MemScope: Analyzing Memory Duplication on Android Systems

Byeoksan Lee, Seong Min Kim, Eru Park, Dongsu Han

KAIST

Abstract

Main memory is one of the most important and valuable resources in mobile devices. While resource efficiency, in general, is important in mobile computing where programs run on limited battery power and resources, managing main memory is especially critical because it has a significant impact on user experience. However, there is mounting evidence that Android systems do not utilize main memory efficiently, and actually cause page-level duplications in the physical memory. This paper takes the first step in accurately measuring the level of memory duplication and diagnosing the root cause of the problem. To this end, we develop a system called MemScope that automatically identifies and measures memory duplication levels for Android systems. It identifies which memory segment contains duplicate memory pages by analyzing the page table and the memory content. We present the design of MemScope and our preliminary evaluation. The results show that 10 to 20% of memory pages used by applications are redundant. We identify several possible causes of the problem.

1. Introduction

Main memory is one of the most important and valuable resources in mobile devices. While resource efficiency, in general, is important in mobile computing where programs run on limited battery power and resources [2], managing main memory is especially critical because it has a significant impact on user experience. For example, the Android operating system reclaims memory from applications or even destroys application processes when it is low on memory. However, reclaiming memory space involves garbage collection that consumes significant CPU cycles, which often increases the response times of applications. Also, terminating applications often results in the loss of valuable application context (e.g., log-in information or cached data), which in turn degrades user experience because users need to perform the same action (e.g., log-in) when the app is reloaded. Additionally, it leads to longer reload time [15].

As users run an increasing number of applications and each application becomes more complex, the memory pressure on mobile devices is increasing. This coupled with the rise of low memory devices [5] makes the problem even more important. Android itself tried to reduce system memory footprint and produced a way to tune the system for low memory devices [6]. However, there is mounting evidence that Android systems do not utilize main memory efficiently, and actually cause page-level duplications in the physical memory [5, 13].

To mitigate the problem, Android has introduced several mechanisms to optimize the system for low memory devices, such as Kernel Same-page Merging (KSM) [8] and zRAM [12]. For example, KSM periodically scans the physical memory and merges pages if their content is the same. Swapping to zRAM uses in-memory swap space; when it needs to swap out pages from the memory, it compresses them into an in-memory swap area. Although these approaches can reduce the memory usage and the number of applications terminated due to low memory [5], they consume power and CPU cycles. Recent studies [14] also suggest that they may actually decrease user experience. More importantly, these approaches are haphazard solutions to the problem since they do not treat or even identify the root cause. Once the root causes of page duplication are identified, they may be used to prevent duplication from happening or apply deduplication more efficiently.

We argue that a systematic approach is required to diagnose and solve the problem. This paper takes the first step in measuring the level of memory duplication and diagnosing the root cause of the problem. To this end, we develop a system called MemScope that automatically identifies and measures memory duplication levels for Android systems. It identifies which memory segment contains duplicate memory pages by analyzing the page table of each application and the memory content.

In summary, this paper makes two contributions:

1. **A useful framework for memory duplication analysis:** Using a combination of existing tools, MemScope reconstructs page tables and their mapped regions. It helps users to track the relationship between events that potentially affect memory duplication by identifying the difference between snapshots of physical memory.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

APSys 2015,, July 27–28, 2015, Tokyo, Japan..

Copyright © 2015 ACM 978-1-4503-3554-6/15/07. . . $15.00.

http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2797022.2797023

2. **A preliminary analysis of memory duplication:** We look at diverse aspects of memory duplication in Android systems. In particular, we find that 10-20% of physical pages are duplicates and many reasons contribute to this duplication. We identify common patterns and sources of page-level memory duplication.

2. System Overview

2.1 MemScope Features

Our goal is to identify how many duplicate physical frames exist on Android systems at a particular time and how they might change over time during a typical life cycle of applications. In addition, we seek to identify where in the virtual memory segment duplicate occurs (e.g., code segment or BSS section) and what kinds of system-level events are correlated with changes in the duplicate frames. Achieving the goal requires analyzing the page table of each process and finger-printing the memory content. To infer the cause of duplicate memory content, the system must also be able to identify the memory segment and correlate with system-level events (e.g., garbage collection).

In summary, MemScope provides the following features:

- **Statistics on duplicate physical memory frames**: MemScope takes a snapshot of physical memory and analyzes page-level duplicate frames. It reconstructs the page tables from the physical memory snapshot and takes a fingerprint of memory content. MemScope then categorizes the duplicate frames into their segment types.
- **Tracking the difference across snapshots**: MemScope allows users to track the differences across multiple snapshots that were taken over time. For this, one can configure MemScope to take the memory snapshot periodically or upon a specified system-level event (e.g., garbage collection). MemScope also records various system-related events (e.g., garbage collection, process creation, low memory killer activation, etc) during the use of the device. This helps users to link events with increase or decrease in duplicate levels of memory content.
- **In-depth analysis on Android-specific data structures**: MemScope detects Android-specific data structures by analyzing the memory layout. Because all Android applications run on dalvik virtual machine, it analyzes dalvik-related data segments, including dalvik heap and bitmap structures. It is able to recover the number of heap objects and their sizes. It also extracts useful information from Android-specific memory regions and relates that information to memory duplication.

2.2 MemScope Overview

MemScope operation consists of two phases: data acquisition and data analysis. In the data acquisition phase, it collects dumps of the physical memory content and a system log that tracks system-level events. In the data analysis phase, it recovers the page table of each process from the memory dump and identifies the duplicate content within the physical memory.

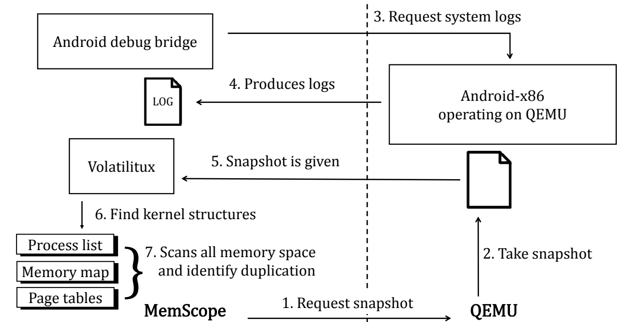

Figure 1: Overview of MemScope

To obtain a physical memory dump (i.e. snapshot), MemScope relies on the support from the hypervisor by running Android on a virtual machine (VM). We take this approach because obtaining a memory dump from a real device is much more difficult and may require hardware support. Another benefit of using VM is that we can configure the machine easily. By changing the boot-up settings, one can easily modify RAM size, change hardware components, and maintain snapshots for reproducibility. We believe that the use of virtual machine does not affect the memory usage.

3. MemScope Design

MemScope is a tool which automatically takes snapshots and retrieves system log. With obtained snapshots, it analyzes how many pages are duplicated and where they are and tracks their changes.

3.1 Data acquisition

MemScope automates data acquisition by combining a variety of tools, including QEMU [9] and Android debug bridge (ADB) [1]. Figure 1 illustrates overall structure of MemScope. When MemScope starts up, it uses QEMU as a hypervisor and creates a VM to run the Android system image specified by the user. During its lifetime, it can take dumps of the VM's physical memory. To take a memory dump, MemScope first suspends the VM and takes a memory snapshot using QEMU monitor, and resumes the VM. MemScope can also repeat this process periodically at a time interval specified by the user.

Android generates rich logs of system events. To keep track of system-level events, MemScope extracts the system log of the Android system using ADB. Among various logs, we only focus on logs from Android system services, such as system server and activity manager. The system log is later used to track which system events affect memory duplication. In addition, we can stop taking snapshots when a specific system event (e.g. low memory killer) occurs.

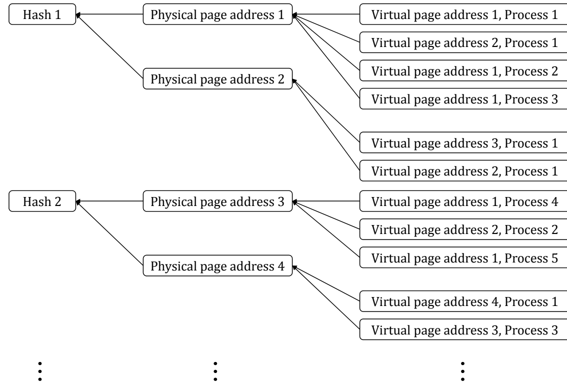

Figure 2: Duplication table structure [13]

3.2 Data analysis

The goal of this phase is to extract as much information as possible about memory pages to help identifying the potential cause of memory duplication. In order to analyze memory duplication from the snapshot, MemScope first locates important data structures and analyzes them to obtain the list of processes, their page tables, and the memory space of each process. This information is contained in `task_struct` and `vm_area_struct` data structures. To locate these data structures, MemScope leverages volatilitux [10], a memory forensic tool for Linux. Given a memory snapshot, it locates these data structures automatically by searching for specific patterns and/or inspecting certain offsets in the memory.

Using this information MemScope takes a hash of each page in the memory and links it to the process and the segment that it belongs to. Figure 2 illustrates the resulting data structure (duplication table) that MemScope produces. For each hash value of the content in a physical page, it lists the physical page address corresponding to the hash value. For each physical page, it reconstructs the process ID and its virtual page address that the physical page is mapped to.

Because volatilitux is a generic memory forensic tool developed for Linux, it only provides process memory layout and memory regions, not specific and detailed information about Android internal data structures. MemScope extracts Android-specific data structures to obtain in-depth information to see the relationship between duplication and Android system state. Because we find that a surprisingly large fraction of duplication comes from the dalvik VM's heap area, we analyze the region in more detail. Specifically, we focus on the dalvik VM's heap objects.

We recover and analyze key data structures that point to all heap objects and are used for garbage collection (`dalvik-bitmap-1` and `dalvik-bitmap-2`). Analyzing these data structures provides us the information about the number of objects allocated, their sizes, and the free space. MemScope locates these data structures related to the dalvik VM that host each Android application.

Unlike Linux data structures, there is no tool to detect Android data structures (i.e. dalvik-related data) automatically. [1] However, recovering the information is not trivial because dalvik-related data is in user memory space, not in the kernel memory space. Thus, we need to know the virtual address of dalvik-related data and translate that address to physical address. Once we know where dalvik-related data is, we can easily translate the address of that data into physical address by utilizing volatilitux.

To locate dalvik-related data structures, MemScope first searches gDvm, a global variable that holds the global state of dalvik VM. It keeps VM heap data structures and configurations of the dalvik VM. Because dalvik VM is implemented inside the libdvm library (i.e. libdvm.so), gDvm is located inside the BSS segment of libdvm.so where uninitialized global variables are stored. However, it is difficult to know the exact memory offset of gDvm within the BSS segment. To resolve this, we rely on instrumentation. We rebuild libdvm after inserting a magic number into gDvm's structure so that we can locate the variable by inspecting the memory content. With this change, we can find dalvik-related data. This allows us to identify the total number of objects allocated in dalvik heap space given a dump of the physical memory.

4. Preliminary Results and Analysis

We install a virtual machine with Android-x86 (Kitkat 4.4.4) and allocate 512MB of RAM to it. MemScope is configured to take snapshots every 60 seconds after booting. LMK(low memory killer) starts working when the system is under memory pressure in order to obtain free memory by destroying applications. In the case of application, it is manually executed and used between every two snapshots until LMK is triggered by the system. We manually operate the applications to mimic normal application usage. Note some apps are already brought up at boot, and executing those apps do not increase the total # of processes. In our case, there are 20 system processes in every snapshot. Upon detecting that LMK is triggered, we take more fine-grained snapshots every 5 seconds to see how LMK affects the memory duplication level.

Through MemScope, We observe the followings:

- **Sub-page level duplication is much higher.** Up to 34.12 % of used memory is duplicated when we divide memory into 1KB-fragments, compared to 21.78 % for page-level duplication.
- **Duplication associated with apps is the dominating factor.** 80 % of total duplication is due to the duplication associated with apps, compared to that of system processes.
- **More than 80% of duplication comes from 5 memory regions.** They are gralloc-buffer, dalvik-heap, dalvik-

[1] We make a couple of assumptions to locate Android-specific data structures: Android is operating on 32-bit machine and each page is 4 kB.

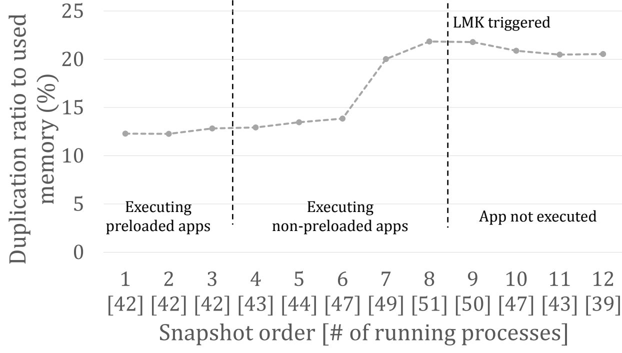

Figure 3: Duplication and its ratio to used memory across time

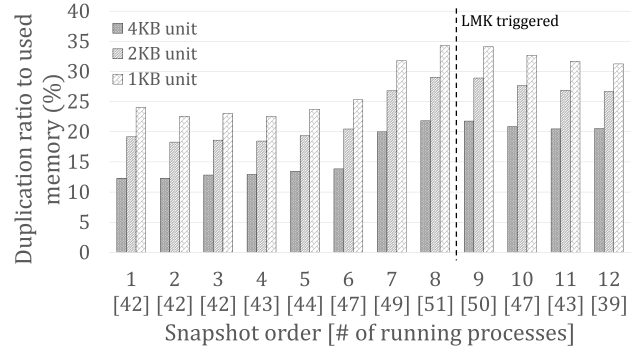

Figure 4: Subpage-level duplication ratio comparison

zygote, BSS segment of libdvm.so library and anonymous, which are the heaps and data sections.

4.1 Page-level and Subpage-level Duplication

We find that the memory duplication level tends to increase as applications are executed until LMK is triggered to reclaim memory. Figure 3 describes duplication across time. At snapshot 1, there is no user-executed applications. All 42 processes are started by the system. 20 of them are system processes such as initd and netd, and the remaining 22 are applications that run at start up, such as the dialer app. We then execute applications in a randomized order during the experiment, because the ordering is not very important for our purpose. Note, executing an app which is already loaded into a process at boot time does not increase the number of running processes. Amazon app and twitter app are such examples.

Right after the system boots 32.1 MB is duplicated while 261.1 MB is used. We can see that 12.3 % of used memory is already duplicated from the beginning. Right before LMK is triggered, about 90.36 MB is duplicated and memory being used by processes are 413.82 MB. Memory used by all processes is obtained by getting the set of valid physical addresses in all page tables. We see that ratio to memory being used is around 21.84 %. It means that there is potential to get at least 45.18 MB, which is 8.82 % out of 512 MB , free memory by deduplication. **This translates to at least one or two apps worth of memory** [3, 14].

Figure 4 shows subpage-level duplication. It shows that when we use smaller duplication unit (e.g 2 kB, 1 kB) there is potential to obtain more free memory by deduplicating them. For example, when the system runs out of memory, it can perform a finer-grain deduplication on a subset of applications and perform delta encoding [11] when the deduplicated page is modified by other apps, instead of activating the low memory killer (LMK) that kills apps. Later, when the apps are brought to foreground, the system can recover the deduplicated page from the information saved. Because most duplicates originate across apps as we show below (rather than system processes) deduplicating with another app is sufficient. We believe that this approach will be more efficient than zRAM that compresses the memory used by a single app. On average, using a 2 kB unit would result in 6 %(24.45 MB) more memory savings and 1 kB unit results in 11 %(42.41 MB) more memory savings than 4 kB unit.

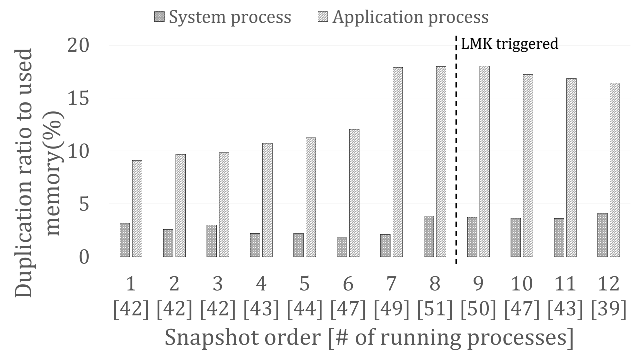

Figure 5: Ratio of system process-associated and application process-associated duplication to memory being used

4.2 System Processes vs. Application Processes

Android applications and non-application processes may exhibit very different characteristics because Android app runs dalvik VM but non-application process does not. To distinguish the two, we classify the children of the zygote process and zygote itself as application processes since zygote is the parent of all applications and runs on dalvik VM. Other processes(e.g. init, netd) are classified as system processes.

In order to identify which process group affects more on duplication, we obtain how many duplicated pages are associated with each group. Figure 5 shows duplication ratio to memory usage associated with each process group. It says that application processes contribute to duplication much more than system processes. On average, 3 % and 14 % of memory usage are duplicated from system processes and application processes respectively. In addition, application group is involved in 80 % of total duplication. The results show that we are better to focus on application processes for deduplication.

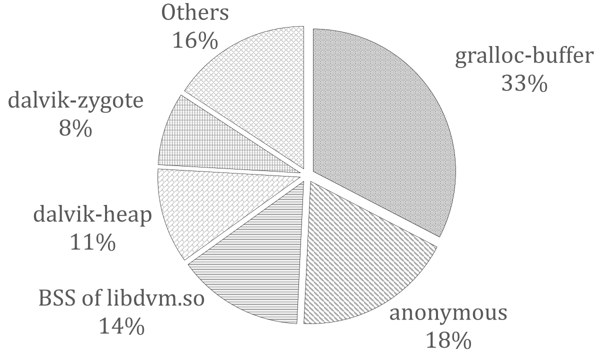

Figure 6: Average duplication portion of memory regions

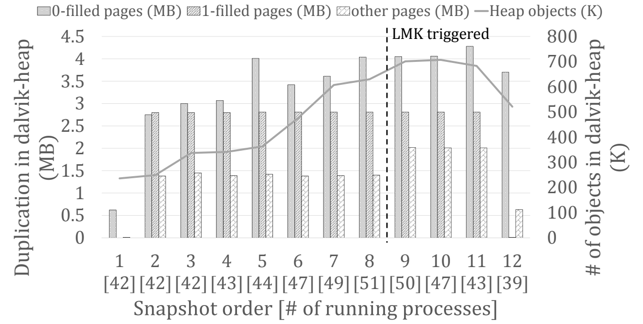

Figure 7: Duplication by content(MB) and # of objects(K) in dalvik-heap

4.3 Duplication in Memory Regions

MemScope allows users to identify the name of the memory segment that duplicated pages appear. Using this information, we investigate which memory regions contains the most duplicate pages. Note, not all memory regions have their area name—we call them anonymous regions.

Figure 6 shows average duplication portion of the five memory regions. We use same snapshot set as earlier figures. We find that top five regions are almost consistent across different snapshots. More than 80 % of duplicated pages come from anonymous, gralloc-buffer, dalvik-heap, dalvik-zygote and BSS segment of libdvm.so library.

Gralloc-buffer is buffer allocated by gralloc HAL (hardware abstraction layer), which is graphic memory allocator implemented by various vendors. In other words, gralloc-buffer is a buffer used for displaying the screen of Android system to user. We observe that most duplicated pages in gralloc-buffer for application are grouped by multiple of three. We discover that Android systems, by default, adopt triple-buffering policy to display buffers since JellyBean so that there are three gralloc-buffers. This is to make rendering smoother, but because the screen does not frequently change for many apps other than those display video, we see duplicate content in display buffers. We believe that by adaptively assigning the buffers (e.g., only do triple buffering upon detection of frequent changes in the buffer), a significant amount of memory (up to 33 %) can be saved.

Dalvik-zygote is one of heap spaces for dalvik VM, which is created when zygote is created. Right after zygote is created, zygote preloads several common Java class objects and resources into dalvik-zygote region. We observe most of duplicate pages in this region have same virtual addresses from different application processes. For example, virtual pages at address `X` from three different applications are mapped to three different physical pages which have same page content. Because every application is forked from zygote, we believe that these pages are generated by copy-on-write (COW). However, duplicate occurs because all apps are initialized with the same set of objects. We believe that by performing the dalvik-zygote initialization at the parent will eliminate the duplicates.

BSS of libdvm.so is the place where global state of dalvik VM is stored. As in dalvik-zygote, there are many duplicate pages which seem to be COWed. The difference is that most of their contents are 1-filled. We suspect that after the child application process is forked, variables in the BSS section are initialized to the same value, which triggers COW. One way to eliminate the root cause is to locate the global variables in the duplicated 1-filled pages and initializing them in the parent process similar to dalvik-zygote.

Dalvik-heap is one of the hot spots where duplication occurs frequently. It is a heap space for dalvik VM and Java objects are allocated here. In other words, duplicated pages in dalvik-heap are due to Java objects that an application uses.

We categorize duplicated pages into three groups based on their content: 0-filled page, 1-filled page and others. Figure 7 shows change of the number of duplicated pages and java objects in dalvik-heap. The number of 0-filled pages dominates in most cases. The number of non-0-filled pages does not show any correlation with the number of objects and it is rather stable. Contrarily, the number of 0-filled pages is more sensitive to the number of heap objects. We hypothesize that when java object is created, enough memory is allocated and only little portion of allocated memory is written. Thus, the rest of the memory remains 0.

In order to identify this we explore Android source code and find java object creation code. `dvmHeapSourceAlloc()` which is used for java object allocation is located at `dalvik/vm/alloc/HeapSource.cpp`. We find that there are two modes for allocating memory.

One is low memory mode and we are going to call the other normal mode. In normal mode memory is allocated using `mspace_calloc()`, which at first `mspace_malloc()` and then `memset()` to 0. This always makes several zero-filled physical pages which are duplicate. On the other hand, in low memory mode it first gets enough memory using `mspace_malloc()` and then returns all pages, except for the first page, back to kernel by using `madvise()` with parameter `MADV_DONTNEED`. With low memory mode on, we expect the amount of duplicate 0-filled pages in dalvik-heap to reduce.

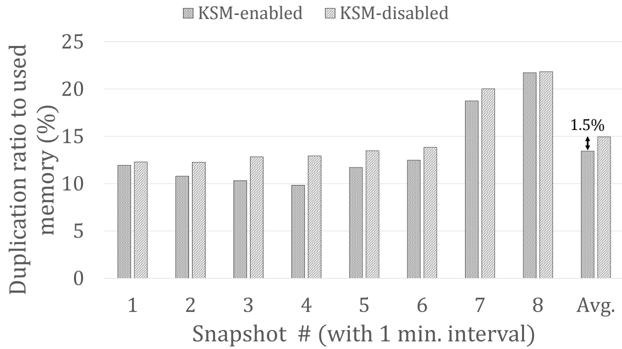

Figure 8: Duplication ratio comparison between KSM-enabled and KSM-disabled snapshots

Anonymous Pages are pages which do not have any name for their memory regions. Many kinds of memory regions are categorized to anonymous page. Application stack and some dynamically allocated memory regions, for example, are types of anonymous pages. We see that anonymous pages account for 18 % of duplicate memory. In addition, 13 % ∼ 30 % of duplication in anonymous region are 0-filled. We guess that these pages are generated by `memset()` to 0 just like the dalvik-heap case. For non-0-filled anonymous pages (e.g reserved by `mmap() with MAP_ANONYMOUS flag`), we need further in-depth research to identify which data is contained or which function accesses to certain page.

5. Implications and Discussions

The observation we made in Section 4 has many implications for memory management. Here, we simply look at how effective KSM is in removing the duplication. We configure KSM to scan 100 pages to merge every 500 ms as [5] suggests.

Figure 8 indicates the difference between KSM-enabled system and default system (KSM-disabled) snapshots with respect to duplication ratio. We find that KSM-enabled system produces about 2 % less duplication ratio than that of default system. The duplication ratio is reduced slightly and still more than 70 % of duplication comes from 5 memory regions mentioned in section 4.3. This indicates that KSM treats every pages equally for selecting candidates of deduplication. It would be better that each page has a different priority so that pages in the 5 regions have higher chance to be scanned and deduplicated. When it is possible, duplication ratio is expected to be reduced by up to 10 % in ideal case.

In addition, if it is possible to identify which system event triggers memory duplication, KSM can use event-driven deduplication policy instead of periodic scanning. Then, we can save the computing power of mobile devices which was serious weakness of KSM.

6. Related Work

Memory deduplication for virtual machines: Memory deduplication is a popular technique to reduce the memory use for virtual machines [11]. Page sharing, delta-encoding, and memory compression have been also used in virtualized environments to reduce the physical memory footprint when running multiple virtual machines. Some of these techniques have been implemented in Android. However, these features are rarely used because energy consumption and user experience are dominating factors in mobile environments.
Memory reclaiming/saving techniques for Android: Android reclaims memory by using two approaches. The first is garbage collection. Android garbage collector is implemented as a part of dalvik and uses mark and sweep algorithm [4]. Second is the low memory killer (LMK). LMK is an android-specific implementation of out-of-memory killer in Linux [7]. It sets a few adjustment value and minimum free memory pairs while booting OS. LMK is triggered if the amount of available memory is not sufficient. LMK kills apps with matching or having a lower adjustment value if the amount of available memory is under a certain threshold.

In Android version 4.4, it supports two new methods for memory management KSM [8] and zRAM [12]. KSM scans pages in given interval and merges pages with same content. zRAM is a swap system which compresses and swaps out pages into RAM based block devices. Although both approaches may improve RAM usability, they take up CPU cycles and power that decreases the battery lifetime and often worsen user experience. The objective of MemScope is helping its users to analyze memory duplication and potentially improve deduplication techniques based on the analysis.
Android memory deduplication: Julino [13] studied page-level memory duplication on Android systems. This is the first study of memory duplication in Android. However, it does not provide any tools and falls short in analyzing its main cause. Recent work [14] proposes a selective memory duplication that carefully selects deduplication pages by observing time-domain and space-domain hints. By using our in-depth observation on Android-specific information, selecting candidate pages for depuplication will be accurate and enhanced.

7. Conclusion and Future Work

Memory usage for Android devices has been of recent interest. Many solutions try to manage main memory more efficiently and even attempt to perform memory deduplication dynamically. However, very few studies examine the current status of main memory use in a systematic way. Even there exists no tool that allows us to analyze the memory duplication in Android systems. To address the problem, we develop MemScope which is a tool to analyze memory duplication for Android systems. It runs on top of QEMU for controllable experiments. The tool automatically takes snapshots of memory and analyzes the amount of duplicate content in memory. We present a design of MemScope and a preliminary analysis of memory duplication for Android systems to demonstrate

the usefulness of the tool. We break down the duplicates into memory regions and identify that most of the duplicate originates from data sections including heap. We further identify potential causes to these duplications and implement user input emulating function for more controllable and accurate experiment. We believe that the MemScope tool and the observations made in the paper will benefit future work in this area.

Acknowledgements

This research was supported by an Institute for Information communications Technology Promotion (IITP) grant funded by the Korean government(MSIP) (No. B0126-15-1078). We also thank the anonymous reviewers and shepherd Mahesh Balakrishnan for their valuable comments.

References

[1] Android Developer's Guide, Android Debug Bridge. http://developer.android.com/tools/help/adb.html, Retrieved March 1st, 2011.

[2] Android Developer's Guide, Managing Your App's Memory. https://developer.android.com/training/articles/memory.html, Retrieved January, 2014.

[3] Average memory usage for google play app categories. http://devsbuild.it/content/Average-Memory-Usage-Google-Play-App-Categories.

[4] Memory management for android apps. http://dubroy.com/memory_management_for_android_apps.pdf.

[5] Running android with low ram. https://source.android.com/devices/tech/low-ram.html.

[6] Tuning android for low ram. http://events.linuxfoundation.org/sites/events/files/slides/android-lowmemory-abs-2014.pdf.

[7] When the kernel runs out of memory. https://events.linuxfoundation.org/slides/2010/linuxcon2010_rientjes.pdf, Retrieved.

[8] Arcangeli, A., Eidus, I., and Wright, C. Increasing memory density by using ksm. In *Proceedings of the linux symposium* (2009), pp. 19–28.

[9] Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), pp. 41–46.

[10] Girault, E. Volatilitux: Physical memory analysis of linux systems, 2010.

[11] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A. C., Varghese, G., Voelker, G. M., and Vahdat, A. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM 53*, 10 (2010), 85–93.

[12] Gupta, N. Compcache: in-memory compressed swapping, 2009.

[13] Julino, J. Analysing page duplication on android. Master's thesis, KIT, 2012.

[14] Kim, S.-H., Jeong, J., and Lee, J. Selective memory deduplication for cost efficiency in mobile smart devices. *Consumer Electronics, IEEE Transactions on 60*, 2 (2014), 276–284.

[15] Prodduturi, R. Effective handling of low memory scenarios in android using logs. Master's thesis, Indian Institute of Technology, 2013.