FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones

Sangwook Shane Hahn, ***Seoul National University;*** **Sungjin Lee,** ***DGIST;***
Inhyuk Yee, ***AIBrain Asia;*** **Donguk Ryu,** ***Samsung Electronics;***
Jihong Kim, ***Seoul National University***

https://www.usenix.org/conference/atc18/presentation/hahn

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

FastTrack: Foreground App-Aware I/O Management for Improving User Experience of Android Smartphones

Sangwook Shane Hahn, Sungjin Lee†, Inhyuk Yee*, Donguk Ryu‡, and Jihong Kim
*Seoul National University, †DGIST, *AIBrain Asia, ‡Samsung Electronics*

Abstract

The quality of user experience on a smartphone is directly affected by how fast a foreground app reacts to user inputs. Although existing Android smartphones properly differentiate a foreground app from background apps for most system activities, one major exception is the I/O service where I/O-priority inversions between a foreground app and background apps are commonly observed. In this paper, we investigate the I/O-priority inversion problem on Android smartphones. From our empirical study with real Android smartphones, we observed that the existing techniques for mitigating I/O-priority inversions are not applicable for smartphones where frequently inverted I/O priorities should be quickly corrected to avoid any user-perceived extra delay. We also identified that most noticeable I/O-priority inversions occur in the page cache and a flash storage device. Based on the analysis results, we propose a foreground app-aware I/O management scheme, called FastTrack, that accelerates foreground I/O requests by 1) *preempting* background I/O requests in the entire I/O stacks including the storage device and 2) *preventing* foreground app's data from being flushed from the page cache. Our experimental results using a prototype FastTrack implementation on four smartphones show that a foreground app can achieve the equivalent level of user-perceived responsiveness *regardless of the number of background apps*. Over the existing Android I/O implementation, FastTrack can reduce the average user response time by 94% when six I/O-intensive apps run as background apps.

1 Introduction

As a highly interaction-oriented device, a smartphone needs to react promptly *without a noticeable delay* to user inputs. In order to minimize a user-perceived delay, which directly affects the quality of user experience on the smartphone, Android smartphones properly differentiate a foreground (or FG) app from background (or BG) apps for most system activities. For example, when CPU cores are allocated, an FG app may be allowed to use one or more CPU cores exclusively while BG apps must share CPU cores with other apps [1, 2]. Such FG app-centric resource management becomes more important for modern Android smartphones because they run more apps at the same time thanks to aggressive multitasking support. As the number of concurrent BG apps increases, an FG app may encounter more interference from BG apps unless the FG app is managed with a higher priority over the BG apps.

Unlike FG app-aware CPU management which has been extensively investigated [3, 4, 5], I/O management on smartphones has not actively considered the quality of user experience issue in designing various I/O-related techniques. FG app-oblivious I/O management was not of a big concern for older smartphones where the number of BG apps is quite limited because of a small DRAM capacity (e.g., 512 MB) [6, 7, 8]. However, on modern high-end smartphones with a large number of CPU cores and a large DRAM capacity (e.g., 8 GB) [9, 10, 11] where the number of BG apps has significantly increased (e.g., from one in Nexus S [12] to more than 8 in Galaxy S8 [13]), FG I/O requests (or FG I/Os) are more likely to be interfered with BG I/O requests (or BG I/Os). Unless FG I/Os are treated with a higher priority over BG I/Os, FG I/Os may have to wait for the completion of a BG I/O. (That is, I/O-priority inversions occur.) In this paper, we comprehensively treat the I/O-priority inversion problem on Android smartphones including its impact on user experience, its main causes and an efficient solution.

Our work is mainly motivated by our empirical observation that I/O-priority inversions between the FG I/Os and the BG I/Os are quite common on Android smartphones. In particular, when an FG app needs a large number of I/Os (e.g., when the app starts), such I/O-priority inversions significantly degraded the response time of the FG app. For example, when five BG apps run at the same time, the app launch time of an FG app can increase by up to four times over when no BG app competes. This large increase in the app launch time of the FG app was rather surprising because the Android system is already designed to handle the FG app with a higher priority. In order to understand why the response time of a higher-priority FG app is affected by the number of BG apps, we have extensively analyzed the complete I/O path of the Android/Linux I/O stack layers on several smartphones using our custom I/O profiling tool, IOPro [14]. From our analysis study, we found that I/O-priority inversions in the page cache and the storage device were main causes of the increased response time of an FG app. We also observed that the current flush policy

1.IO优先级倒置发生在page cache以及flash设备中
2.page cache flush策略没有区分数据页来源

in the page cache, which does not distinguish whether a victim page (to be flushed) is from an FG app or a BG app, significantly impacted the FG app performance.

For frequent I/O-priority inversions on a smartphone, the existing techniques such as [15, 16, 17, 18, 19] may not be applicable because these techniques require a long latency (from the smartphone's viewpoint) to correct the inverted I/O priorities. For example, [19] depends on the priority inheritance protocol [20] to accelerate the completion of the current BG I/O before an FG I/O is started. In our experiment, the FG I/O waited up to 117 ms for the completion of the current BG I/O under the priority inheritance protocol. Obviously, this is too long for a smartphone where a delay of more than a few milliseconds is unacceptable [19]. Furthermore, the efficiency of these existing techniques, which were not specifically developed for smartphones as a target system, is limited in several aspects. For example, they do not fully exploit an important hint such as the type of apps (e.g., foreground or background) and do not take a holistic approach in optimizing the entire I/O stack layers *including the storage device*. Therefore, a different approach is necessary for resolving the I/O-priority inversion problem on smartphones. A solution should meet the fast response time requirement and should better exploit the smartphone-specific hints in a holistic I/O-path-aware fashion.

整体的

In this paper, we propose a new I/O management scheme for Android smartphones, called FastTrack (or FastT in short), which efficiently meets the above requirements on resolving I/O-priority inversions on Android smartphones. The key difference of FastTrack over the existing techniques is that FastTrack takes a more direct approach in fixing the I/O priority inversion problem by *preempting* the current background activity throughout the entire I/O stack layers. By stopping the current background activity *immediately*, FastTrack can quickly service the I/O request from an FG app. FastTrack also modifies the flush policy of the page cache to be *FG app-aware*. For example, when a victim page is selected for the next flush, FastTrack first considers pages that belong to BG apps as victim candidates.

1.暂停当前后台应用
2.优先刷写后台应用page

In order to evaluate the effectiveness of the proposed scheme, we have implemented FastTrack on various Android smartphones, including Nexus 5 [21], Nexus 6 [22], Galaxy S6 [23] and Pixel [24]. Our experimental results show that FastTrack can provide the equivalent level of responsiveness to FG apps regardless of the number of BG apps. For important I/O-intensive app use cases (such as the app launch time, app switch time and app loading time)[1], compared over when no BG app runs, FastTrack can limit an increase in the average response time of an FG app within 27% even when six I/O-intensive BG apps run together. On the other hand, in the default Android implementation, the average response time can increase up to by 2,319%. Because of foreground app-centric I/O management, FastTrack is very effective in decreasing the average response time of an FG app as well. For example, when six BG apps run together, FastTrack can reduce the response time of an FG app by 94% over the default Android implementation.

The remainder of this paper is organized as follows. In Section 2, we report the key results of our empirical study on the impact of BG I/Os on user experience. Section 3 describes the root causes of the I/O-priority inversion problem on smartphones and summarizes the main requirements that must be satisfied by a solution. A detailed description of FastTrack is given in Section 4. Experimental results follow in Section 5, and related work is summarized in Section 6. Finally, Section 7 concludes with future work.

2 Impact of BG I/Os on User Experience

In this section, we empirically analyze how much FG app performance is affected by BG I/Os. We also investigate how often BG I/Os interfere with an FG app. Our empirical study is carried out under various real-life smartphone usage scenarios with 10 different smartphones.

2.1 Evaluation Study Setup

For our study, we collected 10 Android smartphones[2] (with a proper instrumentation function) from different users who are all *heavy* users (almost always carrying their smartphones with them). To avoid possible bias, we have selected the smartphones from seven different manufacturers. Each smartphone is equipped with 4 or more cores and 3 GB or larger DRAM memory which is large enough to actively support multitasking. All of the smartphones also have the latest version of Android (version 7.x) which supports enhanced multitasking features (such as a split screen).

Like other active smartphone users, our study participants used Chrome, Messenger, Camera, Gallery, and Game as their main FG apps. As popular BG apps, cloud backup apps such as Dropbox [25] and OneDrive [26] were popular among the study participants. Furthermore, all the participants enabled an option for an automatic app update. The "background process limit" option was set to "standard limit", which is a default setting.

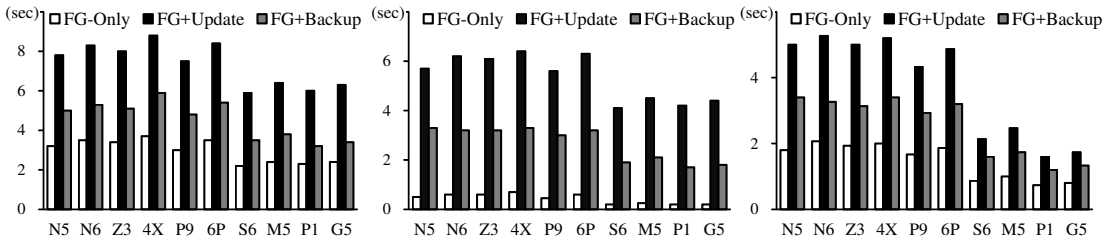
参与者

2.2 Response Time Analysis

In order to understand an impact of BG I/Os on user experience, we conducted a series of experiments on common smartphone usage cases when typical BG apps run. We have measured the user-perceived response times

[1] Since a user must wait for these use cases to complete, their response times directly affect the smartphone user experience quality.

[2] 10 phones include Nexus 5 (N5), 6 (N6), 6P (6P), Z3 (Z3), Redmi 4X (4X), P9 (P9), Galaxy S6 (S6), Mi 5 (M5), Pixel (P1), and G5 (G5).



(a) Gallery app launch time. (b) Camera app switch time. (c) Game app loading time.

Fig. 1: Impact of background I/Os on user experience.

of three usage cases where prompt reaction to user inputs are important – when (i) a new app is launched by a user, (ii) one app is switched to another, and (iii) a launched app is loading required contents. For BG apps, we have selected two BG workloads: `Update` and `Backup`. `Update` downloads and installs multiple apps from the Android market (e.g., as in an auto app update), and `Backup` uploads a large number of files to cloud storage services (e.g., as in Dropbox). These workloads were selected because they are known to generate substantial BG I/Os in a periodic fashion[3] Our background scenarios are automatically invoked in background when a smartphone is connected to a fast network (e.g., Wi-Fi).

Scenario A – Launching `Gallery` App: Launching an app requires to load a relatively large number of files, including executables, libraries, and files. While an app is being launched, a user has to wait until all the required files are loaded from a storage device. In this paper, an app launch refers to a cold start, where an app is launched without any preloaded data. It should be noted that, as the quality and complexity of mobile applications improve, the amount of data to be loaded while launching apps increases as well. In case of `Gallery` app [30], for example, 25 image files, on average, must be preloaded to complete the app's initial display.

Fig. 1(a) depicts the launch time of `Gallery` on the 10 smartphones. Here, the launch time is defined to be the time interval from when an app icon is touched by a user to when all the components are displayed on the screen. Even though there are differences depending on the hardware performance, noticeable launch time degradations are observed in all the smartphones when BG I/Os are issued simultaneously. In N6 with an eMMC storage, the launch times under two BG workloads, `FG+Update` and `FG+Backup`, increase by 2.4 times and 1.6 times, respectively, over a standalone launch (`FG-only`). Similar trends are also observed even in smartphones with faster mobile storage systems. In S6 with an UFS storage that provides higher throughput, the launch times of `FG+Update` and `FG+Backup` increase by 2.6 times and 1.7 times, respectively.

Scenario B – Switching `Camera` App: Switching from one app to another becomes a common feature in smartphones supporting multitasking. Before moving to a new app, a current app should be properly suspended. In the Android platform, the app switch involves the flushing of dirty pages in the page cache to persistent storage, so as to create as many free pages as possible for a new app. A user must wait for an old app to complete flushing its dirty pages before a new app is activated. Therefore, a user may experience a long unpleasant delay between app switches if BG I/Os interfere with the flushing process in the page cache.

We examine the switch time of a `Camera` app [31] when it switches to a home screen app. While the `Camera` app is recording a video for 10 minutes, we measure the time interval from when the home button is pressed to when the home screen is displayed for the next user interaction. Fig. 1(b) illustrates the switch time of `Camera` in S6 – it is less than 1 second when no BG I/Os are being issued, but it increases by 19.5 times under heavy BG I/Os. In N6, the switch time also increases by 11 times compared to when there are no BG I/Os.

Scenario C – Loading `Game` App: After app launching, some apps require additional file loading work before a user interacts with launched apps. One representative example is a `Game` app [32] that has to preload game contents (e.g., stage maps and rendered images) depending on a user's input after it completed the app launching process. This loading process inevitably results in response time delays from the perspective of end users.

In order to understand how much BG I/Os affect the app loading time, we measure the time interval from when the 'story mode' button on a `Game` app [32] is touched by a user to when its loading process is finished. As expected, we observed that the app loading time increases with BG I/Os in all the smartphones. We also confirm that the app loading times tend to be longer in smartphones with less memory over ones with larger memory. For example, on N5 with 2 GB DRAM, the loading time increases by 2.7 times under `Update`. The loading time, however, increases about 2 times only on P1 with 4 GB DRAM.

[3] `Update` is based on an observation that popular mobile apps (such as `Twitter`) are typically updated once every week and the average size of downloaded packages for an app update is about 110 MB [27, 28], and `Backup` is based on a report that a typical smartphone user uploads more than 200 MB of files per day to the cloud storage [29].

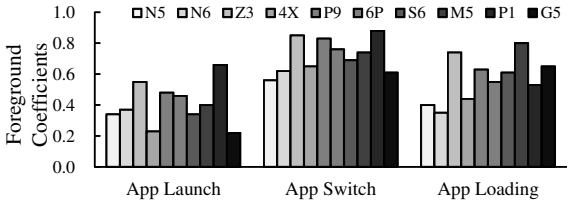


Fig. 2: FG-BG interference analysis results.

2.3 FG-BG Interference Analysis

Although we confirmed that BG I/Os can significantly degrade the quality of user experience of an FG app when they conflict with FG I/Os, if BG I/Os were unlikely to overlap with FG I/Os in practice, our response time analysis in Section 2.2 becomes less meaningful. For example, if most BG I/Os occurred while a smartphone was not actively used by a user, their actual impact on user experience would be negligible, thus making our work useless. In order to evaluate if such conflicts are really happening in real-world settings, we built a simple I/O utility[4] which can tell how much BG I/Os were issued while processing a given FG I/O *req*. If an FG I/O τ was started at t_{start} and completed at t_{finish} , our utility computes the ratio r_τ of the total amount of I/Os from τ to the total amount of I/Os in the interval $[t_{start}, t_{finish}]$. This ratio, we call the foreground coefficient C_{fg}, indicates the proportion of FG I/Os over the total I/Os in $[t_{start}, t_{finish}]$. If C_{fg} is high, it indicates that there is less interference from BG I/Os. For example, if C_{fg} is close to 1, few BG I/Os interfere with the FG I/O.

Fig. 2 shows how much BG I/Os interfere with FG apps on 10 smartphones. For each smartphone, we have collected a month's history of system call usage and computed average foreground coefficients for three use cases (explained in Section 2.2). Fig. 2 shows that a significant portion of BG I/Os can interfere with user's interaction with FG apps. For example, in the app launch scenario, FG I/Os account for only 42% of the total I/Os, which can conflict with 58% of the total I/Os that are requested from BG apps. Similarly, in the app switch scenario and the app loading scenario, FG I/Os are responsible for 77% and 53% of the total I/O requests, respectively. Although the BG I/O portion was reduced over the app launch scenario, the BG I/O portion is still large enough to affect the user experience of an FG app in a significant fashion.

3 Root Causes of User-Perceived Delay

In this section, we analyze the I/O stack of the Android platform to find root causes that are responsible for rapidly increasing user-perceived response time along with an increasing number of I/O-intensive BG apps. We first review the overall architecture of the Android I/O stack, giving a brief explanation of how apps access files in storage media. Then, we explain three major bottlenecks we found through the analysis.

[4]Our monitoring tool is based on strace which is a popular profiling utility for analyzing system calls [33]. Strace provides PIDs of processes that generate I/Os, along with detailed information of relevant I/O system calls. Using the collected information, we can distinguish BG I/Os from FG IOs with their respective I/O traffic amounts.

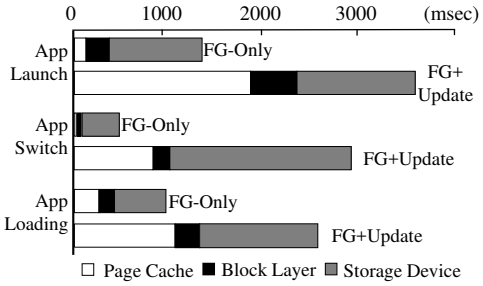


Fig. 3: A breakdown of foreground I/O execution time.

3.1 Overview of Android I/O Stack

As in typical UNIX-like OSes, Android file I/Os (i.e., reads and writes on files) created by an app are delivered to the kernel through system-call interfaces. The Linux kernel checks if corresponding file data is already cached 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/Os with designated logical block addresses (LBAs). Block I/Os are then transferred to the block I/O layer and are put into proper I/O scheduling queues, sync or async queues, according to their types. I/O scheduling algorithms (e.g., CFQ [34]) move ready-to-submit block I/Os to a dispatch queue, which will be sent to the storage device via the eMMC [35] or UFS [36] interface. If the file I/O is for reads, data read from the storage device is stored in the allocated pages, and the data is finally copied to the user-space buffer.

In order to analyze the root causes of performance degradation by BG I/Os, we have measured the execution time of FG I/Os using IOPro. IOPro is capable of measuring the detailed elapsed times of I/O requests across all the Android I/O stacks, including a page cache, a block I/O layer, and a storage device. Fig. 3 shows a breakdown of the I/O execution time observed in the three usage scenarios used in Section 2. Because of the space limit, only the results from S6 are displayed in Fig. 3, but other smartphones also exhibit similar performance trends. When there are no BG I/Os (denoted by FG-Only in Fig. 3), the storage device is a major bottleneck. This is a reasonable result because the storage device is considered the slowest component in the I/O stack hierarchy.

With BG apps running heavily (FG+Update in Fig. 3), we observe that the execution times increased consider-

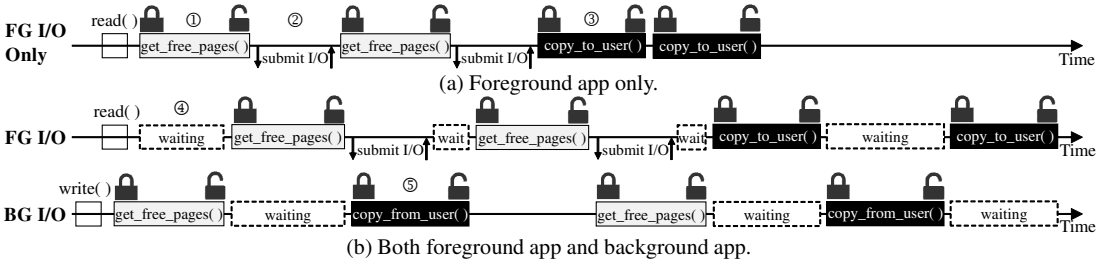


Fig. 4: Impact of lock contention on the I/O latency of the foreground app.

ably across all the layers. In particular, the times spent by the page cache layer have increased by 12 times, on average. For example, in Fig. 3, the portions of the page cache in `FG-Only` were negligible, but these rapidly increased in `FG+Update` to 32%-51% of the total execution times. While the relative portion of the time spent by the storage device has decreased, the total execution time spent by the storage device has significantly increased. For example, in the case of the app switch scenario, the execution time on the storage device has increased by 5 times. One surprising result in our study was that the impact of the block I/O layer on performance was rather negligible compared with the other two layers.

In the following three subsections, we investigate what happens inside the kernel I/O stacks when BG I/Os are heavily issued. Particularly, we focus on analyzing internal I/O activities at the page cache and the storage device layers because they are the main contributors to the increase of the execution times.

3.2 Root Cause 1: Page Allocation

From our performance bottleneck study, we found that *lock contentions in the page cache layer* are responsible for many I/O-priority inversions we have observed. As the first and major root cause of a performance degradation of an FG app under BG apps, we explain the impact of the page allocation module on user experience. When a new I/O request arrives at the kernel, free pages should be assigned first to the I/O request. When new free pages are necessary for serving the incoming I/O request, a free-page allocation module first acquires a *global lock*, *page lock*, for the exclusive access of the page cache during the page reclamation process [37] which is *non-preemptive* [38]. Acquiring free pages is mostly done quickly. However, if there are not sufficient free pages available, it takes a rather long time (e.g., more than 200 ms [19]) to create free pages by evicting dirty pages. Evicting dirty pages require extra writes to the storage device. If FG I/Os are blocked by BG apps that need the free page reclamation process, an FG app has to wait for BG I/Os to finish, thus causing an I/O-priority inversion between the FG app and the BG apps.

Fig. 4(a) illustrates an example where an FG app *F* reads a photo file of 256 KB size from storage media by calling a `read()` system call. We compare two different cases: 1) when *F* runs alone without any BG apps and 2) when *F* runs together with a BG app *B* that writes a large file to the storage device. Without BG apps, the FG 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 [39], 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).

Suppose that the BG app calls the `write()` system call to write data just before `read()` is invoked by the FG app. The *page lock* is grabbed by the BG app first, so the FG app has to wait until it releases the *lock* (④). This could be quite long if dirty page evictions are involved while assigning free pages to the BG app. After the *page lock* is released by the BG app, the FG app is able to acquire the lock, allocates free pages for reads, and then 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 4(b), if the BG app has already acquired the *global lock*, the FG app has to wait again for the lock to be released, which increases 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 BG apps run (e.g., `Update`), many dirty pages are created in the page cache and available free pages quickly run out. If an FG app requests I/Os in such situations, frequent evictions of dirty pages are inevitable.

3.3 Root Cause 2: Page Replacement

Our second root cause comes from a somewhat surprising source. As discussed in Section 3.2, the performance degradation of an FG app from the lock contention mostly occurs when many dirty pages are created in the page cache. When BG apps are read-dominant, such performance degradation is difficult to occur because few dirty pages may exist in the page cache. For example, in ④ of Fig. 4(a), if reclaimed pages were

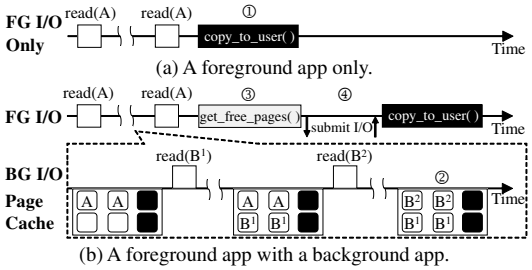


Fig. 5: Impact of page replacement.

clean, no writes to the storage device would be needed. Unlike our reasoning about read-dominant BG apps, our experiments revealed an interesting result that *even read-dominant BG apps can often interfere with an FG app*.

Although it was not straightforward to understand why such unexpected performance degradation occurs, we identified the page replacement policy in the page cache as the main cause. The existing Linux page replacement policy in the page cache works in an FG app-oblivious fashion. That is, the Linux kernel prefers choosing a clean page as a victim because of its cheap replacement cost regardless of whether the owner of the victim page is an FG app or a BG app. Suppose that BG apps want to read a large amount of data from the storage and they need to get more free pages by evicting existing ones from the page cache. In this situation, the Linux kernel often selects clean pages of an FG app even though those clean pages are soon to be accessed. Although choosing a clean page as a victim page is reasonable from minimizing the eviction cost, it is a bad decision for the FG app because a large page cache miss penalty can significantly increase the FG app response time.

Fig. 5 shows a 具体的 concrete example of how a read-dominant BG app negatively affects an FG app. Here, the FG app *F* is assumed to read a file `A` twice by calling `read()`. Again, we compare two different cases: 1) when *F* solely runs and 2) when *F* runs together with a BG app which read a large file B from the storage device. Without BG apps, the FG app can quickly finish the second `read()` by reading the file `A` from the page cache (①). However, when the FG and BG apps run simultaneously, some pages of the file `A` may be evicted from the page cache (②) to create a room for the large file `B`. After the completion of BG reads, when the FG app tries to read the file `A` again, free pages should be allocated (③) and the previously-evicted pages should be read from the storage device again (④). Even worse, from our investigations on real-life app usage scenarios, we observed that many FG apps exhibit high temporal locality, repeatedly referencing the same files. For such an FG app, the existing page cache replacement policy can significantly degrade the user experience by evicting performance-critical hot pages of the FG app.

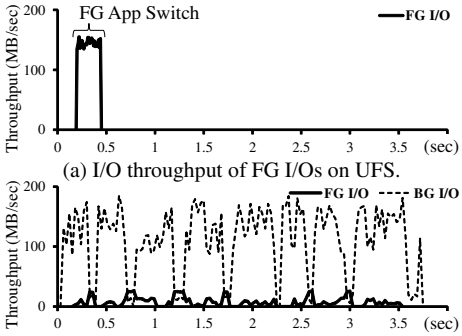


Fig. 6: I/O priority inversions in flash storage device.

3.4 Root Cause 3: Device I/O Scheduling

After our bottleneck study on the page cache layer, we investigated the block layer as a next candidate for the I/O priority inversion problem. We analyzed how the block layer processes I/O requests from when the I/O requests are put into the I/O scheduler queue to when an interrupt handler receives signals notifying the completion of the requests in the storage device. Our investigation revealed that the I/O priority inversion problem occurred *in the storage device* rather than in the block layer.

存储设备内部的调度策略

Once the storage device gets I/Os from a block device driver, it processes them according to its own I/O scheduling algorithm. The storage device generally gives a higher priority to reads than writes because reads have a higher impact on user-perceived response time. For the same type of I/O requests, the storage device process them in an FIFO manner with no preference. Although this generic scheduling policy works reasonably well for equal-priority I/O requests, it causes I/O-priority inversions very frequently because the scheduling policy inside the storage device is not aware of the priority of an I/O request. For example, if FG writes and BG reads are sent to the storage device, the FG writes would be delayed until all the BG reads complete.

1.读优先
2.FIFO

Fig. 6 illustrates the negative impact of a priority-unaware I/O scheduler inside a storage system on the throughput of FG I/Os. It plots the throughputs of FG I/Os and BG I/Os in the app switch scenario, where an FG app writes a large number of files, while huge files are being read in background. Note that the I/O throughputs were measured at the block device driver in order to device-level performance. Unlike the `FG-Only` case shown in Fig. 6(a), a significant degradation of the FG I/O throughput is observed in Fig. 6(b) when FG writes and BG reads are mixed inside the storage system. The app switch scenario, which was completed in 0.45 seconds without BG I/Os, took 3.55 seconds to finish. Our additional experiments showed that the I/O-priority inversion problem within the storage device occurs very frequently whenever FG writes are mixed with BG reads and its impact on an FG app is very serious.

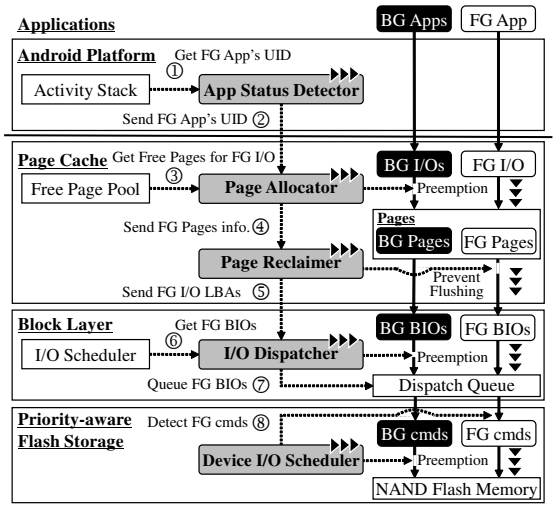


Fig. 7: An overall architecture of FastTrack.

4 Design and Implementation of FastTrack

As explained in the previous section, high-priority FG I/Os are unintentionally delayed by low-priority BG I/Os for various reasons across the entire I/O stack. One of the most commonly used solutions to resolve the I/O-priority inversion problem is to use the priority inheritance protocol that raises a priority of ~~B~~BG I/Os. The priority inheritance, however, is not effective in our cases – it still requires an FG app to wait for BG I/Os to finish, creating long delays to latency-sensitive smartphone users.

An ideal solution to resolve the problem is to create a *vertically-integrated fast I/O path* which is dedicated to serving FG I/Os across the entire I/O stack, including a page cache, a block layer, and a storage device. In other words, if it is possible to quickly preempt BG I/Os upon the arrival of FG I/Os and to deliver them directly to the storage device with minimal interference by I/O stack layers, it would be possible to provide the equivalent level of user-perceived responsiveness as when there is no BG I/O. Key technical challenges here are (1) how to identify FG I/Os from BG I/Os inside the kernel, (2) how to preempt BG I/Os immediately, and (3) how to prevent potential side effects that could occur when creating such a new I/O path.

Keeping these technical challenges in mind, we design the app status-aware I/O management, FastTrack, with five modules as illustrated in Fig. 7. The app status detector obtains the information of the current FG app by monitoring the activity stacks of the Android platform (①) and forwards it to the page allocator (②). Using this, the page allocator is able to identify I/O requests from the FG app, suspending the currently executing BG I/O jobs. The page allocator then grabs a global lock of the page cache, preferentially assigning free pages to FG I/Os, regardless of their arrival time (③). Until the page allocator releases the lock, BG I/Os are postponed.

If there are not enough free pages to handle I/O requests, the page reclaimer evicts kernel pages that belong to BG I/Os as victims, preventing FG pages from being flushed from the page cache (④). After acquiring all the free pages required, the page reclaimer builds up block I/Os for FG 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 I/O dispatcher suspends servicing BG BIOs by limiting I/O queueing and then immediately delivers FG BIOs to the dispatch queue (⑥ and ⑦). When FG BIOs are converted to FG commands (FG cmds) for the storage device, the I/O dispatcher tags an FG I/O flag so that the device I/O scheduler suspends the BG I/O execution (⑧), and FG cmds can be processed immediately in the storage device.

4.1 App Status Detector

In order to identify an FG app among all the apps available in the system, the app status 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., an FG app), the FG app information 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 the Linux processes connected to the FG app. A list in question can be obtained by examining all the processes in Linux's process tree. However, such an exhaustive search on the process tree takes a relatively long time. Therefore, the app status 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 app status detector quickly retrieves a list of FG app's processes.

维持应用UID到进程PID的表

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

4.2 Page Allocator

The page allocator is designed to preferentially allocate kernel pages to I/O requests from an FG app by suspending outstanding BG I/Os. Fig. 8 shows how the page allocator works using the same example in Fig. 4, where the FG app generates read requests to the kernel just after the BG app issued write requests. The page allocator sees if the I/O request is from the FG app or not by

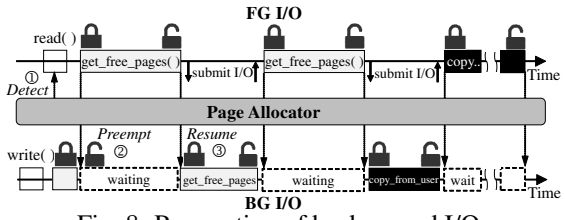


Fig. 8: Preemption of background I/Os.

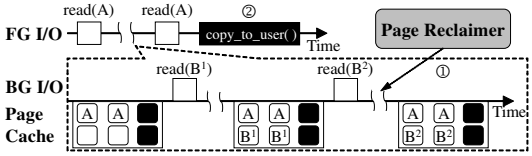


Fig. 9: Prevention of foreground page evictions.

comparing its UID, PIDs, and TIDs with the ones that it previously got from the app status detector (①). If it is from the FG app, the page allocator forces BG 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 FG I/Os, the page allocator resumes the preempted BG I/Os (③). At the same time, the kernel issues FG BIOs to fill up the allocated pages with data read from storage media. In a similar way, the page allocator suspends and resumes data copy operations of BG I/Os between the user and kernel space.

In order to support the prompt preemption and resumption of BG I/Os, we modified the major kernel functions relevant to the page cache, including `alloc_pages()`, `do_generic_file_read()`, and `generic_perform_write()`. These functions are divided into several execution segments. At the end of each segment, the page allocator checks if there are waiting FG I/Os. If so, the page allocator promptly suspends BG I/Os, unlocks the page-cache lock, and yields the CPU for the FG I/Os. After serving FG I/Os, the suspended BG I/Os restart at the point where they were suspended.

While conceptually simple, the implementation of the preemptive page cache raises two technical issues. Firstly, giving the highest favor to FG I/Os does not guarantee the improved response time all the time, and, in the worst case, it may result in serious response time degradation or even application deadlocks. Imagine an application that downloads files from the network and performs certain operations on the download files. The application model of Android requires an app to offload such a typical task to a built-in process that runs as a background service. In case of a file download, a network service process performs downloading files in background on behalf of a user app. If I/O requests from the network service process are preempted for FG I/Os, the execution of the FG app that initiates the file download would be delayed for a long time. Fortunately, the Android system does not allow such dependency between conventional user apps (e.g., game and camera apps) [40]. To avoid the self-harming preemption mentioned above, therefore, it is only necessary to prevent the preemption for BG I/Os from service processes. To do this, the page allocator checks if BG I/Os are issued by services or not and excludes them from the preemption if they are from service processes. This I/O filtering can be done by checking UID because the Android system assigns predefined UIDs to service processes, giving UIDs ranging from 10,000 to 19,999 to user apps [41].

Secondly, performing the preemption at the page cache level is not always possible. Some pieces of the kernel code must run in a special context, called an *atomic context* [42], which does not allow a CPU to go into sleep. Representative examples are interrupt handlers and critical section codes wrapped by spinlocks. The page allocator modifies the page cache functions that are also invoked by other parts of the kernel for various purposes. Thus, the page allocator should disable the preemption if it is called by a caller running in the atomic context. It is straightforward to know whether the page allocation is requested inside the atomic context. In the Linux kernel, a caller function should let the memory allocator know which type of contexts it runs now as an argument (e.g., `GFP_ATOMIC`). The page allocator refers this information and prevents the preemption if the allocation is requested inside the atomic context.

4.3 Page Reclaimer

In addition to preempting BG I/Os to accelerate page allocation for FG I/Os, the page reclaimer improves the performance of FG apps by preventing the eviction of kernel pages belonging to the FG apps.

Fig. 9 illustrates how the page reclaimer operates using the same example in Fig. 5, where the FG app attempts to read the file A twice, while the BG app is heavily reading the large file B. In Fig. 5, the second read to the file A is not hit by the page cache since it was chosen as a victim and was evicted from the page cache (② in Fig. 5). As explained earlier, this is due to the kernel's page replacement that evicts clean pages first, regardless of the status of an app. The page reclaimer prevents such a problem by adopting new replacement priorities for victim selection: *BG clean pages* (highest) > *BG dirty pages* (high) > *FG clean pages* (low) > *FG dirty pages* (lowest). With the new policy, clean pages belonging to BG apps are preferentially evicted when there is insufficient memory. In Fig. 9, the pages labeled by B^1 are evicted for B^1, even though the pages for the file A were least recently referenced (① in Fig. 9).

Keeping FG pages in the page cache wouldn't be effective if an FG app has low temporal locality. In the worst case, it would degrade the performance of BG apps without any performance improvement on an FG app. According to the mobile app workload study [43], how-

ever, the majority of the apps have high degrees of data locality. Thus, the negative effects of the page reclaimer are expected to be minimal in smartphone usages.

4.4 I/O Dispatcher

The primary goal of FastTrack is to create a fast I/O path for FG I/Os in the entire kernel layer. To this end, it is also required to enhance the block I/O layer, together with the page cache layer. Once block I/Os are delivered to the block layer from the page cache, they are put into a sync queue or an async queue in the I/O scheduler according to their types. To accelerate FG BIOs, the 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 I/O dispatcher has to take different strategies to find FG BIOs. FG BIOs can be easily found in the sync queue using the FG app's PID number delivered by the app status 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 to the block layer on behalf of FG processes. Since the PID number is useless to find async FG BIOs, the I/O dispatcher uses LBAs as keys to fetch FG BIOs from the async queue.

Finally, whenever a new BIO enters the sync/async queues, the 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.

4.5 Device I/O Scheduler

In order for FastTrack to achieve its maximum benefit, a storage device, which is at the lowest layer in the I/O stack, needs to be enhanced as well. According to [44, 45, 46], modern flash storage maintains its own internal queue, but is unaware of the status of applications issuing I/Os. To make a storage device FG I/O-aware, we modify an SCSI command set so that it carries an additional flag in a reserved opcode [47] that specifies whether I/O requests are issued by FG apps or not. This flag is used as a hint for a device-level I/O scheduler to decide the execution order of I/O requests staying in the internal I/O queue. In our current design, we assign the highest priority to FG reads, followed by FG writes and BG reads. BG writes are assigned to the lowest priority.

5 Experimental Results

In order to quantitatively evaluate the effect of FastT, we implemented the FastT modules in the Android 7.1.1 and the Linux kernel 3.10.61. Four smartphones, Nexus 5 (N5), Nexus 6 (N6), Galaxy S6 (S6) and Pixel (P1) were used for our evaluation. N5 and N6 use eMMC-based storage devices while S6 and P1 employ UFS-based storage devices. (Note that UFS supports 3 times higher throughput over eMMC.) All the smartphones were connected to the Internet through a 5-GHz Wi-Fi.

We have chosen two background usage scenarios: `Update` for a write-dominant workload and `Backup` for a read-dominant workload. The `Update` scenario updated Hearthstone game [48] downloaded from Play Store, whose size was about 1.5 GB. The `Backup` scenario uploaded 1 GB of data files to cloud storage.

While running BG apps, we executed three FG apps, `Gallery` (app launch), `Camera` (app switch), and `Game` (app loading) discussed in Section 2.2. `Gallery` was a read-dominant workload, `Camera` was a write-dominant workload, and `Game` was a mixed workload. In order to accurately measure performance, all other apps were terminated before the experiment.

5.1 Performance Analysis on Smartphones

As the response time lower limit of an FG app, we first measured user-perceived response time of the FG app when only the FG app ran without any BG apps. To understand the impact of a BG app on performance, we also measured performance when both FG and BG apps ran simultaneously on the unmodified kernel. The above two cases are denoted by FG-only and FG+BG, respectively. We compared the performances of FG-only and FG+BG with four different versions of FastT: PA, PR, ID and FastT$^-$. PA, PR, ID represents FastT with a single main component only, that is, the page allocator, the page reclaimer, and the I/O dispatcher only, respectively. FastT$^-$ employs all of the main components but it uses the existing storage device I/O scheduler[5]. In all the FastT versions we tested, the app status detector was enabled by default.

Fig. 10 shows that, for six different combinations of FG and BG apps, FastT$^-$ reduced the user-perceived response times by 74% over FG+BG, on average. PA exhibited significant performance improvements when BG apps were write-dominant (i.e., `Update`). `Update` required a copy of data from the user space to the kernel, which involved the allocation of free pages in the page cache. PA not only made this acquisition process preemptible, but also gave a higher priority to an FG app so that it got free pages prior to BG apps. By doing this, PA was able to prevent FG I/Os from being blocked by BG writes. Unlike PA, PR mostly contributed to reducing user-perceived response time when the read-dominant BG app (i.e., `Backup`) ran. In our observation, `Backup` required many free pages to load files from the storage before sending them to cloud storage. To create free pages, it often selected clean pages belonging to FG apps as victims, which resulted in the eviction of hot data from the page cache. PR prevented those clean pages from being evicted from the page cache, thereby reducing the

[5]Unfortunately, we cannot access the firmware inside the storage device. For a complete FastT implementation, we use an emulated storage as discussed in Section 5.2.

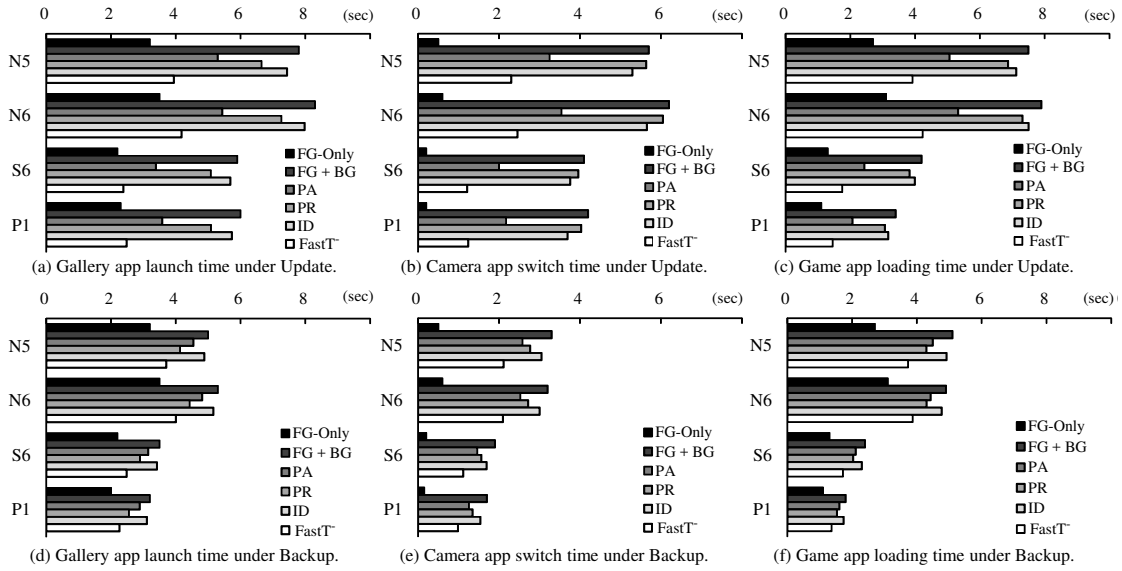


Fig. 10: Response time analysis on smartphones.

number of reads from the storage which were not necessary when only FG apps ran. ID improved the response times by 11% on average, but its impact on performance was negligible compared with PA and PR. This result confirmed our hypothesis that rescheduling I/O requests at the scheduler level was less efficient than doing it at a higher level – a page cache. As expected, by integrating the three techniques, FastT$^-$ exhibited the best performance among all the versions evaluated.

Fig. 10 also shows that FastT$^-$ works more efficiently atop a faster storage device like UFS (used in S6 and P1) than a slower one like eMMC (used in N5 and N6). In our observation, the absolute numbers of I/O latencies reduced by FastT$^-$ are almost the same, regardless of the type of underlying storage devices (i.e., UFS or eMMC). Therefore, the overall improvement ratio by FastT$^-$ becomes more significant for the fast storage, where FG apps generally exhibit shorter response times. This means that as the storage devices evolve in its speed, the effect of FastT becomes more substantial.

Even though FastT$^-$ gave FG I/Os the highest priority combined with a fast I/O path, we still observed that FastT$^-$ showed longer response times than FG-Only in all the scenarios. When we compare Fig. 11(a) and Fig. 6(a), the throughput of FG I/Os was not improved as much as `FG-Only`. This is because FastT$^-$ cannot resolve the I/O-priority inversion problem inside the storage device. Because of a priority-unaware I/O scheduler, for example, FG writes are always delayed by BG reads.

5.2 Performance Analysis on Emulator

Although the evaluation results in Section 5.1 showed that FastT$^-$ is quite effective on real smartphones, FastT$^-$ didn't reveal the full potential of our proposed

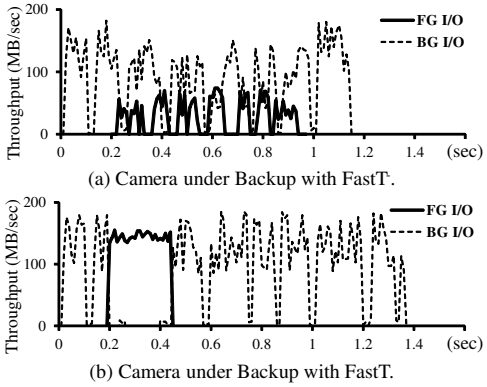
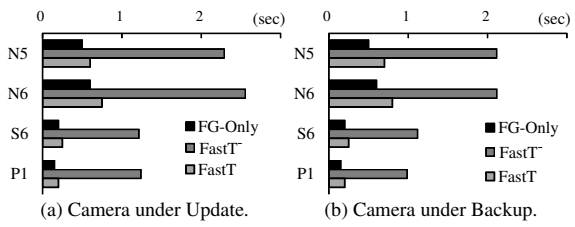


Fig. 11: Storage-level snapshot of FG I/Os and BG I/Os.

FastT because it cannot fully control the storage device-internal I/O scheduler. In order to better understand the real effect of FastT on user experience, it is important to implement the proposed I/O device scheduler (in Section 4.5) with a complete support for the fast I/O path from the Android platform to the storage device. Since storage vendors do not allow to modify their firmware inside their storage devices, we performed evaluations using an emulated storage device with I/O traces collected from real smartphone apps.

We have implemented an emulation layer on top of an off-the-shelf SSD to emulate I/O latency and throughput of eMMC and UFS devices. This work is done by using a storage emulator developed for our prior studies [14]. Then, we collected the app status information, along with I/O traces, while executing scenarios described in Section 5.1 on the smartphone. The collected I/O traces were replayed on the emulated storage. Finally, we implemented the device I/O scheduler on the emulated storage which processed FG I/Os with a higher priority.



(a) Camera under Update. (b) Camera under Backup.

Fig. 12: Response time analysis on emulator.

We compared the performance of three policies: FG-only, FastT$^-$, and FastT, where FastT is FastT$^-$ with the device I/O scheduler. Fig. 12 shows that FastT greatly improved the performance of `Camera` under both `Update` and `Backup`. In the case of `Camera` under `Update` on N6, FastT$^-$ achieved the response time of 2.53 seconds, whereas FastT achieved the response time of 0.75 seconds, which is quite close to 0.6 seconds of `FG-Only`. For `Camera` under `Backup`, similarly, FastT achieved an equivalent response time to `FG-Only`. Compared with Fig. 11(a), in Fig. 11(b), we observe that FG I/Os were processed at a much higher throughput with negligible delays at the storage device level. This result shows that higher performance can be achieved if the storage device is able to handle I/O requests with the app-level priority information.

Finally, Fig. 13 shows how FastT scales when the number of BG apps increases from two to six. In addition to `Update` and `Backup`, we used four more I/O-intensive BG apps for this experiment. As shown in Fig. 13, the normalized app switch time increases from 1.1 to 1.27 only as the number of BG apps increases from 2 to 6. These results indicate that FastT can provide the equivalent level of responsiveness to an FG app regardless of the number of BG apps running with the FG app. Fig. 13 also shows that FastT is effective in improving the app switch time over the existing Android implementation (indicated by `FG+BG`), reducing the app switch time by 94% when 6 BG apps run.

6 Related Work

Various I/O scheduling techniques have been proposed to address the problem caused by BG I/Os [18, 19, 49, 50]. A boosting quasi-async I/O (QASIO) is one of such efforts to provide better I/O scheduling by means of the priority inheritance protocol [18]. QASIO is motivated by an observation that high-priority sync writes are often delayed by low-priority async writes. QASIO improve overall I/O responsiveness by temporarily increasing the priority of async writes over sync ones. A request-centric I/O prioritization (RCP) [19] is proposed which is also based on the priority inheritance protocol. RCP further improves QASIO by prioritizing I/O requests at the page cache layer rather than the block I/O layer.

While still effective, both QASIO and RCP have fundamental limitations in improving I/O responsiveness, in comparison to `FastTrack`. First, both techniques are not aware of FG I/Os and BG I/Os in smartphones, and thus, they are unable to prioritize FG I/Os that have a high impact on user-perceived response times. Second, QASIO and RCP both rely on the priority inheritance protocol. Thus, they cannot remove additional delays required for high-priority I/Os to wait until low-priority ones finish. Therefore, their effectiveness on improving user-perceived latency is limited on highly interaction-oriented devices like smartphones.

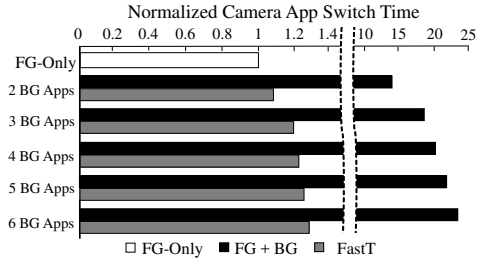


Fig. 13: Scalability of FastT over the varying BG apps.

Foreground app-aware I/O management (FAIO) [49] is the first technique that accelerates FG I/Os by adopting I/O preemption in smartphones. FAIO analyzes FG app information to identify FG I/Os and preempts BG I/Os to quickly process FG I/Os. However, since FAIO uses I/O preemption only at the page cache level, it does not resolve the priority inversion problem at the storage device level. It also fails to prevent performance degradation caused by the aggressive evictions of FG data from page cache under BG I/O intensive workloads.

7 Conclusions

We have presented a foreground app-aware I/O management scheme, FastTrack, which significantly improves the quality of user experience on smartphones by avoiding I/O-priority inversions between a foreground app and background apps on Android smartphones. Unlike the existing techniques, FastTrack employs a preemption-based approach for fast responsiveness of a foreground app. In order to support I/O-priority-based preemption in a holistic fashion, FastTrack reimplemented the page cache in Linux and the storage-internal I/O scheduler which previously operated in a foreground app-oblivious fashion. From a systematic analysis study, these two modules were identified as the root causes of most I/O-priority inversions. Our experimental results on real smartphones show that FastTrack is effective in improving the quality of user experience on smartphones. For example, FastTrack achieved the equivalent quality of user experience of a foreground app regardless of the number of concurrent background apps.

FastTrack can be extended in several directions. For example, we believe that our preemption-based approach can be extended to other computing environments where a strong requirement on the response time exists.

8 Acknowledgments

We would like to thank anonymous referees for valuable comments that greatly improved our paper. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (Ministry of Science and ICT) (NRF-2015M3C4A7065645 and NRF-2018R1A2B6006878). The ICT at Seoul National University provided research facilities for this study. Sungjin Lee was supported by the NRF grant funded by the Korea government (Ministry of Science and ICT) (NRF-2017R1E1A1A01077410) and the DGIST R&D Program of the Ministry of Science and ICT (18-EE-01). (*Corresponding Author: Jihong Kim*)

References

[1] Android Performance Management. https://source.android.com/devices/tech/power/performance.

[2] Android Process. https://developer.android.com/reference/android/os/Process.html.

[3] KWON, Y., LEE, S., YI, H., KWON, D., YANG, S., CHUN, B., HUANG, L., MANIATIS, P., NAIK, M., AND PAEK, Y. Mantis: automatic performance prediction for smartphone applications. In *Proceedings of the USENIX Conference on Annual Technical Conference* (2013).

[4] MITTAL, T., SINGHAL, LOKESH., AND SETHIA, D. Optimized CPU Frequency Scaling on Android Devices Based on Foreground Running Application. In *Proceedings of the Fourth International Conference on Networks and Communications* (2013).

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

[7] RAM Issue on Nexus S. https://forum.xda-developers.com/nexus-s/help/ram-issues-nexus-s-jelly-bean-t1854513.

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

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

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

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

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

[13] Samsung Galaxy S8. https://en.wikipedia.org/wiki/Samsung_Galaxy_S8.

[14] HAHN, S.S., LEE, S., JI, C., CHANG, L., YEE, I., SHI, L., XUE, C.J., AND KIM, J. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter In *Proceedings of the USENIX Annual Technical Conference* (2017).

[15] KIM, H., LEE, M., HAN, W., LEE, K., AND SHIN, I. Aciom: Application Characteristics-aware Disk and Network I/O Management on Android Platform. In *Proceedings of the International Conference on Embedded Software* (2011).

[16] VALENTE, P., AND ANDREOLINI, M. Improving application responsiveness with the BFQ disk I/O scheduler. In *Proceedings of the 5th Annual International Systems and Storage Conference* (2012).

[17] NGUYEN, D. Improving smartphone responsiveness through I/O optimizations. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous* (2014).

[18] 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).

[19] 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).

[20] Priority Inheritance. https://en.wikipedia.org/wiki/Priority_inheritance.

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

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

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

[24] Pixel. https://en.wikipedia.org/wiki/Pixel_(smartphone).

[25] Dropbox. https://en.wikipedia.org/wiki/Dropbox_(service).

[26] OneDrive. https://en.wikipedia.org/wiki/OneDrive.

[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] DRAGO, I., MELLIA, M., MUNAFO, M.M., SPEROTTO, A., SADRE, R., AND PRAS, A. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the Internet Measurement Conference* (2012).

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

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

[32] Dragon Ball Z Dokkan Battle. https://play.google.com/store/apps/details?id=com.bandainamcogames.dbzdokkanww.

[33] strace - trace system calls and signals. https://linux.die.net/man/1/strace.

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

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

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

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

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

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

[40] Android App Dependency Configuration. https://developer.android.com/studio/build/dependencies.html.

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

[42] Atomic context and kernel API design. https://lwn.net/Articles/274695/.

[43] JEONG, S., LEE, K., SON, S., AND WON, Y. I/O Stack Optimization for Smartphones. In *Proceedings of the USENIX Conference on Annual Technical Conference* (2013).

[44] NAM, E.H., KIM, B.S.J., EOM, H., AND MIN, S.L. Ozone (O3): An Out-of-Order Flash Memory Controller Architecture. *IEEE Transactions on Computers* (2013), vol. 60, pp. 653-666.

[45] HAHN, S.S., LEE, S., AND KIM, J. SOS: Software-based out-of-order scheduling for high-performance NAND flash-based SSDs. In *Proceedings of IEEE Symposium on Mass Storage Systems and Technologies* (2013).

[46] JUNG, M., CHOI, W., SHALF, J., AND KANDEMIR, M.T. Triple-A: a Non-SSD based autonomic all-flash array for high performance storage systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2014).

[47] SCSI command. https://en.wikipedia.org/wiki/SCSI_command.

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

[49] HAHN, S.S., LEE, S., YEE, I., RYU, D., AND KIM, J. Improving User Experience of Android Smartphones Using Foreground App-Aware I/O Management. In *Proceedings of the Asia-Pacific Workshop on Systems* (2017).

[50] JAUHARI, R., CAREY, M.J., AND LIVNY, M. Priority-Hints: An Algorithm for Priority-Based Buffer Management. In *Proceedings of the International Conference on Very Large Data Bases* (1990).