Improving User Experience of Android Smartphones Using Foreground App-Aware I/O Management

Sangwook Shane Hahn
Seoul National University
shanehahn@davinci.snu.ac.kr

Sungjin Lee
DGIST
sungjin.lee@dgist.ac.kr

Inhyuk Yee
TmaxSoft
heyork1@gmail.com

Donguk Ryu
Samsung Electronics
du.ryu@samsung.com

Jihong Kim
Seoul National University
jihong@davinci.snu.ac.kr

ABSTRACT

Modern mobile systems are designed to run multiple apps simultaneously to provide a better experience for end users. In such a multi-tasking environment, a foreground app that a user is actually interacting with is often delayed by background ones, which results in significant degradation of user-perceived response time and user experience. Based on detailed analysis of kernel's software stack, we find that the majority of the degradation is caused by the inefficient management of foreground I/Os in the page cache and block I/O layers, and existing techniques like a priority inheritance protocol are not an effective solution to address this. In this paper, we propose a foreground app-aware I/O management scheme that accelerates foreground I/Os by preempting background I/Os in the entire kernel stacks. Our experimental results on smartphones show that the proposed technique reduces the user-perceived response time delay by up to 91%, achieving application's responsiveness close to when a single app solely runs.

1 INTRODUCTION

Modern mobile systems such as smartphones and tablets support multitasking for seamless switching between applications [1, 2]. For example, Android allows one foreground app and up to four background apps to run simultaneously [3]. More recently, Android smartphones start to support a multi-window feature that displays more than one app at the same time on the screen [4]. In such multitasking mobile environments, foreground app-aware resource management is important. For example, if the current interactive session of a foreground app is blocked or delayed by background apps, it may significantly degrade the quality of user experience with mobile systems.

Since the execution behavior of a app is directly affected by how CPUs are managed, foreground app-aware CPU management has been extensively studied. For example, in order to avoid user-perceived delays while running a foreground app, CPU frequency/voltage scheduling policies are differentiated between user-perceived execution intervals and user-oblivious intervals [5]. Unlike CPUs, I/O management has been mostly *foreground app-oblivious* on mobile systems. For older smartphones such as Nexus S [6], I/O requests from a foreground app were rarely interfered with I/O requests from background apps because the number of background apps is quite limited due to a small DRAM capacity (e.g., 382 MB) [7-11]. However, on modern high-end smartphones with a large number of CPU cores (e.g., 8 cores) and a large DRAM capacity (e.g., 8 GB) [12-14], the quality of user experience can be significantly deteriorated because of unexpected collisions between foreground I/O requests and background I/O requests [15].

As a concrete example, consider an app launch scenario where a user must wait until all the required files for an app are loaded from a storage device before the next interaction with the mobile system is initiated. Although an app launch often involves reading a large number of files (e.g., executables, libraries, images, and binary data), in most mobile apps, such a startup phase finishes rather quickly (e.g., less than 1 second). However, when several I/O requests from multiple background apps are issued simultaneously while launching the app, the app launching time may significantly increase over when no background apps interfere with the foreground app. In our experiments with high-end smartphones, the app launch time can increase by up to 4 times over that no background apps.

Various I/O scheduling techniques have been proposed to address the problem caused by background I/Os. Boosting Quasi-Asynchronous I/O (QASIO) is one of such efforts to

APSys'17, Mumbai, India

DOI: 10.1145/3124680.3124721

provide better I/O scheduling by means of a priority inheritance protocol [16]. QASIO prioritizes I/O requests according to their types at the block I/O layer – it gives a high priority to synchronous writes while assigning a low priority to asynchronous writes. Then, if high-priority writes are blocked by low-priority ones, temporarily promotes the priority of the low-priority writes so that they can be serviced quickly. Request-centric I/O prioritization (RCP) [17], more recently proposed, is also based on a priority inheritance protocol, but it further improves QASIO by prioritizing I/O requests at the page cache layer rather than the block I/O layer.

While the aforementioned techniques are rather effective than a default kernel scheduler, they fail to offer high-quality experience to end users because of the following limitations. First, they prioritize I/O requests based on *types of requests* (e.g., synchronous or asynchronous), instead of *types of apps* (e.g., foreground or background) that request the I/O service. If background apps issue synchronous I/Os frequently, a foreground app may be delayed because of background I/O requests. Second, a *priority inheritance protocol is less effective* in a situation where a large volume of I/O traffic is sent by background apps (e.g., downloading apps) while a foreground app issues small I/Os in a sporadic manner (e.g., reading database records). In this case, small foreground I/Os are blocked by bulky background I/Os for a long time because the background I/Os are promoted to a high priority.

In this paper, we take a more direct approach to resolving conflicts between foreground I/O requests and background requests *by preempting background I/O requests whenever foreground I/O requests are issued.* In order to support foreground I/O requests with a higher priority throughout the entire Android I/O stack, our proposed scheme, called *FAIO (foreground app-aware I/O management)*, modifies several layers including the Android platform, the page cache and the block I/O layer. FAIO identifies the current foreground app from the Android platform with a negligible overhead. In the page cache layer, when the foreground app issues an I/O request, FAIO *preempts* the global page-cache lock from the current background I/O request and gives it to the foreground I/O request right away. (This is the key difference from the existing techniques which adopt the priority inheritance protocol.) In the block I/O layer, when an block I/O request is sent to the storage dispatch queue, FAIO sends foreground I/O requests with a higher priority than background I/O requests. By combining these modifications at the three I/O stack layers, FAIO guarantees that foreground I/O requests are not delayed by background I/O requests.

In order to evaluate the effectiveness of the proposed FAIO technique, we have implemented FAIO on various Android smartphones, including Nexus 5 [18], Nexus 6 [19], and Galaxy S6 [20]. Our experimental results show that FAIO significantly improves user-perceived response time. For example, FAIO can reduce the app launch time by 2.68 times, achieving a (nearly) equivalent launch time under no background apps. Furthermore, FAIO reduces delays of app suspension (switching from one app to another) and app loading (loading resources during app running) by up to 72% and 86%, respectively.

The remainder of this paper is organized as follows. In Section 2, we report our key findings through our evaluation study that analyzes the impact of background I/Os on user-perceived response time. Section 3 describes the proposed FAIO in detail. Experimental results using real-world smartphones are presented in Section 4 and Section 5 concludes with future work.

2 EMPIRICAL ANALYSIS OF USER-PERCEIVED RESPONSE TIME

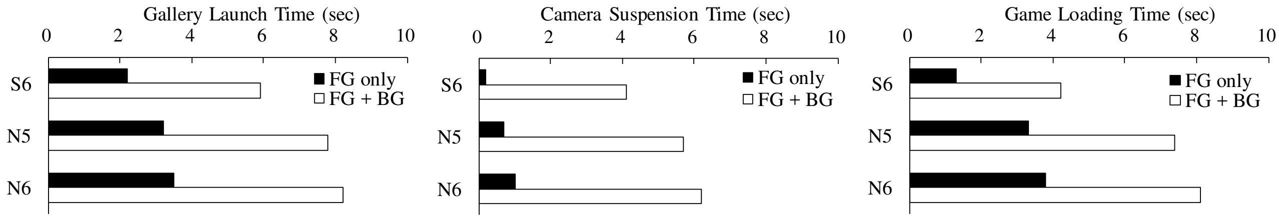
In this section, we first review the overall architecture of the Android I/O stack, giving a brief explanation of how Android apps accesses files in storage media. Then, we demonstrate the impact of background I/Os on response times through experiments on three smartphones. Finally, we introduce a major bottleneck we find by analyzing the Android I/O stack.

2.1 Overview of Android I/O Stack

Android is the most widely-used Linux-based operating system for mobile systems [21]. Android apps are able to access storage media through two top libraries, SQLite [22] and Core Libraries, which are provided by the Android platform. Two libraries are built on top of the Linux I/O stack, comprised of the virtual/local file system, the page cache, and the block I/O layer.

Android file I/Os (i.e., reads and writes on files) created by the top libraries are delivered to the virtual file system through Linux's system call services. After the information of the desired file is obtained from the local file system (e.g., EXT4 [23]), the Linux kernel sees if corresponding file data is already cached or buffered in the page cache. If not, free pages available in the page cache are allocated to individual file I/Os. If the file I/O is for writes, user data is copied to the allocated pages in the kernel.

Before sending I/O commands to an underlying block device, each file I/O is converted into a set of block I/O requests with designated logical block addresses (LBAs). Block I/O requests are then transferred to the block I/O layer and put into proper I/O scheduling queues, synchronous or asynchronous queues, according to their types. I/O scheduling algorithms (e.g., CFQ [24]) move ready-to-submit block I/O requests to a dispatch queue, which will be sent to the block device via the eMMC [25] or UFS [26] interface. If the file I/O is for reads, data read from the storage device is stored in



(a) `Gallery` app launch time.

(b) `Camera` app suspension time.

(c) `Game` app loading time.

Figure 1: Impact of BG I/Os on user experience.

the allocated pages, and the data is finally copied to a buffer in a user space.

2.2 Impact of BG I/Os on User Experience

In order to understand an impact of background I/Os on user experience, we conduct a series of experiments with three Android smartphones, Nexus 5 (N5), Nexus 6 (N6), and Galaxy S6 (S6). We measure user-perceived response time for three real-life usage scenarios of smartphones under heavy background I/Os, which is depicted in Fig. 1. In order to generate background I/Os, we select Android's app update manager as a background app because it is automatically invoked in the background when a smartphone is connected to a Wi-Fi environment. Popular apps such as `Twitter` are reported to be updated every 7 days, on average [27, 28]. Therefore, smartphone users are likely to suddenly experience heavy background I/Os caused by app updates if they install a relatively large number of apps.

Scenario A – Launching a `Gallery` App: As pointed out in Section 1, an app launch requires to read a relatively large number of files (e.g., executables, libraries, and files). In particular, as the quality of mobile contents improves, the amount of data to read while launching an app increases as well. In the case of a `Gallery` app [29], for example, about 500 MB of data has to be preloaded. (To prevent reading the pre-generated thumbnails, we erased all the photos then stored new ones before launching.) We measure its launch time, which is defined to be the time interval from when the app icon is touched by a user to when all the thumbnails of the photos are created.

Fig. 1(a) depicts the launch time of `Gallery` on the three smartphones. Even though there are differences depending on the hardware performance, significant degradation of apps' launch speed is commonly observed on all the smartphones when background I/Os are issued simultaneously. For example, in the Galaxy S6, the launch time (denoted by `FG+BG`) increases to 2.6 times, compared with a standalone launch (denoted by `FG only`).

Scenario B – Suspending a `Camera` App: Switching from one app to another becomes a common feature in smartphones supporting multitasking. Before moving to the new app, the current app should be properly suspended. In the Android platform, the app suspension involves flushing of buffered dirty data to persistent storage, so as to create as much free memory as possible for the new app. For apps that heavily use memory, therefore, lots of writes could be involved that cause degradation of user experience.

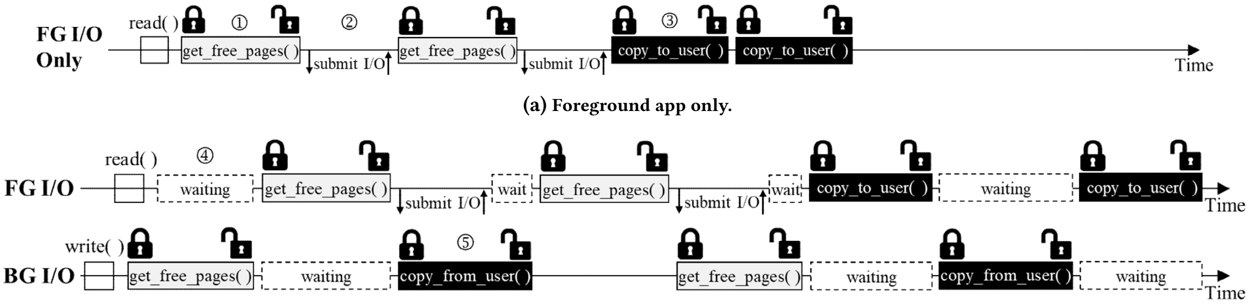
We examine the suspension time of a `Camera` app [30] when it switches to a home screen app. The `Camera` app is recording a video for 10 minutes. To understand an impact of app suspension on user-perceived response time, we measure the time interval from when the home button is pressed (while `Camera` is running) to when the home screen is displayed to get the next user input. Fig. 1(b) illustrates the suspension time of `Camera` – it is less than 1 second when no background I/Os are being issued, but increases to 19.5 times under heavy background I/Os.

Scenario C – Loading a `Game` App: After app launching, some apps require loading additional files to initiate actual app contents. One of the representative examples is a `Game` app [31] that has to preload game contents (e.g., stage maps, rendered images, and character information). This loading process inevitably results in response time delays from the perspective of end users. We measure the loading time of the `Game` app to understand how much background I/Os affect user-perceived response time. As shown in Fig. 1(c), the standalone loading time ends in 2-4 seconds, but under heavy background I/Os, it increases to 4-8 seconds.

2.3 Analysis of Bottlenecks under BG I/O

In order to find bottlenecks causing user-perceived delays, we have analyzed all of the activities in the Android platform as well as the Linux kernel while running background and foreground apps. Surprisingly, we find that the root causes of user-perceived delays are *lock contention in the page cache layer*, rather than I/O scheduling in the block layer.

Once I/O requests arrive at the kernel from user apps, free pages should be assigned first to the I/O requests. This free-page allocation holds a global lock of the page cache until enough free pages are found [32]. Moreover, this free-page acquisition process is *non-preemptive* [33]. In common cases, obtaining free pages is done quickly. However, if there are



(a) Foreground app only.

(b) Both foreground app and background app.

Figure 2: Impact of background apps on the I/O latency of the FG app.

not enough free pages, it could take a relatively long time (longer than 200 *m*s [17]) because it often involves extra I/Os to flush out dirty pages to the persistent storage. Therefore, if foreground I/Os are blocked by background ones which are causing extra I/Os to get free pages from the page cache, the foreground app is inevitably delayed for a long time.

Fig. 2(a) illustrates an example where a foreground app reads a photo file whose size is 256 KB from storage media by calling a `read()` system call. We compare two difference cases: 1) when it solely runs and 2) when there is a background app that writes a large file to the storage device. Without no background apps, the foreground app can quickly get free pages from the page cache (by calling `alloc_pages()` ①). Since the maximum allocation unit of free pages is limited to 128 KB [34], the kernel calls `alloc_pages()` twice, each of which gets 128 KB free pages. After calling each `alloc_pages()`, the kernel sends a read I/O command to the storage device (②), which transfers file contents from the storage to the allocated pages. Finally, data kept in the kernel pages are copied to a user-space buffer in the unit of 128 KB (by calling `copy_to_user()` ③).

In the case where the foreground and background apps run simultaneously, suppose that the background app calls the `write()` system call to write data just before `read()` is invoked by the foreground app. The page lock is grabbed by the background app first, so the foreground app has to wait until it releases the lock (④). This could be quite long if extra I/Os are involved while assigning free pages to the background app. After the page lock is released by the background app, the foreground app is able to acquire the lock, allocates free pages for reads, and releases the lock. Then, it issues a read I/O command to the device. Copying data from the user space to the kernel space (`copy_from_user()`) also requires holding the same global lock of the page cache (⑤). As depicted in 2(b), if the background app already has the global lock, the foreground app has to wait again for the lock to be released, which adds additional user-perceived delays.

Some might argue that the eviction of dirty pages in the middle of calling `alloc_pages()` would rarely occur. In our observation, however, when write-dominant background apps run (for example, update of apps in Section 2.2 and other real-life scenarios in Section 4.1), lots of dirty pages are created in the page cache and available free pages quickly run out. If foreground app requests I/Os in such situations, frequent eviction of dirty pages is inevitable.

3 DESIGN AND IMPLEMENTATION OF FAIO

As pointed out in the previous section, foreground I/Os are often delayed due to lock contention initiated by background I/Os at the page-cache level. A priority inheritance protocol that raises a priority of background I/Os is unable to avoid such delays since it requires a foreground app to wait for background I/Os to finish.

The most promising approach may be to preempt background I/Os upon the arrival of latency-sensitive foreground I/Os at the page cache. This creates a quick I/O path for foreground I/Os so that they are served promptly by the page cache, while suspending less important background I/Os. Key technical challenges here are (1) how to identify foreground I/Os from background ones at the page-cache level and (2) how to preempt background I/Os immediately. Keeping these technical challenges in mind, we design the foreground app-aware I/O management (FAIO) with three modules as illustrated in Fig. 3: a foreground application detector (FG app detector), a foreground-centric page manager (FG page manager) and a foreground-centric I/O dispatcher (FG I/O dispatcher).

The FG app detector obtains the information of the current foreground app by monitoring the activity stacks of the Android platform (①) and forwards it to the FG page manager (②). Using this information, the FG page manager is able to identify I/O requests from the foreground app, suspending the currently executing background I/O jobs (③). The FG page manager then grabs a global lock of the page cache, preferentially assigning free pages to foreground I/Os, regardless of their arrival time (④). Until the FG page manager releases the lock, background I/Os are postponed. After

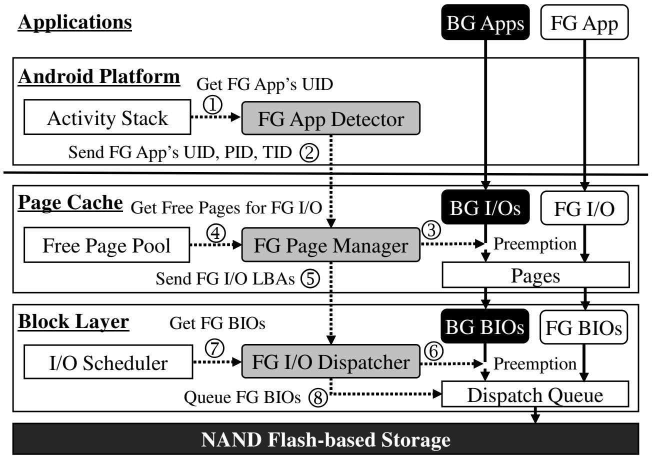


Figure 3: An overall architecture of FAIO.

acquiring all the free pages required, the FG page manager builds up block I/O requests for foreground I/Os (FG BIOs) with designated LBAs, putting them into I/O scheduler's queue in the block layer (⑤). Upon the arrival of FG BIOs, the FG I/O dispatcher suspends servicing BG BIOs by limiting I/O queueing (⑥) and then immediately delivers FG BIOs to the dispatch queue (⑦ and ⑧).

3.1 Foreground App Detector

In order to identify a foreground app among all the apps available in the system, the FG app detector inquires of the Android activity manager holding all of the activities initiated by a user. Whenever a user inputs a command to a phone by touching a screen or an icon, the Android platform creates a new activity, which is a sort of job corresponding to user's command, and puts it into an activity stack in the Android activity manager. Since the top activity on the stack points to the current interactive app with a user (i.e., a foreground app), the information of the foreground app in the system can be easily retrieved.

All of the Android apps have its own unique ID number, called UID, which is assigned when an app was installed in the system. An UID number is different from Linux's process ID (PID). Thus, our next step is to find a list of Linux processes connected to the foreground app. A list of the processes in question can be obtained by examining all the processes in Linux's process tree. However, such a exhaustive search on the process tree takes a relatively long time. Therefore, the FG app detector maintains an UID-indexed table that is updated whenever a new process or thread is created or terminated. Then, using UID as a key, the FG app detector can quickly retrieve a list of foreground app's processes.

Whenever the top activity changes, the FG app detector sends an UID of the new foreground app, along with PIDs and TIDs of related Linux processes, to the Linux kernel via the `sysfs` interface. By doing this, FAIO is able to keep track of the currently executing foreground app.

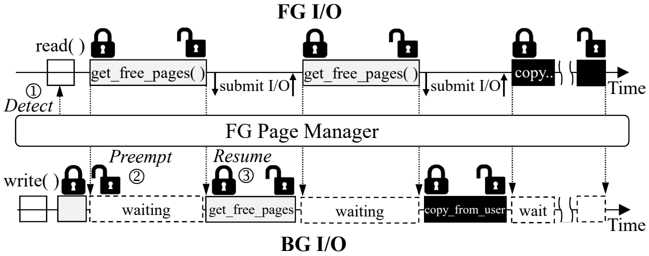


Figure 4: BG I/O preemption by FG page manager.

3.2 Foreground-centric Page Manager

The FG page manager is designed to achieve two main purposes: page allocation control and page access control. The FG page manager receives the foreground app information from the FG app detector and preempts background I/Os when foreground I/Os arrive at the page cache.

Fig. 4 shows how the FG page manager works using the same example as in Fig. 2. Suppose that the foreground app generates a read request to the kernel just after the background app issued a write request. The FG page manager intercepts a system-call invocation (i.e., read ()) and sees if the I/O request is from the foreground app or not by comparing its UID, PIDs, and TIDs numbers with the ones previously received from the FG app detector (①). If the request comes from the foreground app, the FG page manager forces background I/Os to release a global lock of the page cache just after getting a page currently being requested (②). After allocating desired pages to the foreground I/O, the FG page manager resumes the preempted background I/Os (③). At the same time, the kernel issues block I/O requests for the foreground I/O to fill up the allocated pages with data read from storage media. In a similar way, the FG page manager suspends and resumes data copy operations of background I/Os between user and kernel space.

In order to support the prompt preemption and resumption of background I/Os, we modified Linux kernel's page cache-related functions such as `alloc_pages()`, `do_generic_file_read()` and `generic_perform_write()`. For quickly preempting the background I/O, these functions were divided intro several execution segments. At the end of each segment, the FG page manager checks if there is a waiting foreground I/O or not. These checkpoints also help the suspended background I/O restart quickly because it can resume its execution where it was suspended.

Unless two user-installed apps are specially designed to work together, I/Os of a foreground app will not be dependent on those of a background app, thus preempting the background app's I/O won't affect the foreground app's execution. However, for the pre-defined Android system processes such as zygote, wifi and phone [35], the FG page manager does not preempt their I/Os because it may negatively affect the performance of the foreground app. For example, when data

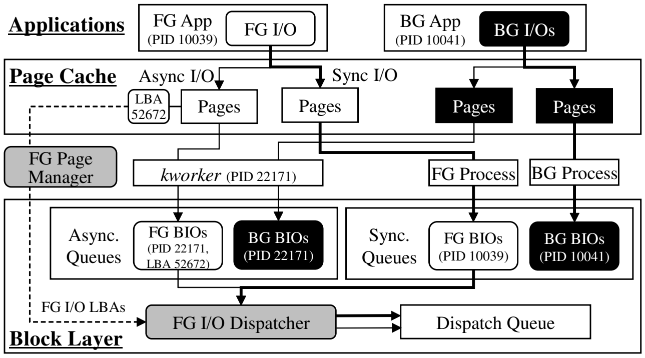


Figure 5: FG I/O acceleration by FG I/O dispatcher.

necessary for the foreground app is downloaded through a network system process, preempting the network system process's I/O may delay the foreground app's execution. In order to avoid such self-harming preemption cases, the FG page manager checks whether background I/Os belong to system processes or user-installed apps through UID and excludes system process I/Os from preemption. Since Android predefines UIDs for system processes and assigns UIDs to user-installed apps between 10,000 and 19,999 [35], the FG page manager is able to determine whether background I/Os belong to system processes or not.

3.3 Foreground-centric I/O Dispatcher

Once block I/O requests are delivered to the block layer from the page cache layer, they are put into a sync queue or an async queue in the I/O scheduler according to their types. To accelerate foreground block I/O requests (FG BIOs), the FG I/O dispatcher looks for FG BIOs in both queues and moves them to the dispatch queue immediately.

Depending on the type of a queue, the FG I/O dispatcher has to take different strategies to find FG BIOs. FG BIOs can be easily found in the sync queue using the foreground app's PID number delivered by FG app detector. In the case of async I/Os, however, the PID number of all async I/Os is the same as the PID of the kworker kthread which delivers async BIOs buffered in the page cache to the block layer on behalf of foreground processes. Since the PID number is not useless to find async FG BIOs, the FG I/O dispatcher uses LBAs as keys to fetch FG BIOs from the async queue.

Fig. 5 illustrates how the FG I/O dispatcher detects FG BIOs and delivers them to the dispatch queue. Suppose that both the process (PID 10039) of the FG app and the process (PID 10041) of the BG app generate sync I/Os and async I/Os simultaneously. For sync I/Os, since the foreground process directly puts FG BIOs to the sync queues, sync FG BIOs maintains the process PID (i.e., PID 10039). However, for async I/O, the PID of async FG BIOs is recorded as 22171, which is kworker khtread's PID. Therefore, the FG I/O dispatcher is unable to find async FG BIOs with the PID of the foreground process. The FG I/O dispatcher receives the LBA number (52672) of async FG BIOs from the FG page manager, and by utilizing the LBA, it is able to find async FG BIOs in the async queue.

Finally, whenever new BIO enter the sync/async queues, the FG I/O dispatcher prevalidates whether it is FG BIO, then directly sends FG BIO to the dispatch queue regardless of its priority in sync/async queues.

4 EXPERIMENTAL RESULTS

For the evaluation of FAIO on real systems, we implemented the FAIO modules in the Android 5.1.1 and the Linux kernel 3.10. Three different smartphones, Nexus 5 (N5), Nexus 6 (N6), and Samsung Galaxy S6 (S6), were used for evaluation. N5, N6, and S6 were equipped with a quad-core CPU and 2 GB DRAM, a quad-core CPU and 3 GB DRAM, and an octa-core CPU and 3 GB DRAM, respectively.

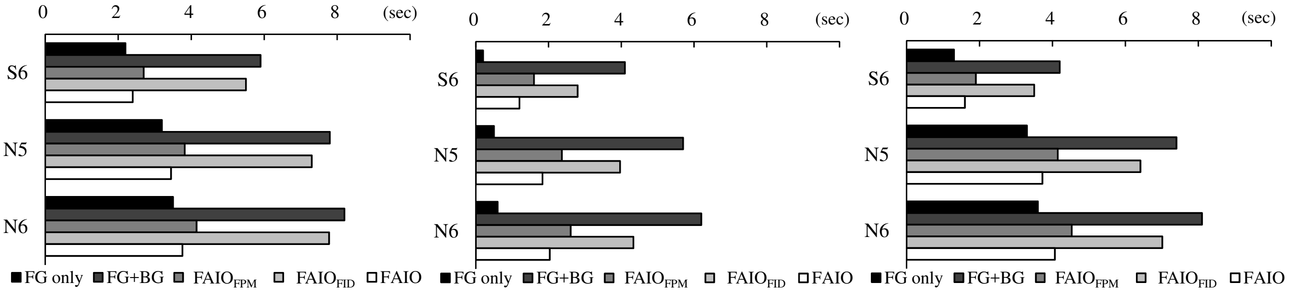
We have chosen two apps for background usage scenarios, `app-update` and `file-download`. The `app-update` scenario updates Hearthstone game [36], from Play Store, whose size is about 1.5 GB. The `file-download` scenario downloads a 3 GB movie file from the FTP server.

While running background apps, we run three foreground apps, `Gallery` (app launch), `Camera` (app suspension), and `Game` (app loading) discussed in Section 2.2. For a fair comparison, before the execution of each scenario in a 5 GHz Wi-Fi connected environment, all other apps except for foreground/background apps are terminated.

4.1 Experimental Results

We compare the performance of five different policies: FG only, FG+BG, FAIO$_{FPM}$, FAIO$_{FID}$, and FAIO. Here, FAIO$_{FPM}$ is FAIO only with FG page manager (FPM), while FAIO$_{FID}$ is FAIO employing FG I/O dispatcher (FID) only. FAIO is with both FPM and FID. FAIO$_{FPM}$, FAIO$_{FID}$, and FAIO all use the FG app detector to detect foreground I/Os.

Fig. 5 shows that FAIO reduces the user-perceive response time delays of `Gallery`, `Camera` and `Game` by up to 91%, 72% and 86% over FG+BG, respectively. In particular, in the case of the app launch times with `Gallery` and `Game`, FAIO achieves a very similar response time as FG only. Unlike `Gallery` which is a read-dominant workload, for `Camera` often issues writes to the storage device, FAIO shows slightly increased response times than FG only. This is mainly due to the inefficient device queue management of UFS and eMMC storage. If foreground reads are mixed up with background writes inside UFS's or eMMC's device queues (e.g., in the cases of `Gallery` and `Game`), the reads are serviced prior to the writes with a high priority. However, if foreground and background writes are mixed in the queue (e.g., `Camera`), they are handled in a FIFO manner according to their arrival time. Thus, even if FAIO immediately forwards foreground writes to the



(a) `Gallery` **app launch time.** **(b)** `Camera` **app suspension time.** **(c)** `Game` **app loading time.**

Figure 6: Impact of FAIO on user-perceived response time.

device at the kernel level, they are blocked inside the device by outstanding writes previously issued by background apps.

One of the interesting observations is that the impact of FID on performance improvement is negligible compared with FPM. For example, with FAIO$_{FID}$, the response times of the three scenarios are reduced by only 7%, 29%, and 21% over FG+BG, respectively. This shows that I/O optimization at the block layer has a limited effect; instead, managing foreground I/Os at a higher level (i.e., the page cache level) is more effective in reducing response time delays.

Finally, Fig. 5 also shows that FAIO works more efficiently atop a faster storage device like UFS (used in S6) than a slower one like eMMC (used in N5 and N6). In our observation, the absolute numbers of I/O latencies reduced by FAIO are almost the same, regardless of the type of underlying storage devices (i.e., UFS or eMMC). Therefore, the overall improvement ratio by FAIO becomes more significant for the fast storage, where foreground apps generally exhibit shorter response times. This means that as the storage devices get faster, the effect of FAIO becomes more substantial.

5 CONCLUSIONS

In this paper, we presented a foreground app-aware I/O management technique, called FAIO, which accelerated foreground I/Os by preempting background I/Os in the kernel I/O stacks. FAIO was motivated by our empirical findings that background I/Os significantly affected user experience. By monitoring an activity stack in the Android platform, FAIO detected I/O requests from a foreground app. With this information, the improved page cache and block I/O modules for FAIO immediately delivered the foreground I/Os to the storage device with minimum interference from in-flight background I/Os. Our experimentals showed that FAIO reduced the user-perceived response time delay by up to 91%.

FAIO achieved high performance improvement for read-dominant workloads, but the performance improvement for writes was rather limited due to inefficient device-level I/O scheduling. As future work, we plan to develop a new device-level I/O scheduling policy, which is tightly integrated with the kernel scheduler and thus can handle foreground I/Os more efficiently inside the device.

6 ACKNOWLEDGMENTS

We would like to thank Tian Guo, our shepherd, and anonymous referees for their valuable comments which greatly improved our paper. This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Ministry of Science, ICT and Future Planning (MSIP) (NRF-2015M3C4A7065645). The ICT at Seoul National University provided research facilities for this study. (*Corresponding Author: Jihong Kim*)

REFERENCES

[1] Hackborn, D. Multitasking the Android Way. https://android-developers.googleblog.com/2010/04/multitasking-android-way.html.

[2] iOS Human Interface Guidelines - Multitasking. https://developer.apple.com/ios/human-interface-guidelines/features/multitasking.

[3] Background Execution Limits. https://developer.android.com/preview/features/background.html.

[4] Multi-Window Support. https://developer.android.com/guide/topics/ui/multi-window.html.

[5] Song, W., Sung, N., Chun, B., and Kim, J. Reducing Energy Consumption of Smartphones Using User-Perceived Response Time Analysis. In *Proceedings of the International Workshop on Mobile Computing Systems and Applications* (2014).

[6] Nexus S. https://en.wikipedia.org/wiki/Nexus_S.

[7] Anwar, A., and Tanveer, O. Performance Optimization For Android. https://www.slideshare.net/arslantumbin/performance-optimization-for-android-32797106.

[8] Background Optimizations. https://developer.android.com/topic/performance/background-optimization.html.

[9] Optimizing Foreground App Performance on Nexus S. https://www.reddit.com/r/Android/comments/1wqcuh/how_do_i_make_android_manage_foreground_apps.

[10] RAM Issue on Nexus S. https://forum.xda-developers.

com/nexus-s/help/ram-issues-nexus-s-jelly-bean-t1854513.

[11] Performance Drop on Nexus S. http://forums.whirlpool.net.au/archive/1999853.

[12] The First Android Smartphone in the World with 8 GB of RAM. http://bgr.com/2017/01/05/asus-zenfone-ar-release-date.

[13] Asus ZenFone AR. https://www.asus.com/us/Phone/ZenFone-AR-ZS571KL.

[14] Android Mobile Phones with 6 GB RAM. https://www.techmanza.in/6gb-ram-mobile.html.

[15] Williams, A. How Much RAM Does a Phone Need?. http://www.trustedreviews.com/opinions/how-much-ram-does-a-phone-need.

[16] Jeong, D., Lee, Y., and Kim, J. Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2015).

[17] Kim, S., Kim, H., Lee, J., and Jeong, J. Enlightening the I/O Path: A Holistic Approach for Application Performance. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2017).

[18] Nexus 5. https://en.wikipedia.org/wiki/Nexus_5.

[19] Nexus 6. https://en.wikipedia.org/wiki/Nexus_6.

[20] Samsung Galaxy S6. https://en.wikipedia.org/wiki/Samsung_Galaxy_S6.

[21] Android, the World's Most Popular Mobile Platform. https://developer.android.com/about/android.html.

[22] Saving Data in SQLite Databases at Android Platform. https://developer.android.com/training/basics/data-storage/databases.html.

[23] Manthur, A., Cao, M., and Bhattacharya, S. The New ext4 File System: Current Status and Future Plans. In *Proceedings of Linux Symposium* (2007).

[24] Completely Fair Queueing. https://en.wikipedia.org/wiki/CFQ.

[25] Embedded MultiMediaCard (e.MMC). http://www.jedec.org/standards-documents/technology-focus-areas/flash-memory-ssds-ufs-emmc/e-mmc.

[26] Universal Flash Storage (UFS). http://www.jedec.org/standards-documents/focus/flash/universal-flash-storage-ufs.

[27] Kumar, U. Understanding Android's Application Update Cycles. https://www.nowsecure.com/blog/2015/06/08/understanding-android-s-application-update-cycles.

[28] Twitter Version History. https://www.apk4fun.com/history/2699.

[29] QuickPic Gallery. https://play.google.com/store/apps/details?id=com.alensw.PicFolder.

[30] Android Camera API. https://developer.android.com/guide/topics/media/camera.html.

[31] Kingdom Story: Brave Legion. https://play.google.com/store/apps/details?id=com.nhnent.SK10392.

[32] Physical Page Allocation. https://www.kernel.org/doc/gorman/html/understand/understand009.html.

[33] Blocking I/O. http://www.makelinux.net/ldd3/chp-6-sect-2.

[34] Memory Mapping and DMA. https://static.lwn.net/images/pdf/LDD3/ch15.pdf.

[35] Predefined UIDs for Android Processes. https://android.googlesource.com/platform/frameworks/base/+/master/core/java/android/os/Process.java.

[36] Hearthstone. https://play.google.com/store/apps/details?id=com.blizzard.wtcg.hearthstone.