# Buffered FUSE: optimising the Android IO stack for user-level filesystem

Sooman Jeong and Youjip Won*

Hanyang University,
#507-2, Annex of Engineering Center,
17 Haengdang-dong, Sungdong-gu, Seoul, 133-791, South Korea
E-mail: 77smart@hanyang.ac.kr
E-mail: yjwon@hanyang.ac.kr
*Corresponding author

**Abstract:** In this work, we optimise the Android IO stack for user-level filesystem. Android imposes user-level filesystem over native filesystem partition to provide flexibility in managing the internal storage space and to maintain host compatibility. The overhead of user-level filesystem is prohibitively large and the native storage bandwidth is significantly under-utilised. We overhauled the FUSE layer in the Android platform and propose buffered FUSE (bFUSE) to address the overhead of user-level filesystem. The key technical ingredients of buffered FUSE are: 1) extended FUSE IO size; 2) internal user-level write buffer; 3) independent management thread which performs time-driven FUSE buffer synchronisation. With buffered FUSE, we examined the performances of five different filesystems and three disk scheduling algorithms in a combinatorial manner. With bFUSE on XFS filesystem using the deadline scheduling, we achieved the IO performance improvements of 470% and 419% in Android ICS and JB, respectively, over the existing smartphone device.

**Keywords:** Android; storage; user-level filesystem; FUSE; write buffer; embedded systems.

**Reference** to this paper should be made as follows: Jeong, S. and Won, Y. (2014) 'Buffered FUSE: optimising the Android IO stack for user-level filesystem', *Int. J. Embedded Systems*, Vol. 6, Nos. 2/3, pp.95–107.

**Biographical notes:** Sooman Jeong received his BS degrees in Computer Science from Hanyang University, Seoul, Korea, in 2003. He worked for Samsung Electronics as a Software Engineer. He is currently working toward his MS degree in the Department of Computer Science, Hanyang University, Seoul, Korea. His research interests include optimising I/O stack of smartphones.

Youjip Won received his BS and MS degrees in Computer Science from the Seoul National University, Korea, in 1990 and 1992, respectively. He received his PhD in Computer Science from the University of Minnesota, Minneapolis, in 1997. After receiving his PhD degree, he joined Intel as a Server Performance Analyst. Since 1999, he has been with the Department of Computer Science, Hanyang University, Seoul, Korea, as a Professor. His research interests include operating systems, file and storage subsystems, multimedia networking, and network traffic modelling and analysis.

## 1 Introduction

Smart devices have rapidly spread to the public and expanded their territory into all home electronics including phones, TVs, cameras, and game consoles. As they have become so closely related to our everyday lives, improving the performance of these smart devices will have a significant impact on society. A recent study on Android smartphones suggests that storage is the biggest performance bottleneck (Kim et al., 2012a).

An Android-based device is no longer a simple media player but has become a professional content creator. One piece of clear evidence for this trend is a recent introduction of an Android-based digital camera with a high resolution sensor (up to 17 mega-pixels) (Galaxy Camera, http://www.samsung.com/us/photography/galaxy-camera). In the future, smart devices including phones, TVs, and cameras are projected to acquire and play movies at ultra-HD (3,840 × 2,160 @ 60 frames/sec) resolution.

Android apps will require more IO bandwidth to handle such high definition multimedia contents.

The Android platform partitions its storage into multiple logical partitions. There are two main storage partitions in the Android: data and sdcard. Data partition (`/data`) is used to harbour text files, e.g., SQLite database tables, and sdcard partition (`/sdcard`) harbours multimedia files, e.g., pictures, mp3 files and video clips. In recent smartphones, both of these partitions share the same storage device. Android storage has gone through several generations of evolution. In the early models, as the partition names suggest, data partition was at internal NAND storage device (eMMC) and sdcard partition was at external sdcard (HTC Dream, http://en.wikipedia.org/wiki/HTC_Dream). In the later generation of Android-based smartphones (Samsung Galaxy S2, http://www.samsung.com/global/microsite/galaxys2/html/), data partition and sdcard partition resided at the same physical device, internal NAND-based storage and each was allocated a 'fixed' size logical partition. Until this generation, the Android IO stack imposed EXT4 (Mathur et al., 2007) over data partition and VFAT (Microsoft Corporation, 2000) filesystem over sdcard partition. VFAT filesystem is de facto standard in embedded multimedia devices, e.g., cameras, camcorders, TVs, etc. Therefore, for PC compatibility, it is mandatory that sdcard partition adopts VFAT filesystem despite its technical deficiencies (Lim et al., 2013). However, this fixed partition-based storage configuration causes a significant underutilisation of the storage space.

Recent Android platform imposes user-level filesystem (FUSE) for sdcard partition instead of VFAT. This allows sdcard to freely manoeuvre between the two respective partitions. The FUSE filesystem exists as a virtual filesystem layer on top of EXT4, the native filesystem. While this configuration provides flexibility in managing the internal storage space, it causes significant overhead on the FUSE, decreasing the performance of IO accesses to user partition. For example, in Galaxy S3, sequential write performance in FUSE imposed filesystem is one fourths of that in native EXT4 filesystem partition.

The IO access to the Android storage can be categorised into two types: data partition accesses and sdcard partition accesses. While the IO accesses to both partitions suffer from excessive overhead, the source of inefficiency differs vastly. For data partition accesses, majority of IOs are created by SQLite operation and the inefficiency lies in the excessive filesystem journaling (Lee and Won, 2012; Jeong et al., 2013b). In this work, we focus on optimising the Android IO stack for handling multimedia files. Sdcard partition is the default storage partition for mp3, video files, photographs, and user-downloaded documents, e.g., pdf files. We found that the IO accesses to sdcard partition suffer from significant overhead and the apps utilise only 25% of the raw NAND storage bandwidth.

We examined the behaviour of FUSE, native filesystem, buffer cache, and IO scheduler for user partition accesses and found that FUSE layer is the dominant source of overhead. FUSE entails significant overhead mostly due to the following three reasons:

1 command fragmentation

2 excessive context switch

3 loss of access correlation.

We propose buffered FUSE (bFUSE) framework, which effectively addresses the above mentioned three technical issues. bFUSE framework consists of:

1 extended FUSE IO unit

2 FUSE buffer managed by independent thread

3 time driven FUSE buffer synchronisation.

The Android IO stack consists of FUSE, native filesystem (EXT4), IO scheduler for block device (CFQ), and underlying eMMC storage. We focus on finding the optimal combination in the IO stack design choices. We examined the behaviour of five filesystems (EXT4, XFS, F2FS, BTRFS, and NILFS2) and three IO schedulers provided by standard Linux kernel (CFQ, deadline, and NOOP) in a combinatorial manner. By using bFUSE and deadline-based IO scheduler and changing the filesystem to XFS, the IO performance (storage throughput) increases by 470% over the baseline (existing smartphone). With this optimisation, we are able to achieve 38 Mbyte/sec sequential write bandwidth and utilise 99% of the raw device bandwidth in the Android storage stack.

The remainder of this paper is organised as follows. Section 2 presents the background. The behaviour of the Android IO stack is analysed in Section 3. bFUSE is introduced in Section 4. Section 5 provides optimisations for the Android IO stack. Section 6 presents the results of our integration of the proposed schemes. Section 7 describes other works related to the study, and Section 8 concludes this paper.

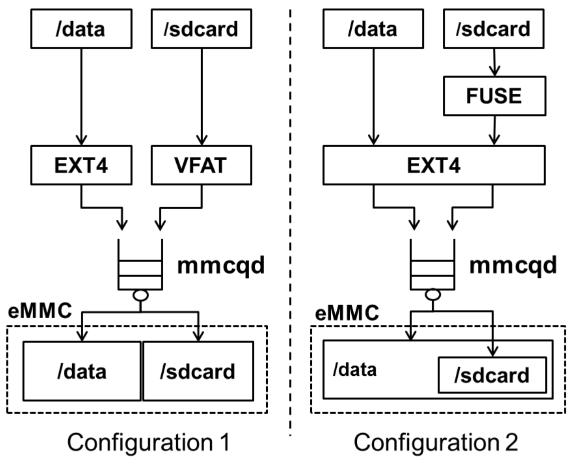
## 2 Background

### 2.1 *Storage partitions for android-based smartphones*

Android manages several filesystem partitions: `/recovery`, `/boot`, `/cache`, `/system`, `/data` and `/sdcard`. The `/system` partition contains Android-executable files and pre-installed applications. The `/cache` partition is used for updating firmware. The `/boot` and `/recovery` partitions are used for maintaining boot image. There are two types of main partition: '`/data`' and '`/sdcard`'. `/data` partition harbours SQLite tables, SQLite journals, and apps. `/sdcard` partition usually harbours pictures, video clips, and mp3 files, which are mostly user created contents. We call `/sdcard` an 'sdcard partition' or a 'user partition'.

Typical accesses to the data partition (`/data`) and the user partition (`/sdcard`) are small (4 Kbyte) synchronous writes and large buffered writes (> 64 Kbyte), respectively (Lee and Won, 2012). The IOs to '`/data`' partition are mostly caused by SQLite database operation and excessive number of IO operations are generated due to the uncoordinated interactions between SQLite and EXT4 (Lee and Won, 2012). Sdcard partition does not suffer from journaling of journal anomaly (Jeong et al., 2013b), but its performance is severely degraded by excessive overhead on the FUSE.

**Figure 1** Storage configuration for Android-based smartphones



There are two different approaches in configuring data and user partitions. Earlier smartphone models locate data and user partitions at different logical partitions (Google Nexus S, http://www.google.com/nexus/s/; Samsung Galaxy S, http://www.samsung.com/global/microsite/galaxys/index_2.html; Samsung Galaxy S2, http://www.samsung.com/global/microsite/galaxys2/html/). Configuration 1 in Figure 1 corresponds to this configuration. In this configuration, data and user partitions are formatted with different filesystems, EXT4 and VFAT, respectively. Recently, the Android adopted FUSE to manage both data and user partitions at the same physical partition (Google Galaxy Nexus, http://www.google.com/nexus/4; Samsung Galaxy S3, http://www.samsung.com/ae/microsite/galaxys3/en/index.html). This is to dynamically adjust the partition size, allowing more flexibility in utilising the storage space, and to maintain host-compatibility to transfer files to and from the host PC via multimedia transfer protocol (MTP) (Osborne et al., 2010). In this configuration, `/sdcard` is imposed with FUSE. FUSE imposed filesystem looks and works like an independent logical partition, and VFS mounts this partition with FUSE filesystem type. FUSE imposed filesystem resides in the existing concrete filesystem as a normal file (or a directory). Configuration 2 in Figure 1 illustrates this configuration. By mounting user partition (`/sdcard`) with FUSE, the system allows the users to freely manoeuvre a proportion of data and user partitions. However, overhead caused by the FUSE virtualisation service is prohibitively large.

Table 1 illustrates the information on data and user partitions of six Android-based smartphones. The models are sorted in chronological order. In earlier models, e.g., Nexus One, data partition uses raw NAND and is managed by log structured filesystem such as YAFFS2. The more recent smartphones, e.g., Nexus S (Google Nexus S, http://www.google.com/nexus/s/), Galaxy S (Samsung Galaxy S, http://www.samsung.com/global/microsite/galaxys/index_2.html), and Galaxy S2 (Samsung Galaxy S2, http://www.samsung.com/global/microsite/galaxys2/html/), began to use FTL loaded NAND storage device such as eMMC as their internal storage. In these models, data partition and user partition reside in separate partitions. In the most recent Android-based smartphones, such as Galaxy Nexus and Galaxy S3, data and user partitions reside in the same partition. User partition is mounted with FUSE and resides in the data partition as a normal directory (and files).

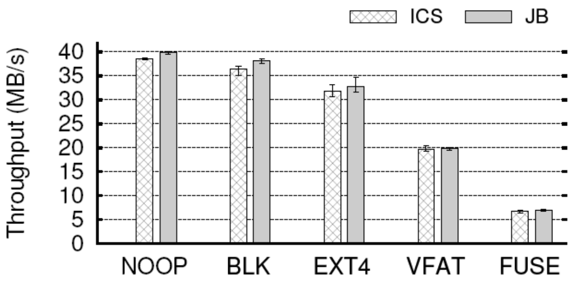
**Table 1** Storage configuration of various smartphone models

| *Model name* | *Year* | *Internal storage* | | *Ext-storage* |
|---|---|---|---|---|
| | | */data* | */sdcard* | */extSdCard* |
| Nexus One (http://en.wikipedia.org/wiki/Nexus_One) | 2010 | YAFFS2 | VFAT | VFAT |
| Nexus S (http://www.google.com/nexus/s/) | 2010 | EXT4 | VFAT | - |
| Galaxy S (http://www.samsung.com/global/microsite/galaxys/index_2.html) | 2010 | RFS | VFAT | VFAT |
| Galaxy S2 (http://www.samsung.com/global/microsite/galaxys2/html/) | 2011 | EXT4 | VFAT | VFAT |
| Galaxy Nexus (http://www.google.com/nexus/4) | 2011 | EXT4 | FUSE | - |
| Galaxy S3 (http://www.samsung.com/ae/microsite/galaxys3/en/index.html) | 2012 | EXT4 | FUSE | VFAT |

## *2.2 Filesystem in user space*

FUSE (File system in user space, http://fuse.sourceforge.net/) is a widely used framework that has been created to enable the filesystem to be developed in the user space without modifying kernel source code. The FUSE framework is especially helpful in realising virtual filesystems. FUSE allows programmers to directly control filesystem operation using simple application interface that is equivalent to `mount`, `open`, `read`, or `write` in the legacy system calls. On the other hand, FUSE causes additional context-switch and memory copy, which lead to significantly low performance. Some performance may be regained by increasing the speed of processor, memory, and bus. Rajgarhia and Gehani (2010) showed that FUSE exhibits similar performance to the existing kernel level filesystem on a desktop PC.

**Figure 2** Overhead of individual storage layers in eMMC on Galaxy-S3



Notes: NOOP: raw device with NOOP scheduler,
BLK: raw device with CFQ scheduler,
EXT4: EXT4 filesystem with CFQ scheduler,
VFAT: VFAT filesystem with CFQ scheduler,
FUSE: FUSE with EXT4 and CFQ scheduler.

**Table 2** Configuration options for each experiment

| *Label* | *NOOP* | *BLK* | *EXT4* | *VFAT* | *FUSE* |
|---|---|---|---|---|---|
| IO scheduler | NOOP | CFQ | CFQ | CFQ | CFQ |
| Filesystem | | | EXT4 | VFAT | EXT4 |
| FUSE | | | | | Yes |

## *2.3 Dissection of the IO stack overhead*

We overhauled the IO stack for sdcard partition access and analysed the overhead caused by each layer of the storage stack. We conducted an experiment on a commercially available smartphone model, Samsung Galaxy S3 (http://www.samsung.com/ae/microsite/galaxys3/en/index.html) (Samsung Exynos 4412 1.4 GHz Quad-core, 2 GB RAM, 32 GB eMMC[1]), running Android 4.0.4 (ICS) and Android 4.1.2 (JB). The version of each of the Android kernel is 3.0.15 and 3.0.30, respectively. The data partition resides at internal flash storage and is formatted with EXT4 filesystem. The user partition is imposed with FUSE and actually resides at the data partition. The default IO scheduler in the Android is complete fair queuing (CFQ) (Axboe, 2007).

We generated sequential write workloads with 512 KB unit size and 512 MB file size using Mobibench (Jeong et al., 2013a). First, we measured the performance of the raw device. We ‘`open()`’ the `/sdcard` partition and generated the workloads. To minimise interference on the storage layers, we used the IO scheduler, ‘NOOP’, instead of the default ‘CFQ’. CFQ is known to be more CPU demanding than NOOP (Kim et al., 2009). This experiment is labelled NOOP. Second, we changed the disk scheduler to CFQ and still used the raw device. This is to measure the overhead of disk scheduler. This is labeled BLK. In the third experiment (EXT4), we measured the filesystem overhead. We formatted the storage device with EXT4 filesystem and measured the performance. In the fourth experiment (VFAT), we used VFAT instead of EXT4. In the fifth experiment (FUSE), we mounted the FUSE partition, which resides on EXT4, and measured the IO performance. This is the default configuration for the user partition in Galaxy S3. Table 2 summarises the labels and the respective experiment operations.

**Figure 3** IO overhead breakdown

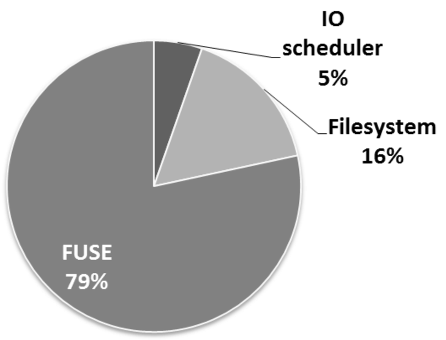


Figure 2 illustrates the results. The labels in the x-axis denote the five experiments. In ICS and JB, the raw device (NOOP) yielded sequential write bandwidth of 39 Mbyte/sec and 40 Mbyte/sec, respectively. The performance decreased by 7% and 5% in ICS and JB, respectively, when CFQ disk scheduler was used (BLK). The performance decreased by 14% and 16%, respectively, when EXT4 filesystem was used. In the default configuration for the user partition of Galaxy S3, i.e., when we imposed FUSE on the filesystem, we obtained sequential write bandwidth of only 7 Mbyte/sec on both Android versions. This is less than 20% of the physical bandwidth of the raw device.

In the Android storage stack of JB version, the overhead of FUSE, EXT4, and IO scheduler account for 79%, 16%, and 5% of the total overhead, respectively (Figure 3), and only 20% of the raw device performance is available to the applications. The IO accesses to the user partition suffer from excessive software overhead. The dominant fraction of the overhead is caused by FUSE. The overhead of filesystem, EXT4, is more significant in a mobile device with NAND-based storage than in a desktop with an HDD. In a desktop with an HDD, it is commonly accepted that the filesystem overhead is 5% of the total overhead (Giampaolo, 1998). When the Android platform adopts FUSE instead of VFAT to manage the user partition, the IO performance decreases by 50%. Based on these observations, we carefully conjecture that from Google's point of view, the flexible and efficient space management must be more important than storage performance in current smartphone models.

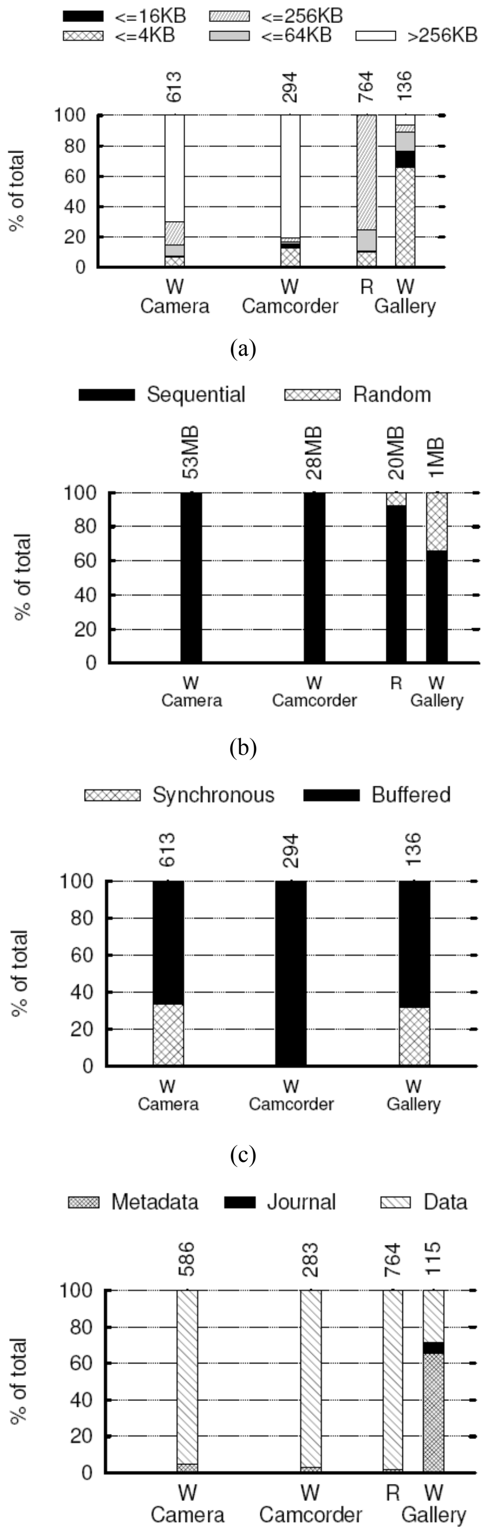
## 3 The Android IO stack for the user partition

### *3.1 IO characteristics of the user partition accesses*

To design an IO subsystem, it is mandatory to acquire a thorough understanding on the IO workload characteristics for the respective storage partition. We selected the most popular Android apps that use sdcard partition: camera, camcorder, and gallery. For camera application, we took pictures consecutively for one minute. For camcorder, we recorded a video for one minute, with full HD resolution (1,920 × 1,080 @ 30 fps). For gallery, we scrolled the thumbnails and displayed each picture for one minute. We ran the experiment on Galaxy S3 with the user partition mounted with FUSE. We used mobile storage analysis tool (MOST) (Jeong et al., 2013a) to collect and analyse the trace. MOST can extract the file type (journal vs. data file) and block type (data vs. metadata) information for a given block access. In this study, we examined the IO size distribution, spatial locality (random vs. sequential), fraction of synchronous IO, and fraction of accesses for each filesystem region (metadata, journal, and data) for a given Android apps. This information provides an important guideline in designing and optimising the filesystem and the respective storage stack. The following summarises IO characteristics of the IOs to the user partition.

- *Most of IOs are over 256 Kbyte in size*. In camera and camcorder, 70% of the write requests to the block device are larger than 256 Kbyte [Figure 4(a)]. This is an important characteristic in designing the filesystem as well as the storage device. In particular, the current FUSE splits the incoming IO requests into 4 KB units. When the incoming request is large, this splitting behaviour of the FUSE causes significant overhead. For the data partition accesses in Android, 64% of the write operations is 4 Kbyte (Jeong et al., 2013b).
- *Over 90% of the total IOs are sequential*. For sdcard partition accesses, sequential IOs account for over 90% of the total IOs [Figure 4(b)]. The percentages were near 100% for camera and camcorder. We have observed an entirely different access characteristic in the data partition accesses. In the data partition accesses, random writes constitute a dominant fraction (75%) of the entire write operations.
- *Over 70% of the total IOs are buffered*. Buffered IOs accounted for over 70% of all IOs in all three applications [Figure 4(c)]. The percentage of buffered IOs reached 100% for camcorder application. This is in direct contrast to the data partition access characteristics (Jeong et al., 2013b). In the data partition accesses, 70% of the writes are synchronous. They are caused by SQLite database behaviour.

**Figure 4** IO distribution of file types, block types, IO modes, randomness, and IO size, (a) IO size (b) locality (c) IO modes (d) block types



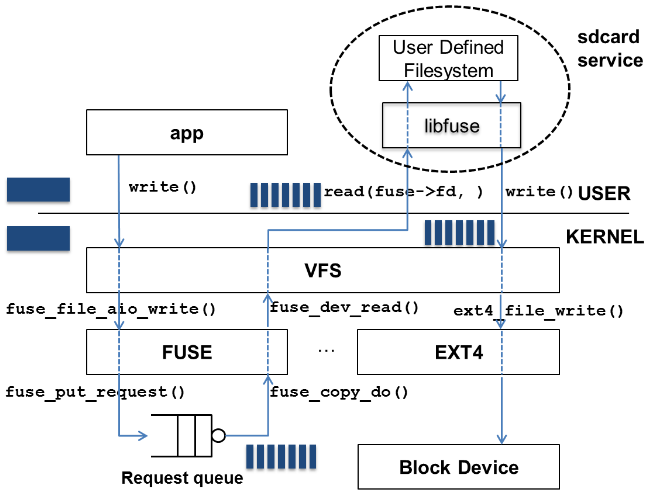
Note: The number at the end of each bar indicates the number of respective block IO for R (read) and W (write).

- *Journal and metadata writes are negligible*. We categorised the IO accesses into three groups based on the type of filesystem region: data, metadata, and journal. Accesses to the filesystem journal and metadata are negligible. Data writes account for 95% and 97% of the total IO counts for camera and camcorder, respectively [Figure 4(d)]. In the case of gallery, data reads account for almost 100% of the total read counts and data writes account for 30% of the total write counts.

## 3.2 *Anatomy of the FUSE behaviour*

We observed that 79% of all IO overhead, which is all IO accesses to sdcard partition, is caused by FUSE (Figure 3). We examined the call path of accessing the sdcard partition to analyse the cause for this overhead. Figure 5 schematically illustrates the mechanism for a '`write()`' system call. When the application makes a write system call to the FUSE partition, VFS forwards the request to the FUSE layer. The FUSE layer inserts this request to its own request queue, which resides in the kernel. The OS hands over the CPU to another thread after it inserts the write request to the queue. User-level FUSE library reads the FUSE request queue through VFS interface. FUSE library is run by its own thread. The FUSE layer splits the IO request in the request queue into 4 Kbyte units and returns them to the FUSE library. The FUSE library translates the incoming IOs into the filesystem operations appropriate for EXT4 (or other existing concrete filesystem where FUSE resides) and creates the request.

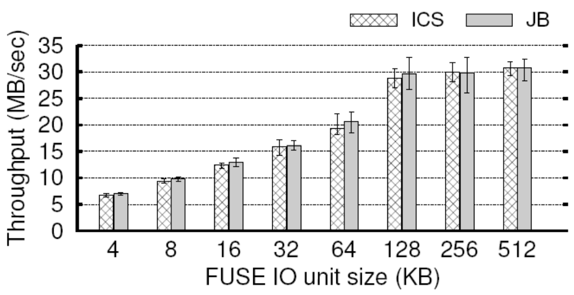
**Figure 5** `write()` path of the FUSE framework (see online version for colours)



We identified the main source of FUSE's inefficiency to be IO fragmentation. The FUSE library removes an IO request from the FUSE mount filesystem in 4 Kbyte units, which we call 'FUSE IO units'. Since the default FUSE IO unit is 4 Kbyte, the FUSE library splits a large IO request from the application into multiple IO requests. This IO fragmentation increases the system call processing overhead and the number of context switches. Also, IO fragmentation can dismantle the spatial locality that the original IO stream bears. Since large IO requests are split into multiple requests, it is possible that a set of split requests are interleaved with other IO streams, e.g., journal buffer flush and metadata synchronisation. As a result, the aggregate stream may look 'more' random. It is critical for NAND-based flash storage to properly identify the randomness and hotness of the underlying traffic (Lee et al., 2008; Hsieh et al., 2006). Loss of access locality may significantly aggravate the inefficiency of the underlying storage behaviour.

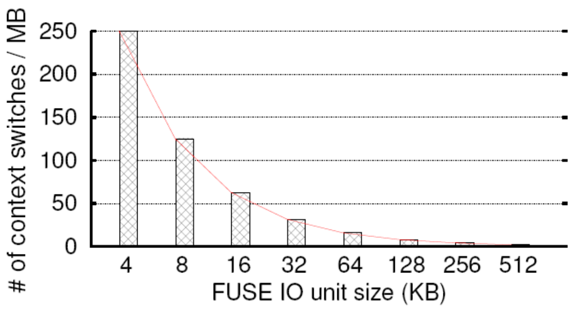
In the Android IO stack, the side effect of IO fragmentation amplifies dramatically because most of sdcard accesses are large sized (> 256 Kbyte) and mobile devices have relatively low CPU and memory copy performance. Subsequently, the overhead of going through the FUSE layer becomes prohibitively large. We physically examined the overhead of IO fragmentation. We varied the FUSE IO size from 4 Kbyte to 512 Kbyte and measured the throughput for sequential write. IO size denotes the unit size into which the FUSE library splits the IO request at the request queue. Figure 6 illustrates the results. On both Android versions, with 4 Kbyte FUSE IO unit size, the internal storage yielded only 7 Mbyte/sec. With 512 Kbyte IO unit size, the internal storage yielded 30 Mbyte/sec sequential write throughput.

**Figure 6** Throughput vs. FUSE IO unit size



Notes: Sequential buffered write, internal eMMC, file: 512 Mbyte, record-size: 512 Kbyte.

**Figure 7** Number of context switches vs. FUSE IO size (see online version for colours)



Notes: Sequential buffered write, internal eMMC, file: 512 Mbyte, record size: 512 Kbyte.

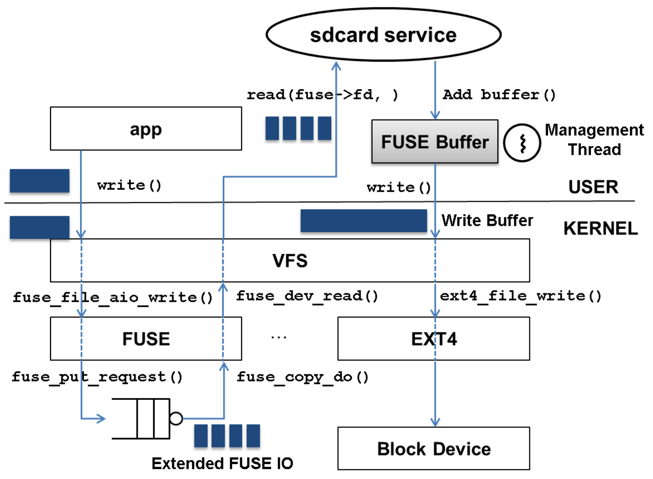
We examined the number of context switches for each FUSE IO unit size using Mobibench (Jeong et al., 2013a). Figure 7 illustrates the results. The total number of context switches decreases as the IO unit size increases.

## 4 Buffered FUSE

### *4.1 Design*

We carefully examined the IO workload characteristics of sdcard accesses in the Android platform and designed a novel user-level file system mechanism, called bFUSE, to effectively address the IO fragmentation issue of the FUSE. bFUSE is specifically designed for the Android IO stack where a dominant fraction of IOs are large sized (> 256 Kbyte) sequential (and buffered) writes. bFUSE practically eliminates all overhead of the FUSE layer and increases the efficiency of the FUSE while preserving the flexibility in the storage space management.

**Figure 8** `write()` path of bFUSE framework (see online version for colours)



Despite its dramatic performance impact, the design and implementation of bFUSE are rather simple and straight forward. bFUSE consists of the following key ingredients. First is extended FUSE IO unit size. In bFUSE, the FUSE IO unit increases from 4 Kbyte to 512 Kbyte. Note that about 70% of the IO accesses are larger than 256 Kbyte [Figure 4(a)]. Second, we introduce a FUSE buffer between the FUSE library and VFS. Upon reading a request from legacy FUSE request queue, the sdcard service immediately issues the request to the underlying filesystem. In bFUSE, the FUSE library takes incoming requests from request queue in kernel and places them in the FUSE buffer. Third, we introduce management thread for the FUSE buffer which periodically flushes the FUSE buffer contents to the underlying filesystem. Thread-based FUSE buffer management enables the FUSE library, i.e., FUSE, to process IO requests concurrently with the FUSE buffer management exploiting multiple cores of modern mobile CPU. Figure 8 illustrates the organisation of bFUSE framework. The FUSE buffer resides at the user space (within the FUSE library).We implemented FUSE buffer at the user address space instead of the kernel space to reduce the number of system calls and to dynamically adjust the FUSE buffer size.

### *4.2 Implementation*

A FUSE buffer is an array of fixed size entries. An entry consists of buffer header and buffer data. Buffer header contains file descriptor, offset, and size. Buffer data is a fixed size region to hold data blocks for write operations. A sufficiently large region, 2 Mbyte, is allocated to an entry so that it can hold consecutive write requests in a single entry. Figure 9 illustrates the structure of the buffer.

When the FUSE library inserts a request to the FUSE buffer, it searches the FUSE buffer (write buffer) entries for the same file descriptor. If the incoming request and the existing FUSE buffer entry form a consecutive region of a file, the FUSE library places the data blocks of the incoming request in the data block region of the existing entry and updates the size field. When the data region of the existing entry is full, the FUSE library allocates a new entry in the FUSE buffer.

**Figure 9** Structure of bFUSE (see online version for colours)

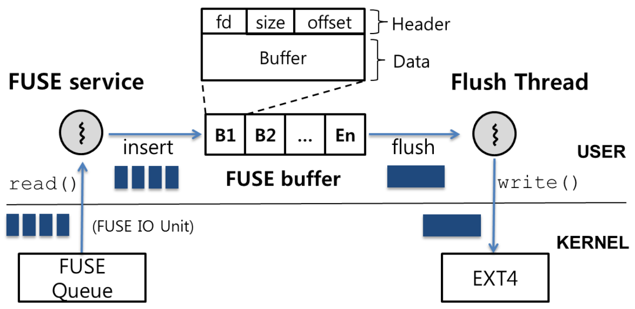
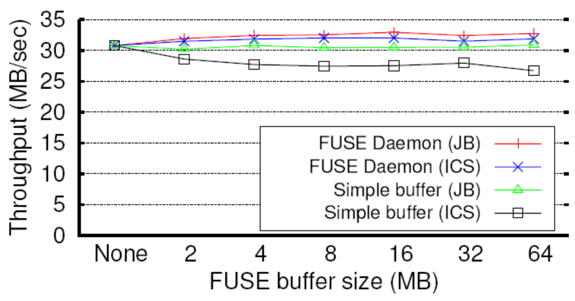


**Figure 10** Effect of separate thread management under various FUSE buffer sizes (see online version for colours)



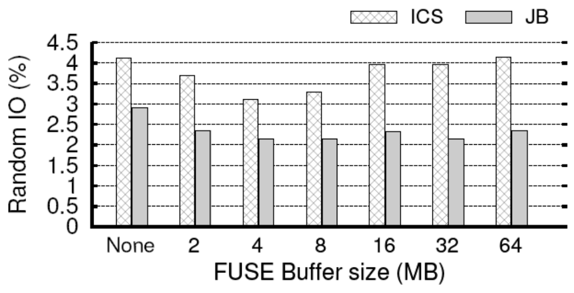
Notes: Sequential buffered write, internal eMMC, filesystem: EXT4, file size: 512 Mbyte, record size: 512 Kbyte.

bFUSE may negatively affect the overall IO performance due to memory copy overhead and increased latency to flush IO requests to the filesystem. The size of the region is user configurable. We performed a series of experiments to determine the right size for the data region. We managed the bFUSE with producer-consumer paradigm and allocated a separate thread for flushing the buffer. Figure 9 illustrates the structure.

The FUSE buffer manager regularly flushes buffer entries. In addition, it flushes buffer entries for a given file when the file is closed, or when the request reaches the end of the file, or when `fsync()/fdatasync()` is called. The performance of the user partition accesses relies heavily on the interval at which the FUSE buffer manager flushes the contents. We examined the IO performance under various flush intervals and found that 10 msec yields the best performance. In all experiments, flush interval of FUSE buffer manager was set to 10 msec. We measured the throughput of sequential buffered writes (512 KB FUSE IO unit size) both with and without the FUSE buffer manager. Figure 10 illustrates the results. With single threaded implementation of bFUSE, throughput actually decreases with introduction of bFUSE, without the FUSE buffer manager. With the FUSE buffer manager, we achieved 8% increase in performance. Based on the detailed examination of the process, we believe the reason for this performance gain is reduced randomness in the incoming IOs. The objective of the FUSE buffer is to coalesce multiple IO requests into one so that we can reduce the aggregate system call overhead and make the larger fraction of IOs sequential. Most NAND flash storages are designed to exploit sequentiality of the incoming IOs (Lee et al., 2005) to improve the IO performance and to lengthen the storage life time.

We examined the ratio of random IOs under various FUSE buffer size. Figure 11 illustrates the results. We can see that as the ratio of random IOs decreases, the throughput increases. With 8 Mbyte FUSE buffer, the ratio of random IOs to the entire IOs is the smallest.

**Figure 11** Fraction of random IOs (IO count)

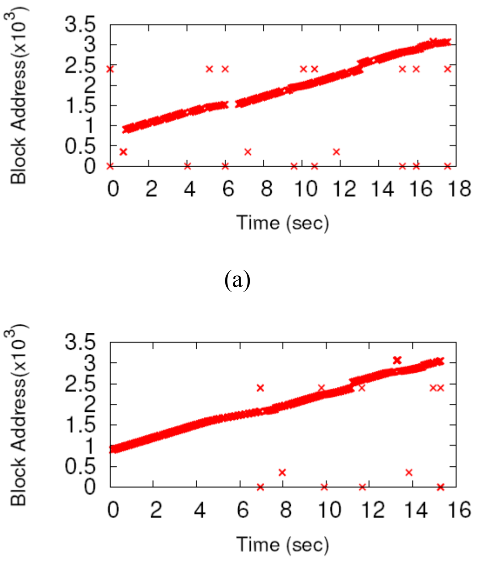


Notes: Sequential buffered write on internal eMMC through FUSE with different size of FUSE IO unit. File size: 512 Mbyte, record size (app): 512 Kbyte.

The FUSE buffer successfully reduces the number of journal and metadata updates and the number of system calls reducing the system call overhead. Also, by coalescing multiple write requests into one, there is less chance that sequential IOs issued by the application are inter-mixed with other IO streams, e.g., journal flush. We examined the block access trace for a sequential write of 512 Mbyte file with and without FUSE buffer. Figure 12 illustrates the results. Without the FUSE buffer, writing a file consists of higher number of smaller write bursts [Figure 12(a)]. With the FUSE buffer, writing a file consists of lower number of larger write bursts. Also, with the write buffer, there are fewer journal updates and metadata updates. Occasional updates in the 0–0.5 × $10^3$ LBA range and 2.5 × $10^3$ range are for metadata updates and journal updates, respectively.

The FUSE buffer in the FUSE layer brings a number of improvements to the Android storage stack: reductions in the number of `write()` system calls, the number of context switches, randomness in the underlying IO traffic, and filesystem journaling overhead.

**Figure 12** Effect of FUSE buffer on randomness of the IO accesses, (a) without FUSE buffer (b) with FUSE buffer (see online version for colours)

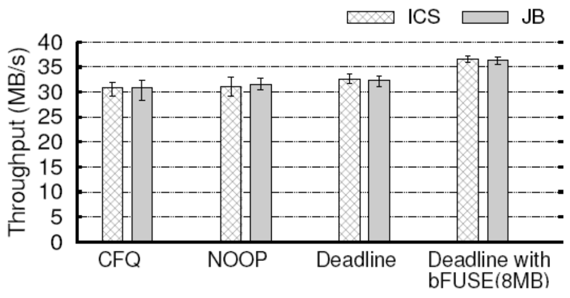


(a)

(b)

Notes: Sequential buffered write on internal eMMC through FUSE with/without bFUSE. Base filesystem: EXT4, file size: 512 Mbyte, record size (app): 512 Kbyte.

**Figure 13** Effect of IO scheduling policy



Notes: Sequential buffered write on internal eMMC through FUSE with three different IO scheduler. File size: 512 Mbyte, record size: 512 Kbyte.

## 5 Optimising the Android storage stack

In an effort to optimise the entire IO stack, we examined the throughput of the user partition under five filesystems, EXT4, XFS, F2FS, BTRFS, and NILFS2; and three IO schedulers, NOOP, CFQ, and deadline; in a combinatorial manner.

## *5.1 IO scheduler*

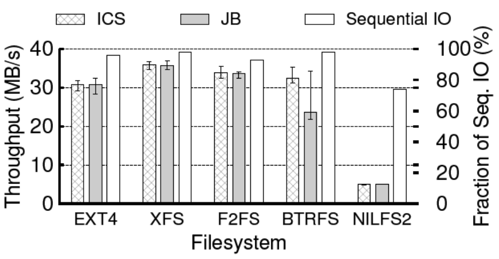
Linux kernel currently provides CFQ, NOOP, and deadline IO schedulers, with CFQ being the default. These schedulers are designed for HDDs and should be reexamined in the context of NAND-based storage devices (Wang et al., 2013; Shen and Park, 2013). We compared the performance of CFQ, NOOP, and deadline as the IO scheduler. Figure 13 shows the results. The default IO scheduler, CFQ, yielded the worst performance among the three, with deadline performing 6% better than CFQ. With 8 Mbyte FUSE buffer and deadline driven scheduling, the sequential write throughput increases by 12% compared to the CFQ IO scheduler.

## *5.2 Filesystem*

We examined the performance of five widely used filesystems in Linux: EXT4, BTRFS, NILFS, XFS, and F2FS. We examined how well these filesystems match with the FUSE framework. XFS (Sweeney et al., 1996) is a journaling filesystem designed for enterprise class storage systems. BTRFS (Rodeh et al., 2012) is copy-on-write-based filesystem and is envisioned as the future filesystem for Linux OS. NILFS2 is a log-structured filesystem (Konishi et al., 2006). We included NILFS2 to examine the effectiveness of the legacy log structured filesystem on the Android platform. Flash Friendly Filesystem, F2FS (F2FS Patch on LKML, https://lkml.org/lkml/2012/10/5/205), is the youngest among the five, specifically designed for flash-based storage.

`mkfs` tools for F2FS, NILFS2, and BTRFS are ported for ARM processor. Since F2FS is recently merged into the standard Linux kernel 3.8 (F2FS File-System Merged Into Linux 3.8 Kernel, http://www.phoronix.com/scan.php?page=news_item&px=MTI1OTU), we backported F2FS to Linux 3.0.15 (Android ICS for Galaxy S3) and Linux 3.0.30 (Android JB for Galaxy S3), respectively. We measured the performance of sequential writes (buffered IOs). IO size was 512 Kbyte. FUSE IO unit size was 512 Kbyte and FUSE buffer was not used. Figure 14 shows the throughput for the different filesystems.

**Figure 14** Throughput vs. fraction of sequential IO counts



Notes: Sequential buffered write on internal eMMC through FUSE with five different base filesystems. File size: 512 Mbyte, record size: 512 Kbyte, FUSE IO Size: 512 Kbyte, without FUSE buffer.

Without the FUSE buffer, XFS and F2FS exhibited 17% and 10% improvements, respectively, over EXT4 on both Android versions. BTRFS exhibited 5% improvement over EXT4 on ICS while the performance decreased by 20% on JB. NILFS2 performed significantly worse than the rest. In addition to the write requests to the storage device, there were occasional kernel generated IO requests, such as journal updates and metadata updates. These requests negatively affect the filesystem performance because it breaks the sequentiality in the underlying IO stream. Figure 14 also shows the fraction of sequential IOs to all IO requests to the filesystem. The filesystem's performance coincides with the fraction of sequential IOs.

## *5.3 Analysis*

We analysed block level access patterns in these filesystems. Figure 15 shows the time series of block accesses in writing 512 Mbyte. X-axis is time and Y-axis is the logical block address. It only shows `write()` operation. Sequential write patterns are commonly observed in all filesystem block traces. IO requests in the lower LBA region ($0{\sim}0.5 \times 10^3$) are for metadata updates.

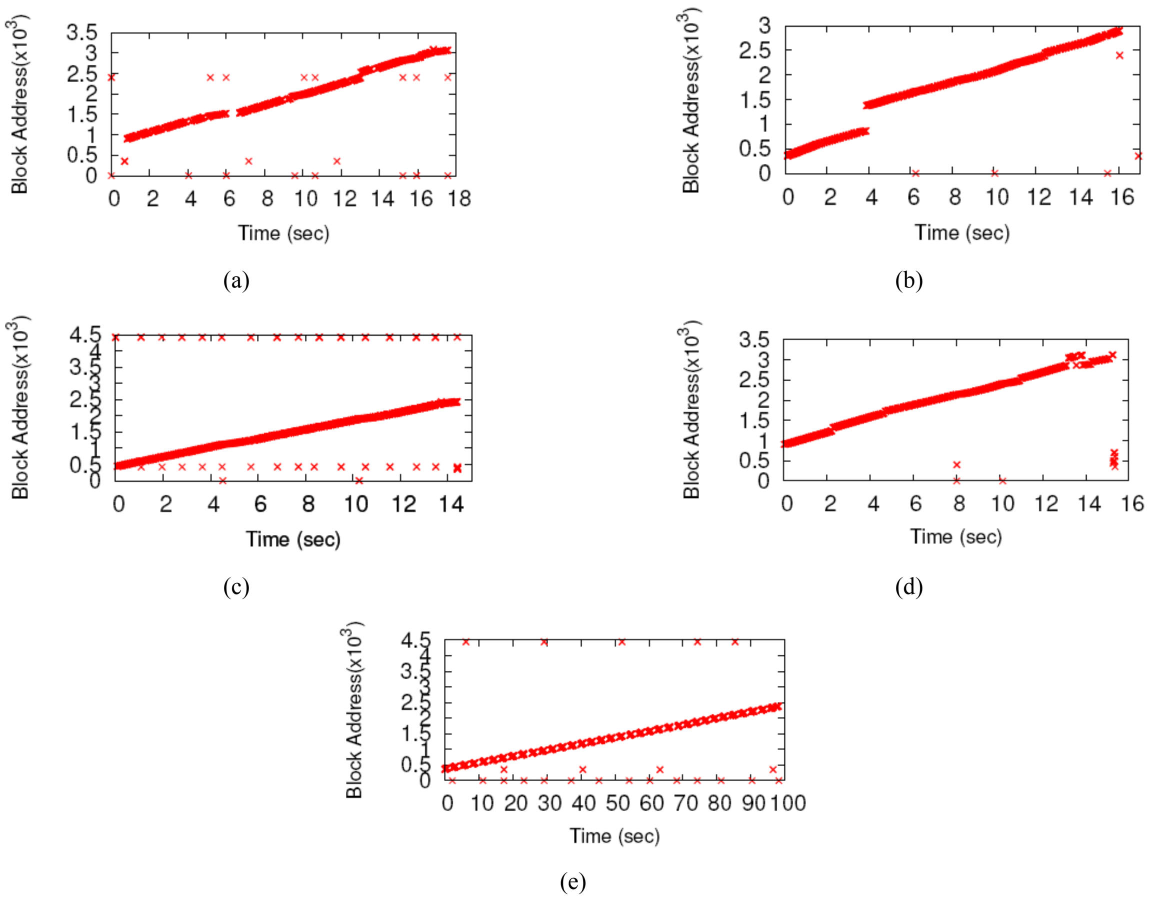
The FUSE buffer and extended FUSE IO unit are designed to mitigate or to remove the overhead caused by IO fragmentation. We examined in detail how these features contribute to minimising the overhead of FUSE. We generated sequential write workloads and measured the number of system calls, the number of block IO requests issued to the NAND storage device, and the number of metadata and journal updates. Without extended FUSE IO unit, writing 512 Mbyte consists of 128,000 write system calls since FUSE library splits the incoming IO requests into 4 Kbyte units. By increasing the FUSE IO unit size to 512 Kbyte, the number of write system calls decreases by $\times$ 128 (from 128,000 to 1,000). With 8 Mbyte FUSE buffer, we further reduce the number of system calls by $\times$ 4 (from 1,000 to 249). With bFUSE (extended FUSE IO and a FUSE buffer), we are able to achieve sheer $\times$ 512 reduction in the number of system calls (from 128,000 to 249). On average, bFUSE coalesces four write system calls from the application into one write system call with the help of the FUSE buffer. IO requests shown in Table 3 are the requests issued to the internal storage. Since the maximum IO size is 512 Kbyte in eMMC standard (e-MMC, 2011), the number of IO requests remains the same regardless of FUSE buffer. In data partition accesses, journal and metadata update operations constitute a dominant fraction of all IO operations (Lee and Won, 2012). We found that for the user partition accesses, i.e., buffered writes, the journal and metadata update overhead is negligible.

**Table 3** The number of system calls, IO operations and metadata/journal updates

| *Filesystem* | | *EXT4* | *XFS* | *F2FS* | *BTRFS* | *NILFS2* |
|---|---|---|---|---|---|---|
| # of pwrite() | FUSE | 128,000 | 128,000 | 128,000 | 128,000 | 128,000 |
| | Nobuf | 1,000 | 1,000 | 1,000 | 1,000 | 1,000 |
| | bFUSE | 249 | 249 | 249 | 249 | 249 |
| # of IO request | FUSE | 1,048 | 1,121 | 1,044 | 1,041 | 1,077 |
| | Nobuf | 1,032 | 1,112 | 1,048 | 1,016 | 1,070 |
| | bFUSE | 1,030 | 1,099 | 1,049 | 1,011 | 1,059 |
| Metadata/journal | FUSE | 23 | 11 | 26 | 15 | 153 |
| | Nobuf | 12 | 12 | 33 | 9 | 137 |
| | bFUSE | 11 | 6 | 35 | 9 | 142 |

Note: FUSE: baseline, nobuf: bFUSE without FUSE buffer, bFUSE: bFUSE with 8 Mbyte FUSE buffer.

**Figure 15** Block access pattern, sequential buffered write on internal eMMC through FUSE with five different base filesystems, (a) EXT4 (b) XFS (c) F2FS (d) BTRFS (e) NILFS2 (see online version for colours)



Notes: File size: 512 Mbyte, record size: 512 Kbyte.

## 6 Combined study

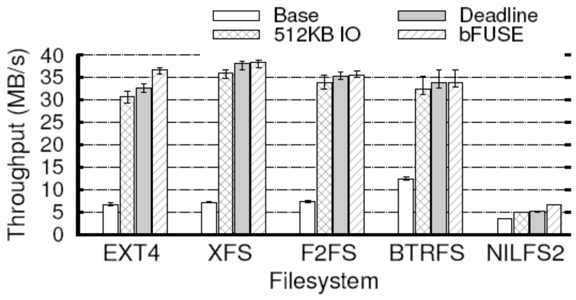
We examined the performance of the user partition accesses under each combination of storage stack options. There are five different filesystems, EXT4, XFS, F2FS, BTRFS, and NILFS2; and three IO schedulers, NOOP, CFQ, and deadline. We tested the storage throughputs under 4 Kbyte to 512 Kbyte FUSE IO unit sizes and 2 Mbyte to 64 Mbyte FUSE buffer sizes. It is reported that IO characteristics of Android-based smartphones are not sensitive to hardware models (Kim et al., 2012a; Lee and Won). Galaxy S3 with Android 4.0.4 (ICS) and Android 4.1.2 (JB) are used as baseline. In the baseline configuration, FUSE IO unit size is 4 Kbyte, and the default IO scheduler is CFQ.

Figure 16 shows the performances of five filesystems. Experiment for each filesystem was conducted in four steps: First, we tested the baseline configuration, 'base', which is CFQ IO scheduler with FUSE IO unit of 4 Kbyte. Second, FUSE IO unit is increased to 512 Kbyte, shown as '512 KB IO'. Third, the IO scheduler is changed to deadline, keeping FUSE IO unit at 512 Kbyte. This step is labelled 'deadline'.

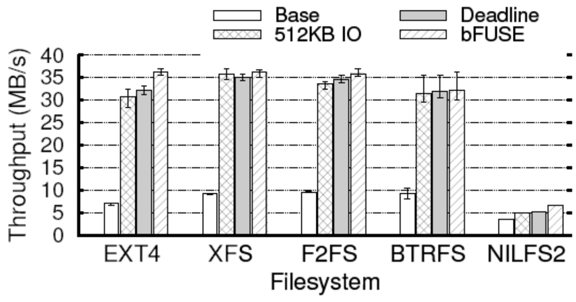
Fourth, the FUSE buffer is set at 8 Mbyte (with 512 Kbyte FUSE IO and deadline IO scheduler). This step is labelled 'bFUSE'.

In EXT4 filesystem on both Android versions, extended IO size led to about 350% performance enhancement, change of IO scheduler added about 5% enhancement, and bFUSE added 12% more enhancement. Performance enhancements in XFS, F2FS, and BTRFS for each step were similar. It is worth noting that a FUSE buffer needs to be managed by a separate thread so as not to block the caller. Otherwise, the performance gain becomes less significant.

**Figure 16** Effect of individual optimisation options, (a) ICS (b) JB



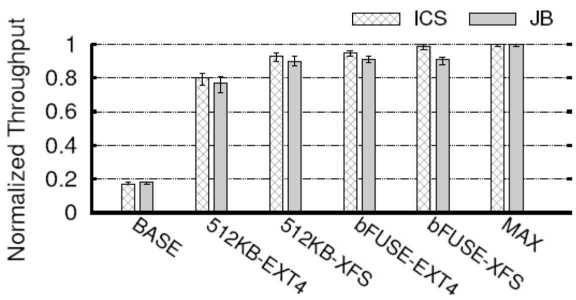
(a)



(b)

Notes: Sequential buffered write on internal eMMC with different filesystem. File size: 512 Mbyte, record size (app): 512 Kbyte.

**Figure 17** Throughput of optimised android storage stack for user partition accesses



Notes: Sequential buffered write on internal eMMC. IO scheduler: deadline, file size: 512 Mbyte, record size (app): 512 Kbyte.

Finally, we combine and summarise the results of all experiments (Figure 17). For ease of comparison, the throughput was normalised to the raw device throughput, MAX (38.6 Mbyte/s and 39.8 Mbyte/s for ICS and JB, respectively). In EXT4 with bFUSE (512 Kbyte FUSE IO size and 8 Mbyte FUSE buffer), we achieved 95% and 91% of the raw device bandwidth for ICS and JB, respectively. With XFS filesystem and the bFUSE (512 Kbyte FUSE IO unit, 8 Mbyte FUSE buffer, deadline scheduler), we eliminated most of the software overhead and achieved 99% and 91% of the raw device throughput for ICS and JB, respectively. This corresponds to spectacular improvement of 470% and 419% from the original baseline configuration for ICS and JB, respectively.

## 7 Related work

In the past few years, many studies have examined the performance of NAND flash-based storage devices. Kim et al. (2012a) showed that the determinant of mobile device performance is not the network speed but storage, which is contrary to popular belief.

Kim et al. (2012b) conducted a study on improving buffer cache management in order to improve IO performance of smartphones using low-cost flash memory. They developed a new buffer cache replacement scheme, spatial clock, which addresses the issue of spatial locality that has been neglected in other algorithms. These algorithms, including LRU, clock (Bensoussan et al., 1972)], Linux2Q (Bovet and Cesati, 2005), CFLRU (Park et al., 2006), LRUWSR (Jung et al., 2008), FOR (Lv et al., 2011), and FAB (Jo et al., 2006), focused only on reducing write cycles. Spatial clock is based on the clock algorithm, but page frame is aligned with the logical sector number. Alignment was achieved using AVL tree (Adelson-Velskii and Landis, 1963).

Lee and Won (2012) performed comprehensive analysis on the Android IO workload. They performed trace driven analysis on 14 apps from various categories, e.g., contact, calendar, gallery, camera, twitter, etc. They found that typical IOs to data partition (`/data`) and sdcard partition (`/sdcard`) are 4 Kbyte write followed by `fsync()` and large size buffered write (> 64 Kbyte), respectively. They also found that SQLite and EXT4 interact in an unexpected way and put significant stress on the underlying storage. On the other hand, Chiang and Lo (2006) proposed a component-based VFAT file system on embedded system, which improved performance of VFAT by supporting RAM-based storage and disk-based storage. They not only saved system memory but also improved performance of read and write operation on resource-limited embedded system environment.

Studies have been done on the relationship between the actions of FTL and the unnecessary writes generated by the EXT4 journal. Jang and Lim (2012) asserted that FTL should be located on the host side instead of NAND controller to be more efficient, taking into account the resource constraints (RAM and CPU) for page mapping table and redundant writing by the journal. They modified MTD and JBD of Linux 2.6.35 and compared the performance of host-side FTL to that of controller-side FTL on NANDsim. Random write IOPS improved about 20% to 30%. Although this study is significant in that it presents a solution to overcoming hardware limitations of FTL as well as performance reduction caused by excess journaling, it does not provide specific instructions on implementing the proposed method nor explains the cause for performance enhancement. Falaki et al. (2010) analysed various smartphone usage patterns. On top of the performance increases achieved by these earlier works (Kim et al., 2012a; Lee and Won, 2012), bFUSE provides an effective solution to further enhance the performance of the Android IO stack.

## 8 Conclusions

The role of smartphones is changing from a simple office assistant, which handles contact management, schedule management, and e-mails, to a professional device for creating various types of multimedia contents, replacing cameras and camcorders. There is a significant demand for higher performance storage subsystem to harbour better quality pictures and videos. The Android storage stack allocates a separate storage partition for user-created multimedia files. The Android storage stack for the user partition, as it currently stands, is an uncoordinated collection of software layers with excessive overhead. It can draw only 20% of the storage performance. In this study, we focus our efforts on optimising the Android IO stack for sdcard partition accesses. We characterised the IO workload to sdcard partition, examined the overhead of individual software layers, and found the optimal storage configuration for sdcard partition accesses. We found that a dominant fraction of software overhead is caused by FUSE. We propose bFUSE, bFUSE, to address the performance issues of the legacy FUSE. We examined the performances of five filesystems and three disk schedulers to find the best match for the bFUSE framework. After applying 512 Kbyte FUSE IO unit and 8 Mbyte FUSE buffer in bFUSE, and with XFS in the deadline scheduling mode, the proposed Android IO stack achieved 470% and 419% performance improvements in Android 4.0.4 (ICS) and Android 4.1.2 (JB), respectively, over the existing state of the art smartphone model with the default options. The observed performance corresponds to 99% and 91% of the raw device bandwidth in ICS and JB, respectively. The proposed Android IO stack opens up a new opportunity for the existing smartphone storage to handle higher definition multimedia contents such as ultra-high definition quality video recording (3,840 × 2,160 @ 60 frames/sec).

## Acknowledgements

This work is sponsored by IT R&D programme MKE/KEIT. [No. 10035202, Large Scale hyper-MLC SSD Technology Development], and by IT R&D programme MKE/KEIT. [No. 10041608, Embedded system Software for New-memory-based Smart Device].

## References

Adelson-Velskii, M. and Landis, E.M. (1963) *An Algorithm for the Organization of Information*, Vol. 146, pp.263–266, Defense Technical Information Center, in Russian, English translation by Myron J. Ricci in *Soviet Math. Doklady*, Vol. 3, pp.1259–1263, 1962.

Axboe, J. (2007) 'CFQ IO scheduler', in Presentation at *Linux. Conf. AU*, January.

Bensoussan, A., Clingen, C.T. and Daley, R.C. (1972) 'The multics virtual memory: concepts and design', *Communications of the ACM*, Vol. 15, No. 5, pp.308–318.

Bovet, D. and Cesati, M. (2005) *Understanding the Linux Kernel*, O'Reilly Media, Incorporated, Sebastopol, CA, USA.

Chiang, M-L. and Lo, C-J. (2006) 'Lyrafile: a component-based vfat file system for embedded systems', *International Journal of Embedded Systems*, Vol. 2, No. 3, pp.248–259.

Embedded Multi-Media Card (e-MMC) (2011) Electrical Standard (4.5 Device), June.

F2FS File-System Merged into Linux 3.8 Kernel [online] http://www.phoronix.com/scan.php?page=news_item&px=MTI1OTU (accessed 8 November 2013).

F2FS Patch on LKML [online] https://lkml.org/lkml/2012/10/5/205 (accessed 8 November 2013).

Falaki, H., Mahajan, R., Kandula, S., Lymberopoulos, D., Govindan, R. and Estrin, D. (2010) 'Diversity in smartphone usage', in *Proc. of the 8th International Conference on Mobile Systems, Applications, and Services*, ACM, pp.179–194.

File system in user space [online] http://fuse.sourceforge.net/ (accessed 8 November 2013).

Galaxy Camera [online] http://www.samsung.com/us/photography/galaxy-camera (accessed 8 November 2013).

Galaxy S2 [online] http://www.samsung.com/global/microsite/galaxys2/html/ (accessed 8 November 2013).

Giampaolo, D. (1998) *Practical File System Design with the Be File System*, Morgan Kaufmann Publishers Inc., San Francisco, CA.

Google Galaxy Nexus [online] http://en.wikipedia.org/wiki/Nexus_4 (accessed 8 November 2013).

Google Nexus S [online] http://en.wikipedia.org/wiki/Nexus_S (accessed 8 November 2013).

Hsieh, J-W., Kuo, T-W. and Chang, L-P. (2006) 'Efficient identification of hot data for flash memory storage systems', *TOS*, Vol. 2, No. 1, pp.22–40.

HTC Dream [online] http://en.wikipedia.org/wiki/HTC_Dream (accessed 8 November 2013).

Jang, B. and Lim, S-H. (2012) 'Storage subsystem implementation for mobile embedded devices', in Park, J.H., Jeong, Y-S., Park, S.O. and Chen, H-C. (Eds.): *Embedded and Multimedia Computing Technology and Service, Lecture Notes in Electrical Engineering*, Vol. 181, pp.197–204, Springer, Netherlands.

Jeong, S., Lee, K., Hwang, J., Lee, S. and Won, Y. (2013a) 'AndroStep: Android storage performance analysis tool', in *ME13: In Proceedings of the First European Workshop on Mobile Engineering, Lecture Notes in Electrical Engineering*, Aachen, Germany, 66 February, Vol. 215.

Jeong, S., Lee, K., Lee, S., Son, S. and Won, Y. (2013b) 'I/O stack optimization for smartphones', in *Proc. of the USENIX Annual Technical Conference*.

Jo, H., Kang, J.U., Park, S.Y., Kim, J.S. and Lee, J. (2006) 'FAB: flash-aware buffer management policy for portable media players', *IEEE Trans. on Consumer Electronics*, Vol. 52, No. 2, pp.485–493.

Jung, H., Shim, H., Park, S., Kang, S. and Cha, J. (2008) 'LRU-WSR: integration of LRU and writes sequence reordering for flash memory', *IEEE Transactions on Consumer Electronics*, Vol. 54, No. 3, pp.1215–1223.

Kim, H., Agrawal, N. and Ungureanu, C. (2012a) 'Revisiting storage for smartphones', in *Proc. of the 10th USENIX Conference on File and Storage Technologies*, San Jose, CA, USA, February.

Kim, H., Ryu, M. and Ramachandran, U. (2012b) 'What is a good buffer cache replacement scheme for mobile flash storage?', in *Proceedings of the 12th ACM SIGMETRICS/Performance joint international conference on Measurement and Modeling of Computer Systems*, ACM, pp.235–246.

Kim, J., Oh, Y., Kim, E., Choi, J., Lee, D. and Noh, S.H. (2009) 'Disk schedulers for solid state drives', in *EMSOFT 2009: 7th ACM Conf. on Embedded Software*, pp.295–304.

Konishi, R., Amagai, Y., Sato, K., Hifumi, H., Kihara, S. and Moriai, S. (2006) 'The Linux implementation of a log-structured file system', *SIGOPS Oper. Syst. Rev*., July, Vol. 40, No. 3, pp.102–107.

Lee, K. and Won, Y. (2012) 'Smart layers and dumb result: IO characterization of an android-based smartphone', in *EMSOFT 2012: In Proc. of International Conference on Embedded Software*, Tampere, Finland, 7–12 October.

Lee, S., Shin, S., Kim, Y-J. and Kim, J. (2008) 'LAST: locality-aware sector translation for NAND flash memory-based storage systems', *SIGOPS Oper. Syst. Rev*., Vol. 42, No. 6, pp.36–42.

Lee, S.W., Park, D.J., Chung, T.S., Lee, D.H., Park, S.W. and Songe, H.J. (2005) 'FAST: a log-buffer based ftl scheme with fully associative sector translation', *The UKC*, August.

Lim, S-H., Lee, S. and Ahn, W-H. (2013) 'Applications IO profiling and analysis for smart devices', *Journal of Systems Architecture*, October, Vol. 59, No. 9, pp.740–747.

Lv, Y., Cui, B., He, B. and Chen, X. (2011) 'Operation-aware buffer management in flash-based systems', in *Proceedings of the 2011 International Conference on Management of Data*, ACM, New York, NY, USA, pp.13–24.

Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A. and Vivier, L. (2007) 'The new ext4 filesystem: current status and future plans', in *Proc. of the Linux Symposium*, Ottawa.

Microsoft Corporation (2000) *FAT32 File System Specification* [online] http://microsoft.com/whdc/system/platform/firmware/fatgen.mspx (accessed 8 November 2013).

Nexus One (Google/HTC) http://en.wikipedia.org/wiki/Nexus_One (accessed 8 November 2013).

Osborne, R., Van Zoest, A., Robinson, A., Fudge, B., Srinivasan, M., Fry, K. et al. (2010) *Media Transfer Protocol*, 16 February, US Patent 7,664,872.

Park, S., Jung, D., Kang, J., Kim, J. and Lee, J. (2006) 'CFLRU: a replacement algorithm for flash memory', in *Proc. of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ACM, New York, NY, USA, pp.234–241.

Rajgarhia, A. and Gehani, A. (2010) 'Performance and extension of user space file systems', in *Proceedings of the 2010 ACM Symposium on Applied Computing, Sierre, Switzerland, SAC '10*, ACM, New York, NY, USA, pp.206–213.

Rodeh, O., Bacik, J. and Mason, C. (2012) 'BTRFS: the Linux B-tree filesystem', IBM Research Report, July.

Samsung Galaxy S [online] http://www.samsung.com/global/microsite/galaxys/index_2.html (accessed 8 November 2013).

Samsung Galaxy S3 [online] http://www.samsung.com/ae/microsite/galaxys3/en/index.html (accessed 8 November 2013).

Shen, K. and Park, S. (2013) 'FlashFQ: a fair queueing I/O scheduler for flash-based SSDs', in *Proc. of the USENIX Annual Technical Conference*.

Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M. and Peck, G. (1996) 'Scalability in the XFS file system', in *Proc. of the USENIX Annual Technical Conference*, USENIX Association, Berkeley, CA, USA, p.1.

Wang, H., Huang, P., He, S., Zhou, K. and Li, C. (2013) 'A novel I/O scheduler for SSD with improved performance and lifetime', in *Mass Storage Systems and Technologies (MSST)*.