Boosting Read-ahead Efficiency for Improved User Experience on Mobile Devices

Yu Liang[†], Yajuan Du[*], Chenchen Fu[†], Riwei Pan[†], Liang Shi[‡], Chun Jason Xue[†]
[†] Department of Computer Science, City University of Hong Kong
[*] School of Computer Science and Technology, Wuhan University of Technology
[‡] College of Computer Science, Chongqing University

ABSTRACT

Read-ahead schemes of page cache have been widely used to improve read performance of Linux systems. As Android system inherits the Linux kernel, traditional read-ahead schemes are directly applied in mobile devices. However, read request sizes and page cache sizes in mobile devices are much smaller than other platforms, which may decrease read-ahead efficiency and hurt user experience. The read-ahead efficiency is defined as hit pages / all pre-fetched pages in a sequential read. To study the efficiency of traditional read-ahead in mobile devices, this paper first observes that many pre-fetched pages are unused in page cache, which causes high page cache eviction ratio with high extra access latency. Then, this paper analyzes the factors that closely relate to the access latency. It is found that there exists a trade-off between read-ahead size and access latency. A size-tuning scheme is then proposed to explore this trade-off. Experimental results on real mobile devices have shown that the proposed scheme can reduce the number of pre-fetched pages and improve the efficiency of read-ahead without decreasing the page cache hit ratio.

1. INTRODUCTION

Read performance is critical to user experience on mobile devices because a quick response is expected for read operations, such as launching applications [1][2]. Page caches companied by read-ahead scheme can largely decrease read latency and have been widely applied in Linux systems. As Android system inherits the Linux kernel, traditional read-ahead scheme is directly applied in mobile devices. However, mobile devices have limited resources [5] and their request sizes and cache sizes have a lot of differences from servers, which leads to an inefficiency of read-ahead. For example, when pre-fetching 100 pages into the page cache, only about 3 pages of them will be read by the requests again. The other 97 unused pages stay in the page cache, easily leading to frequent evictions. When page cache is full, some pages have to be evicted from the page cache first. These evictions induce many extra latencies. To quantitatively show this influence, the latencies of launching Twitter and Facebook in three situations are collected on a real Android mobile device (HuaweiP9 mounted with fourth extended filesystem (Ext4) and flash friendly file system (F2FS)), as shown in Figure 1.

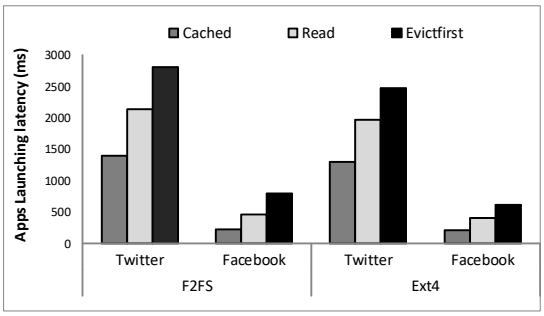


Figure 1: The latencies of launching Twitter and Facebook in three situations.

In the "Cached" case in Figure 1, cache hit happens and the requested data pages can be directly accessed. It is implemented by re-launching the app when it is still in memory [1]. In the "Read" case, cache miss happens and cache has enough space to launch the app immediately. It is implemented by launching the app after cleaning the page cache. In the "Evictfirst" case, cache miss happens and cache is full. Some pages have to be evicted from the cache to release space first. It is implemented by launching the app after sequentially launching 20 apps. The results show that the latency of launching app is the shortest in the "Cached" case. Compared to "Read" case, the "Evictfirst" case causes a longer launching latency for the listed two apps. This additional latency in the "Evictfirst" case is mainly caused by the page cache eviction.

In recent years, the works focusing on read ahead schemes can be classified into two types. The first type predicts the next app to use according to the information, such as user location [12][4] and then pre-fetches the data of the predicted next app [14]. The second type modifies the read-ahead scheme to pre-fetch pages according to other rules, such as the disk layout [6][7].

Focusing on read-ahead schemes on mobile devices, this paper first obtains three observations based on experiments in real mobile devices. Then, related factors that induce benefit and cost of current read-ahead scheme are analyzed and it is figured out that there exists a trade-off between read-ahead size and access latency. Finally, a size-tuning method is proposed to find a proper maximum size of read-ahead with which the eviction ratio can be reduced as much as possible without increasing the number of I/O requests.

In summary, this paper makes the following contributions:

[1]Check it by the command **dumpsys meminfo**.

Ewili'18, October 4th 2018, Torino, Italy .
Copyright held by Owner/Author.

- Observed that the inefficiency of read-ahead schemes are caused by the small request size and cache size in mobile devices;
- Analyzed the factors related to the access latency and identified that there is a trade-off between read-ahead size and access latency;
- Proposed a size-tuning scheme to find a proper maximum read-ahead size by exploiting the above trade-off. Experimental results show that this size can improve the efficiency of read-ahead without decreasing page cache hit ratio.

The rest of this paper is organized as follows. Section 2 describes background information and motivation. Several observations on real mobile devices and analysis are presented in Section 3. Section 4 proposes a size-tuning scheme to exploit the trade-off between read-ahead size and access latency and evaluates it on real mobile devices. Section 5 lists related works. This paper is concluded in Section 6.

2. BACKGROUND AND MOTIVATION

In this section, an overview of Android I/O stack, read-ahead scheme and file systems of mobile devices are presented. Besides, the request sizes of mobile devices and servers are compared.

2.1 Overview of Android I/O Stack

Android is a mobile operating system developed by Google, based on the Linux kernel and designed primarily for mobile devices. Figure 2 illustrates an architecture of Android I/O stack, including user space, Linux kernel, and device.

Application is the program to perform coordinated functions for the benefit of user, such as YouTube, Twitter, etc.

VFS Layer is designed for allowing applications to access different types of file systems in a uniform way.

Page Cache is used to provide quick accesses to cached pages and improve overall performance. The sizes of page cache in mobile devices are usually smaller than in server.

File System is used to control how data are stored and retrieved. It could affect the efficiency of read-ahead and hit ratio of page cache through data layout.

Generic Block Layer translates host requests into block I/O (bio). One bio is an ongoing I/O block device operation.

I/O Scheduler Layer re-organizes I/O operations to decide the order of submitting to the storage devices.

Flash Storage is an electronic non-volatile storage medium. It constitutes device with the above layer.

2.2 Read-ahead Scheme

The read-ahead scheme is designed for improving the page cache hit ratio by pre-fetching next few pages in a sequential read operation. When a request is a sequential read operation, read-ahead will be conducted to pre-fetch pages with the size 2 or 4 times of that read operation. If read-ahead hits, the size of pre-fetched pages will be increased by 2 times of the last pre-fetch operation until reaching the maximum size of read-ahead (default 128KB). If read-ahead misses, the size of read-ahead will be degraded by 2 pages (8KB) until reaching the minimal size of read-ahead (default 16KB).

Read-ahead is located in the same layer as page cache, and above the file system layer, but its hit ratio depends on the specific file system. This is because that the read-ahead scheme is based on an assumption that sequential read operation on file level is also sequential on the logical address of devices. However, the data layout depends on the specific file system.

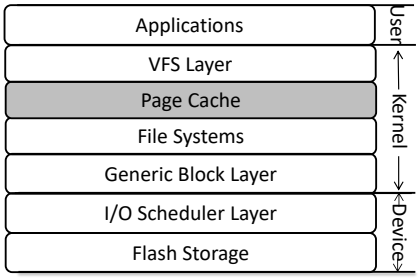


Figure 2: Android I/O stack overview.

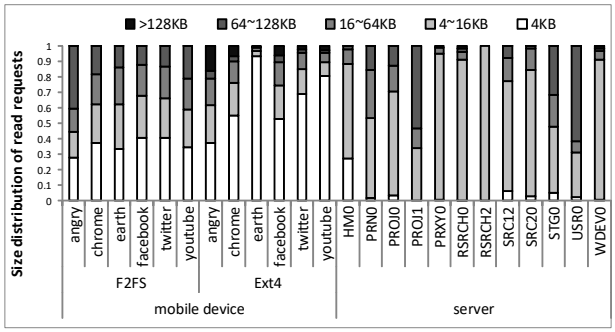


Figure 3: Request size distribution of read operations. The traces of mobile devices are collected in the accordingly file system (F2FS on Nexus9 and Ext4 on Nexus6). The traces of servers are the typical traces, which are also used in [10][13].

2.3 File System on Mobile Devices

There are two popular file systems on current mobile devices: Ext4 and F2FS. These two file systems have different influences on read performance because of their log and update schemes.

Ext4 was born as a series of backward-compatible extensions to Ext3 [11]. It is the most popular file system on current mobile devices. Ext4 adopts in-place update, which writes update data at the old logical addresses.

F2FS [9] is another widely used file system on mobile devices. If the free space is enough, F2FS buffers a sequence of data in the cache and then writes all the changes at the end of the log sequentially. Meanwhile, the original data are marked as invalid. This log and update scheme needs a garbage collection (GC) to reclaim space. However, when space is almost full, to improve performance, it adopts threaded log scheme, which reuses the space of invalid data without GC.

Since different file systems and space status will affect the data layout and thus affect the efficiency of read-ahead, two file systems and two space status in F2FS will be used in following experiments.

2.4 Request Sizes Comparison

To introduce the motivation of this paper, the comparison of request sizes between mobile devices and servers are

presented in Figure 3. These results are collected by **blktrace**[3]. The results show that most of the request sizes of Linux server are large, e.g. 4-64KB or more than 64KB. However, most of the request sizes of mobile devices are smaller than 64KB. Android directly inherits the read-ahead from Linux kernel, in which the maximum size of the read-ahead is 128KB (32 pages). This leads to a lot of unused pre-fetched pages. For example, to sequentially read 7 pages, 156 (4+8+16+32+32+32+32) pages will be pre-fetched and the efficiency of read-ahead is 4.5%. These redundant pages will induce extra access latencies and thus affect the user experience.

3. OBSERVATIONS AND ANALYSIS

In this section, the number of pre-fetched pages will be observed on real mobile devices. The experimental scenarios and setup will be presented first. And then, the page cache eviction ratio and extra access latency induced by these pre-fetched unused pages will be presented. Finally, the factors that closely relate to the extra access latency are analyzed. Based on the analysis, it is found that there exists a trade-off between read-ahead size and extra access latency.

3.1 Experimental Scenarios and Setup

The 120 experimental scenarios and the experimental setup used are presented in this section.

3.1.1 Experimental Scenarios

To comprehensively study the efficiency of read-ahead on the mobile devices, data are collected in 120 scenarios. Each scenario is represented by 5 capital letters. The configurations are listed in Table 1.

To figure out the influence of the file system on the hit ratio of the page cache and read-ahead, both two mainstream file systems of mobile devices, F2FS and Ext4, are involved in our experiments. To show the influence of read-ahead on the page cache, we compare the hit ratio of the page cache when read-ahead is turned on and turned off. Moreover, the logging ways are different in F2FS when the storages (Flashes) are full and have enough space, while logging ways are the same in Ext4. So we add these two scenarios (F and E) in F2FS and just one scenario (E) in Ext4. Additionally, cleaning cache is a critical step for testing the hit ratio of the page cache and read-ahead because that the cache is not empty[2] after restart operation. Furthermore, launching and using apps are different types of operations. Launching apps mainly generates sequential read operations while using apps mainly generates random and some sequential read and write operations. Thus these two scenarios have different influences on the hit ratio of the page cache and read-ahead. As users' common operations, both of them need to be tested. Finally, different apps have different using patterns, so 7 popular apps (including browser, map, game, multimedia and social apps) are selected in the experiments.

For clarity, the letters with specific order present experimental scenarios. For example, in the results, **FRESF** represents that the testing file system is **F**2FS and **R**ead-ahead is turned on, and storage has **E**nough space, **F**acebook is tested after re**S**tarting the mobile device. According to the results of our experiments, some interesting observations will be shown in the next subsection. Moreover, the reasons for these observations will be analyzed.

[2]Check it by command **free -m** in current mobile devices.

Table 1: Interpretation of the capital letter represented scenarios.

Order	Letter	Meaning
1	F	F2FS
	E	Ext4
2	R	With readahead
	N	Without readahead
3	E	Enough space (more than 50% of capacity)
	F	Full in F2FS (less than 5% of capacity)
4	S	Test after restart
	C	Test after clean cache
5 Launching Apps	M	Launching 20 Apps
	T	Launching Twitter
	F	Launching Facebook
5 Using Apps	C	Using Chrome
	T	Using Twitter
	F	Using Facebook
	A	Using Angrybird
	M	Using Google Map
	E	Using Google Earth
	Y	Using YouTube

Table 2: Platforms used in experimentations.

Name	System	Storage
Nexus9	CPU: NVIDIA Denver Memory: 2GB OS: Linux 3.4.0 Android: 5.0.1	eMMC16GB Data partition: 10.8GB File system : F2FS
Huawei P9	CPU: ARM's Cortex-A72 Memory: 3GB OS: Linux 3.10 Android: 6.0	eMMC32GB Data partition: 25.1GB File system : F2FS or Ext4
Nexus6	CPU: Krait 450 Memory: 3GB OS: Linux 3.10.40 Android: 6.0.1	eMMC 32GB Data partition: 26GB File system : Ext4

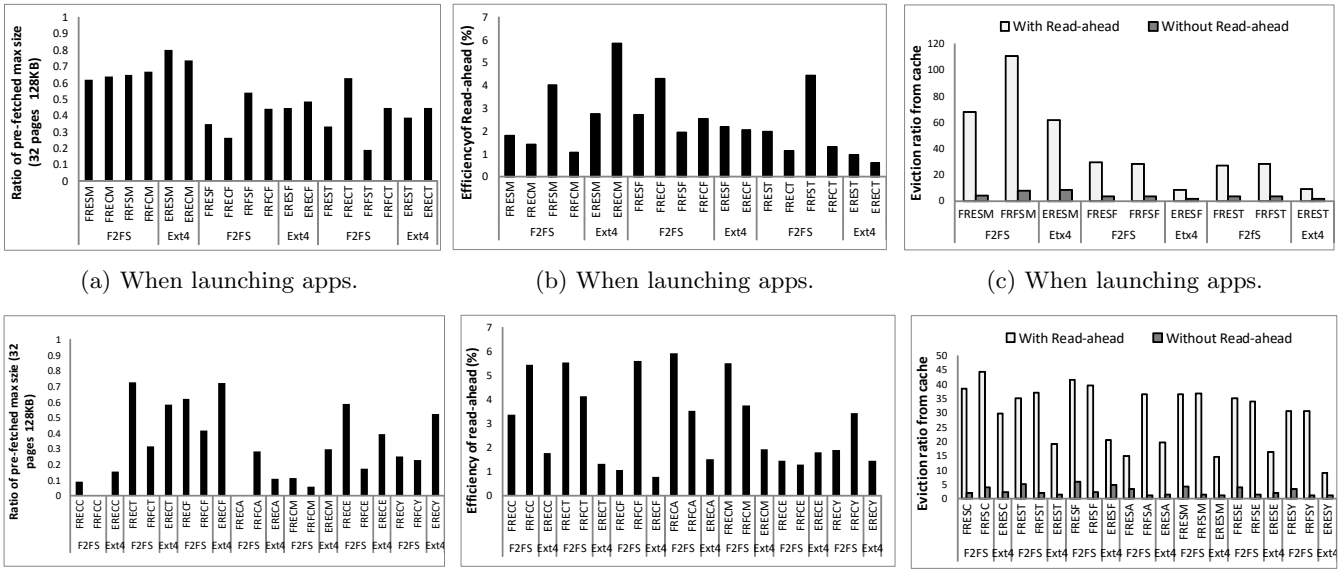
3.1.2 Experimental Setup

To compare the influence of different file systems, three different Android mobile devices are used in our experiments. Their configuration details are shown in Table 2. Nexus9 employs F2FS as default file system of the data partition. The default file system of data partition in HuaweiP9 is also F2FS, but it can be converted to Ext4. So it can be used to do the influence comparison of F2FS and Ext4. In Nexus6, the default file system of its data partition is Ext4. Launching and using apps are conducted on the data partition, so their file systems will affect the experimental results.

3.2 Observations

In this section, a set of experiments are conducted on HuaweiP9 with Ext4 and F2FS file systems to comprehensively study efficiency of read-ahead schemes in mobile devices. According to the experimental results, three observations have been found.

The first observation is that many unused pages are pre-fetched into page cache by read-ahead. The ratio of max-sized (128KB) pre-fetch when launching apps and using apps are shown in Figure 4a and Figure 4d respectively. The results show that there are a lot of max-sized pre-fetch in both launching and using app cases. When using apps, the ratio can reach 72%; When launching apps, the ratio can reach 80%. However, according to Figure 3, most of the request sizes are smaller than 64KB in mobile devices. This



(a) When launching apps. (b) When launching apps. (c) When launching apps.

(d) When using apps. (e) When using apps. (f) When using apps.

Figure 4: Read-ahead efficiency and page cache eviction ratio on HuaweiP9. If the first letter is F, that means the file system is F2FS. Otherwise, the file system is Ext4. The second letter R means read-ahead scheme is turned on. If the third letter is E, that means the storage has enough space. Otherwise, the storage is full (more than 95%). If the fourth letter is S, conducting restart operation before tests. Otherwise, cleaning page cache before tests. For example, FRES: file system, with read-ahead, enough space and restart before tests.

means that too many unused pages are pre-fetched. The efficiencies of read-ahead when launching and using apps are shown in Figure 4b and Figure 4e respectively. The average efficiency is 3% in F2FS and 2% in Ext4. This means that most of pre-fetched pages are unused.

The second observation is that the eviction ratio induced by read-ahead is unexpectedly high, especially in F2FS. The page cache eviction ratios[3] in different scenarios are shown in Figure 4c and 4f. In F2FS, when launching 20 apps, the eviction ratio could be 1.1. This is because that some of the pre-fetched pages have been evicted from the cache, even though when there is no request. The eviction ratio in F2FS is larger than Ext4, no matter when launching apps or using apps. One of the major reasons is that when processing a request, the data read into page cache by F2FS is larger than by Ext4.

The third observation is that page cache eviction has induced extra access latency. According to the results in Figure 1, the latencies in "Evictfirst" case could be 23.8% and 41.5% longer than in "Read" case when launching Twitter and Facebook, respectively.

In a word, many unused pages are pre-fetched into the limited page cache, which induces high page cache eviction ratio and thus leads to extra access latency.

3.3 Access Latency Analysis

Based on the above observations, this subsection analyzes the factors that are closely related to the access latency.

The latency of reading a page is different in four cases, including "Cached", "Read", "Evictfirst1", and "Evictfirst2". Their latencies are listed in Table 3.

In "Cached" case, the requested page is in the page cache, therefore it can be directly accessed from the page cache. The latency of read operation only includes the latencies of "Check cache" and "Read from cache". Both the latencies of "Check cache" and "Read from cache" are on nanoseconds level. Hence, the latency of read operation is very short.

In "Read" case, cache miss happens but the page cache has enough space for the requested page, thus this page can be read into page cache immediately. So the read operation includes the latencies of "Check cache", "Create page", "Read from device" and "Read from cache". The latency of "Read from device" is on microseconds level while other parts are on nanoseconds level. The latency of read operations is longer than "Cached" case.

In "Evictfirst1" case, the page cache is full, some pages have to be evicted from the cache to release space first, so that the requested page can be read into the cache. In this case, the evicted page is clean (it has not been updated), thus it can be removed directly. The latency of read operation includes the latencies of "Check cache", "Remove one", "Create page", "Read from device" and "Read from cache". Only more than "Read" case by "Remove one", which is on nanoseconds level. So the latency of read operation is similar to the "Read" case.

In "Evictfirst2" case, the page cache is full and the evicted page is dirty (it has been updated), thus it needs to be written back to the secondary storage first. The latency of read operation includes all the latencies. The latencies of "Write back" and "Read from device" are on microseconds level. Hence, the latency of read operations is the longest.

Read-ahead can improve read performance by improving the page cache hit ratio and reducing I/O operations. This means read-ahead could increase the number of "Cached" case. However, if the efficiency of read-ahead is too low, the unused pages will occupy the page cache. In other words, the inefficiency of read-ahead will increase the number of

[3]eviction ratio = evicted pages / requested pages

Table 3: The comparison of latency of read operations with four seances. "ns" represents the latency is on nanosecond level and "us" represents the latency is on microseconds level. "Y" represents that this kind of latency is included.

	Check cache	Evict page	from cache	Create page	Read from device	Read from cache
		Write back	Remove one			
Latency	ns	us	ns	ns	us	ns
Cached	Y					Y
Read	Y			Y	Y	Y
Evictfirst1	Y		Y	Y	Y	Y
Evictfirst2	Y	Y	Y	Y	Y	Y

Evictfirst1 and Evictfirst2. The efficiency of read-ahead can be improved by reducing the size of read-ahead. However, if the read-ahead size is too small, it will decrease the page cache hit ratio and increase the number of I/O requests, which thus can increase the access latency. There is a trade-off between read-ahead size and access latency.

4. SOLUTION AND EVALUATION

In this section, a size-tuning scheme is proposed to exploit the trade-off between read-ahead size and access latency. To evaluate this scheme, a case study is implemented on HuaweiP9 with F2FS and Ext4.

4.1 Size-tuning Scheme

Size-tuning scheme aims to find a proper maximum size of read-ahead (MSR) to reduce the pre-fetched pages without increasing the number of I/O requests. The size-tuning scheme calculates the MSR based on the maximum request size denoted as *maxreqsize*. On mobile devices, as shown in Figure 3, almost all of the request sizes are smaller than 128KB (32 pages). The *maxreqsize* is 32 pages, which is calculated based on the average mobile workloads.

It is not tricky to find the proper MSR, because that too small value would increase the number of I/O while too large value would induce a lot of unused pages pre-fetched. For example, in original read-ahead (by default MSR is 32 pages), to sequentially request 32 pages (maxreqsize), 956 (4+8+16+32...+32) pages will be pre-fetched by 32 read-ahead operations. Each read-ahead operation is one I/O operation. Actually, the first four I/O will pre-fetch 60 pages (>maxreqsize). The pages other than the first 32 pages are unused in this sequential read operation. If MSR is tuned to 16 pages, 492 (4+8+16+16...+16) pages will be pre-fetched by 32 read-ahead operations. The first four I/O will pre-fetch 44 pages (>maxreqsize). If MSR is tuned to 8 pages, 252 (4+8+8+8...+8) pages will be pre-fetched by 32 read-ahead operations. The first four I/O will pre-fetch 28 pages (<maxreqsize). In this case, the fifth I/O will be needed to pre-fetch the remaining 4 pages to meet the maxreqsize.

Based on the above discussion, a size-tuning scheme is proposed to calculate MSR. The equation for calculating the new maximum size of read-ahead (NMSR) is as follows:

$$\arg\min_x \{4 * \sum_{i=0}^{x} 2^i \geq maxreqsize\} \quad (1)$$

$$\arg\min_{NMSR} \{\sum_{i=0}^{x} min\{4 * 2^i, NMSR\} \geq maxreqsize\} \quad (2)$$

The x represents the I/O number. The minimum number of I/O can be calculated by Equation 1. Substitute the x obtained by Equation 1 into Equation 2,the minimum NMSR can be calculated. Note that the size-tuning scheme is offline.

4.2 Evaluation Results

To evaluate this size-tuning scheme, a case study is implemented on HuaweiP9. As shown in Figure 3, *maxreqsize* is 32. According to above two equations, the NMSR is calculated as 16 pages. In this section, the page cache hit ratio, the average size of read-ahead, and the efficiency of read-ahead are compared when with original MSR and NMSR. We only the influences on Ext4 and F2FS file systems as examples. The evaluation results are shown in Figure 5.

Figure 5a and 5d show that the hit ratio of the page cache with NMSR is very close to the original hit ratio. This means that introducing NMSR will not impact the hit ratio of the page cache in both launching and using apps cases.

Figure 5b and 5e show that with NMSR, the average request size pre-fetched by read-ahead can be reduced. This means that the amount of the pre-fetched pages are most likely to be reduced.

Figure 5c and 5f show that the efficiency of the read-ahead can be significantly improved in some cases. When launching 20 apps, the efficiency of read-ahead can be improved by 30 times and 6.5 times in F2FS and Ext4 respectively. This is because there are a lot of sequential read operations. Launching Facebook and launching Twitter also mainly generate sequential read operations, but their results do not present significant improvements. This is because the time of launching one app is too short to show the benefits.

In summary, the proposed NMSR can improve the efficiency of read-ahead without reducing the page cache hit ratio and thus improve the read performance.

5. RELATED WORK

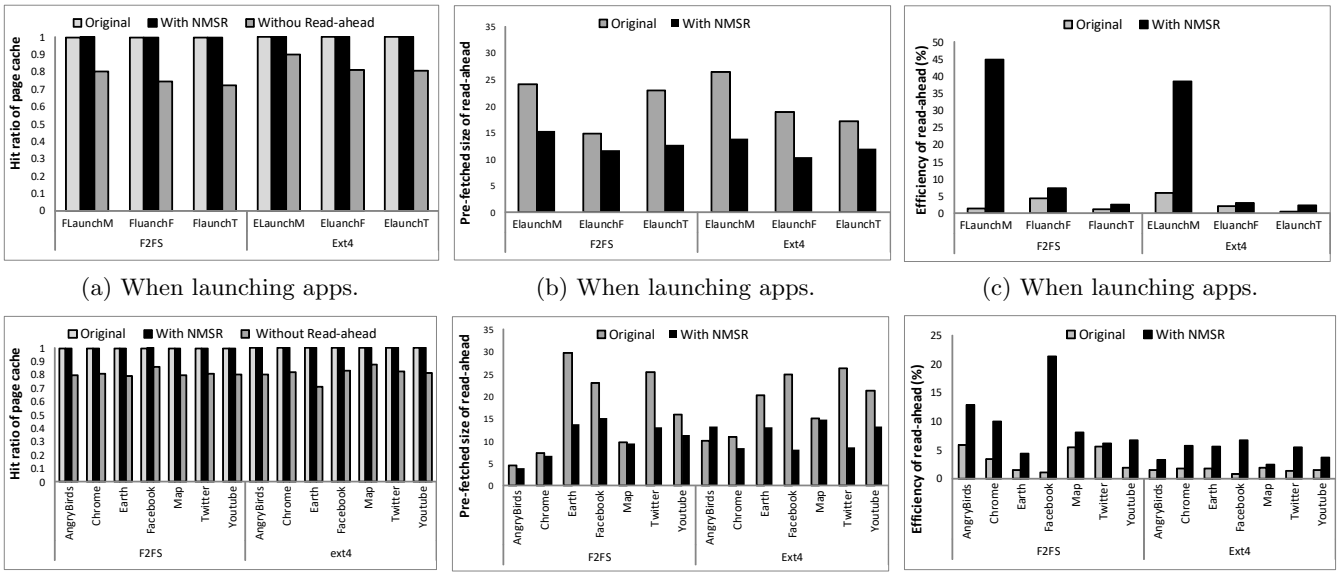
This paper, to our knowledge, is the first to study the efficiency of read-ahead schemes on mobile devices. There are two groups of works related to the read-ahead scheme optimization. The first group aims to predict the next few apps according to the user's information. The second group is to change the read-ahead scheme according to other rules.

Predicting the usage of applications

If the usage of applications is known exactly, user experience could be much improved. Matsumoto et al.[12] predicted the application usage according to the usage patterns. Parate et al.[14] not only predicted the application usage, but also pre-fetched the requested pages for the predicted application. Chung et al.[4] reduced the application launching time by terminating "unbeneficial" background applications.

Read-ahead improvement according to other rules

Some read-ahead schemes use other rules to pre-fetch pages for a high accuracy. Jiang et al.[6] proposed a new read-ahead scheme to pre-fetch pages according to both data layout and access history on disk. Khaleel et al.[7] proposed average least frequency used removal (ALFUR) by using intelligent agent to improve the performance and speed of browsing web sites using internet. Kim et al.[8] proposed



(a) When launching apps. (b) When launching apps. (c) When launching apps.

(d) When using apps. (e) When using apps. (f) When using apps.

Figure 5: The comparison of the page cache hit ratio and performance between the original read-ahead and with NMSR.

a novel scheme to properly harness the swapping to mobile systems to provide extra usable memory by reclaiming inactive pages and improving memory utilization.

Few previous works discuss the efficiency of read-ahead schemes. However, we find the efficiency of read-ahead on mobile devices is very low, which leads an extra latency. The proposed tuning read-ahead size scheme can solve this problem and is easily implemented.

6. CONCLUSION

Read-ahead schemes have been widely used to improve read performance in Linux kernel, but the traditional read-ahead size is too large for mobile devices. This work observes that many unused pages are pre-fetched in page cache by traditional read-ahead, which causes a high page cache eviction ratio with extra access latency. Based on these observations, this paper further analyzes the factors that closely relate to the access latency. According to the analysis, it is found that there is a trade-off between read-ahead size and access latency. To exploit this trade-off, this paper proposes a size-tuning scheme. To evaluate this scheme, a case study is implemented on real mobile devices. The experimental results show that the proposed method can significantly reduce pre-fetched pages. On average, the efficiency of read-ahead has been improved by 2.9 times in using app case and by 6.5 times in launching app case without decreasing page cache hit ratio. In the future work, we will propose a scheme to tuning the zooming pace and the stop time of read-ahead.

7. ACKNOWLEDGMENTS

This work is supported by National Science Foundation of China (NSFC) 61772092 and 61572411.

8. REFERENCES

[1] Daily tip: How to make your iphone camera launch instantly. http://www.tipb.com/2011/04/20/daily-tip-iphone-camera-launch-instantly-jailbreak/.

[2] Ios 5 slowing iphone 4 and 4s complaints. http://www.phonesreview.co.uk/2011/10/25/ios-5-slowing-iphone-4-and-4s-complaints/.

[3] A.D.Brunelle. Block i/o layer tracing: blktrace. HP, Gelato-Cupertino, CA, USA, 2006.

[4] Chung, Y.-F., Lo, Y.-T., and King, C.-T. Enhancing user experiences by exploiting energy and launch delay trade-off of mobile multimedia applications. *ACM Trans. Embed. Comput. Syst. 12*, 1s (2013), 37:1–37:19.

[5] Ji, C., Chang, L.-P., Shi, L., Gao, C., Wu, C., Wang, Y., and Xue, C. J. Lightweight data compression for mobile flash storage. *ACM Trans. Embedded Comput. Syst. 16* (2017), 183:1–183:18.

[6] Jiang, S., Ding, X., Xu, Y., and Davis, K. A prefetching scheme exploiting both data layout and access history on disk. *Trans. Storage 9*, 3 (2013), 10:1–10:23.

[7] Khaleel, M. S. A., Osman, S. E. F., and Sirour, H. A. N. Proposed alfur using intelegent agent comparing with lfu, lru, size and pccia cache replacement techniques. In *ICCCCEE* (2017), pp. 1–6.

[8] Kim, S.-H., Jeong, J., and Kim, J.-S. Application-aware swapping for mobile systems. *ACM Trans. Embed. Comput. Syst. 16*, 5s (Sept. 2017), 182:1–182:19.

[9] Lee, C., Sim, D., Hwang, J. Y., and Cho, S. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)* (2015), pp. 273–286.

[10] Li, Q., Shi, L., Xue, C. J., Wu, K., Ji, C., Zhuge, Q., and Sha, E. H.-M. Access characteristic guided read and write cost regulation for performance improvement on flash memory. In *FAST* (2016), USENIX Association, pp. 125–132.

[11] Mathur, A., Cao, M., Bhattacharya, S., and Dilger, A. The new ext4 filesystem : current status and future plans.

[12] Matsumoto, M., Kiyohara, R., Fukui, H., Numao, M., and Kurihara, S. Proposition of the context-aware interface for cellular phone operations. In *5th International Conference on Networked Sensing Systems* (2008), pp. 233–233.

[13] Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., and Rowstron, A. Migrating server storage to ssds: Analysis of tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer Systems* (2009), EuroSys '09, ACM, pp. 145–158.

[14] Parate, A., Böhmer, M., Chu, D., Ganesan, D., and Marlin, B. M. Practical prediction and prefetch for faster access to applications on mobile phones. UbiComp '13, ACM, pp. 275–284.