MARS: Mobile Application Relaunching Speed-Up through Flash-Aware Page Swapping

Weichao Guo, Kang Chen, Huan Feng, Yongwei Wu, *Member, IEEE*, Rui Zhang, and Weimin Zheng, *Member, IEEE*

Abstract—The approach for fast application relaunching on the current Android system is to cache background applications in memory. This mechanism is limited by the available memory size. In addition, the application state may not be easily recovered. We propose a prototype system, MARS, to enable page swapping and cache more applications. MARS can speed up the application relaunching and restore the application state. As a new page swapping design for optimizing application relaunching, MARS isolates Android runtime Garbage Collection (GC) from page swapping for compatibility and employs several flash-aware techniques for swap-in speedup. Two main components of MARS are page slot allocation and read/write control. Page slot allocation reorganizes page slots in swap area to produce sequential reads and improve the performance of swap-in. Read/Write control addresses the read/write interference issue by reducing concurrent and extra internal writes. Compared to the conventional Linux page swapping, these two components can scale up the read bandwidth up to about 3.8 times. Application tests on a Google Nexus 4 phone show that MARS reduces the launching time of applications by 50 $\sim$ 80 percent. The modified page swapping mechanism can outperform the conventional Linux page swapping up to four times.

Index Terms—Mobile system, application launching, page swapping, flash storage

1 Introduction

SMARTPHONES and the applications running on top of them have played an important role in our daily life. Nowadays people are spending more time on smartphones than ever. Undoubtedly responsiveness is a crucial factor for user experiences with mobile applications.

As more and more applications are available on smartphones and users tend to interact with multiple applications, the launching time of applications should not be neglected. It is quite obvious that the launching procedures may include I/O or networking requests, graphics rendering, location sensor, as well as some calculation for initialization. Each of them needs time to be completed. According to a two-month trace from 12 Android users, most mobile application interactions only last about tens of seconds to several minutes. Negative impact on user experience will be inevitable if the launching costs more than the interactive time, e.g., over tens of seconds.

Some previous work attempts to reduce the launching time perceived by users. For example, in [1] authors used prediction and pre-launching to reduce the response time during the start of an application. However, pre-launching cannot reduce the intrinsic time cost by the initialization work during launching, but the perceived time instead. And if the prediction fails, the launching remains sluggish and system resources are wasted as well. The prediction precision must be high or the benefit of pre-launching could be dropped down.

We are also seeking ways to reduce the perceived time for application launching by improving the performance of application relaunching. We learn from observation that, users usually run the same application multiple times and use several applications simultaneously. For example, users might be obsessed with *Facebook* and launch it multiple times for daily use. It is quite possible that users might click the URL shared by their friends. This will usually launch a web browser. And the user will quickly return to the facebook application. In such scenarios, some applications are relaunched again and again. It provides a new opportunity to improve the launching time during the relaunching procedure.

Based on the observation, caching applications in memory is an obvious approach. This is in fact how the system works in current mobile platforms. For example, Android can cache background applications in memory and simply kill them for reclaiming memory in the case of low memory. This is because Android manages the life cycle of every application and provides APIs to save the application specific state. Thus, the state information can be used to restore the application after relaunching. Caching applications makes the relaunching instantly. Restorable state makes the low memory killer harmless.

However, there are still two issues of application caching: the memory size and recoverability of complex applications.

- *W. Guo, K. Chen, H. Feng, Y. Wu, and W. Zheng are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST) Tsinghua University, Beijing 100084, China, Research Institute of Tsinghua University in Shenzhen, Shenzhen 518057, China, and Technology Innovation Center at Yinzhou, Yangtze Delta Region Institute of Tsinghua University, Ningbo 315000, Zhejiang. E-mail: {gwc11, feng-h11}@mails.tsinghua.edu.cn, {chenkang, wuyw, zwm-dcs}@tsinghua.edu.cn.*
- *R. Zhang is with the Department of Computing and Information Systems, The University of Melbourne, Australia. E-mail: rui.zhang@unimelb.edu.au.*

Manuscript received 12 Nov. 2014; revised 19 Apr. 2015; accepted 23 Apr. 2015. Date of publication 30 Apr. 2015; date of current version 10 Feb. 2016. Recommended for acceptance by K. Li.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2015.2428692

As for the limited size of memory on mobile devices, only moderate numbers of applications can be cached. Applications still need to be killed for reclaiming memory. It requires a long time to relaunch an application and restore it to the previous same state, which will surely do harm to user experience. In addition, the assumption of restoring to the previous same state cannot be easily achieved when the applications become much more complex. For example, the APIs provided by Android to store the application state require input as a bundle object (a mapping from string values to parcelable objects). It is hard to efficiently export the running state information of game applications into a bundle object. Thus, we argue that the mobile applications should not be simply killed.

Page swapping can mitigate the above two issues. It provides more memory space for caching applications and the full state of them. Applications now do not need to relaunch from scratch. Instead a load is needed to rebuild the memory image of corresponding application. As the bootstrap code does not need to be executed again, the relaunching could be fast.

It is possible to enable page swapping on Android for fast application relaunching. Unfortunately, Android has disabled page swapping in Linux kernel for the considerations of flash storage and battery life. The most critical reason is the incompatibility between page swapping and Android runtime garbage collection (runtime GC).[1] Another challenge is the speed of page swapping. Most current smartphones are equipped with eMMC[2] for energy efficiency and high-density storage capacity. However, the efficiency of reloading data from swap area in this type of storage is usually slow. Thus, speed up page swapping is significant to improve the performance of application relaunching.

To address these two issues, this paper proposes MARS, a prototype system implementing the application relaunching oriented redesign of Linux page swapping on Android. MARS has embraced several new designs to build a flash-aware page swapping mechanism for application relaunching speedup.

1) MARS has built a runtime GC compatible page swapping mechanism by retrofitting Android runtime and the Linux page swapping.
2) MARS has redesigned the data organization in swap area and carefully controlled the read and write operations involved in swapping to avoid the performance problem.
3) We have tested our prototype system. Compared to the conventional Linux page swapping, the two components, page slot allocation and read/write control in MARS can scale up the read bandwidth to about 3.8 times. Application tests on a Google Nexus 4 phone show that MARS reduces the launching time of applications by about 50 to 80 percent. The modified page swapping mechanism can outperform the conventional Linux page swapping up to four times.

The remainder of the paper is structured as follows. Section 2 describes the motivations we built MARS and the technical challenges encountered. Section 3 presents the design and implementation of MARS. We evaluate MARS in Section 4 followed by related work in Section 5 and conclusion in Section 6.

2 Motivation and Technical Challenges

Applications running on top of smartphones have different characteristics from those running on desktop or servers. Moreover, application relaunching is far more frequent on smartphones than on other platforms. In this paper, MARS is proposed to improve the user experience by enhancing the performance of application launching. In this section, we illustrate our motivations of building MARS by answering the following questions:

- Is slow application launching a serious problem?
- What are the limitations of state-of-the-art solutions?
- Why building MARS instead of using (enabling) the original Linux page swapping?

2.1 Application Launching Analysis

Unlike long running tasks on server platforms, it is quite common to observe plenty of short and frequent interactions of mobile applications. In fact, user experience is more important than performance for mobile applications. A user experience research [2] reveals that 10 seconds is the maximum waiting time before distraction happens. Even for up-to-date smartphones, the launching time of some popular applications reaches up to tens of seconds.

We collect a two-month trace from 12 Android users. An application usage tracer was installed on volunteers' Android phones (with their permissions) from June to August 2014. Fig. 1 shows the CDF of application usage durations. Over 60 percent application interactions last from several seconds to tens of seconds. And about 85 percent of the interactions last less than 5 minutes. Some similar results have been reported in [3]. The results support the view that the durations of mobile interactions are inherently short and applications are frequently relaunched. In addition, we have observed that users tend to use multiple applications simultaneously. This is usually due to the usage patterns of smartphones. Users tend to take out of their smartphones, use them for a while and then put them back.

We choose various popular applications on Android market to quantify the issue of application launching. Fig. 2 shows the *launching time* of these applications. The launching time is measured as the timespan from the time point when a user taps on the shortcut icon of an application to the point the user is able to interact with the application. Half of these applications take more than 5 seconds. Only one of them takes less than 2 seconds before the interaction is available. The results suggest that the launch really takes time and hurts the application experiences.

In order to take a closer look at the work done during the application startup, we use Android *dmtracedump* and *traceview* tool to help profiling the launching process

1. We denote this as runtime GC to make it different from the flash GC (flash block garbage collection performed by the flash translation layer in the flash storage for reclaiming dirty blocks).

2. Embedded Multi Media Card, a form of NAND flash used for main storage on the vast majority of mobile devices.

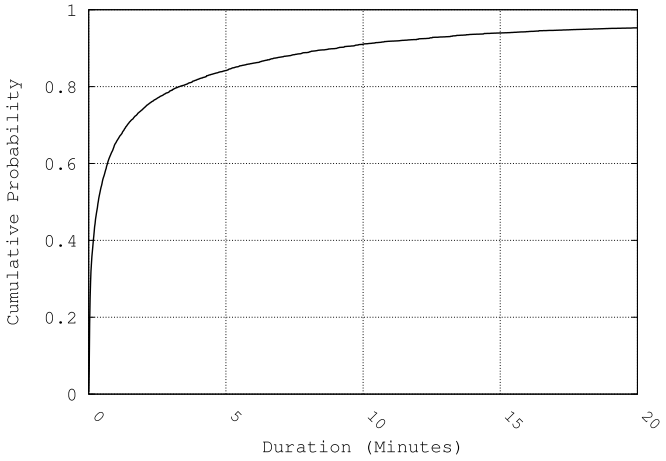


Fig. 1. Application usage for 12 Android users over 2 months. We have developed an Android application, called "Astat", to trace the application interleaving on Android phones. The application usage traces are collected by Astat installed on the smartphones with the permission of the volunteers. There are 25,948 application interactions in the traces. The number of the durations less than 10 seconds is 10,345. There are 16,946 application interactions that last less than 1 minute. Many mobile applications last only a few seconds but used very frequently, such as messaging, weather report, and utility applications. For example, the application Wechat has 1,624 interactions that are less than 10 seconds in the traces.

of an application. Taking the game application of *PvZ 2* as an example, Fig. 3 is a screenshot of its *traceview*. Some representative function calls are marked for clarification. Each parallel timeline represents a thread. For this game application, most work is done by the "GL Thread". There is a lot of initial work using OpenGL library for graphic rendering. Other work like network and I/O is mixed up with the initial process. The traditional network and I/O prefetching schemes [4], [5] cannot solve this issue because most of them overlap with graphic rendering. This shows that the application start procedure is in fact intrinsic and hard to avoid during startup from scratch.

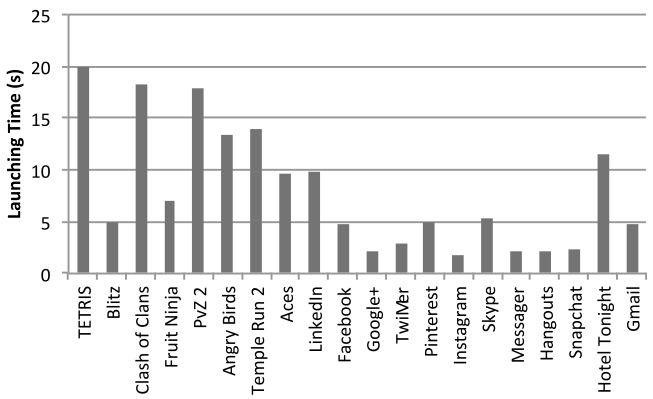


Fig. 2. Launching time of some popular applications on Android. The launching time of applications is measured by instrumenting the Android application framework. We choose the "startActivity" function of Activity Class as the start time point and the "handleResumeActivity" function of ActivityThread Class as the end time point. However, some applications show a splash screen first and call back the "onResume" function of Activity Class prematurely. For example, the game application update scene rendered far behind the activity resumed. We further instrument the InputDispatcher Class to trace the time point where the application is able to interact with the user.

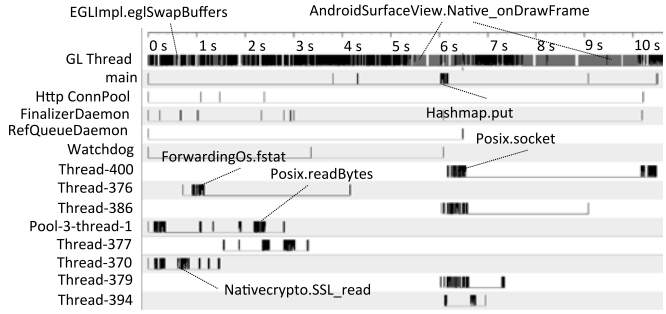


Fig. 3. Game application PvZ 2 startup profiling with Android traceview. Android traceview is a graphical viewer for application profiling by Android dmtracedump. There is a timeline on the top of the panel. Each thread's execution is shown in its own row, with time increasing to the right. Each method is shown in a different color (colors are reused in a round-robin fashion starting with the methods that have the most inclusive time). Some key methods are marked.

2.2 Limitations of Current Solutions

The current design of mobile systems makes it difficult to reduce time of launching or relaunching from scratch. Taking Android as an example, we can see the details in application life cycle as well as memory management. The operating system rather than the developers manages the life cycle of an application. This means that when the foreground application is switched to the background, the application will either be killed or cached based on the current utilization ratio of memory. Android employs a low memory killer for immediately reclaiming memory held by background applications. The low memory killer chooses an application to kill based on the badness value of the process. The badness value is a numeric score calculated by Linux out of memory (OOM) killer, which judges whether a process should be chosen as the candidate for termination. To restore to the exact previous state of a relaunching application after being killed, Android provides APIs for developers to save and restore state information on the system. The assumptions here are: 1) developers should know exactly how to use the system APIs to save state information; 2) it is feasible and efficient to export the state information into data input format as required by these APIs. However, for some applications like games, it is uneasy to restore the previous same state because the APIs (only support bundle objects) are limited and the state to be saved might be huge. This difficulty becomes worse with the increasing complexity of mobile applications. It does bring some burden to application developers who have to carefully consider what kind of state should to be saved. In addition, relaunching application takes the same long time as launching the application from scratch and perhaps longer due to the included restoring work.

Caching background applications in memory for fast application relaunching is a conventional solution in mobile operating systems. This conventional caching approach is limited by the available memory of smartphones. Some applications holding large amount of memory may be killed immediately after caching in background. Thus caching background application to boost application launching has limited effects.

There have emerged some application launching prediction based pre-launching solutions to this issue. A

representative one is FALCON [1], which is a novel solution to improve user experience during the application launching on Windows Phone (WP). The basic idea of FALCON is to predict the timing of using an application. Thus, it can pre-launch the predicted application before the user starts it manually. Through this way, the perceived launching time could be reduced. The prediction is based on context signals and access patterns. As such sets of preloaded prediction rules are diverse for different users, FALCON trains individual context triggers to improve accuracy. FALCON takes both the expected latency benefit and energy cost into account so as to reduce the possibility of unexpected side effects. With FALCON, the limitation of available memory does not matter if the prediction is accurate. However, bootstrapping FALCON to achieve a precision of 80 percent needs over eight months training according to its evaluation. Furthermore, the predictability of application launching varies for different users. These factors limit the effectiveness of FALCON greatly.

Our solution extends the state caching mechanism by using page swapping. Thus, the method is not based on any imprecise information such as prediction. Using page swapping, we extend virtual memory to hold much more background applications, i.e., applications do not need to be killed immediately. Therefore, the performance of page swapping is the key to the practicability of this solution. In the next section, we will discuss the technical challenges on this issue.

2.3 Technical Challenges

To enable page swapping on Android for fast application relaunching is challenging in two aspects, i.e., compatibility and performance.

For the compatibility issue, page swapping in fact conflicts with Android runtime. Android applications are running on Android runtime (Dalvik before Android 4.4 or experimental ART on Android 4.4). Android runtime employs a radical GC. It may be triggered not only when allocating new objects on heap but also when the application is in the background. Page swapping is especially disastrous with Android runtime. A full runtime GC cycle will not be performed until the runtime has run out of allowed heap. However, at that time, garbage objects likely occupy most of the heap. Since the garbage objects are usually not touched until runtime GC is started, those objects should be more likely to be swapped out if page swapping is enabled. When runtime GC finally runs, there would be a ridiculous swap storm, pulling in all these objects only to discover that they are indeed garbage and should be discarded. This shows why page swapping conflicts with runtime GC. As the application life cycle is controlled by the operating system, a low memory killer is more efficient than page swapping in reclaiming memory. In addition, the extra flash writes in page swapping reduce flash storage and battery life. So the original Linux page swapping is turned off on Android. To enable page swapping, we must retrofit it to adapt to Android runtime.

For the performance issue, the swap-in performance of the conventional Linux page swapping on Android is unacceptable for fast application relaunching. This is due to the characteristic of eMMC (Embedded Multi Media Card) on smartphones, the slow small scattered reads as well as the interference of reads and writes.

TABLE 1
Read Bandwidth of Google Nexus 4 eMMC

Record Size	Sequential (KB/s)	Scattered (KB/s)
4 KB	63,071	6,839
16 KB	61,105	11,676
64 KB	64,619	18,658
256 KB	61,648	29,330
1 MB	63,957	55,352

- *Slow small scattered reads*. Due to the unrevealed internal design, a small scattered read is up to ten times slower than a sequential read on eMMC (about 60 MB/s for 4 KB sized sequential read and 6 MB/s for 4 KB sized scattered read as shown in Table 1). However, Linux page swapping does not guarantee that the pages to swap in are adjacent when applications are relaunched on Android. In Linux page swapping, the swap area (a swap partition or a swap file) is divided into a number of page-sized slots. Each slot is called a "page slot" or "swap slot". Even though the Linux page swapping tries to allocate page slots in cluster (a number of continuous page slots), it becomes difficult when swap partition becomes messy. In addition, Linux page swapping uses an approximate Least Recently Used (LRU) algorithm to choose memory pages to swap out. The successive pages to swap out are not always from the same background application.
- *Read/write interference*. 1) Concurrent reads and writes may degrade each other's performance as mingled reads and writes would interrupt the internal pipelining of the flash storage. However, swap-in and swap-out often happens at the same time in Linux page swapping. Relaunching usually requires swap-in operations and the performance cannot be guaranteed under such circumstance. 2) Potential *garbage collection* of flash storage (flash GC) will generate extra reads and writes, which also brings the interference.

To address these compatibility and performance issues, we have designed and implemented MARS to replace Linux page swapping on Android in order to improve the performance of application relaunching.

3 Design and Implementation

3.1 MARS Overview

MARS is a retrofitted Linux page swapping compatible to Android system. The design goal of MARS is to gain swap-in speedup for faster application relaunching. The architecture of MARS is shown in Fig. 4. There are two main components in MARS: page slot allocation and read/write control. The former aims to reduce scattered reads, while the later reduces the interference of flash reads and writes. As the low-end NAND flash storage on smartphones is the target of speeding up swap-in, the main efforts fall into the following aspects:

- Modification on Android runtime. 1) As we have illustrated that Linux page swapping conflicts with

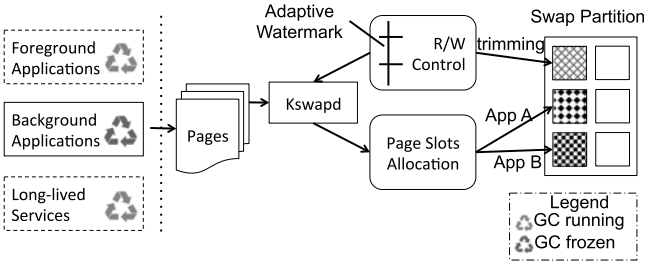


Fig. 4. The architecture of MARS. The isolation of runtime GC and page swapping falls into two parts. One is no GC for background applications, and the other is no page swapping for foreground applications and long-lived services. The page slot allocation tries to generate sequential reads upon swap-ins. The read/write control aims to eliminate the interference of flash reads and writes during page swapping.

runtime GC, we first disable the "idle runtime GC" of background applications to mitigate this issue. 2) And then the application launching events are collected as the hint information to guide the read/write control component. 3) Since page swapping is employed for reclaiming memory in MARS, low memory killer is disabled.

- Modification of Linux page swapping. As conventional page swapping is not optimized for smartphones, we modify several aspects of Linux page swapping. The first is to make the foreground applications and long-lived services bypass the page swapping to avoid potential conflicts with runtime GC. We also reorganize and manage the swap partition to utilize the characteristics of eMMC on smartphones. An adaptive free memory watermark is added to help control the page swapping behaviors. The writeback mechanism of page cache is also modified slightly for reducing flash writes during the application relaunching. Finally, we use background trimming method to reduce the extra reads and writes of flash GC.

3.2 Retrofitting for Compatibility

To resolve the conflicts between Linux page swapping and Android runtime GC, we isolate them rather than make them collaborate with each other. This is due to the design requirements of Android runtime GC. Android runtime GC runs more frequently than Java runtime in order to keep memory footprint small. So we try to make the runtime GC stay simple and fast instead of adding complexity and delay. To isolate Linux page swapping and Android runtime GC, MARS needs to retrofit both Linux page swapping and Android runtime framework.

For background applications, MARS modifies the Android runtime GC behavior to be compatible with page swapping. A full runtime GC on Android usually takes several hundreds of milliseconds. It is a long time that greatly degrades the runtime performance. After Android 2.3, only partial runtime GCs are performed when the application is running. Android runtime shrinks the heap size by doing explicit full runtime GCs periodically when the application is idle, i.e., the application is running in the background. In MARS, the idle runtime GC is frozen until the application comes back to the foreground. Thus, runtime GC will never confuse page swapping.

For foreground applications and long-lived services running on Android runtime framework, MARS filters the memory pages of these processes to isolate Linux page swapping from Android runtime GC. Linux page swapping employs a page replacement algorithm like 2Q [6], an implementation of Least Recently Used approximation. This algorithm maintains two queues of memory pages—the active list and the inactive list. For each page, there are two flag bits—"PG_active" and "PG_referenced". When a page is visited, set its "PG_referenced" flag bit. If its "PG_referenced" flag bit has been set, it means that this page is referenced frequently. If this page is on the inactive list, it should be moved to the active list. "PG_referenced" flag bit will be cleared if the page has not been visited after a period of time. Pages on the active list will be moved out to the inactive list if their "PG_referenced" flag bit has been cleared for a long time. Only the pages in the inactive list are the candidates for page swapping. MARS modifies the "page_referenced" function to always set the "PG_reference" flag bit of the pages for foreground applications and long-lived services. For other processes not running on Android runtime framework, they do not conflict with page swapping. So MARS just takes these processes the same as those of background Android applications.

In addition, background applications may be killed by low memory killer on Android for reclaiming memory. We disable the low memory killer as it is unnecessary and incompatible in MARS.

3.3 Page Slot Allocation

The main drawback of eMMC on smartphones is its poor scattered I/O performance. Previous researchers [7] have reported the poor scattered write performance of SD cards, another kind of low-end NAND flash on smartphones. As MARS focuses on speeding up swap-in, we measured the read bandwidth of the eMMC on a Google Nexus 4 phone using *IOZone*, a well-known I/O benchmark tool for measuring the performance of storage devices or systems. The results shown in Table 1 indicate that the scattered read operation is up to ten times slower than the sequential read operation on the eMMC of Google Nexus 4 phone.

As shown in Table 1, the difference between sequential and scattered read becomes negligible as the record size increases. This makes an insight for speeding up swap-in. The method is to organize pages of an application in a sequential Logic Block Address (LBA) sequence in swap area and make them swap in together.

Linux page swapping already tries to cluster swap pages by allocating them sequentially in swap partition. However, this cannot guarantee that the read operations are sequential while doing swap-in considering how the original Linux page swap allocates a page slot. The procedures of allocating a page slot are listed below.

1) If there is any free page slot from the current cluster of continuous free page slots, the page slot will be allocated.
2) Try to find a new cluster meeting the requirements if the current cluster has no free page slots, and then allocate a new page slot from the new cluster.
3) Otherwise simply allocate a free page slot.

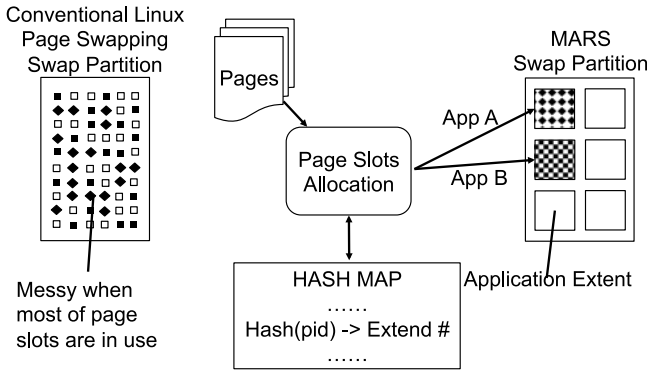


Fig. 5. Page slot allocation. The size of an application extent is 1 MB. One application may have more than one application extents. But there is only one application extent that is currently serving the swap-outs for an application in the hash map. The application extent should be removed from the hash map when all the page slots have been allocated and a new free application extent should be added in the hash map.

This scheme is in fact a straightforward way to generate sequential swap-in reads based on information of reference locality. But it does not fit MARS for the following reasons.

1) The allocation method cannot allocate sequential page slots when lots of page slots are in use. This is not the case for traditional servers as swap area is only used when working set is larger than the main memory. Considering performance, server applications rarely use swap area. However, in the context of MARS, we want most of the swap area is in use as we intend to hold background applications as many as possible.
2) We have shown that users intend to use multiple applications simultaneously. This will reduce the locality of memory reference.

The page slot allocation scheme of MARS is shown in Figs. 5 and 6. MARS reorganizes the page slots in the swap area into application extents. An application extent is an aligned cluster of continuous page slots. The default size of an application extent is 1 MB. To swap out a page, MARS finds the process that the page belongs to and allocates a page slot from the corresponding application extent. If there is no free page slot in the application extent, MARS adds a new application extent for that process. By using different extents for different applications, when an application is switched to the foreground, the extent for the corresponding application will be loaded into the memory. Thus, MARS forces the swap-in read operations to be sequential.

The mappings of applications and application extents are recorded in a key-value table as shown in Fig. 5. The key is a hashed process ID of an application and the value is the pointer to the current extent of corresponding application. Page slot allocation is also responsible for reclaiming the application extents that has no valid page slots. These invalid application extents will be trimmed (further illustrated in Section 3.4) and then placed in a free extent list for the upcoming application extent allocations.

3.4 Read/Write Control

Previous researchers [8], [9], [10] pointed out that concurrent reads and writes interfere with each other in flash memory based Solid State Drives (SSD) and cause performance degradation severely. This issue is also serious in eMMC. The main reason is that mingling reads and writes would interrupt the internal pipelining of a *flash memory plane* in the flash storage.

We measure the I/O performance of concurrent 64 KB reads and writes on the eMMC of Google Nexus 4 to verify the phenomenon of flash storage. The results shown in Table 2 reveal that the interference of concurrent reads and writes on the eMMC interrupt each other and greatly degrade the I/O performance. However, swap-in and

Algorithm:PageSlotAllocation

Input: $page$ to swap out, $flags$ to set in swap map
Output: the allocated page slot number $offset$

```
1: offset ← 0
2: key ← get_process_id(page)
3: possible_extents ← extents_hash[key]
4: for extent ∈ possible_extents do
5:     if extent.key = key then
6:         if extent.free_pages = 0 then
7:             hash_del(extent)
8:             list_add_tail(used_extents, extent)
9:             if list_empty(free_extents) then
10:                break
11:            end if
12:            extent ← free_extents.next
13:            extent.key ← key
14:            list_del(free_extents, extent)
15:            hash_add(extents_hash, extent)
16:        end if
17:        offset ← get_cur_page(extent)
18:        swap_map[offset] ← flags
19:        break
20:    end if
21: end for
```

Fig. 6. Page slot allocation algorithm. The swap map is used to record the state of each page slot, such as used, scanning or discarding. If this algorithm finds out an available page slot, it returns the index of the page slot, "offset", which is a positive integer. It returns 0 if there is no free page slots. The function "get_process_id" returns the ID of the process which the page belongs to. The function "get_cur_page" returns the first unallocated page slot sequentially and maintains the free page counts of the application extent. In our implementation, the hash map has 1,024 buckets by default and stores all the entries that hash to the same bucket in an array. The algorithm used here has a constant complexity and brings neglected overhead. Thus, the performance is close to the original page slot allocation.

TABLE 2
Concurrent Reads and Writes Bandwidth (in KB/s) of Google Nexus 4 eMMC

	Seq. Write	Scat. Write	None
Seq. Read	34,071+14,153*	37,084+3,866*	64,619
Scat. Read	11,624+4,420*	4,503+4,359*	18,658
None	16,386	5,758	

Note: "Seq." is sequential and "Scat." is scattered.
The numerical value on the left side of "+" denotes read bandwidth, and the numerical value on the right side of "+" denotes write bandwidth.

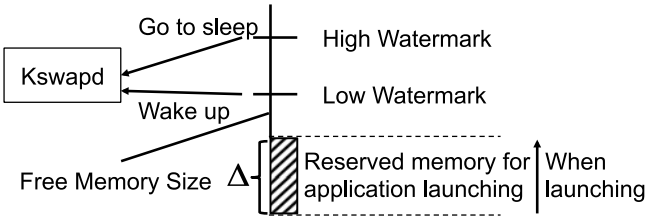


Fig. 7. Adaptive free memory watermark. The reserved memory will complement the free memory pages for swap-in. This part of memory will never be used when there is no application launching. The reserved memory size Δ is determined by the maximum value of all the background applications' working set size.

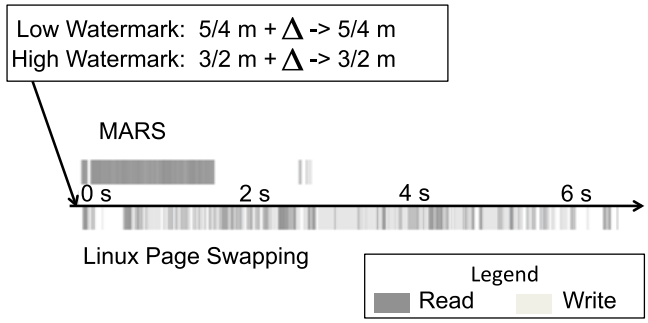


Fig. 8. I/O trace when application relaunching. For MARS, the two watermarks are reduced Δ when the application PvZ 2 is relaunching. As the original page swapping the two watermarks are fixed, this small change makes a big difference of the I/O pattern of page swapping. There are two phases of page swapping caused by application relaunching in MARS. One is the swap-in phase during application relaunching, and the other is the swap-out phase afterwards. The swap-in and swap-out is performed simultaneously in the original page swapping.

swap-out are often triggered simultaneously in Linux page swapping. A common case is to swap out pages to make room for some pages to swap in. As discussed before, this will greatly reduce the performance.

MARS decouples swap-out and swap-in by using a novel swapping control mechanism. Asshown in Fig. 7, MARS controls the *kswapd*, kernel swap daemon in Linux, with adaptive free memory watermarks—low watermark and high watermark. In Linux kernel, the watermarks are determined by the *min_free_kbytes* m, which is the pool size of reserved page frames in kernel. (Low watermark is $5/4 \cdot m$ and high watermark is $3/2 \cdot m$ by default). When the free memory is lower than the low watermark, the *kswapd* threads will be waked up to reclaim memory. The duty of *kswapd* is to keep the free memory higher than the high watermark. The *kswapd* also tries to reclaim memory periodically. In MARS, when there is no application launching, the low and high watermarks are increased by an variation Δ, which is the maximum value of the total swap-in size of all the background applications. Δ is usually less than 100 MB which should not induce performance degradation for a smartphone with 2 GB RAM. The total swap-in size of a background application is actually its swap-out size. In such a way, MARS reserves enough free memory for any background application relaunching. During application relaunching, the low and high watermarks are reset as the original value. Thus, the *kswapd* should not be triggered to swap out upon swap-in. In addition, we also modify *pdflush* threads, the page cache synchronization daemons in kernel, to turn off the writeback of dirty pages during application launching.

Taking the application *PvZ 2* relaunching as an example, the interleaving of swap-in and swap-out when application relaunching is different under MARS and the conventional Linux page swapping. We employ Android systrace tool[3] to record the I/O activities of the eMMC on the phone when the application *PvZ 2* is relaunching. The interleaving of reads and writes is shown in Fig. 8.

During application relaunching, MARS sets the low/high watermark to the original ones. The reserved memory pages for application launching are then used for swap-in. The relaunching time of MARS is about 1.6 seconds. The recorded I/O activities in 2 to 4 seconds of the timeline in MARS are I/O operations generated by the application. For the conventional Linux page swapping, frequent I/O interleaving reduces the performance of swap-in greatly.

We modify the Android application framework to notify the Linux kernel at each time point of application launching. The application launching state is indicated by Android runtime through the Linux proc filesystem.

Besides the interference of reads and writes generated in software level, the flash GC may also induce internal reads and writes that will interrupt the normal I/O operations. The flash GC is caused by the characteristic of flash storage. Flash storage cannot update data in place. Instead, they must first erase a large flash block,[4] and then write to pages within the erased block. The old flash pages are no longer needed (also called stale pages). Before erasing a flash block, the valid flash pages (not stale) should be copied to another flash block and remapped by Flash Translate layer (FTL). This is why the process of flash GC involves reading and rewriting data internally.

In MARS, we employ the *discard*[5] command to inform the FTL about which flash pages should be invalid. As the *discard* command itself costs the bandwidth of the eMMC, this command should not be issued at any time. MARS issues *discard* commands in the background when no page swapping is performed currently. The *discard* requests of adjacent LBAs are merged to make a batch operation.

A simple and low overhead scheme for discarding the invalid flash pages is to discard the invalid application extents, called "trim". As this task should be done periodically, we assign it to the *kswapd* service. When *kswapd* has no page swapping tasks, it will execute the function of discarding the invalid application extents before going to sleep.

3.5 Implementation

We built our prototype system MARS on Android 4.4 with the kernel version of ARM Linux 3.4. There are about 200

3. The Systrace tool combines data from the Android kernel such as the CPU scheduler, disk activity, and application threads to generate an HTML report that shows an overall picture of an Android devices system processes for a given period of time.

4. A flash block is composed of a number of flash pages.

5. *Discard* is a sub-operation defined in eMMC 4.5 Standard [11]. It allows an operating system to inform the FTL which blocks of data are no longer considered in use and can be wiped internally.

LOCs (Lines Of Codes) coding efforts within the Linux kernel and about 40 LOCs of modifications on Android runtime and application framework. The implementation of MARS entails three main parts: retrofitting for compatibility, page slot allocation and read/write control. It is recommendable to Android community for its portability and ease of maintenance.

The implementation of disabling background runtime GC and low memory killer are located in the Android application framework module and the *lowmemorykiller* kernel module. The application launching state is traced in Android application framework. And the kernel is notified through the *proc* file system. Besides the writes of swap-out, the page cache sync writes of pdflush threads should be blocked during application launching. The default size of the hash table we choose is 1,024 with considering both the lookup efficiency and memory overhead.

4 Evaluation

We evaluate the performance of MARS against the original Linux page swapping and the current solution on Android. The evaluation aims to answer the following three key questions:

- What is the improvement from each component of MARS for swap-in performance compared to the conventional Linux page swapping?
- Does MARS architecture design improve relaunching performance among a variety of mobile applications?
- Is MARS an effective solution to speed up mobile application relaunching for daily use?

The experimental setup and methodology are described in Section 4.1. Then we present our evaluation to answer the three questions in Sections 4.2, 4.3, 4.4 respectively.

4.1 Methodology

Devices and system settings. All the evaluations are performed on a Google Nexus 4 phone equipped with Qualcomm Snapdragon^{TM}S4 Pro CPU, 2 GB RAM, and 16 GB eMMC internal storage. Linux kernel is compiled with the option "CONFIG_BLK_DEV_IO_TRACE" enabled for tracing block level I/O. We use a 2 GB swap file as the swap area. The *swappiness* value is set to *60* by default. Swappiness [12] is a parameter that controls the relative weight given to swap out runtime memory, as opposed to dropping pages from the system page cache. It can be set to values between 0 and 100 inclusively. A low value causes the kernel to avoid swapping, and a higher value causes the kernel to use swap space. The *page_cluster* is set to *3* by default, which means that 8 (2^3) pages will be paged in together [12]. The settings of two parameters settings can impact the page swapping behaviors. The runtime is set to default on Android 4.4, Dalvik virtual machine, in all our evaluations.

Benchmarks. The micro-benchmarks test the I/O latency and bandwidth in page swapping for evaluating the effectiveness of each MARS component. These two micro-benchmarks are measured by *blktrace* [13] tools. The trace data are collected in memory to minimize the interference caused by tracing itself. Then the collected trace is processed with *blkparse* [13] and *btt* [13]. The application benchmark is the relaunching time of applications for evaluating the performance of MARS. The relaunching time is measured by instrumenting Android runtime framework. The timespan of an application relaunching is from the time point when the application icon is tapped to the time point when the user is able to interact with the application. To evaluate the effectiveness of MARS for daily use, we replay the collected application usage traces on the test smartphone under baseline, the conventional Linux page swapping and MARS. Then we collect all the application relaunching times. The applications involved in all the evaluations are picked from the most popular ones of various categories on Android market. The picked applications are listed as follows.

- *Social*. Facebook, LinkedIn, Google+, Twitter, Instagram, Pinterest, Fancy, Banjo, and Beautylish.
- *Game*. Fruit Ninja, PvZ 2, Temple Run 2, Angry Birds, Blitz, and Aces.
- *Utility*. Hotel Tonight, Expedia, The Weather Channel, ES File Explorer, and Google Maps.
- *Productivity*. Evernote, Gmail, QuickOffice, and Google Drive.
- *Communication*. Skype, Hangouts, Facebook Messenger, and Snapchat.

For all our experiments, we report results averaged over five different runs. All the evaluations are performed on the phone with good Wi-Fi connection.

4.2 Micro-Benchmarks

Micro-benchmarks are used to test the effectiveness of the two components in MARS. One is the page slot allocation and the other is the read/write control. The former is designed for generating more sequential reads to increase read bandwidth of eMMC during page swapping. We can observe the effectiveness brought by page slot allocation by measuring the occupied bandwidth (throughput) of swap-in. The read/write control of MARS aims at reducing the interference of concurrent reads and writes for good swap-in performance. The latency of read or write operations can reflect the interference of parallel I/O sequences. Without the interrupts of concurrent writes, the latency of read operations should be kept low.

We instrumented the Linux kernel to trace the swap-in/swap-out records. Each record entry contains: timestamp, action (swap-in or swap-out), page slot number and the corresponding process name. We implemented a swap-in/swap-out trace replayer that can simulate the original Linux page swapping and MARS on the test smartphone. We first traced the swap-in/swap-out on the smartphone under normal usage. Then we replayed the same trace with the original Linux page swapping, MARS without read/write control, and MARS with read/write control. The latency and occupied bandwidth under different page slot allocation schemes and read/write control strategy were measured by *blktrace* tools.

The results of occupied bandwidths of swap-in are shown in Fig. 9. For the Linux page swapping, the read bandwidth of swap-in is about 6.94 MB/s. This is due to the slow small-scattered read operations and the interference of concurrent write operations. With added component of page slot allocation (MARS without read/write control), the

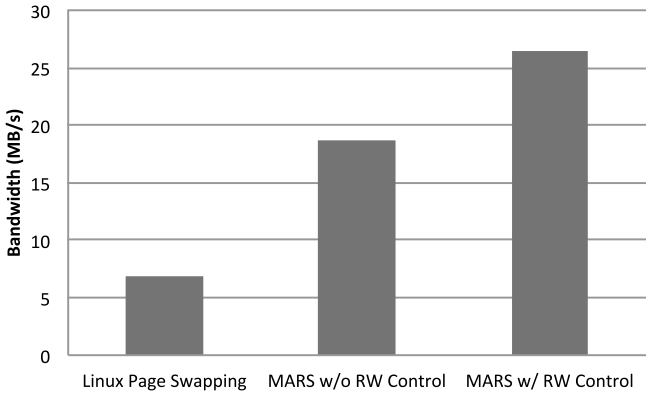


Fig. 9. The read bandwidth when page swapping. The read occupied bandwidth is obtained by the statistical processing of "blkparse" and "btt" with the I/O operations collected by "blktrace". It reflects the throughput of swap-in. For eliminating the noise of "blktrace" itself, the trace data are kept in memory without writing to the persistent storage.

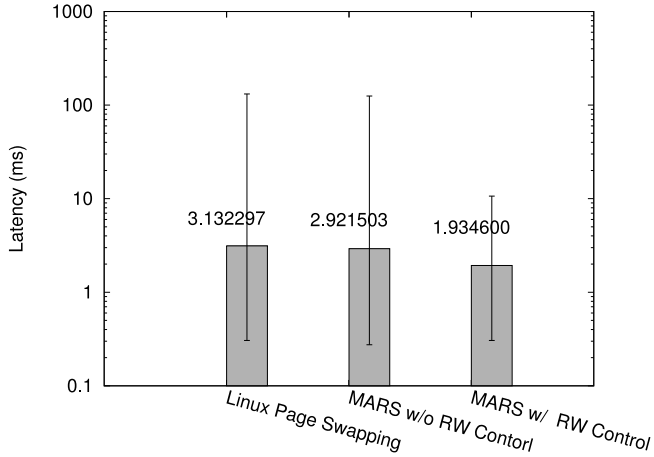


Fig. 10. The read latency when page swapping. The read latency is given by the "blkparse" with parsing the block I/O traces.

swap-in throughput of MARS is 18.76 MB/s. This indicates that the page slot allocation scheme of MARS is much better than the Linux page swapping for improving swap-in performance. Enhanced with read/write control, the swap-in throughput of MARS is increased to 26.48 MB/s. This demonstrates that read/write control can reduce the average latency of read operations.

Fig. 10 shows the results of block I/O traces. The read latency of eMMC varies from less than 1 millisecond to more than one hundred milliseconds. For Linux page swapping, the average read latency is about 3.1 milliseconds, and ranges from 0.3 to 131 milliseconds. This is due to the read/write interference we illustrate in Section 3. For MARS without read/write control, the situation of interference is similar to the Linux page swapping. The read latency ranges from 0.3 to 125 milliseconds, and the average is 2.9 milliseconds. MARS with read/write control shows its effectiveness by reducing 1 millisecond read latency. The average is about 1.9 milliseconds. More importantly, it reduces the variation of read latency. The maximum read latency is reduced to 10.6 milliseconds.

In summary, the design of MARS architecture achieves our initial goals. The two components of MARS, page slot allocation and read/write control, have tackled the technical challenges demonstrated in Section 2. They are specially designed for application relaunching and comparatively different from the conventional Linux page swapping. The page slot allocation of MARS improves the read performance on eMMC. The read/write control reduces the interference of write operations involved in the swap-in procedure. They both can speed up the swap-in greatly.

4.3 Application Benchmark

In this section, we evaluate the performance of MARS among a variety of applications on Android. The applications are the most popular application listed in Section 4.1. We measure the relaunching time of these applications under MARS and the Linux page swapping. The launching time of application starting from scratch is measured as the baseline. For MARS and the Linux page swapping, the system status is kept nearly the same when relaunching occurs. There are about 75 percent page slots are in use. The amount of memory pages of the application that have been paged out varies from 20 MB to more than 60 MB. Fig. 11 shows the relaunching time clustered histogram of the applications we measured. Fig. 12 shows the relaunching time CDF of all the applications on the most popular application list.

As shown in Figs. 11 and 12, MARS speeds up the application relaunching time by 2 to 4 times. The relaunching time under MARS is less than 2 seconds over 80 percent applications on the list. The application relaunching under Linux page swapping is slower than that under MARS without exception. Some of the results are even worse than the baseline. For those applications with launching time larger than 10 seconds, the advantage of MARS is obvious.

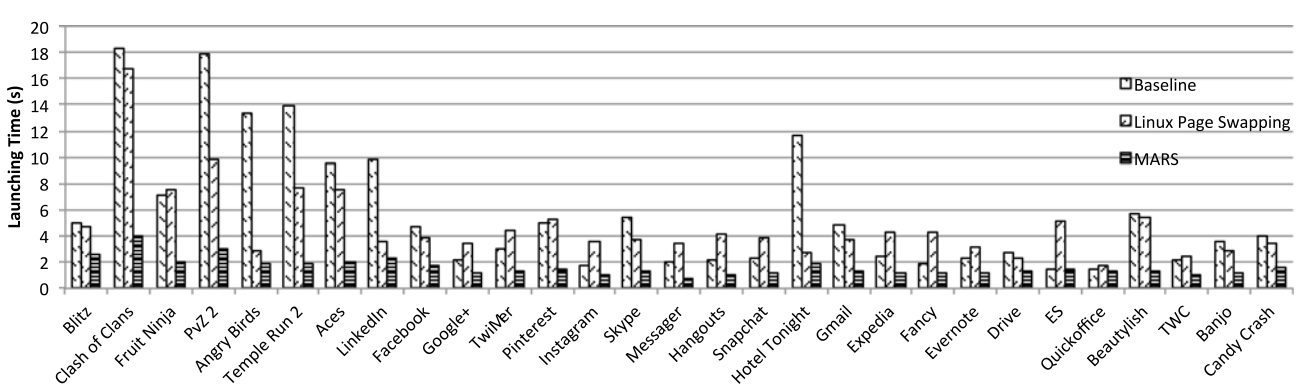


Fig. 11. Relaunching time under Linux page swapping and MARS. The relaunching time can be measured in the same way as the launching time. The baseline is the launching time of applications on Android without MARS.

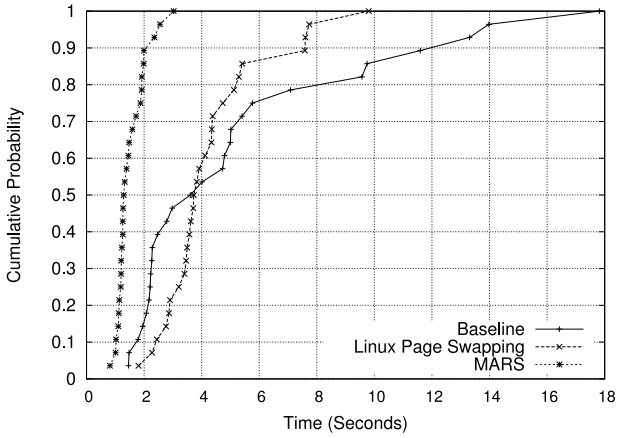


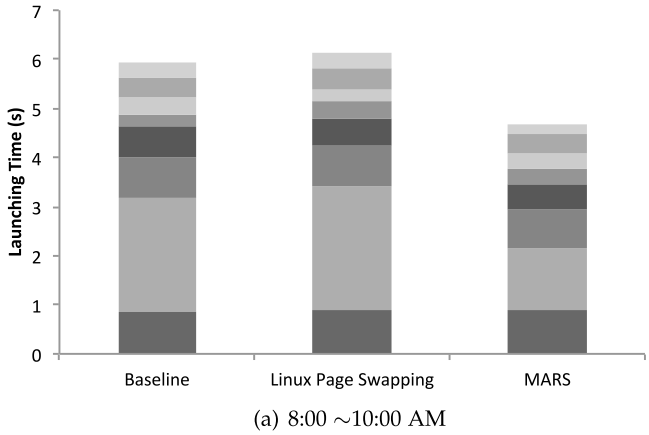
Fig. 12. CDF of applications relaunching time.

Overall, MARS outperforms the Linux page swapping and improves the application relaunching on smartphones.

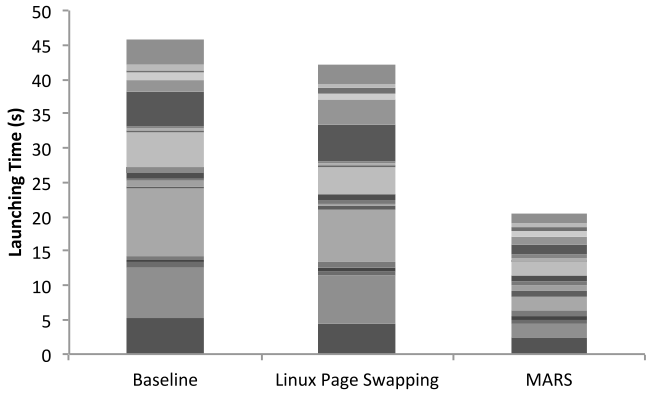
4.4 MARS Effectiveness for Daily Use

In this section, we evaluate the effectiveness of MARS for daily use. A large scale of Android application usage data collected by Yahoo [14] shows that a typical Android device has an average of 95 applications installed, and 35 of them are actually used each day. The data shows that Android users interact with applications for an average of 100 times a day and that up to six applications an hour are accessed at peak hours usually in the late afternoon. We wonder whether MARS can be always effective to reduce the application launching time in such a situation. We also deploy a two-month application usage trace on Android phones of 12 volunteer (mentioned in Section 2). We choose a one-day trace of a representative individual from the collected traces and replay the interactions manually on the test smartphone with no page swapping, the conventional Linux page swapping, and MARS. The interactions happen in two time periods, 8:00 to 10:00 AM and 8:00 to 10:00 PM. The other interactions are used for warming up the test smartphone to the preposition for testing. The launching times are collected the same as the application benchmark. Fig. 13 shows the stacked histogram of each test case.

As shown in Fig. 13, the effectiveness of MARS is different. From 8:00 to 10:00 AM, there are only seven interactions and three used applications. And the second interaction dominates the results shown in Fig. 13a. It is a weather application which is usually used in morning according to this individual's convention. The effectiveness of MARS is significant from 8:00 to 10:00 PM. There are 21 interactions and eight used applications. As shown in Fig. 13b, MARS reduces about 50 percent of the launching time. The effectiveness of MARS depends on the usage of each individual. MARS is effective for the individuals who use smartphone a lot. For the ones who rarely interact with mobile applications, the effectiveness of MARS would be visible after a long period of time.



(a) 8:00 ∼10:00 AM



(b) 8:00 ∼10:00 PM

Fig. 13. Effectiveness for daily use. In this stacked histogram, each block in each column represents the launching time of an application interaction. For this individual we chose, the smartphone is used more heavily at night than in the morning. In the 8:00 ∼ 10:00 AM, there is only one application relaunching speeded up by MARS. The effectiveness of MARS is obviously in the 8:00 ∼ 10:00 PM.

4.5 Overhead

The overhead of MARS mainly includes extra storage capacity used, flash storage wearing out, and power consumption. As the flash storage capacity on smartphones becomes more and more plenty, e.g., 8 or 16 GB eMMC on Google Nexus 4, the extra storage capacity cost of MARS, and 1 or 2 GB for swap partition is acceptable. In this section, we focus on the additional flash storage wear rate and power consumption of MARS.

Obviously, the additional flash writes are produced by page swapping in MARS. The number of swapped pages is determined by the interleaving and working set of applications, i.e., relaunching an application from background when parts of its memory pages are swapped out and allocating memory for a running process when there is no enough free pages held. However, not each application interleaving causes the additional flash writes. For the evaluation of daily use in Section 4.4, there are 1 of 7 and 7 of 21 application relaunchings cause additional flash writes for 8:00 to 10:00 AM and 8:00 to 10:00 PM respectively in one day. The working set of applications on smartphone varies from about 10 MB (e.g., messaging application) to over 100 MB (e.g., game applications). Fig. 14 shows the working set of some Android applications measured on the test phone in our evaluation.

The write endurance of Multi-Level Cell (MLC) NAND flash is usually 3,000 to 10,000 write/erase cycles [15]. For eMMC on Google Nexus 4 [16], [17], its MLC NAND Flash is approximately 10,000 write/erase cycles. According to a

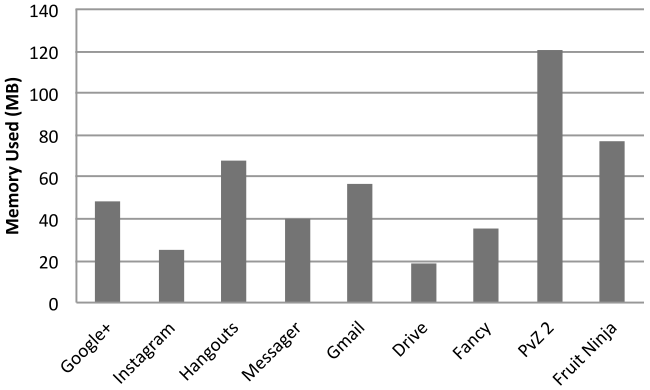


Fig. 14. Some Android applications memory usage. The memory used of applications are the PSS (Proportion Set Size) of calculated by Android 'procrank' tool, which can be seen as the applications' working set size.

block I/O trace of daily used Android devices [18], there is about 1,837 MB data written for a period of 24 hours. On the wearing overhead of MARS, considering the worst case in our daily usage evaluation, there are about 10,000 MB (100 times of application interleaving a day [14] and swapping 100 MB data for each application) data written for page swapping in one day. Even in such a case, equipped with the wear leveling techniques, the write endurance of MLC NAND flash would enable the applications read and write data for more than tens of years on smartphones like our test phone, Google Nexus 4. Thus, the write endurance overhead of MARS is acceptable.

The additional flash writes induced by MARS cost additional power simultaneously. According to the power consumption measurement of read/write operations in a smartphone [19] and our validation with PowerTutor [20], Android requires 70 to 3,300 μJ/KB to complete the I/O operations, including sequential/scattered reads or writes. The main parts of power consumption are CPU and DRAM energy cost instead of the eMMC device in the whole I/O stack. Sequential operations are more energy-efficient than the scattered ones. For the case in our daily usage evaluation (about 20 percent application interleaving caused page swapping, i.e., 2,000 MB data read and written for page swapping in one day), the daily energy overhead of MARS mainly contains two parts: the read energy cost of swap-in, and the write energy cost of swap-out.

Taken 70 μJ/KB for the read energy cost and 360 μJ/KB for the write energy cost, the daily energy overhead of MARS is 880.64 J, which is about 3.07 percent of our test phone's battery power. Our test phone embeds a 3.8 V, 2100 mAh battery [21]. It means that MARS induces about 3 percent energy overhead in the daily use of smartphones like Google Nexus 4. This is not a significant amount for the whole energy cost in daily usage.

As discussed above, MARS does introduce potential extra costs on smartphones including more storage capacity, flash storage wearing out and power consumption. We think that as the current flash storage spaces are relative large and becoming even larger. The extra storage needed for page swapping is small and should not be an issue. The reads and writes during page swapping degrade the life of flash storage. However, with the improvement of flash controller techniques like wear leveling and NAND flash upgrade, the lifetime of flash far exceeds the service life of a smartphone. It is tolerable to have these additional write operations introduced by MARS. Based on an analysis [22] of power consumption in a smartphone, the main part of energy cost on smartphones is LCD screen display. Flash read/write operations cost only a small portion of battery power. In addition, MARS can speedup the application relaunching which can save the energy with reduced consumption of LCD screen display. Thus, we believe that the energy cost of MARS is negligible.

5 Related Work

There are various work related to MARS design and implementation. They can be categorized into similar work on desktop PC, mobile application launching prediction, virtual memory GC improvement as well as the performance of NAND flash based storage optimization.

Studies for fast application launching with system optimizations on desktop PC are distinguished to various circumstances and requirements [5], [23], [24]. A representative one is SuperFetch [24], an extension of Windows Vista. It continually analyzes application behavior and usage patterns and attempts to load commonly used libraries and application components into memory before they are required. As the memory size of smartphones is usually limited, SuperFetch does not suit for smartphones. Addressing the inefficiency of the HDD-aware application launchers on SSDs, Joo et al. [5] proposed an SSD-aware application prefetching scheme, which tries to overlap the computation time with the SSD access time during application launching. By contrast, MARS caches the applications to skip the computation of initial process and focuses on speeding up page swap-in on low-end NAND flash storage.

Mobile users intend to install more and more applications on their smartphones with the increasing numbers of available mobile applications. A bunch of application launch prediction approaches [1], [25], [26], [27] have been proposed. The prediction results can be used for pre-launching applications to reduce the perceived launching time. FALCON [1] predicts the timing of application launching using user context such as locations and temporal access patterns. FALCON also considers both cost and benefit to choose the most appropriate prediction. On the contrary, APPM [27] only uses historical application usages in prediction without any power-hungry or privacy-sensitive contexts. We believe that the prediction precision and/or long-term training requirements limit the effectiveness of such prediction based approaches. Thus, MARS does not rely on any imprecise information.

The poor interaction of garbage-collected applications with virtual memory systems has been studied in [28] and [29]. BC [28] cooperates with the virtual memory manager to perform in-memory full-heap collections using summary information ("bookmarks") recorded from evicted pages. To address the swap thrashing when Java virtual machines (JVMs) cannot adapt their application heap sizes to fit in RAM, CRAMM [29] dynamically adjusts heap sizes to maximize throughput while minimizing page swapping. Such approaches do not fit for smartphones,

and can degrade application responsiveness seriously. The heap size of Android runtime is only tens of MBs. There is no need to swap pages for applications running in foreground. In MARS, we isolate the runtime GC and page swapping, i.e., no page swapping for long-lived services or foreground applications, and no runtime GC for background applications.

The design of MARS has embraced past work on investigating the use of NAND flash storage for virtual memory and page cache in Linux. FlashVM [30] is a core virtual memory subsystem built in the Linux kernel that uses dedicated flash for paging. Similar to MARS, FlashVM also employs the *discard* command to reduce the extra internal writes on flash devices. The design of background flash trimming in MARS draws on the usage of *discard* command in FlashVM. SpatialClock [31] is a buffer cache replacement algorithm designed for mobile devices. It addresses the well-known low write-throughput problem for small, scattered writes on micro SD cards and transfers scattered writes to sequential ones. SpatialClock produces sequential writes to improve the performance of buffer cache replacement. By contrast, MARS generates sequential reads to speed up swap-in. Some other flash-aware cache replacement schemes have been proposed. FOR [32] focuses on the asymmetric read and write operation time of flash storage, calculates the Inter Operation Distance (IOD) [33] and *recency* values of read and write operations separately to get the replacement weight of each page. BPLRU [34] enhances the random write performance of flash storage by improving write buffer management with block-level LRU scheme and page padding technique in the FTL. CFLRU [35] tries to evict a clean page rather than a dirty page for reducing costly writes. The replacement scheme in MARS is still 2Q [6] LRU. Taking replacement cost into account should be a complementally future work.

6 Conclusion

This paper presents the design, implementation, and evaluation of MARS, a retrofitted Linux page swapping for fast application relaunching on smartphones. MARS tackles the conflicts between the Linux page swapping and Android runtime GC. To speed up swap-in upon application relaunching, MARS adapts the Linux page swapping for the performance characteristics of low-end flash storage on smartphones. Addressing the issue of slow small-scattered reads, MARS reorganizes the swap area and employs a new page slot allocation scheme for generating sequential reads. Addressing the issue of read/write interference, we have developed the read/write control component for reducing the concurrent writes upon swap-in. Compared to the conventional Linux page swapping, the component of page allocation can scale up the read bandwidth to about 2.7 times. In addition, read/write control can bring another improvement and scale up the read bandwidth to 3.8 times.

We have demonstrated that using page swapping to extend the conventional caching scheme for fast application relaunching is practicable in smartphones. Micro-benchmarks have verified that both components of MARS achieve the initial design goals. Evaluation results show that MARS reduces the launching time of Android applications by 50 to 80 percent.

Acknowledgments

This work is supported by Natural Science Foundation of China (61433008, 61373145, 61170210, U1435216), National High-Tech R&D (863) Program of China (2012AA012600), Chinese Special Project of Science and Technology (2013zx01039-002-002). This work is partially supported by Australian Research Council (ARC) Discovery Project DP130104587 and Australian Research Council (ARC) Future Fellowships Project FT12010083. Yongwei Wu is the corresponding author.

References

[1] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu, "Fast app launching for mobile devices using predictive user context," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Serv.*, New York, NY, USA, 2012, pp. 113–126.

[2] J. Nielsen. (2009). Powers of 10: Time scales in user experience. [Online]. Available: http://www.nngroup.com/articles/powers-of-10-time-scales-in-ux

[3] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "LiveLab: Measuring wireless networks and smartphone users in the field," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 3, pp. 15–20, Jan. 2011.

[4] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson, "Informed mobile prefetching," in *Proc. 10th Int. Conf. Mobile Syst., Appl., Serv.*, New York, NY, USA, 2012, pp. 155–168.

[5] Y. Joo, J. Ryu, S. Park, and K. G. Shin, "FAST: Quick application launch on solid-state drives," in *Proc. 9th USENIX Conf. File Stroage Technol.*, Berkeley, CA, USA, 2011, pp. 19–19.

[6] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. 20th Int. Conf. Very Large Data Bases*, San Francisco, CA, USA, 1994, pp. 439–450.

[7] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. 10th USENIX Conf. File .Storage Technol.*, Berkeley, CA, USA, 2012, pp. 17–17.

[8] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Washington, DC, USA, 2011, pp. 266–277.

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

[10] C. Dirik and B. Jacob, "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, New York, NY, USA, 2009, pp. 279–289.

[11] (2011). V. Tsai, emmc v4.41 and v4.5 architecture for high speed functions and features. [Online]. Available: http://www.jedec.org/sites/default/files/Victor_Tsai.pdf

[12] D. Bovet and M. Cesati, *Understanding the Linux Kernel*. Oreilly & Associates Inc, Sebastopol, CA, US, 2005.

[13] (2014). Blktrace [Online]. Available: http://linux.die.net/man/8/blktrace

[14] (2014). Yahoo, How android users interact with their phones?. [Online]. Available: http://yahooaviate.tumblr.com/image/95795838933

[15] (2014). Flash memory write endurance. [Online]. Available: http://en.wikipedia.org/wiki/Flash_memory#Write_endurance accessed 22.10.2014

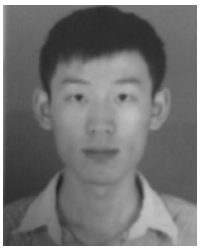
[16] (2014). Toshiba mlc nand. [Online]. Available: http://toshiba.semicon-storage.com/ap-en/product/memory/nand-flash/mlc-nand.html accessed 22.10.2014

[17] DSstar. (2014). Toshiba releases research on mlc nand flash memory. [Online]. Available: http://www.taborcommunications.com/dsstar/04/0518/108104.html accessed 26.10.2014

[18] K. Lee and Y. Won, "Smart layers and dumb result: IO characterization of an android-based smartphone," in *Proc. 10th ACM Int. Conf. Embedded Softw.*, New York, NY, USA, 2012, pp. 23–32.

[19] J. Li, A. Badam, R. Chandra, S. Swanson, B. Worthington, and Q. Zhang, "On the energy overhead of mobile storage systems," in *Proc. 12th USENIX Conf. File Storage Technol.*, Berkeley, CA, USA, 2014, pp. 105–118.

[20] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. 8th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codes. Syst. Synthesis*, New York, NY, USA, 2010, pp. 105–114.
[21] (2014). Nexus 4 teardown. [Online]. Available: https://www.ifixit.com/Teardown/Nexus+4+Teardown/11781
[22] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, Berkeley, CA, USA, 2010, pp. 21–21.
[23] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud, "Intel turbo memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems," *Trans. Storage*, vol. 4, no. 2, pp. 4:1–4:24, May 2008.
[24] MICROSOFT. (2008). Windows PC Accelerators. [Online]. Available: http://www.microsoft. com/whdc/system/sysperf/perfaccel.mspx
[25] C. Shin, J. Hong, and A. K. Dey, "Understanding and prediction of mobile application usage for smart phones," in *Proc. ACM Conf. Ubiquitous Comput.*, New York, NY, USA, 2012, pp. 173–182.
[26] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "Falling asleep with angry birds, facebook and kindle: A large scale study on mobile application usage," in *Proc. 13th Int. Conf. Human Comput. Interaction Mobile Devices Serv.*, New York, NY, USA, 2011, pp. 47–56.
[27] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin, "Practical prediction and prefetch for faster access to applications on mobile phones," in *Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput.*, New York, NY, USA, 2013, pp. 275–284.
[28] M. Hertz, Y. Feng, and E. D. Berger, "Garbage collection without paging," in *Proc. ACM SIGPLAN Conf. Programm. Lang. Des. Implementation*, New York, NY, USA, 2005, pp. 143–153.
[29] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Cramm: Virtual memory support for garbage-collected applications," in *Proc. 7th Symp. Oper. Syst. Des. Implementation*, Berkeley, CA, USA, 2006, pp. 103–116.
[30] M. Saxena and M. M. Swift, "Flashvm: Virtual memory management on flash," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, Berkeley, CA, USA, 2010, pp. 14–14.
[31] H. Kim, M. Ryu, and U. Ramachandran, "What is a good buffer cache replacement scheme for mobile flash storage?" in *Proc. 12th ACM SIGMETRICS/PERFORM. Joint Int. Conf. Meas. Modeling Comput. Syst.*, New York, NY, USA, 2012, pp. 235–246.
[32] Y. Lv, B. Cui, B. He, and X. Chen, "Operation-aware buffer management in flash-based systems," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, New York, NY, USA, 2011, pp. 13–24.
[33] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Modeling Comput. Syst.*, New York, NY, USA, 2002, pp. 31–42.
[34] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," in *Proc. 6th USENIX Conf. File Storage Technol.*, Berkeley, CA, USA, 2008, pp. 16:1–16:14.
[35] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proc. Int. Conf. Compilers, Archit. Synthesis Embedded Syst.*, New York, NY, USA, 2006, pp. 234–241.



Weichao Guo received the BE degree from the Harbin Institute of Technology, China, in 2011. He is working towards the PhD degree with the Department of Computer Science and Technology, Tsinghua University, Beijing, China. He is currently working on flash array storage techniques towards enhancing reliability and performance. His research interests include distributed systems, storage techniques, mobile and operating systems.

Kang Chen received the PhD degree in computer science and technology from Tsinghua University, Beijing, China in 2004. Currently, he is an associate professor of computer science and technology at Tsinghua University. His research interests include parallel computing, distributed processing, and cloud computing.

Huan Feng received the BE degree from Beijing Jiaotong University, China in 2011. She is now working towards the PhD degree in the Department of Computer Science and Technology at Tsinghua University in China. She is currently working on mobile performance optimization problems. Her research interests include storage, mobile operating system, and mobile computing.

Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences in 2002. He is currently a professor in computer science and technology at the Tsinghua University of China. His research interests include parallel and distributed processing, and cloud storage. He has published more than 80 research publications and has received two Best Paper Awards. He is currently on the editorial board of the *International Journal of Networked and Distributed Computing* and *Communication of China Computer Federation*. He is a member of the IEEE. He can be reached at: wuyw@tsinghua.edu.cn.

Rui Zhang received the bachelor's degree from Tsinghua University in 2001 and the PhD degree from the National University of Singapore in 2006. He is an associate professor and reader in the Department of Computing and Information Systems at the University of Melbourne. Before joining the University of Melbourne, he has been a visiting research scientist at AT&T labs-research in New Jersey and at Microsoft Research in Redmond, Washington. Since January 2007, he has been a faculty member in the Department of Computing and Information Systems at the University of Melbourne. Recently, he has been a visiting researcher at Microsoft Research Asia in Beijing regularly collaborating on his Future Fellowship project funded by Australian Research Council. His research interest is data and information management in general, particularly in areas of indexing techniques, moving object management, web services, data streams, and sequence databases.

Weimin Zheng received the BS and MS degrees, in 1970 and 1982, respectively, from Tsinghua University, China, where he is currently a professor of computer science and technology. He is the research director of the Institute of High Performance Computing at Tsinghua University, and the managing director of the Chinese Computer Society. His research interests include computer architecture, operating system, storage networks, and distributed computing. He is a member of the IEEE.