Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices

Daeho Jeong, *Samsung Electronics Co.;* Youngjae Lee and Jin-Soo Kim, *Sungkyunkwan University*

https://www.usenix.org/conference/fast15/technical-sessions/presentation/jeong

This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).

February 16–19, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-201

Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices

Daeho Jeong[*+], Youngjae Lee[+], and Jin-Soo Kim[+]

[*] *Samsung Electronics Co., Suwon, Korea* [+] *Sungkyunkwan University, Suwon, Korea*

Abstract

Providing quick system response for mobile devices is of great importance due to their interactive nature. However, we observe that the latency of file system operations increases dramatically under heavy asynchronous I/Os in the background. A careful analysis reveals that most of the delay arises from an unexpected situation where the file system operations are blocked until one or more asynchronous I/O operations are completed. We call such an I/O – which is issued as an asynchronous I/O but has the property of a synchronous I/O as some tasks are blocked on it – *Quasi-Asynchronous I/O (QASIO)*.

We classify the types of dependencies between tasks and QASIOs and then show when such dependencies occur in the Linux kernel. Also, we propose a novel scheme to detect QASIOs at run time and boost them over the other asynchronous I/Os in the I/O scheduler. Our measurement results on the latest smartphone demonstrate that the proposed scheme effectively improves the responsiveness of file system operations.

1 Introduction

Mobile devices such as smartphones and tablet PCs have become one of the most popular consumer electronics devices. According to the Gartner's survey, the worldwide shipments of mobile devices are estimated at 2 billion units in 2013 and are expected to be nine times higher than those of traditional PCs by 2015 [7].

A key feature in providing satisfactory user experiences on mobile devices is fast responsiveness. Since most applications running on mobile devices interact with users all the time, quick system response without any noticeable delay is of great importance. In spite of the increase in storage capacity and significant improvement in its performance, storage is still often blamed for impairing end-user experience in mobile devices [9].

Several efforts have been made to understand the I/O characteristics of popular mobile applications and their implications on the underlying storage media, NAND flash memory [9, 12]. Based on such investigations, various optimizations have been proposed for the I/O stack of the mobile platform including file systems, I/O schedulers, and block layers [10, 8, 11, 18]. However, previous approaches mostly view the problem from the perspective of increasing throughput or enhancing the lifetime of NAND flash memory.

In this paper, we focus on the latency of file system operations such as `creat()`, `write()`, `truncate()`, `fsync()`, etc. under heavy I/O load. We observe that when the system has lots of I/O requests issued asynchronously, the latency of these file system operations increases dramatically so that the responsiveness of applications is severely degraded. In our evaluations with one of the latest Android-based smartphones, the time to launch an application has been slowed down by a factor of 2.4 in the worst case when a large amount of file writes is in progress in the background.

This problem is projected to become worse in the future as the peripherals of mobile devices continue to adopt newer and more advanced technology. For example, the latest smartphones are equipped with Wi-Fi 802.11ac (1Gbps) and USB v2.0 (480Mbps) modules which can generate the I/O traffic of several tens of megabytes per second. It is very likely for a user to run an application while downloading some large files in the background through Wi-Fi or USB connections. In this case, the responsiveness of the foreground task will be affected significantly by massive asynchronous I/O operations.

Some degree of delay is inevitable when the foreground task accesses files under heavy asynchronous I/Os as long as they share the same storage device. Surprisingly, however, we find out that most of the delay arises from an unexpected circumstance where the file system operations are *unintentionally* blocked until one or more *asynchronous* I/O operations are finished. This phenomenon contradicts the conventional wisdom that an asynchronous I/O operation can be performed at any time as no one waits for it. Since asynchronous I/Os have lower priority than synchronous I/Os and handling of asynchronous I/Os is optimized not for latency but for throughput, the responsiveness of the foreground task can be highly affected when it has to wait for the completion of asynchronous I/Os. Our measurement with a high-end smartphone shows that a single invocation of a seemingly benign `creat()` or *buffered* `write()` system call can take more than 1 second, when its execution is blocked due to pending asynchronous I/Os (cf. Table 2). It is common that the worst case delay of these system calls increases to several seconds for low-end smartphones with worse storage performance.

To address this problem, we introduce a new type of I/O operations called *Quasi-Asynchronous I/O (QASIO)*. QASIOs are defined as the I/O operations which are issued asynchronously, but should be treated as synchronous I/Os since other tasks are blocked on them. Note that some of asynchronous I/Os are promoted to QASIOs *at run time* when a task gets blocked on them. Since the execution of the blocked task depends on QASIOs, QASIOs should be prioritized over (true) asynchronous I/Os for better responsiveness. To the best of our knowledge, this work is the first study to discuss the dependency between file system operations and asynchronous I/O operations.

We propose a novel scheme to detect QASIOs and boost them in the Linux kernel. First, we analyze three problematic scenarios where the responsiveness of applications is severely degraded due to QASIOs through extensive investigation across the entire storage I/O stack of the Linux kernel encompassing virtual file system (VFS), page cache, Ext4 file system, JBD2 journaling layer, I/O scheduler, and block layer. Then, we classify the types of direct or indirect dependencies between file system operations and QASIOs. We also present how to detect each type of dependency in the Linux kernel. Finally, we devise a mechanism to dynamically prioritize QASIOs in the CFQ I/O scheduler, a de-facto I/O scheduler in the Linux kernel.

We have implemented and evaluated the proposed scheme on one of the latest Android-based smartphones, *Samsung Galaxy S5*. Our evaluation results with microbenchmarks show that the worst case latency of `creat()`, `fsync()`, and buffered `write()` is reduced by up to 98.4%, 87.1%, and 90.2%, respectively. In real workloads, the worst case launch time of the CONTACTS application is decreased by 44.8% under heavy asynchronous I/Os.

The rest of this paper is organized as follows. Section 2 explains some background to understand how the Linux kernel handles file I/O operations. Section 3 describes three problematic scenarios and Section 4 introduces QASIOs. The design and implementation details of how to detect QASIOs and boost them in the Linux kernel are presented in Section 5. Section 6 demonstrates evaluation results and Section 7 discusses the related work. Finally, Section 8 concludes the paper.

2 Background

2.1 I/O in the Android Platform

Android is one of the most widely used mobile platforms in the world. Android provides an application framework that allows a plenty of apps to operate simultaneously. An Android app consists of different type of components such as *Activity*, *Service*, *Broadcast Receiver*, and *Content Provider* which have an individual entry point and needs to be executed as a separate task. An app provides not only user interfaces, but also various background services performed by the request of the other components of the app or even by the other apps through *Intents* [2]. Recently, Android devices begin to support multi-window mode [16] beyond simple multi-tasking. The multi-window mode allows a user to divide one display screen into two and perform different tasks on each screen. For these reasons, a large number of tasks can run simultaneously at any moment in the Android system, yielding a large amount of I/Os at the same time.

2.2 Linux Kernel I/O Path

The Linux kernel is the core of the Android platform, which is responsible for managing system resources such as CPUs, memory, and various I/O devices. In particular, the storage I/O stack is one of the most complex parts in the Linux kernel as each file system operation is processed with the help of various layers such as virtual file system (VFS), Ext4 file system, page cache, JBD2 journaling layer, I/O scheduler, and block layer. In this subsection, we briefly describe a step-by-step procedure for processing a `write()` system call as shown in Figure 1. We choose the `write()` system call because it is slightly more complicated to handle compared to other system calls as it manipulates data and metadata at the same time. We assume that the default *ordered* journaling mode is used in the Ext4 file system.

1. Invoke a `write()` system call: A task passes the file descriptor, the address of the user data buffer, and the write size to the kernel through `write()`. This information is transmitted to the VFS layer. VFS updates necessary fields (such as timestamps and file size) of the file metadata (i.e., inode) by obtaining a JBD2 journal handle. The journal handle is obtained each time the metadata is modified to record the updated metadata in the journaling area. Then, VFS copies the requested data into the corresponding page in the page cache. The calling task returns from `write()` as soon as its data is copied into the page cache.

2. Make pages of file data dirty: The pages of file data written by VFS are marked as dirty. The Linux kernel accumulates these dirty pages up to a certain threshold.

3. Flush out dirty pages: If a dirty page stays for more than the expiration time or the amount of dirty pages

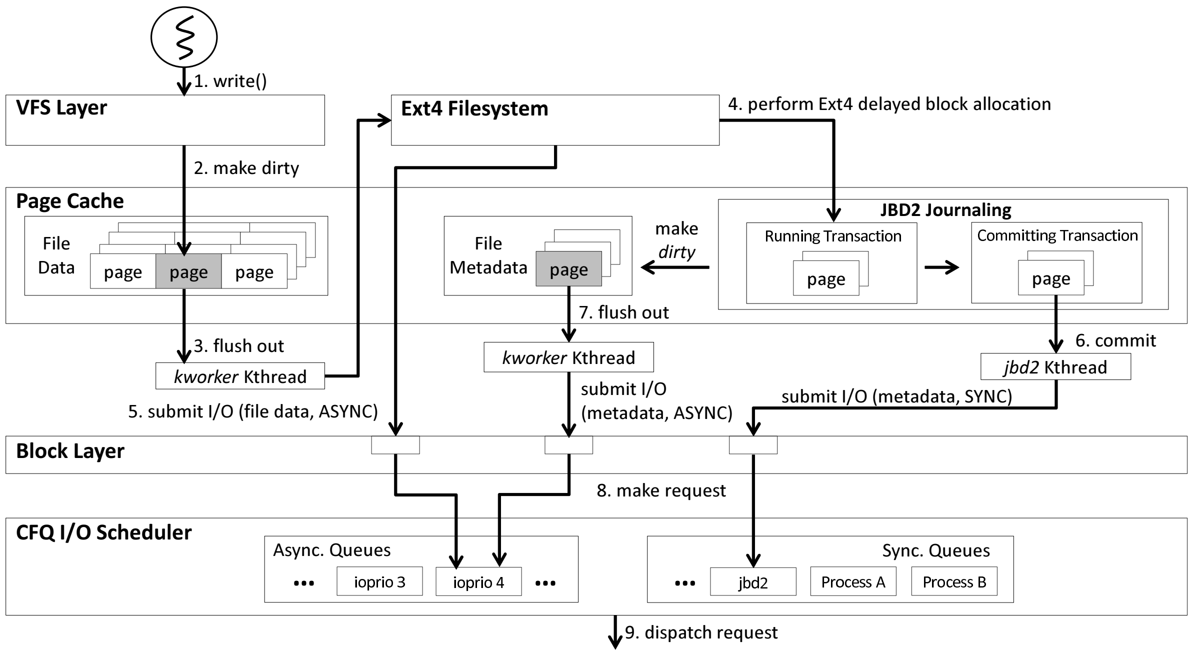

Figure 1: The processing of the `write()` system call in the Linux kernel

exceeds the background dirty ratio, they are forcibly flushed. The *kworker* kernel thread is executed periodically or synchronously in order to satisfy these requirements and asks the Ext4 file system to flush dirty pages if necessary.

4. Perform Ext4 delayed block allocation: By default, Ext4 performs block allocation when the file data is flushed from the page cache. During block allocation, Ext4 needs to modify file system metadata such as block bitmaps and group descriptors. Since these metadata should be also written into the journaling area, Ext4 obtains the journal handle of the running transaction and requests to the JBD2 module to manage them. Note that Ext4 associates a buffer head with each metadata page.

5. Submit dirty pages of file data: After block allocation, Ext4 submits dirty data pages to the block layer as *asynchronous* I/Os.

6. Commit JBD2 journaling: The running transaction is changed to the committing transaction after a predetermined time or a certain amount of buffer heads is gathered by the *jbd2* kernel thread. The metadata pages belonging to the committing transaction are submitted as *synchronous* I/Os by *jbd2*, because they must be written into the storage device rapidly for ensuring the file system integrity.

7. Flush out dirty metadata: After journal commit completes, the metadata pages included in the committing transaction are changed to the dirty state again, which enables checkpointing the metadata pages. Finally, *kworker* flushes out these metadata as *asynchronous* I/Os.

8. Make a request: Physically adjacent pages with the same I/O property among the submitted I/O requests are merged into one request through the block layer. These requests are forwarded to the I/O scheduler.

9. Dispatch a request: CFQ is the default I/O scheduler in the Linux kernel. CFQ has separate queues for synchronous I/Os and asynchronous I/Os. Synchronous I/O requests generated from a process is entered into the *synchronous* CFQ queue which is provided for each process. On the other hand, the *asynchronous* CFQ queue is shared by processes having the same I/O priority. Since most asynchronous I/Os are submitted by *kworker* (I/O priority 4) as shown in Figure 1, they are put into the same asynchronous CFQ queue. When dispatching a request, CFQ first selects a CFQ queue and then processes I/O requests in the selected queue in the order of logical sector number. Note that all the synchronous queues have higher priority than asynchronous queues in CFQ.

3 Problem and Motivation

This section presents three real-life scenarios as motivating examples which show reduced responsiveness under heavy asynchronous I/Os. The performance results of these scenarios are obtained from our test device (refer to Section 6 for details).

Scenario A: Launching the CONTACTS App:
The app start delay is a simple metric which shows the responsiveness of mobile devices. We observe that the

start time of the CONTACTS app gets slower and varies with a large standard deviation when a 4GB file is created in the background simultaneously. With 500 times of repeated tests, the worst case app start time increases by 140.0% over that of the normal case where there is no background I/O traffic.

Scenario B: Burst Mode in the CAMERA App:
The burst mode in the CAMERA app is a continuous high-speed shooting mode and is supported by many smartphones nowadays [19]. Our test device, *Samsung Galaxy S5*, shoots up to 30 shots by touching and holding on the shot icon. The next burst shot is possible shortly after the images taken by the first burst shot are processed. When the image size of each shot is large (8MB in the tested case), the intermittent delays occur between shots after the first burst shot. Finally, the burst shot performance is degraded by 19.0% than the ideal performance.

Scenario C: Installing the ANGRY BIRDS App:
The final scenario is to install the ANGRY BIRDS app downloaded from the Google Android market, when a 4GB file is written in the background. Since downloading the package file depends on the network performance, we measure the time to install the app from the package file pre-downloaded in the local storage. We observe that the average installation time increases by 35.0% when there are asynchronous I/Os in the background.

The underlying problem:
Many synchronous I/Os, such as `read()`'s or `write()`'s followed by `fsync()`, are issued during installing or launching an app. As mentioned in Section 2.2, the CFQ I/O scheduler gives higher priority to these synchronous I/Os over asynchronous I/Os. In spite of this, it is inevitable for the foreground task to experience some delay under heavy asynchronous I/Os for the following reasons. First, there can be a contention in holding a lock to modify the file system metadata. Second, it is possible that another asynchronous I/O is already in progress in the storage device when a synchronous I/O is dispatched. Third, there can be no room in memory or request queues for additional I/Os.

However, after investigating the three scenarios carefully, we find out that file system operations issued by the foreground task are significantly delayed by another reason. Although the specific condition is slightly different, the root cause of the problem is the same; the progress of a file system operation is blocked by *undispatched* asynchronous I/Os. This is an unexpected situation in the Linux kernel, resulting in an unpleasant consequence that the foreground task waits for the completion of asynchronous I/Os queued in the I/O scheduler. Since those asynchronous I/Os are not dispatched yet, the delay could be long as they may be served last by the I/O scheduler. In the next section, we show when this happens in more detail.

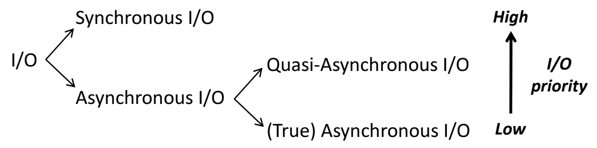

Figure 2: The classification of I/O requests

4 Quasi-Asynchronous I/O (QASIO)

This section defines a new type of I/O called quasi-asynchronous I/O (QASIO) and describes the types of dependencies on QASIOs. We also revisit the problematic scenarios shown in Section 3 to demonstrate how those scenarios are related to QASIOs.

4.1 Definition of Quasi-Asynchronous I/O

The Linux kernel traditionally categorizes I/O requests into the following two classes:

- *Synchronous I/O*: An I/O request is called synchronous when the calling task is blocked until the I/O request is completed. For this reason, the I/O scheduler such as CFQ treats synchronous I/Os in preference to asynchronous I/Os for better responsiveness. Typically, synchronous I/Os are created by `read()`, `fsync()`, and `sync()` system calls. However, `write()`'s can be made synchronous by opening a file with the `O_SYNC` flag. The *jbd2* kernel thread also generates synchronous I/Os when it commits journal data.
- *Asynchronous I/O*: Writing data to a file opened without the `O_SYNC` flag creates asynchronous I/Os. Asynchronous I/Os are flushed together by the *kworker* thread to maximize I/O throughput. They are handled in low priority in the I/O scheduler because no tasks waits for them. In this way, tasks can freely enjoy the benefits of the buffered I/O. Asynchronous I/Os are also produced when the file system metadata is written into the original location after journal commit.

In this paper, we introduce a new class of I/O called *quasi-asynchronous I/O (QASIO)* as depicted in Figure 2. A QASIO is defined as the I/O which is seemingly asynchronous but has the synchronous property because one or more tasks are waiting for its completion. This seems to be impossible in theory, but we show in the next subsection that it happens frequently in practice. Note that whether an I/O request is synchronous or asynchronous is determined when it is submitted to the block layer. In contrast, an existing asynchronous I/O is promoted to a QASIO *at run time* when a task gets blocked due to the asynchronous I/O. For better responsiveness, QASIOs should be given the higher priority than other (true) asynchronous I/Os.

4.2 Types of Dependencies on QASIO

Each task can have a *direct* or an *indirect* dependency on QASIOs. The direct dependency occurs when the execution of a task is blocked due to (quasi-) asynchronous I/Os. Figure 3 illustrates the situation where task A has a direct dependency on a QASIO. To identify when such a dependency exists, we have conducted an extensive analysis of the Linux kernel and the dynamic I/O patterns generated by file system calls. According to our analysis, we have identified the following four types of direct dependencies on QASIOs:

- *When modifying a metadata page* ($\mathbf{D}_{meta}$): This type of dependency can occur when a task invokes a file system call which modifies a metadata page (such as inodes, group descriptors, block bitmaps, inode bitmaps, and directory entries in Ext4). The target metadata page, made dirty by itself or the other tasks, may be already submitted as an asynchronous I/O by the *kworker* thread.
- *When modifying a data page* ($\mathbf{D}_{data}$): When a task appends data partially within a data page, it can be blocked since the target data page may be already flushed out asynchronously by the *kworker* thread. The task cannot proceed its execution until the data page hits the storage.
- *When guaranteeing data to be written back* ($\mathbf{D}_{sync}$): A task needs to wait for the completion of asynchronous I/Os when synchronizing or truncating the previously-issued file data in `fsync()` or `truncate()`. When performing `fsync()`, all the previous buffered writes are issued synchronously *as long as* they are still in the page cache. If calling `fsync()` is late or there are too many dirty pages in the page cache, some of them can be already flushed out as asynchronous I/Os. In this case, `fsync()` should wait until those asynchronously-issued I/Os are done.
- *When completing discard commands* ($\mathbf{D}_{discard}$): Currently, the *jbd2* kernel thread issues discard commands *asynchronously* for deallocated blocks, unlike other journal blocks which are issued synchronously. Hence, its execution is blocked on every journal commit until all the discard commands are completed. This delay in turn can affect the responsiveness of the foreground task (cf. $\mathbf{I}_{jcommit}$).

Sometimes, it is also possible that the execution of a task is being delayed due to another task that has a direct dependency on QASIOs. For example, Figure 3 shows

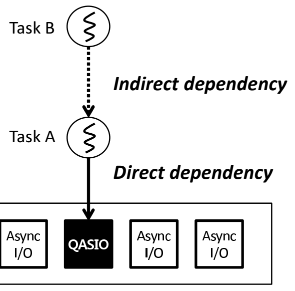

Figure 3: Direct and indirect dependency on QASIO

that task B is blocked because task A cannot make any progress due to the direct dependency on a QASIO. In this case, we call that task B has an indirect dependency on a QASIO. Typically, this situation arises when task A is blocked holding a resource that task B requires. Unlike the direct dependency, it is difficult to list all the possible types of indirect dependencies since the delay due to QASIOs can be propagated to other tasks in diverse and complicated ways. However, we found the following two types of indirect dependencies related to the JBD2 journaling which has a significant impact on the performance.

- *When unable to obtain a journal handle due to* $\mathbf{D}_{meta}$ *or* $\mathbf{D}_{data}$ ($\mathbf{I}_{jhandle}$): In Ext4, a task should obtain a journal handle to modify a metadata page or a data page. As mentioned before, the task can be blocked if the target page is already issued asynchronously, creating the $\mathbf{D}_{meta}$ or $\mathbf{D}_{data}$ type of dependency on QASIOs. Sooner or later, the transaction including the journal handle is started to be committed but the transaction is locked because the blocked task holds the journal handle. In this case, another task which attempts to perform any file operation is blocked since it fails to obtain a new journal handle.
- *When unable to complete* `fsync()` *due to* $\mathbf{D}_{discard}$ ($\mathbf{I}_{jcommit}$): This type of indirect dependency is observed only for the task that invokes `fsync()`. The `fsync()` system call needs to wait until the journal commit is completely done to ensure that the metadata of the corresponding file is written into the storage device. However, the processing time of the journal commit can be significantly prolonged since the *jbd2* kernel thread usually has a direct dependency of $\mathbf{D}_{discard}$ due to asynchronously-issued discard commands.

Whenever a foreground task interacting with a user has a direct or an indirect dependency on QASIOs, its execution has nondeterministic *hiccups* and the user can encounter sluggish responsiveness. Despite that there is room for additional I/Os in memory and request queues, the processing of system calls is blocked by the stacked

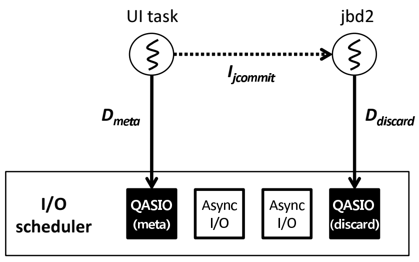

Figure 4: Dependencies on QASIOs in Scenario A

asynchronous I/Os in the request queue. More serious problem is that the lag due to QASIOs occurs not only in `fsync()`, but also such system calls as `creat()`, `chown()`, `unlink()`, and even *buffered* `write()`, where a user does not expect any delay. Depending on the storage performance, we observe that a single invocation of the `creat()` system call takes up to several seconds due to its dependency on QASIOs.

4.3 Revisiting Problematic Scenarios

We now take a closer look at the problematic scenarios in Section 3 and its relationship with QASIOs. Table 1 summarizes the major dependencies on QASIOs observed in each scenario.

4.3.1 Scenario A: Launching the CONTACTS App

When an app's UI shows up in the Android platform, several file system calls are made to update its states into databases (e.g., SQLite) or files (e.g., xml files) persistently. In our test device, launching the CONTACTS app is accompanied by a series of system calls such as `rename()`, `write()`, `fsync()`, and `unlink()`. These system calls all need to update the metadata. Therefore, they can have the $\mathbf{D}_{meta}$ type of dependency on QASIOs under bulky asynchronous I/Os, when they try to modify the metadata page which is being written back.

The UI task of the CONTACTS app has another indirect dependency of $\mathbf{I}_{jcommit}$ to QASIOs. At the end of each journal commit, the *jbd2* kernel thread has a direct dependency ($\mathbf{D}_{discard}$) due to the asynchronously-issued discard commands. If the journal commit is delayed due to $\mathbf{D}_{discard}$, the `fsync()` system call performed by the UI task is delayed as well since it cannot return until the metadata modification of the synchronized file is completely committed. Therefore, the UI task has two kinds of dependencies on QASIOs as depicted in Figure 4.

4.3.2 Scenario B: Burst Mode in the CAMERA App

After the first burst shot completes, several services are executed through the *Intents* transferred from the Android platform. One of them is the thumbnail maker task which generates thumbnails of the taken images. Since the thumbnail size is very small, the thumbnail maker task writes data in a small size (e.g., 1KB) repeatedly. This results in partial writes to the same data page, which can be blocked if the target data page is already being written back. Hence, the thumbnail maker task can have the $\mathbf{D}_{data}$ type of dependency on QASIOs.

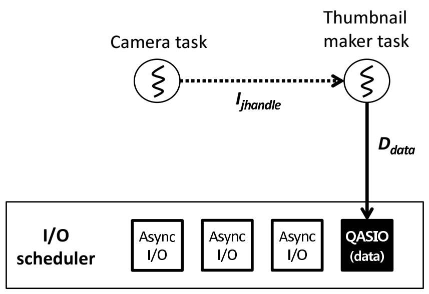

Figure 5: Dependencies on QASIOs in Scenario B

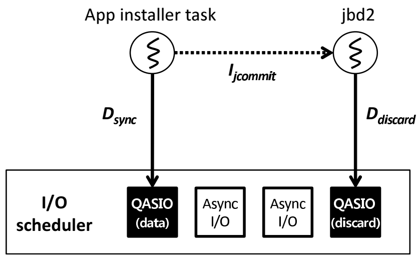

Figure 6: Dependencies on QASIOs in Scenario C

Since the thumbnail maker is running in the background, it should not affect the responsiveness of the foreground CAMERA task. However, the problem is that the CAMERA task has an indirect dependency of $\mathbf{I}_{jhandle}$ to QASIOs via the thumbnail maker as shown in Figure 5. The thumbnail maker obtains a journal handle of the running transaction before copying the thumbnail image data to the data page, and falls into a sleep by the $\mathbf{D}_{data}$ dependency. After a certain period of time, the journal commit is started but the *jbd2* thread falls into the *locked* state since the thumbnail maker went asleep due to $\mathbf{D}_{data}$ with holding the journal handle of the transaction to be committed. When the CAMERA wants to acquire a journal handle to write additional images for the subsequent burst shots, it is eventually blocked. This is because the JBD2 journaling module does not give out any journal handle under the condition that the committing transaction is locked.

4.3.3 Scenario C: Installing the ANGRY BIRDS App

The dependencies on QASIOs arisen in this scenario are illustrated in Figure 6. The App installer task issues a number of *buffered* `write()` system calls in order to save the extracted data of the downloaded app to the app repository. To prevent data loss on sudden power failures, the App installer task invokes `fsync()` so that all the written data are flushed into the storage device. The `fsync()` system call is meant to write the data pages belonging to the corresponding file *synchronously*. However, when a large file is written in the background, the page cache is filled with dirty pages. In this case, the data pages to be `fsync()`'ed can be flushed out *asynchronously* by the *kworker* thread before the App installer invokes the `fsync()` system call. This leads to the $\mathbf{D}_{sync}$ type of dependency on QASIOs.

For the same reason as in scenario A, this scenario also has an indirect dependency of $\mathbf{I}_{jcommit}$ during `fsync()` due to asynchronously-issued discard commands.

5 Boosting QASIOs

In this section, we explain how to detect QASIOs efficiently at run time and boost them for better responsiveness during file system operations.

5.1 Design

When a task has a direct or an indirect dependency on a QASIO, its execution is blocked until the corresponding QASIO completes. This situation is somewhat similar to the priority inversion problem in scheduling where a high priority task cannot make any progress as a low priority task has a resource it requires. In this case, as the priority inheritance protocol does, the best way we can do to minimize the waiting time of the task is to give a higher priority to the QASIO and complete it quickly.

However, it is not easy in the current design of I/O schedulers since they do not know the presence of QASIOs and thus they have no idea of which one to boost. The various dependencies between tasks and QASIOs are formed *dynamically* at run time across various upper layers such as VFS, page cache, and Ext4 file system. This suggests that we have to have a run time mechanism which can detect QASIOs in the upper layers and notify the I/O scheduler to prioritize them.

The requirements for boosting QASIOs can be summarized as follows:

- *Req.(1): When a task is blocked waiting for the completion of an asynchronous I/O, the kernel should be able to give a hint about the existence of QASIO to the I/O scheduler.*
- *Req.(2): Upon the receipt of the hint from the kernel, the I/O scheduler should prioritize them among asynchronous I/Os.*

The *Req. (1)* is independent of the I/O scheduler used in the kernel, but *Req. (2)* needs to be re-implemented for each I/O scheduler. In this paper, we only show the implementation based on the CFQ I/O scheduler. However, the design can be easily applied to the other I/O

Table 1: The major dependencies on QASIOs in each scenario discussed in Section 4.3. (This table shows the major dependencies only. Sometimes other dependencies can occur as well. (B) represents that the dependency is associated with the background task, but the foreground task also has an indirect dependency on QASIOs.)

	Direct				Indirect	
Scenario	$\mathbf{D}_{meta}$	$\mathbf{D}_{data}$	$\mathbf{D}_{sync}$	$\mathbf{D}_{discard}$	$\mathbf{I}_{jhandle}$	$\mathbf{I}_{jcommit}$
A	✓			✓(B)		✓
B		✓(B)			✓	
C			✓	✓(B)		✓

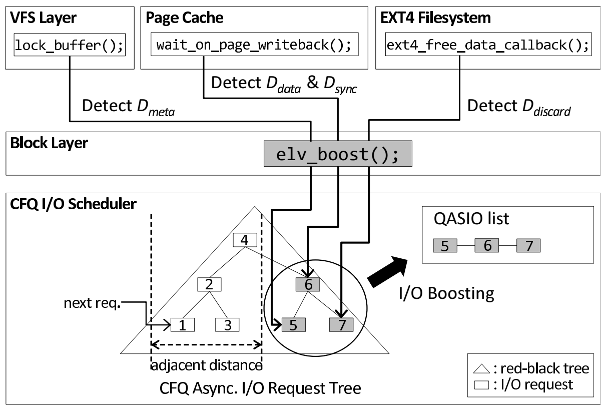

Figure 7: Implementation overview for detecting and boosting QASIOs in the Linux kernel

schedulers. Figure 7 overviews our implementation for detecting and boosting QASIOs in the Linux kernel.

5.2 Detecting QASIOs

In this subsection, we present how QASIOs can be detected at run time in the Linux kernel. Since an indirect dependency on a QASIO occurs only when there is another direct dependency on the same QASIO, we only focus on detecting direct dependencies on QASIOs. If the direct dependency is resolved, the associated indirect dependency is terminated as well.

As we have seen in Section 4.2, there are four types of direct dependencies on QASIOs. Each can be detected as follows:

- *Detecting* $\mathbf{D}_{meta}$: In Ext4, a task accesses a metadata page through the associated buffer head structure. Before a task modifies a metadata page, it obtains an exclusive lock to the metadata page using the `lock_buffer()` kernel function. However, if the buffer head is already submitted to the block layer, the state of the buffer head is changed into the *locked* state and the execution of the task that attempts to acquire the same lock is suspended. Note that failing to obtain the lock does not necessarily mean that the target metadata page is submitted as an asynchronous I/O. Therefore, we have to double check whether the locked buffer head is being processed as an asynchronous I/O before making a decision.

- *Detecting* $\mathbf{D}_{data}$ *and* $\mathbf{D}_{sync}$: The dependency types of $\mathbf{D}_{data}$ and $\mathbf{D}_{sync}$ can be detected in the same location in case of Ext4. When a task wants to guarantee some data to be written back for synchronizing or truncating a file or to perform a partial write to a data page, it checks whether the previously-issued I/O has reached the storage device using the kernel function `wait_on_page_writeback()`. If necessary, the task waits in this function until the previous I/O is finished. Similar to the $\mathbf{D}_{meta}$ case, we should check whether the previous I/O is submitted as an asynchronous I/O.
- *Detecting* $\mathbf{D}_{discard}$: The kernel function `ext4_free_data_callback()` is registered to the JBD2 journaling module as a callback function. This function is called whenever the block deallocation is required at the end of journal commit. Unlike the above two cases, the *jbd2* kernel thread submits asynchronous discard operations directly in the callback function and then falls asleep waiting for the completion of those I/Os. Since it is obvious that *jbd2* generates QASIOs, no additional check is necessary.

To detect QASIOs in `lock_buffer()` and `wait_on_page_writeback()` functions, we should be able to quickly confirm whether the buffer head or the page, now being accessed, is issued as an asynchronous I/O. Since the Linux kernel has no way to represent this information, we added a special buffer head flag and modified the `submit_bh()` and `ext4_bio_write_page()` functions so that they set the flag when submitting asynchronous I/O requests. `submit_bh()` and `ext4_bio_write_page()` are used to submit a buffer head and a data page, respectively, to the underlying block layer. This special flag is unset after the I/O completes.

Algorithm 1 outlines how the actual detection of QASIOs is implemented in the function `wait_on_page_writeback()`. In the original implementation, a task simply waits in `wait_on_page_writeback()` until the writeback of the target page completes. Instead, we check whether the target page is submitted as an asynchronous I/O (lines 1–2) and if it is the case, we send the sector number of the detected QASIO to the I/O scheduler (lines 3–4). Note that even if the I/O scheduler is notified of the presence of a QASIO, it may fail to find it in the request queue (line 5) for the following two reasons.

Algorithm 1 A modified algorithm for `wait_on_page_writeback()` for detecting D_{data} and D_{sync}

```
 1: while the target page is already submitted do
 2:     if the page is issued as async. I/O then
 3:         extract the start sector number from the page
 4:         send the start sector number to I/O scheduler
 5:         if I/O scheduler fails to find the I/O then
 6:             set a timer of several clock ticks
 7:         end if
 8:     end if
 9:     wait until the page I/O completes or the timer expires
10: end while
```

First, the I/O request can be already dispatched to the storage device by the low-level device driver. In this case, we have no choice other than wait for the I/O completion. The second case is that the I/O request is staying temporarily in the *plug list* of the task, not in the request queue of the I/O scheduler. The plug list keeps I/O requests generated by a task for a short period time on the stack space of the task in order to increase the possibility of creating a larger request and to decrease a lock contention in the I/O scheduler [3]. Since the plug list is a private area that cannot be searched, we keep checking periodically until all the I/O requests in the plug list are flushed to the request queue. QASIOs can be detected similarly in `lock_buffer()` and `ext4_free_data_callback()` functions.

5.3 Prioritizing QASIO

Since QASIOs should be processed urgently, we give a higher priority to QASIOs than all the (true) asynchronous I/Os, but not more than any other synchronous I/Os. This is because we do not want that the boosting of QASIOs interferes with the responsiveness of synchronous I/Os. The actual implementation of handling QASIOs proceeds as follows.

Once a QASIO is detected, the information on the QASIO (i.e., start sector number) should be delivered to the I/O scheduler. For this purpose, we have added a new interface called `elv_boost()` in the elevator layer of the Linux kernel (cf. Figure 7). `elv_boost()` is an abstract interface which invokes a pre-registered function specific to each I/O scheduler, `cfq_boost_req()` in our case with the CFQ I/O scheduler.

For each asynchronous queue, we maintain a separate list of I/O requests called *QASIO list* as shown in Figure 7. In `cfq_boost_req()`, we traverse red-black trees of all the asynchronous queues and look for the I/O request which contains the received sector number of the QASIO. If it is found, an entry for the QASIO is inserted into the corresponding QASIO list.

In the original CFQ scheduler, whenever an asynchronous CFQ queue is selected for dispatching a request, CFQ finds the nearest request, specified by the `next_req` pointer, from the last processed request in the red-black tree and then sequentially dispatches I/O requests from that position. In our implementation, however, CFQ checks the QASIO list first and then dispatches QASIO requests if any. Moreover, if the QASIO list is empty, the current asynchronous queue yields the chance of I/O dispatching to another asynchronous queue which has a non-empty QASIO list. In this way, QASIOs are dispatched before any other asynchronous I/O requests.

6 Evaluation

This section presents the evaluation results with five microbenchmarks, three real-life scenarios, and two I/O benchmarks for Android.

6.1 Methodology

Our evaluation has been conducted on one of the latest smartphones, *Samsung Galaxy S5*, equipped with Exynos 5422 (including quad Cortex-A15 and quad Cortex-A7 ARM CPUs and Mali-T628 MP6 GPU), 2GB DRAM, and 16GB eMMC flash storage. It runs the Android platform version 4.4.2 (KitKat), with the Linux kernel 3.10.9. In Android, the dirty page expiration time is set to 2 seconds and the background dirty ratio to 5% by default.

In order to investigate the impact of QASIOs on the latency of various file system operations, we have used the following in-house microbenchmarks:

- M1: M1 iterates the creation of a 4KB file 500 times. In each iteration, M1 opens the same file with `creat()`, and then writes 4KB of data to the file using the buffered `write()`. Finally, it performs `fsync()` and closes the file with `close()`. Note that this microbenchmark mimics the storage I/O patterns of a database system such as SQLite [12].
- M2: M2 is the same as M1 except that the file size is increased to 1MB and the number of iterations is set to 200. The file data (1MB in size) is written using a single `write()` system call.
- M3: M3 creates a new file with `creat()` and repeats a 1KB-sized `write()` until the file size reaches 300MB.
- M4: In each iteration, M4 truncates the 2MB file created in the previous iteration to zero length using `truncate()`, recreates the same sized file using `write()`, and then closes the file with `close()`. This is repeated 500 times. The file for the first iteration is created manually before the execution.
- M5: M5 creates a single 4KB file by performing `creat()`, `write()`, `fsync()`, and `close()`, while another task truncates an existing 8GB file and writes 8GB of data again to the file.

We run all the microbenchmarks except M5 while a 8GB file is written in the background in order to generate asynchronous I/O operations. In addition to these microbenchmarks, the proposed scheme is evaluated with real-life scenarios discussed in Section 3 and two representative I/O benchmarks in Android, Antutu and RL-Bench.

6.2 Microbenchmarks

Figure 8 compares the total elapsed time of each microbenchmark according to the type of dependency boosted. The results are normalized to NONE in which no special handling is performed for QASIOs. Each $\mathbf{D}_{data}$+$\mathbf{D}_{sync}$, $\mathbf{D}_{meta}$, and $\mathbf{D}_{discard}$ represents the case where only the specified type of dependency is detected and boosted. Note that $\mathbf{D}_{data}$ and $\mathbf{D}_{sync}$ types cannot be boosted separately, as they are detected in the same location (cf. Section 5.2). Finally, ALL means that all kinds of optimizations are applied for QASIOs. Table 2 presents the latency of key file system operations before and after applying the optimizations for QASIOs. Overall, we can see that boosting QASIOs improves the total elapsed time by up to 83.1%. The proposed scheme also reduces the worst case latency of each file system operation by up to 98.4%. The detailed analysis on the result of each microbenchmark is as follows.

In M1, when all the dependency types are boosted, the total elapsed time is reduced by 83.1% as the average latency of `creat()` and `fsync()` is improved by 99.1% and 63.7%, respectively. In each iteration of M1, `creat()` modifies metadata pages and also incurs discard operations as it creates the file with the same name. Hence, `creat()` and `fsync()` will have the $\mathbf{D}_{meta}$ and $\mathbf{I}_{jcommit}$ dependency, respectively. This is why the most of reduction in the total elapsed time comes when $\mathbf{D}_{meta}$ and $\mathbf{D}_{discard}$ types are boosted.

The M2's results show the similar trend as M1 except that boosting $\mathbf{D}_{data}$+$\mathbf{D}_{sync}$ is as effective as $\mathbf{D}_{meta}$ or $\mathbf{D}_{discard}$. As the file size becomes larger, it is likely that *kworker* flushes out some of data pages asynchronously before `fsync()` is called. Consequently, unlike in M1, `fsync()` will have the $\mathbf{D}_{sync}$ dependency in M2.

In M3, the most dominant operation affecting the overall performance is the *buffered* `write()`. From Table 2, we can observe that the latency of `write()` is reduced by 47.4% on average and by 90.2% in the worst case. `write()` suffers from the $\mathbf{D}_{data}$ dependency since 1KB of data is written into a data page partially. Therefore, the most of performance improvement is achieved by boosting the $\mathbf{D}_{data}$ type of dependency as Figure 8 illustrates.

In M4, the key file system operation affected by QASIOs is `truncate()`. If the file data is already issued

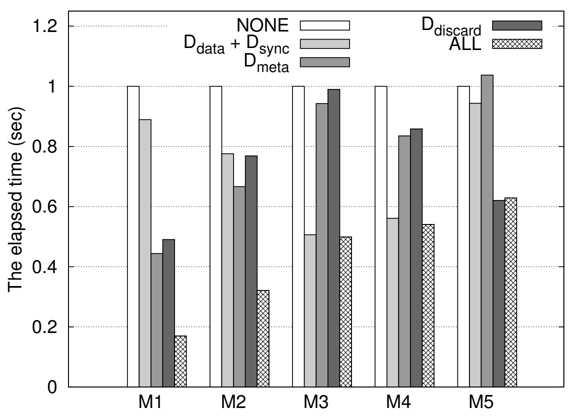

Figure 8: The normalized total elapsed time of each microbenchmark according to the dependency type boosted

asynchronously, `truncate()` should wait for the completion of those asynchronous I/Os. Hence, boosting the $\mathbf{D}_{sync}$ type of dependency is most helpful for M4.

In case of M5, *jbd2* has the $\mathbf{D}_{discard}$ type of dependency on asynchronously-issued discard commands as another task truncates a large file. In this case, calling `fsync()` to synchronize just 4KB of data takes 13.27 seconds on average due to the $\mathbf{I}_{jcommit}$ dependency. However, when we boost the $\mathbf{D}_{discard}$ type, the latency is decreased to 6.85 seconds.

Since QASIOs are prioritized over other asynchronous I/Os in the I/O scheduler, boosting QASIOs can have a negative impact on the throughput of asynchronous I/Os. To investigate this effect, we have measured the throughput of creating a 8GB file in the background while performing the M1 microbenchmark. According to our measurement results, the throughput is decreased by 15.4%, from 46.2MB/s to 39.1MB/s. We believe this is acceptable considering that the total elapsed time of the foreground task in M1 is improved by 83.1%.

6.3 Real-life Scenarios

Figure 9 depicts the impact of boosting QASIOs in three real-life scenarios described in Section 3. On the right side of Figure 9, we also show the total time spent on waiting for the completion of QASIOs. These times are measured in the kernel functions described in Section 5.2 where each type of dependency is detected. In Figure 9(a) and (c), the phrase "with bg. I/O" indicates the case where a 4GB file is created in the background simultaneously in order to generate asynchronous I/Os.

In Scenario A, we have measured the time to launch the CONTACTS app. Under heavy asynchronous I/Os, the launch time is increased by 29.4% on average. In the worst case, the app start is slowed down by a factor of 2.4. However, boosting QASIOs effectively reduces the worst case launch time by 44.8%. Similar to the M1

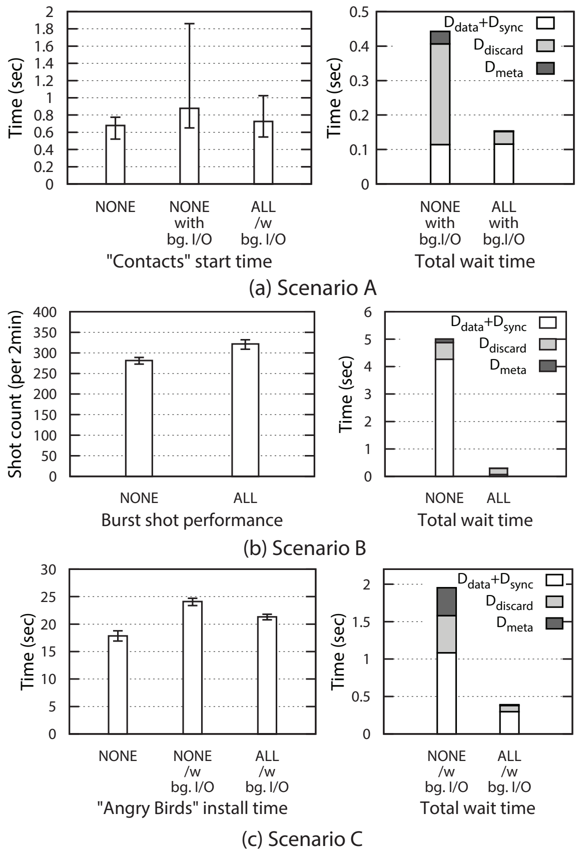

Figure 9: Results of three real-life scenarios

microbenchmark, the most of improvement comes from boosting $\mathbf{D}_{meta}$ and $\mathbf{D}_{discard}$, as the CONTACTS task has the $\mathbf{D}_{meta}$ and $\mathbf{I}_{jcommit}$ dependency on QASIOs. The total wait time on the corresponding kernel function of $\mathbf{D}_{meta}$ and $\mathbf{D}_{discard}$ has been reduced by 96.1% and 87.4%, respectively.

In Scenario B, we can see that the shot count in the burst mode is improved by 14.4% on average when we boost QASIOs. The thumbnail maker has the $\mathbf{D}_{data}$ dependency on QASIOs, however the total wait time due to $\mathbf{D}_{data}$ is reduced by 98.4%. As the direct dependency between the thumbnail maker and QASIOs is resolved quickly, the burst mode performance of the CAMERA app has been improved as well.

Finally, the average installation time of the ANGRY BIRDS app is slowed down by 35.0% under the heavy asynchronous I/Os in the background. However, we observe that the average installation time is improved by 11.5% through the boosting of QASIOs. As mentioned in Section 4.3.3, the ANGRY BIRDS app has the $\mathbf{D}_{sync}$ and $\mathbf{I}_{jcommit}$ dependency on QASIOs. Accordingly, the most of reduction in the installation time comes from boosting the $\mathbf{D}_{sync}$ type. Boosting $\mathbf{D}_{discard}$ and $\mathbf{D}_{meta}$ also contributes to reducing the installation time since some

Table 2: The latency of key file system operations in each microbenchmark. (The time unit is millisecond for M1 – M4, while it is second for M5)

Opt	M1				M2				M3		M4		M5	
	`creat()`		`fsync()`		`creat()`		`fsync()`		`write()`		`truncate()`		`fsync()`	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
NONE	119.57	1435.54	98.24	1119.62	109.01	1397.64	172.05	1417.82	0.19	1813.54	62.60	1632.15	13.27	13.87
ALL	**1.02**	**39.15**	**35.64**	**144.69**	**3.90**	**22.52**	**69.24**	**298.83**	**0.10**	**177.40**	**12.85**	**334.57**	**6.85**	**7.11**

metadata modifications and discard operations are performed during the app install procedure.

6.4 I/O Benchmarks for Android

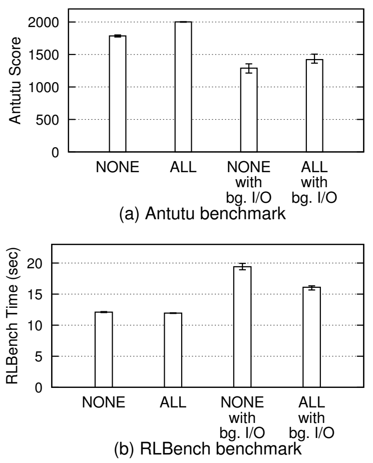

Figure 10: Results of Antutu and RLBench

Finally, we have evaluated the proposed scheme with two famous I/O benchmarks for Android, Antutu [6] and RLBench [15]. Antutu is a comprehensive Android benchmark which provides several modes for measuring the performance of CPU, memory, storage system, etc. In this evaluation, we use the *Database IO* mode that estimates the storage I/O performance with database workloads. The Antutu benchmark reports the final score as a result of the performance measurement, where the higher score means the better performance.

RLBench is a benchmark for measuring the performance of Android devices under SQLite workloads. Since it makes SQLite generate storage I/O intensive workloads, the RLBench's result is closely related to the storage I/O performance in general. Unlike Antutu, RLBench shows the elapsed time to process the predefined set of SQL queries, hence the lower time means the better performance.

The results of Antutu and RLBench are displayed in Figure 10. We run each benchmark with (labeled as "with bg. I/O") and without asynchronous I/O operations. The asynchronous I/Os are generated by writing a 4GB file in the background while running the benchmarks.

In Antutu, the base score is measured at 1,785 when there is no asynchronous I/O operations in the background. Due to asynchronous I/Os, the score is decreased to 1,289. However, the proposed scheme improves the score to 1,421 which is smaller than the base score by 20.4%. An interesting observation is that boosting QASIOs improves the performance of Antutu by about 12.0% even when there is no asynchronous I/Os in the background. This means Antutu itself issues asynchronous I/Os and its performance is also affected by QASIOs.

The result of RLBench also shows that the proposed scheme successfully improves the storage I/O performance. When there are asynchronous I/O operations in the background, the elapsed time is reduced by 17.1% by boosting QASIOs.

7 Related Work

Mobile devices usually employ NAND flash memory as the media of the main storage system. Kim et al. show that the storage performance is a limiting factor for the performance of several common applications for mobile devices through extensive experiments with various flash storage systems [9]. Since NAND flash memory shows very different characteristics compared to hard disk drives, prior work attempts to revisit various operating system mechanisms and policies which have been optimized for rotating media. As a result of these efforts, several file systems [4, 13] and I/O schedulers [14, 17] have been proposed which are optimized for the characteristics of flash storage.

Recently, many researches have focused on optimizing the I/O stack of the Linux kernel in accordance with the I/O characteristics of SQLite, a lightweight transactional database engine provided by the Android platform. Since most applications heavily utilize SQLite to keep their application-specific data persistently, the overall performance of mobile applications is known to highly depend on the SQLite's performance. In particular, Lee et al. have observed that running SQLite on top of the Ext4 file system produces very inefficient I/O patterns to the storage system [12]. Based on the observation, Jeong et al. propose the elimination of unnecessary metadata journaling, external journaling, and a polling-based I/O mechanism to improve the journaling efficiency [8]. Similarly, Shen et al. propose the enhanced journaling mechanism for the Ext4 file system in the SQLite environment to solve the so-called *journaling of journal* problem [18].

In this paper, we focus on the fact that the responsiveness of file system operations is severely degraded when the system has lots of storage I/O operations asynchronously issued. This is an inherent problem in handling I/O requests, being independent of file systems and I/O schedulers. Therefore, our approach is largely orthogonal to previous researches. Note that the proposed scheme reduces the latency of `fsync()` successfully, which is known to be performed frequently by SQLite. Therefore, boosting QASIOs will be also helpful in improving the performance of SQLite as exemplified in the result of RLBench.

The recent Linux kernel allows to update data pages while they are under writeback by disabling the *stable pages* feature, which successfully eliminates any $\mathbf{D}_{data}$ dependency [5]. However, this can be unacceptable in future mobile devices since the hardware-supported encryption during I/O is seriously being considered. Also, the $\mathbf{D}_{discard}$ dependency can be removed if a userspace program issues a *fstrim* command in a batch manner at convenient times when the device is idle, as introduced in the recent Android platform [1]. However, this may not be a complete solution since the I/O performance of the underlying flash storage can be suddenly degraded if discard commands are not issued at a proper time.

8 Conclusions

This paper introduces a new type of I/O called Quasi-Asynchronous I/O (QASIO). The QASIO is the I/O operation which is seemingly asynchronous but has the synchronous property since one or more tasks are blocked until the I/O operation is completed. As the system handles asynchronous I/Os in the perspective of maximizing throughput not latency, the responsiveness of the blocked tasks is significantly degraded. In particular, in mobile devices where most applications interact with users all the time, the quality of user experiences suffers from QASIOs.

In order to address this problem, we propose a novel scheme to detect QASIOs and boost them in the Linux kernel. We have implemented and evaluated the proposed scheme on the latest Android-based smartphone, *Samsung Galaxy S5*. By performing various microbenchmarks, real-life scenarios, and Android I/O benchmarks, we confirm that boosting QASIOs is effective in improving the responsiveness of file system operations. We plan to analyze the effect of boosting QASIOs on more diverse platforms including servers and low-end smartphones.

9 Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Youjip Won, for their valuable comments. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MSIP) (No. 2013R1A2A1A01016441).

References

[1] Android 4.3 update brings trim to all nexus devices. `http://www.anandtech.com/show/7185/android-43-update-brings-trim-to-all-nexus-devices`.

[2] Android developers. `http://developer.android.com`.

[3] Explicit block device plugging. `http://lwn.net/Articles/438256`.

[4] F2fs: Introduce flash-friendly file system. `https://lwn.net/Articles/518718/`.

[5] Optimizing stable pages. `http://lwn.net/Articles/528031`.

[6] ANTUTU LABS. AnTuTu Benchmark. `https://play.google.com/store/apps/details?id=com.antutu.ABenchMark5`.

[7] GARTNER, INC. Gartner says worldwide traditional PC, tablet, ultramobile and mobile phone shipments are on pace to grow 6.9 percent in 2014. `http://www.gartner.com/newsroom/id/2692318`.

[8] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O stack optimization for smartphones. In *USENIX ATC* (2013), pp. 309–320.

[9] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *FAST* (2012), pp. 1–14.

[10] KIM, H., AND SHIN, D. Optimizing storage performance of android smartphone. In *ICUIMC* (2013).

[11] KIM, W.-H., NAM, B., PARK, D., AND WON, Y. Resolving journaling of journal anomaly in android i/o: multi-version b-tree with lazy split. In *FAST*, pp. 273–285.

[12] LEE, K., AND WON, Y. Smart layers and dumb result: IO characterization of an android-based smartphone. In *EMSOFT* (2012), pp. 23–32.

[13] LU, Y., SHU, J., AND WANG, W. ReconFS: a reconstructable file system on flash storage. In *FAST* (2014), pp. 75–88.

[14] PARK, S., AND SHEN, K. FIOS: a fair, efficient flash I/O scheduler. In *FAST* (2012), pp. 1–15.

[15] REDLICENSE LABS. RL Benchmark: SQLite. `https://market.android.com/details?id=com.redlicense.benchmark.sqlite`.

[16] SAMSUNG. How do i use multi window mode (multitasking) on my Samsung Galaxy Note II? `http://www.samsung.com/us/support/howtoguide/N0000004/8829/62396`.

[17] SHEN, K., AND PARK, S. FlashFQ: A fair queueing I/O scheduler for flash-based SSDs. In *USENIX ATC* (2013), pp. 67–78.

[18] SHEN, K., PARK, S., AND ZHU, M. Journaling of journal is (almost) free. In *FAST* (2014), pp. 287–293.

[19] WIKIPEDIA. Burst mode (photography). `http://en.wikipedia.org/wiki/Burst_mode_(photography)`.