MB
Market1	8.8 MB	**Average**	**20.5 MB**

The application caching is the motivation of the need for securing more memory by using memory deduplication. The additional memory can be used for caching more application so that the possibility to launch an application from the cached state can increase. Therefore, the user experience of a mobile smart device will be improved.

The hint the application caching can provide to memory deduplication is that the data stored in cached applications are stable. Since a cached application no longer interacts with users until it goes into foreground, the application does not need to be allocated CPU. Accordingly, the cached application never changes the content of the pages belonging to the applications. Since memory deduplication is based on the content of each page, this hint can rule out the target pages to be inspected for memory deduplication and can also determine the frequency of scanning of those stable pages. The details are explained in Section III-A.

2) Fork-dlopen Execution Model

Many mobile application frameworks adopt *fork-dlopen* execution model rather than *fork-exec* execution model. The fork-exec model is traditionally used application execution model in Unix-like systems. In this model, a parent process forks a child process and calls an exec-like system call to replace the address space of the child process into a new one as depicted in Fig. 3. Loading an executable binary and its associated libraries occur during this phase. The drawback of this model is that when an application uses a lot of libraries, loading the libraries slows down the launch time of an application. Since mobile applications are highly dependent to many application framework libraries [18], their launch time can be slowed down.

Fork-dlopen execution model is better than the fork-exec model in this case. Using this model, a parent process already loads commonly used application libraries in its address space. When to run a new application, it forks a child process by

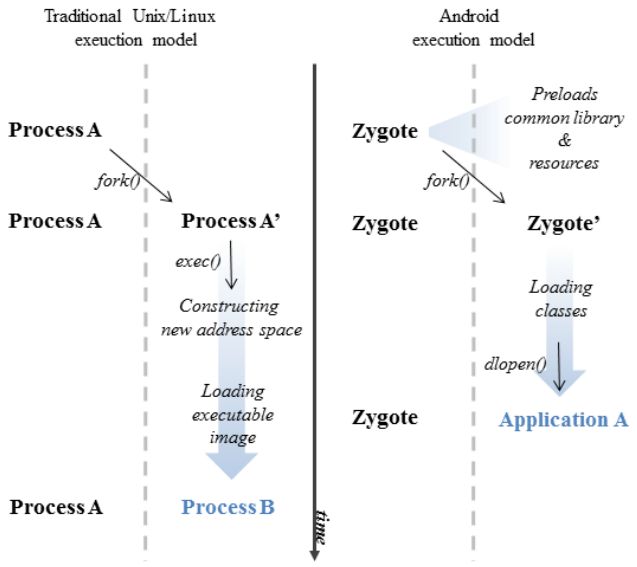

Fig. 3. Traditional Unix/Linux execution model (fork()-exec()) vs. Android execution model (fork()-dlopen())

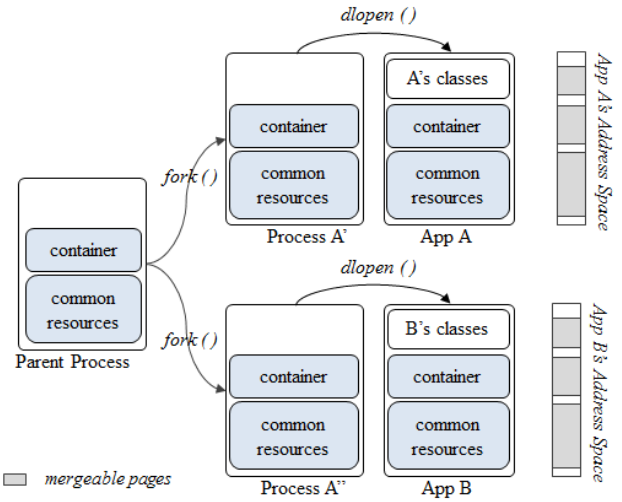

Fig. 4. Similar address space of applications by using fork-dlopen execution model

invoking the fork system call. Then, it only loads an executable of the desired application and calls the main function of the application as shown in Fig. 3. Accordingly, the number of binary loads is greatly reduced compared to the fork-exec model [19]. Many mobile application frameworks adopt this model due to the benefit of reducing application launch time.

The main characteristic of this fork-dlopen execution model is that applications have mostly the same address space layout as illustrated in Fig. 4. Since many commonly used libraries and resource files are already linked in the parent's address space, applications forked from the parent still have the same files linked at the same addresses. In addition, identical pages are usually found within the forked memory regions. By using this hint, the proposed scheme can get rid of pages that never provide sharing chances. The details are explained in Section III-B.

3) Memory Deduplication

Memory deduplication (or content-based page sharing) is a technique to secure free memory by merging identical pages. When there are multiple pages having the same content, multiple contexts (e.g., processes or virtual machines) can share one page so that the rest of the pages can be saved. To avoid data inconsistency from any modification to the shared page, page table entries to the shared page are copy-on-write protected. Any context performing updates to the shared page causes copy-on-write break and the context have a private copy of the shared page.

Since memory deduplication is based on the contents of pages, inspecting memory pages in a system is essential. Usually, a thread is designated to inspect machine memory and find out identical pages. When identical pages are found, the thread conducts page merging. Since physical memory management is the role of operating system kernel, the thread runs at kernel-level.

From the perspective of applications, the main cost of memory deduplication is copy-on-write protection. When merged (shared) pages are read-only accessed, it does not incur any overhead accessing the merged pages. When writes occur on the merged pages, however, the writes causes page fault and memory copies to perform copy-on-write.

In mobile systems, this copy-on-write overhead is not significant in practice. Fig. 1 also includes the launch time of applications when each application's memory pages are deduplicated so copy-on-write protected (Cached+Deduped). Since memory pages are copy-on-write protected, launching the application may cause several copy-on-writes. This overhead, however, does not overrun the benefit of application caching. In the figure, the launch time of applications whose memory pages are deduplicated is still faster than spawning a new process.

The main overhead of memory deduplication from the perspective of system is scanning the page frames to find out identical pages. To find out same pages, every combination of pages should be compared in byte-by-byte granularity. This overhead is non-trivial. This overhead can be serious problem in battery-powered mobile devices. Accordingly, even if a mobile system has many identical pages, memory deduplication cannot be easily adopted without resolving the high scanning overhead.

4) Memory Duplication in Smart Devices

Basically, memory deduplication is based on the assumption that a system has many duplicated (identical) pages. This assumption is mostly true in virtualized environment. Indeed, memory deduplication is actively used in virtualized environments [4], [5], [8], [9]. Since a physical machine hosts multiple virtual machines (VMs) and the software as well as the data used in VMs can be similar [10]. Accordingly, by merging those identical pages, the physical system can obtain additional free pages.

Whether many identical pages exist in mobile smart devices is an important issue because if no memory duplication occurs,

memory deduplication is useless. A mobile device usually runs not multiple operating systems but a single operating system. Within a single operating system, memory pages are unlikely duplicated because operating system tries to minimize duplication of memory pages. For example, when multiple processes read the same file, an operating system maintains a single copy of the file within its page cache and the content is copied to each process. When a process forks, a child process shares the pages of its parent process albeit in copy-on-write protection.

Memory duplication in mobile systems actually occurs due to the fork-dlopen execution model. Since mobile applications are highly application-framework dependent, they share a lot of shared libraries. It causes each application has similar address space layout as shown in Fig. 4. It means that when a data structure consisting of multiple pointers is stored in a process's heap, the contents of the data structure are the same across multiple processes [11]. In addition, cached applications are at the same state waiting for the event to make them to be in foreground. Accordingly, the states of variables are mostly the same across cached applications. For example, a lock variable is frequently locked or unlocked when an application is in foreground. But, the lock variable is unlocked while the application is in cached state. For these reasons, a mobile system has many identical pages and memory deduplication can be applied to mobile systems to secure more memory [6].

B. Related Work

Waldspurger introduced content-based page sharing in a commodity virtual machine monitor [4]. It finds out identical pages and merges them based on the contents of pages. It also provides a knob to throttle the ratio at which a page scanning thread works. By adjusting the scanning speed (e.g., 100 pages per 100 ms), the overhead can be reduced. But, when the inspection thread works slowly, it cannot quickly gather same pages [12]. This scheme also uses a hash table to reduce the number of memory comparisons. When two pages have the same hash value, then it compares both pages in byte granularity.

Kernel same page merging (KSM) is memory deduplication technique used in Linux kernel [5]. It not only targets kernel virtual machines (KVMs) but also processes running on the host Linux kernel. This scheme uses a red-block tree to reduce the number of memory comparisons.

To avoid the scanning of pages, several studies targets memory deduplication on cache pages of storage [8], [13], [14]. When multiple VMs read pages having the same block number, those pages are deduplicated because those have the same content. These approaches, however, are limited to cache pages while most of the sharing chances are on heap or data sections within a process in mobile systems.

To reduce scanning overhead, Sharma *et al* [16] proposed scanning dirtied pages only scheme. When clean pages are scanned once, no additional scanning is required.

Chen et al [17] exploits page access characteristics to reduce the number of memory comparisons during memory deduplication. Pages having similar access patterns have high likelihood of deduplication. This scheme, however, requires hardware modification that generates the foot print of access pattern of a page.

Memory deduplication in mobile is first mentioned in Cells [15]. Because it enables multiple mobile systems to be hosted on a single smartphone device, the system can experience high memory shortage. Memory deduplication is applied to overcome the memory shortage. No attempts to minimize the overhead related to the memory deduplications were made.

III. Selective Memory Deduplication

Although a mobile smart device has many identical pages which can be deduplicated for saving memory, CPU costs caused by memory deduplication should be resolved. Otherwise, excessive use of CPU will drain battery quickly.

The main idea of the proposed efficient memory deduplication scheme is to prune the target pages to be inspected for memory deduplication. By getting rid of unlikely to be deduplicated pages, the deduplication thread does not need to consume unnecessary CPU cycles. By carefully selecting the pages not to be inspected, the amount of memory savings provided by the memory deduplication can be the same as that of an uncontrolled scheme.

The hints are arisen from the specific characteristics of mobile applications. The first subsection gives the idea of pruning pages in each process based on the scheduling information. The next subsection provides how the proposed scheme selects likely to be deduplicated pages within an application. The last subsection gives implementation issues related to the proposed ideas.

A. Pruning Pages in Time Domain

Since memory deduplication is based on the contents of pages, scheduling information can give a great hint to whether to inspect a certain memory pages or not. Every content update on a process's memory pages occurs by store instructions on CPU. Accordingly, when a process is not allocated CPU, it is guaranteed that the memory pages belonging to the process are not updated. From this point, when pages in a process are inspected for memory deduplication, until the process is scheduled on CPU, no additional inspection is needed [16].

This simple pruning of memory pages based on the scheduling information, however, have a problem. For example, a foreground application is given CPU allocation to show its initial screen. Then the application waits for input commands by a user. During this time, the process for the application is not allocated CPU. At this moment, if pages within the process are deduplicated, the following progress of the application may update the contents of the pages so that the deduplicated pages will be broken their copy-on-write protection. In this regard, low-level scheduling information can cause unnecessary memory deduplication and overhead associated with the deduplication.

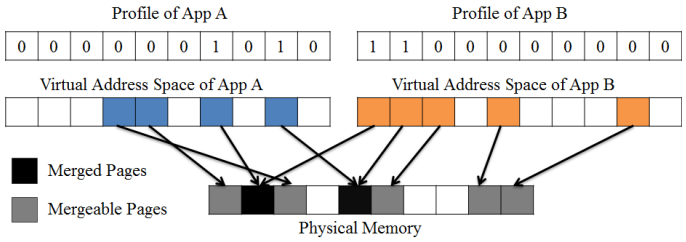

Fig. 5. Virtual page profile of each application. Each profile consists of virtual pages which are merged by memory deduplication.

In mobile smart devices, more accurate high-level information can be used for the hints instead of the scheduling information. Recall that cached applications are not given CPU allocation for a long time in several to hundred seconds. Accordingly, the memory pages of a cached application are stable for a long time. By using this characteristic, when an application goes into the cached state, its memory pages become the target of memory deduplication. In other words, memory pages of a foreground application are ruled out of target of memory deduplication.

In addition, once a cached application's memory pages are scanned, no additional scanning is needed. The reason for this is that memory pages are not updated until the process is given CPU allocation. Accordingly, when some pages of them are deduplicated, those pages are stable.

Service applications, which are given CPU allocation even when the applications are not visible to users, are outlier of the proposed time-domain pruning. For example, an MP3 player application periodically runs in background. Memory pages of such service applications, however, are unlikely to be deduplicated because these applications are live so that variables have different values to other cached applications.

B. Pruning Pages in Space Domain

While the pruning in time domain gets rid of unnecessary pages to be inspected in process-granularity, pruning in space domain excludes certain pages within each process. The pages that should be excluded are selected based on a priori knowledge as follows.

1) Intra-Application Hint

Recall that most duplicated pages are found on data section, such as a heap and library data sections. The main characteristic of these regions is that the regions already exist in the address space of the parent process. Memory regions that are created after spawning a new process unlikely generate memory duplication because the content stored in such region is totally private. Because merging pages occur across processes, totally private pages cannot find its counterpart for page merging. On the other hand, the memory regions which already exist in the parent process provide many memory deduplication chances because multiple application processes have the same regions and the contents within them are also similar to each other.

In this regard, an assumption for pruning pages within a process is that pages shared with other processes are limited to some pages. In order to validate this assumption, the stability of sharing is measured. Stability of sharing indicates that when an application progresses, a fixed set of pages give memory savings by memory deduplication.

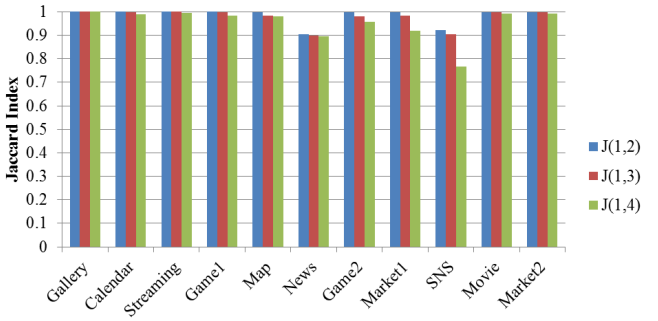

Fig. 6. Stability of deduplicated pages within an application by calculating similarity of deduplicated virtual pages over time. J(1, 2) denotes that the first and second sets of deduplicated virtual pages are used to calculate similarity by using Jaccard's index.

To measure the stability, well-known mobile applications are used. For each application, after an application is started, (i) memory deduplication is conducted once and the pages (virtual pages) that are merged by deduplication are recorded (the first profile or P_1) as depicted in Fig. 5. Then, to make changes on the state, (ii) the application is given several commands. After that, (iii) the shared virtual pages are recorded again (the second profile or P_2). The (ii) and (iii) phases are repeated three times. From this measurement, each application gives four sets of deduplicated virtual pages each of which is denoted as P_1, P_2, P_3, and P_4.

If the four sets of an application are the same (or similar) of each other, deduplicated virtual pages are stable within the lifetime of the application. For the similarity metric, *Jaccard index* [20] is used which provides a similarity value of two sets from (1).

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{1}$$

Fig. 6 shows the measured Jaccard indices of well-known applications. J(1,2) denotes P_1, P_2 are used to calculate Jaccard index. Except for the SNS application, other applications have 0.9 - 1 of their Jaccard indices. Since the value is closed to 1, every set of virtual page profiles is similar to each other within an application.

The result indicates that if the deduplication thread knows each application's profile, a set of virtual pages that are likely to be deduplicated, the thread can inspect only virtual pages within the set. Accordingly, many virtual pages not belonging to the profile are skipped.

When to build a profile of each application is also a challenging issue. In order to build a profile of an application, full scan of virtual pages of an application is necessary. Since mobile devices usually connected to power outlet for recharging, the full scanning can be performed at that moment without the power concern. By applying this approach, not

TABLE II
SIMILARITY OF A SET OF DEDUPLICATED VIRTUAL PAGES ACROSS APPLICATIONS

	Phone	Calendar	Market1	Movie	News	Browser	Game1	Streaming	Game2	SNS	Gallery	Map	Market2	Alarm
Phone		0.66	0.31	0.28	0.79	0.26	0.42	0.58	0.74	0.73	0.49	0.85	0.63	0.86
Calendar	0.66		0.33	0.22	0.59	0.20	0.31	0.44	0.60	0.57	0.40	0.63	0.82	0.62
Market1	0.31	0.33		0.21	0.31	0.22	0.33	0.39	0.27	0.34	0.31	0.30	0.33	0.31
Movie	0.28	0.22	0.21		0.28	0.30	0.26	0.31	0.28	0.30	0.33	0.28	0.21	0.28
News	0.79	0.59	0.31	0.28		0.27	0.43	0.61	0.77	0.74	0.49	0.86	0.62	0.87
Browser	0.26	0.20	0.22	0.30	0.27		0.26	0.30	0.24	0.29	0.30	0.27	0.20	0.27
Game1	0.42	0.31	0.33	0.26	0.43	0.26		0.47	0.38	0.45	0.39	0.42	0.31	0.42
Streaming	0.58	0.44	0.39	0.31	0.61	0.30	0.47		0.55	0.62	0.52	0.60	0.44	0.60
Game2	0.74	0.60	0.27	0.28	0.77	0.24	0.38	0.55		0.71	0.52	0.81	0.60	0.74
SNS	0.73	0.57	0.34	0.30	0.74	0.29	0.45	0.62	0.71		0.56	0.79	0.53	0.75
Gallery	0.49	0.40	0.31	0.33	0.49	0.30	0.39	0.52	0.52	0.56		0.50	0.38	0.50
Map	0.85	0.63	0.30	0.28	0.86	0.27	0.42	0.60	0.81	0.79	0.50		0.61	0.86
Market2	0.63	0.82	0.33	0.21	0.62	0.20	0.31	0.44	0.60	0.53	0.38	0.61		0.62
Alarm	0.86	0.62	0.31	0.28	0.87	0.27	0.42	0.60	0.74	0.75	0.50	0.86	0.62	

only making an initial profile but also updating a profile can be done. Applications having low stability, (SNS in Fig. 6) can take advantage of this update for having more accurate profile.

2) Inter-Application Hint

Maintaining profiles for each application, however, has high memory overhead. For example, when an application has 3 GBs of its memory address space, it can have at most 786,432 virtual pages. If a profile is represented as a bit for each virtual page, the profile needs 96 KBs of memory space. Accordingly, when the number of applications increases, the memory space for maintaining the profile for each application becomes significantly large.

One of the ways to minimize this storage overhead is to maintain one profile (global profile) for all applications. Hence, the global profile is union of each application's profile. To make this approach viable, virtual pages likely to be deduplicated should be similar across applications. In addition, the global profile should contain minimal false positive virtual pages. Therefore, a virtual page exists in the global profile, but the virtual page does not exist in an application's profile.

First, to know whether applications have similar footprint of shared virtual pages, the Jaccard indices of well-known applications are calculated as depicted in Table II. As shown in the table, actually, virtual pages merged by memory deduplication are different from each other since the Jaccard index values are in the range of 0.2 to 0.87. The average similarity across the applications is only 0.47. This means that each application's profile is neither different from nor similar to each other. This, however, necessarily means that maintaining a global profile is inefficient. When a virtual page is set in the global profile, if the virtual page does not exist in an application's address space, this virtual page is not a target of scanning.

In this regard, more important metric for measuring the overhead when to maintain one global profile is the portion of false positive. Hence, the portion of virtual pages which are set in the global profile but are unlikely to be deduplicated in an application. This portion of app A can be calculated by (2), where GP is the global profile; MP_A is a set of mergeable pages of app A; P_A is a set of merged pages of app A.

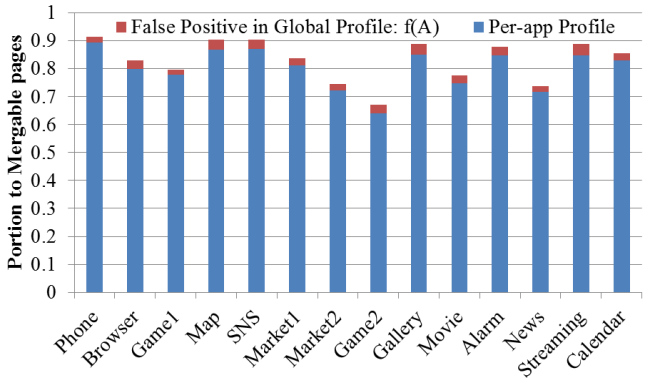

Fig. 7. The portion of per-app profile and false positive in the global profile to mergeable pages within each application.

$$f(A) = \frac{(GP \cap MP_A) - P_A}{MP_A} \tag{2}$$

Fig. 7 shows the portion of per-application profile and false positive of using global profile to the mergeable (anonymous) pages for each application. Hence, the portion of per-application profile is how much per-application profile skips unlikely deduplicated pages, and the portion of false positive is how many pages are unnecessarily scanned by using the global profile instead of using per-application profile. As shown in the figure, the portion of per-application profile is 64% - 89% across the application. This result indicates that from 11% to 36% of memory pages are skipped during scanning. In addition, the portion of false positive is very low at 2% - 4% of the mergeable pages. The result indicates that using one global profile is reasonable since only a few pages are needlessly scanned.

The global profile is also maintained as the per-application profile is maintained. When a smart device is plugged to power outlet, it performs full scan and sets a bit associated to a virtual page which is deduplicated.

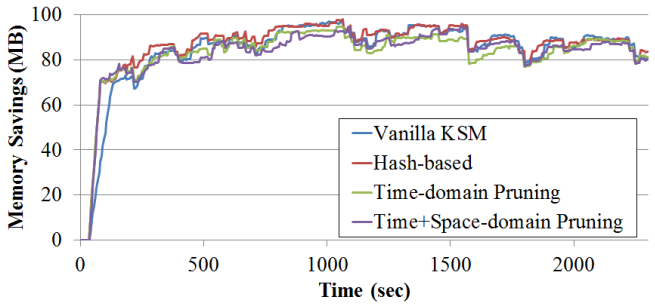

Fig. 8. Memory savings while a real-user workload is running.

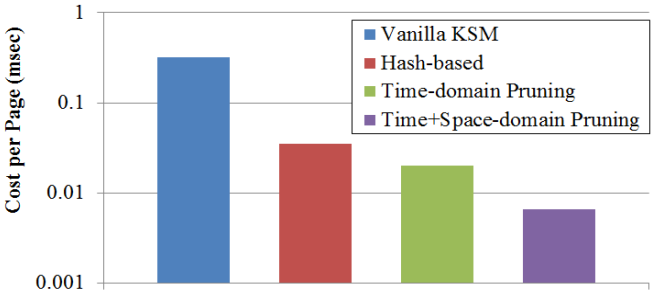

Fig. 9. Computation time for scanning one page in each scheme (*time spent by memory deduplication thread* / *# of pages scanned*).

C. *Scheduling Interference*

Excluding pages unlikely to be deduplicated from the scanning target does not necessarily mean that the overhead of memory deduplication is completely eliminated. The deduplication thread continuously works whenever some pages become the target of inspection. When the memory deduplication thread is active, it still contends for CPU with other applications.

Many operating system kernels provide fair-based scheduling. Accordingly, when the deduplication thread is given the same priority to the other applications, the scheduling contention increases the runqueue wait time of applications. This results in the increased launch time of applications. In this regard, not only reducing the target of memory deduplication, but also careful scheduling of the memory deduplication thread is also important.

One of ways of minimizing the scheduling contention is to degrade the scheduling class of the memory deduplication thread. For example, the thread is only allocated CPU when no runnable processes exist. This, however, may increase the complexity of existing operating system.

An easier way of minimizing the scheduling contention is to give low priority to the deduplication thread. However, if the priority is too low, the progress is too slow that it can lose chances of memory deduplication [12]. This will result in killing more cached applications than that giving proper priority to the deduplication thread. In this regard, careful selection of the priority is important so that the priority satisfies both minimized scheduling contention and sufficient speed of memory deduplication.

TABLE III
CHARACTERISTICS OF THE MERGED PAGES

Region	Same virtual address	Different virtual address
Anonymous Pages (heap, bitmap, et al.)	64.1%	24.5%
Library data section	10.6%	0.8%

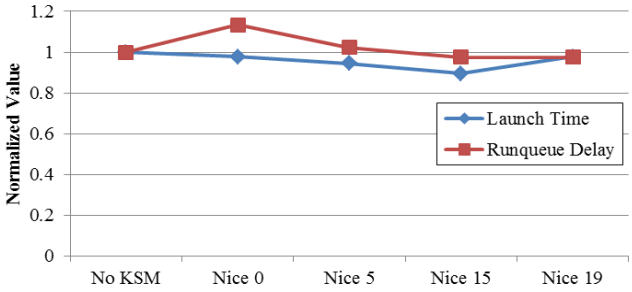

Fig. 10. Normalized launch time and runqueue delay of the proposed scheme with varying the nice (priority) value of ksmd from 0 (high priority) to 19 (low priority).

IV. EVALUATION

This section shows the effect of the proposed memory deduplication scheme. First, the proposed scheme is evaluated for how well it reduces the computation cost while ensuring memory savings as much as that provided by original KSM. The next subsection provides how the proposed schemes improve user experience and mitigate memory pressure of the mobile smart devices. Finally, the last subsection gives the evaluation of the overheads caused by the memory deduplication.

A. *Experimental Environment*

The proposed scheme is implemented in an open source-based mobile operating system. Since the mobile operating system uses Linux kernel, the proposed scheme is applied to KSM in Linux kernel. The evaluation is conducted on a well-known commodity smartphone which is equipped with 1GB of main memory and 1.2 GHz dual-core CPU.

All evaluations are performed by running a real world workload scenario. The real-world workload consists of twenty representative mobile applications. Those applications are classified into several groups such as social network service (SNS), game, market, streaming service, video player, web-browser, news, calendar, map, and some basic application suite for providing cellular phone functionalities. Each run of a workload invokes 200 application launches in a fixed sequence [21]. The portion of each application is determined by the smartphone usage characteristics [22].

The memory deduplication thread (*ksmd*) periodically scans 100 pages and goes to sleep for 20 milliseconds by default. This configuration is used in all experiments.

The proposed scheme is compared with *vanilla KSM* which uses a red-black tree index and *hash-based KSM* which uses a hash table instead of the tree. The case memory deduplication is disabled is denoted as *no KSM*.

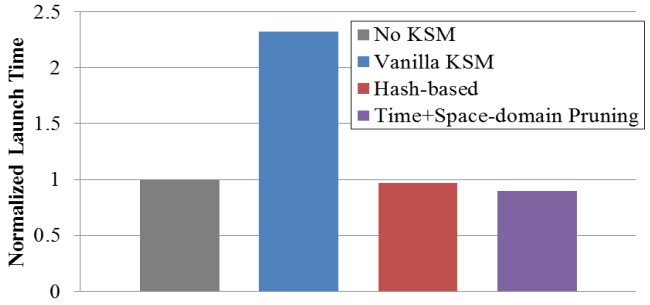

Fig. 11. Normalized launch time of applications

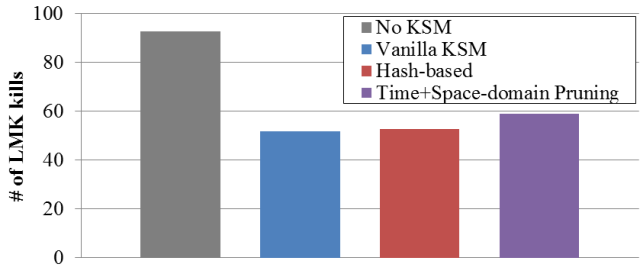

Fig. 12. The number of process terminations (# of LMKs performed).

The full scan is only performed at the beginning of a workload. Full scan of an application's address space is also performed when the application is firstly launched during the workload running. Each full scan updates the global profile.

Except for the evaluation in Section IV-C, the nice value (priority) of ksmd is assigned 15. This value is obtained from the parameter sensitivity test described in Section IV-C.

B. Memory Savings and Computation Cost

The primary objective of memory deduplication is securing free memory pages. Therefore, proposed memory deduplication schemes should secure free memory pages as many as that ensured by the vanilla KSM. Fig. 8 shows the amount of memory savings achieved while running the workload. As shown in the figure, the proposed scheme shows similar memory savings to vanilla KSM and hash-based KSM.

Fig. 9 shows the cost of memory deduplication. Hence, the value shown in the figure is average time to scan a page for deduplication in each scheme. The cost of the proposed scheme is reduced to only 0.2% of vanilla KSM and 0.19% of hash-based KSM. Since the proposed scheme skips pages unlikely deduplicated, the average cost of scanning is greatly reduced.

Table III shows the characteristics of the merged pages by memory deduplication. As shown in the table, the majority of merged pages belong to anonymous page regions such as heap and bitmap (metadata for free regions in the heap) regions. Data sections in libraries also show 11% of merged pages. In addition, 74% of merged pages have the same virtual address of each other.

C. Scheduling Parameter

As depicted in Section III-C, careful selection of scheduling parameter of the memory deduplication thread (ksmd) is important. This subsection tests the sensitivity of the

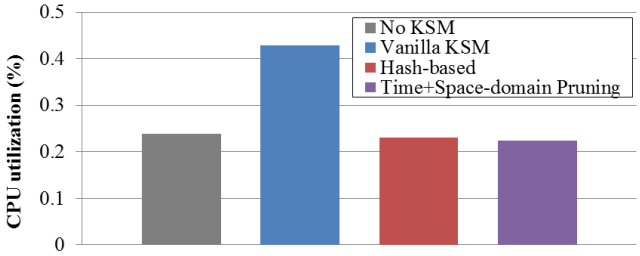

Fig. 13. CPU utilization while the workload is running.

scheduling parameter (i.e., priority). Fig. 10 shows launch time and runqueue delay of the ksmd thread. The values are normalized to the case that memory deduplication is disabled. The runqueue delay excludes the delay of ksmd. As shown in the figure, when the priority of ksmd is high (nice 0), the dealy of all applications is increased due to scheduling contention. This offsets the advantage of securing more memory by memory deduplication. When the priority is low (nice 19), the runqueue delay is low, but the launch time is increased. When the priority of ksmd is low, the progress of memory deduplication is too slow to timely secure additional memory. In this experiment, the best priority parameter (nice value) is 15. In that case, the proposed scheme shows the lowest launch time with low runqueue delay.

D. Performance Benefit

This section compares the performance benefit of the proposed scheme with vanilla KSM and hash-based KSM. Fig. 11 shows normalized launch time of each scheme. The values are normalized to that of the no KSM case. As shown in the figure, the vanilla KSM shows the longest launch time of applications. Since vanilla KSM shows the highest cost of scanning, high CPU contention between applications and ksmd increases the launch time. Hash-based KSM shows faster launch time than the no KSM case by 3%. The proposed scheme shows the fastest launch time by 10% compared to the no KSM case. Because the proposed scheme has the lowest scanning cost, minimal scheduling contention happens.

The main benefit of memory deduplication is the ability to cache more applications in memory. Fig. 12 shows the number of application kills (LMKs) while the workload is running. Fewer kills mean that more applications are cached in memory. As shown in the figure, the no KSM case shows the largest application kills. Accordingly, many applications are re-launched from newly spawning processes. By using memory deduplication, fewer applications are killed. This indicates that additional memory contributed by memory deduplication is used for caching more applications. Although the proposed scheme shows slightly more application kills, it shows shorter launch time than other schemes.

E. Overhead

The final test of the proposed scheme is whether the memory deduplication consumes more power. Since power consumption of a smartphone is generally proportional to the CPU usage [3], the CPU utilization is measured while each workload is running and depicted in Fig. 13. As shown in the

figure, the vanilla KSM shows the highest CPU utilization. Since this scheme has high scanning cost, the cost is directly reflected to the CPU utilization. The proposed scheme shows lower CPU utilization than the no KSM case. While the proposed scheme consumes more CPU, the benefits of reducing application launch time offsets the additional CPU consumption. Note that killing an application or launching an application by spawning a new process needs additional CPU time than switching cached application from background to foreground.

V. Conclusion

In spite of the benefits of memory deduplication, this technique is not adopted into smart mobile devices because of its high computation cost. Its massive computations result in quick battery drain so that the operational time of smart devices is shortened.

In this paper, a cost effective memory deduplication scheme is proposed that scans memory pages selectively by using several heuristics coming from the specific features of mobile smart devices. With these heuristics, the proposed memory deduplication schemes can reduce the computation cost while it ensures almost the same memory savings that secured by previous memory deduplication schemes. By using the additional memory for caching more applications, the evaluation result shows that the proposed scheme improves application launch time without incurring additional CPU consumption.

References

[1] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng, "Rigorous rental memory management for embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 12, issue 1, pp. 43:1-43:21, Mar. 2013.

[2] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," *in Proc. Int. Conf. Mobile Systems, Applications, and Services*, pp. 113-126, 2012.

[3] C. Yoon, D. Kim, W. Jung, C. Kang, and H. Cha, "AppScope: application energy metering framework for android smartphones Using Kernel Activity Monitoring," *in Proc. USENIX Annu. Technical Conf.*, 2012.

[4] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 181–194, Dec. 2002.

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

[6] S. Kim, J. Jeong, and J. Lee, "Efficient memory deduplication for mobile smart devices," *in Proc. IEEE Int. Conf. Consumer Electronics*, pp. 25-26, Jan. 2014.

[7] G. Lim, C. Min, Y. Eom, "Virtual memory partitioning for enhancing application performance in mobile platforms," *IEEE Trans. Consumer Electron.*, vol. 59, no. 4, pp. 786-794, Nov. 2013.

[8] G. Miłós, D. G. Murray, S. Hand, and M, A. Fetterman, "Satori: enlightened page sharing," *in Proc. USENIX Annu. Technical Conf.*, 2009.

[9] 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," *Commun. ACM*, vol. 53, no. 10, pp. 85-93, Oct. 2010.

[10] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: exploiting page sharing for smart colocation in virtualized data centers," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 3, pp. 27-37, July 2009.

[11] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman, "An empirical study of memory sharing in virtual machines," *in Proc. USENIX Annu. Technical Conf.*, 2012.

[12] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "XLH: more effective memory deduplication scanners through cross-layer hints," *in Proc. USENIX Annu. Technical Conf.*, pp. 279-290, 2013.

[13] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: running commodity operating systems on scalable multiprocessors," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 412-447, Nov. 1997.

[14] K. Miller, F. Franz, T. Groeninger, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "KSM++: using I/O-based hints to make memory-deduplication scanner more efficient," *In Proc. Workshop on Runtime Environments, Systems, Layering and Virtualized Environments*, 2012.

[15] J. Andrus, C. Dall, A. Van't Hof, O. Laadan, and J. Nieh, "Cells: a virtual mobile smartphone architecture," *in Proc. ACM Symp. Operating Systems Principles*, pp. 173-187, 2011.

[16] P. Sharma, and P Kulkarni, "Singleton: system-wide page deduplication in virtual environments," *in Proc. the 21st Int. Symp. on High-Performance Parallel and Distributed Computing*, pp. 15-26, 2012.

[17] L. Chen, Z. Wei, Z. Cui, M. Chen, H. Pan, and Y. Bao, "CMD: classification-based memory deduplication through page access characteristics," in Proc. ACM SIGPLAN/SIGOPS Int. Conf. Virtual Execution Environments, pp. 65-76, 2014.

[18] A. Gutierrez, R. G. Dreslinski, T. F. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-system analysis and characterization of interactive smartphone applications," *in Proc. IEEE Int. Symp. Workload Characterization*, pp. 81-90, 2011.

[19] C. Jung, D. Woo, K. Kim, and S. Lim, "Performance characterization of prelinking and preloading for embedded systems," *in Proc. ACM & IEEE International Conference on Embedded Software*, pp. 213-220, 2007.

[20] P. Tan, M. Steinbach, V. Kuma, *Introduction to Data Mining*, 1st ed., Addision-Wesley, 2005.

[21] J. Jeong, H. Kim, J. Hwang, J. Lee, and S. Maeng, "DaaC: device-reserved memory as an eviction-based file cache," *in Proc. 21th Int. Conf. Compilers Architecture and Synthesis for Embedded Systems*, Tampere, Finland, pp. 191-200, 2012.

[22] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin, "Diversity in smartphone usage," *in Proc. Int. Conf. Mobile Systems, Applications, and Services*, pp. 179-194, 2010.

BIOGRAPHIES

Sung-hun Kim received his B.S. degree in Computer Engineering from Sunghyunkwan University (SKKU) in 2012 and M.S. degree in Information Technology Convergence Department from SKKU in 2014. He is currently a Ph.D. candidate in the IT Convergence Department in SKKU. His current research interests include operating systems, multi-core systems, storage systems and mobile systems.

Jinkyu Jeong received his B.S. degree in Computer Science from Yonsei University and Ph.D. degree from Korea Advanced Institute of Science and Technology (KAIST) in 2013. He is currently an assistant professor in the department of Semiconductor Systems Engineering at Sungkyunkwan University. His research interests include real-time systems, operating systems, virtualization, and embedded systems.

Joonwon Lee received his B.S. degree in Computer Science from Seoul National University in 1983 and M.S. and Ph.D. degrees from the Georgia Institute of Technology in 1990 and 1991, respectively. He is currently a professor in Sungkyunkwan University (SKKU). Before joining SKKU, he was a professor at the Korea Advanced Institute of Science and Technology from 1992 to 2008. His current research interests include low power embedded systems, system software, and virtual machines.