Optimizing Storage Performance of Android Smartphone

Hyukjoong Kim,
College of Information & Communication Engineering
Sungkyunkwan University, Korea
+82-10-9489-8974
wangmir@skku.edu

Dongkun Shin,
College of Information & Communication Engineering
Sungkyunkwan University, Korea
+82-10-6235-3641
dongkun@skku.edu

ABSTRACT

Recently, mobile platform devices such as smartphone and tablet have spread widely. These devices have embedded NAND flash storage devices. For example, recent smartphones use embedded multimedia cards (eMMC) to store application and data. The performance of smart devices is strongly related with the embedded storage. Recent products of eMMC provide several special features for higher performance. In this paper, we investigated the performance-related features of eMMC device at Android-based smartphone. First, we study the effect of *packed command* which is introduced at eMMC 4.5 specification. Second, we examine the performance degradation by *Least Significant Bit (LSB) backup* on MLC eMMC devices. Finally, we observed the performance difference under different ext4 file system configurations such as flex group. From experiments, we found that the storage subsystem of current Android platform needs further optimizations considering the special features of eMMC.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Secondary storage, Storage hierarchies

General Terms

Measurement, Performance

Keywords

eMMC, Android, Smartphone, Storage, IO system, Ext4, NAND Flash, MLC

1. INTRODUCTION

Mobile platform devices such as smartphones and tablets have recently become the dominant personnel computing devices. Generally, these platforms adopt NAND flash storage devices. For example, Google's Android-based smartphones use eMMC (embedded Multi-Media Card) to store user's application and data as well as the platform software. As the need for higher performance at these mobile platform devices, the storage IO performance cannot be overlooked as examined at a previous study [18].

The architecture of NAND flash-based storage has been moved from pure NAND flash memory to *fusion NAND* flash device. The pure NAND requires special software, called flash translation layer (FTL), to handle all idiosyncrasies of flash memory such as address translation, bad block management, error correction, etc. As shown in Figure 1, the pure NAND flash memory just handles the read or write requests sent form host system while all higher-level operations must be performed by host system.

However, the fusion device such as eMMC has a micro-controller and RAM that allow it to operate FTL internally. Therefore, the host system can consider the fusion device as a traditional block device like hard disk drives, and we can use legacy file systems and IO subsystems without modifications. The weak point of eMMC is that eMMC cannot optimize its performance exploiting the host information. For example, the software module to handle pure NAND can exploit several semantic information of host operating system since it runs at the host system. However, eMMC communicates with host through a standard MMC bus which does not transfer any additional host information except the block request information. To overcome such a limitation, recent eMMC standard specification includes several extended interfaces through which host can transfer information to eMMC device [2].

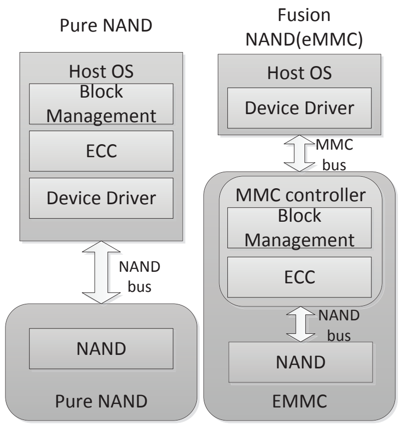

Figure 1. Difference between eMMC & Pure NAND.

In this paper, we study the extended features of eMMC 4.5, examine whether current Android smartphones exploit these features efficiently, and propose the related techniques to optimize the storage performance. Firstly, we investigate the internal features of eMMC such as *Least Significant Bit (LSB) backup* issue on Multi-Level Cell (MLC) based NAND flash. Because of MLC's characteristics, if power failure is occurred during on write operation, *paired page* that was already written will also be corrupted. To handle this problem, eMMC back up written paired page to Single-Level Cell (SLC) buffer. This behavior can produce additional write, therefore we evaluate how LSB backup can affect to the performance. Secondly, we observe *packed command* that is the function of eMMC 4.5 standard. The role of packed command is ‘packing’ several IO requests at device-driver level. It is similar to Native Command queuing (NCQ) [3] using on hard disk or SSD. In our works, we evaluate the usage and impact of packed command on Android device, and find optimization points. As third, we study about *Flex group* on Ext4 file system [8]. Flex group is introduced to unify a bunch of block groups in order to handle several block group’s metadata into single management. We evaluate how this feature can affect to the performance of eMMC. Additionally, Using on the overall observations, we make *Driver-level Block IO Tracer* as Linux kernel patch for eMMC to look up deeper level of eMMC’s IO behavior.

The rest of this paper is organized as follows. In Section 2, we describe necessary background about MLC NAND, eMMC Standard, Ext4 file system. The observations on relationship between Android and eMMC are presented in Section 3. The conclusion and future works are described in Section 4.

2. BACKGROUND

2.1 MLC Feature

NAND Flash memory can be separated to two types, SLC and MLC. SLC stores 1-bit per cell. But in case of MLC, 2 or 3 bits are stored in one cell. From this architectural difference, MLC has higher capacity and cheaper price but worse lifespan compared to SLC.

In case of 2-bit MLC, one page is related to another page called *paired page*. The problem is that when power failure is occurred during programming MSB, page corruption is appeared not only MSB page but also LSB page that was already written. It is heavy danger on reliability, therefore LSB page should be backed up to prepared SLC buffer at MSB page programming. This behavior is called as LSB backup.

2.1.1 LSB Backup

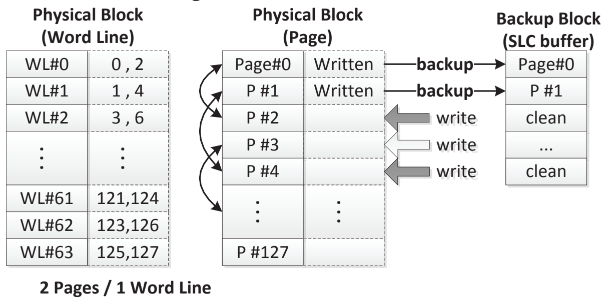

Figure 2. Paired page on word line and the behavioral description of LSB backup.

Figure 2 describes the behavior of LSB backup. In case of 2-bit MLC, 2 pages are stored in one word line, and thus, single block that has 64 word lines can have 128 pages. Paired pages are formed as physical block’s figure with word line describes. Word line #0 has page #0 and #2, and word line #1 has page #1 and #4 as paired pages. If additional write operation is arrived when page #0 and #1 is already written, like as Figure 2 shows, LSB backup should be performed to SLC buffer. Writing page #2 will go with backing up page #0 to SLC buffer and writing page #4 will be with page #1’s backup. However, writing page #3 does not need to back up because it is LSB by itself.

With LSB backup, at most, single write operation needs 2 actual page writes. And also, when the situation that several banks perform interleaving [9], its harm can be more significant problem when interleaving is paused because single SLC buffer will be saturated with backup actions on many MLC banks. Figure 3 describes the behavior of interleaved write and non-interleaved write because of LSB backup. 4 banks are paired to interleave and 8 pages perform data program. Data processing (PC) integrates several jobs to prepare data program like commands, data transferring, etc. and these jobs cannot be overlapped. Figure shows that first 4 pages are interleaved. Although unit program time of single page is (PC + PG), programming 4 pages take only PC + PG × 4 by interleaving. However, because second 4 pages require LSB backup, and LSB backup cannot be overlapped, the latency of second 4 pages are much longer than first.

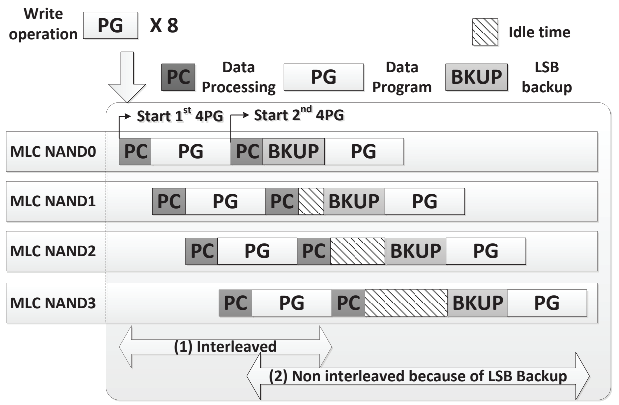
Write operation
PG X 8
Idle time
PC Data Processing
PG Data Program
BKUP LSB backup
Start 1st 4PG
Start 2nd 4PG
MLC NAND0 PC PG PC BKUP PG
MLC NAND1 PC PG PC BKUP PG
MLC NAND2 PC PG PC BKUP PG
MLC NAND3 PC PG PC BKUP PG
(1) Interleaved
(2) Non interleaved because of LSB Backup

Figure 3. Behavior of Interleaved write & Non-interleaved write because of LSB backup. Total 8 programs, first 4 programs are interleaved; others are not able to be interleaved because LSB backup cannot be overlapped.

Because word line is tied with two pages that is 3 steps far from each other, write operation more than 4 pages can reduce LSB backup overhead gradually because paired pages are programmed at the same time. On the situation that several banks are interleaved, that threshold will be 4 super pages rather than 4 pages. Accordingly, enlarging IO size is important thing to improve performance.

2.2 eMMC Standard

eMMC is, as mentioned above, the device that combines NAND flash memory and controller. eMMC should be based on JEDEC’s eMMC Standard, and can use standard interface and MMC driver. The newest version is eMMC 4.5, and we review key features like *Context ID, Packed command, Trim and Discard.*

2.2.1 Context ID

Context ID is formed with 15 IDs that can be sent with IO operation. Through these IDs, OS-level data information is able to go to storage. This information is used to determine data’s tendency, random or sequential, small or large, document or media, etc. And then storage device can handle data area more efficiently.

Context ID is already supported, but not used yet. It has variety of potential application, and should be investigated more.

2.2.2 Packed command

Packed command is the function that can ‘pack’ plural IO requests. With this function, IO operation on OS acts like NCQ thus large sequential writes can have benefits of interleaving. But in case of small random writes, packed command is not used because they have no profits on interleaving [1]. In fact, small random write also can enjoy the advantage on packed command a little bit.

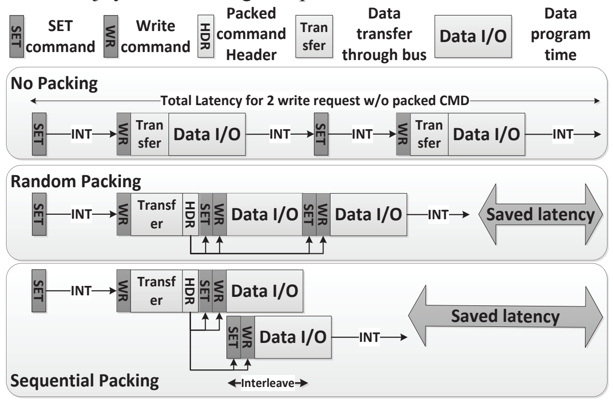

Figure 4. Packed command behavior on 2 write-operations. Sequential write packing enable interleaving, random write packing has advantage also by reducing interrupt.

As Figure 4 shown, the fundamental activities of write operation are performed as follow, sending set command, receiving interrupt, sending write command and data, and receiving another interrupt. In this sequence, write operation should receive two interrupts per single write. However, if using packed command, set commands and write commands of packed IO requests are all stored in packed header and additional interrupts are not required. Therefore, two interrupts per single write can be reduced into two interrupts per single packed request. Previous research presented that interrupt overhead will be enlarged because improvement of IO performance is too rapid [10]. In the result, saving a number of interrupts is not ignorable and will more important on future.

2.2.3 *Trim & Discard*

On the situation of previous storage IO system based on HDD, file system had no responsibility to inform data deletion to storage because HDD can physically overwrite data. However, in case of NAND-based storage, because storage cannot overwrite data into same physical area, information of data deletion is important things. If data deletion is not informed, the storage device may consider deleted data as valid data, thus useless copies are created. Trim command is employed by Solid State Disk (SSD) because of this phenomenon [21] and eMMC also adopts trim command as standard. Trim command informs data deletion to storage device, and then storage device invalidates that and then does not copy useless data.

Although trim command is useful function, it has a limitation. Trim command has a responsibility to return null value when host OS sends read operations on already ‘trimmed’ address. This limitation makes storage handle an additional data structure to manage ‘trimmed’ data, therefore it enlarges IO latency. In order to handle this problem, discard command is suggested. Discard does not require such null value at read operation, accordingly we can use discard operation with no additional management when reliability problem is not presented.

2.3 Ext4 File System

Ext4 file system is what Android platform uses for main partition’s file system. In this paper, we focus on *Flex group* that is a block group management technique on Ext4 file system.

Figure 5 shows the architecture of block group with flex group. Flex group is the unit that handles one or many block groups, and at formatting partition, tuning the number of block groups on single flex group is available. When block groups per flex group is more than one block group, all block groups metadata are stored in flex group’s first block group thus metadata are stored on limited logical area. This phenomenon means spatial locality is considered on storing metadata. And this also means IO performance can be improved depending on storage architecture.

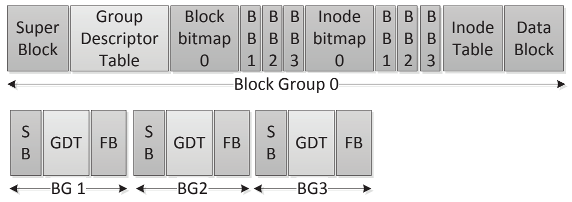

Figure 5. Block group architecture with Flex group on Ext4 file system: Block bitmaps, inode bitmaps and inode tables all are in block group 0.

Previous research emphasized that the increases of the number of block groups in flex group improved IO performance on the workload that has large amount of transactions [11]. However this result can be unlikely according to Flash Translation Layer (FTL) on eMMC. eMMC that uses hybrid mapping FTL like FAST [17] is able to achieve reasonable improvement by data workload considering spatial locality [12], but in case of page mapping FTL like DFTL [16], there are few advantages because of out-of-place management on storage area.

3. OBSERVATION

Observations are three contents. Firstly we investigate eMMC structure and LSB backup on MLC NAND, and secondly we study about effects and optimization points of packed command, lastly, we evaluate the effect of flex group depending on devices.

3.1 Tools & Experimental Setups

3.1.1 *Drivel-level Block IO Tracer*

Blktrace [13] is widely used tracing tool to observe IO behaviors of block devices. However, because blktrace is located on IO scheduler, it cannot observe lower driver’s behavior. For example, packed command that is performed on block interface driver of eMMC, separating set command overhead from write or read operation on host driver are not able to be observed from blktrace. And also, blktrace don’t report about trim command. From this reason, we develop Driver-level Block IO Tracer. Figure 6 describes the observing points and observing behaviors of blktrace and Driver-level Block IO Tracer. Blktrace traces IO scheduler, especially request queue’s behavior, on the other hand, Driver-level Block IO tracer traces device driver level command handling.

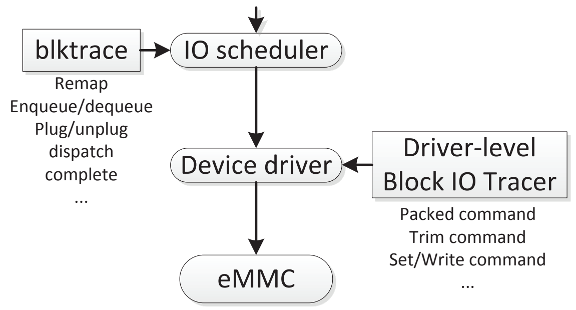

Figure 6. Differences between blktrace & Driver-level Block IO tracer.

Driver-level Block IO Tracer is located on eMMC 4.5 standard driver, and it observes write/read, trim/discard, packed command and their corresponding interrupt behavior. Only problem is tracing latency that takes 50~100 micro seconds per IO. This latency cannot be ignored at small random IO, Therefore we avoid using this tool in case of small random IO operation.

3.1.2 Experimental Setups

We use GT-I9100, GT-I9300 and Pandaboard [6] as target devices. First two devices are commercial Android smartphone on Samsung, and the latter is android development board. Target devices have 16GB eMMC, especially GT-I9300 and Pandaboard use same model of eMMC. Pandaboard is alternative device to GT-I9300 because GT-I9300 is unable to re-partition storage. Therefore every option is equivalent to GT-I9300.

Tiobench [24], uFLIP [5] and postmark are used for benchmark workload. Tiobench creates threaded IO workload, depending on option, it can simply generate random and sequential I/O workload. uFLIP directly sends I/O operation into bio structure rather than through file system, Thus it is useful to evaluate raw device performance. Postmark is mail server benchmark. It creates certain number of working files and transactions (create, delete, append, and read). Real workloads are composed with local synchronization on Google Drive and application install jobs.

In order to trace the files that are observed by Driver-level Block IO Tracer, we use Android Block IO Semantic Analyzer [22]. This tool translates block I/O operation into corresponding file, consequently, we can re-match I/O pattern to file system level.

3.2 Internal features & Effects

In this section, we investigate internal features and effects of LSB backup on eMMC. GT-I9300 and uFLIP are used for this experimentation. Based on previous work that estimates internal architecture of NAND-based storage with uFLIP [23], we find size of page and super page, and then evaluate the effects of LSB backup. We perform write operations with various IO size and aligned to logical address of eMMC device.

Figure 7 describes the bandwidth of write operation with IO size from 512byte to 8MB. The graph is separated into three parts, and the first part named ‘Under page size’ appears much lower performance compared to second part and third part that are ‘Under super page size’ and ‘Interleaved’. This phenomenon shows that page size is 8KB. The reason is as follows. Because the minimum write unit of NAND-based storage is page size, although write operation is performed with IO size under page size, it takes same latency to write with page size. It means ‘under page size’ IO brings low bandwidth compared to ‘above page size’ IO. On the other hand, at the second region, because its IO size is larger than page size, the bandwidth should be improved gradually depending on IO size due to increasing of interleaving writes. But as shown in figure, the performance is not that improved until IO size is larger than 128KB. As explained on Section 2.1.1, we can assume that this is because of LSB backup. In case of under super page size, every write operation is single page operation at the aspect of each bank, therefore LSB backup is always presented when MSB pages are writing. Therefore, we also assume that the super page size is 128KB based on figure. Above 128KB, the performance keeps increasing. This is due to interleaving and decreasing of the number of LSB backup because LSB and MSB pages are written at the same time. However, Android, and its Linux kernel perform write operation with maximum size of 512KB due to IO scheduler policy. Thus, packed command is important function to ‘pack’ up diffuse write operation into large size IO at ‘device driver’.

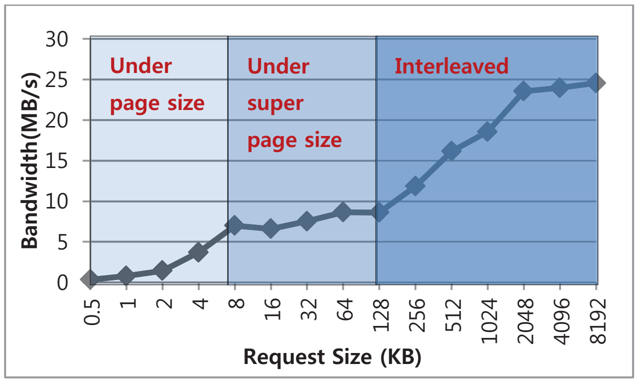

Figure 7. Write bandwidth depending on IO size (uFLIP, aligned random IO).

3.3 Effects of Packed Command

On this experimentation, we use GT-I9300 as target device and tiobench as test benchmark. In case of packed command on sequential write, tiobench writes 2GB with 8MB block units and on single thread (*tio_seq*). On the other hand, experimentation of packed command on random write, tiobench writes 2GB write with 4KB block units on 4 threads (*tio_rand*). And Google Drive and application install workload are used for real workload.

3.3.1 Packed command on Sequential Write

Packed command can pack 62 IO requests per single packed command according to standard, however, another limitation is what each eMMC device has own limitation of write size. In case of our target device, the limitation on single write size is 4MB. It means the maximum number of IO request per single packed request is 8 due to the maximum size of IO at IO scheduler, 512KB. At this point, default option only allows packing sequential write with no limitation on the number of packed IO request. Packed request supposes a unified IO request that covers packed IO requests and packed IO requests means internal IO requests on packed request.

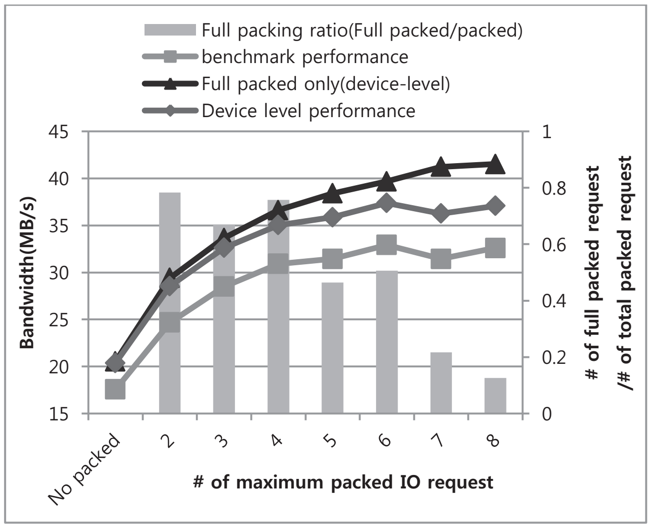

Figure 8. Bandwidth & full packing ratio depending on # of maximum packed IO request (enable packed command on sequential write only, *tio_seq*).

Figure 8 shows bandwidth and full packing ratio by limiting the number of maximum packed IO request. Full packing ratio means the portion of packed request that contains maximum IO request compared to total packed request. Workload on this experimentation is *tio_seq*, and benchmark performance is reported bandwidth from tiobench. Device level performance is measured from Device-level Block IO Tracer and full packed only bandwidth stands for the performance of full packed request only. It is also measured from Device-level Block IO Tracer. The gap between benchmark performance and device level performance can be treated as OS latency.

The bandwidth on benchmark performance is saturated since the number of packed IO request is 4 on the Figure 8. However, full packed only bandwidth, the pure performance of packing is still increasing and full packing ratio is decreasing. This result implies that packed command is not fully utilized because IO requests are not fully packed.

Table 1. Reason of halt on packing sequential write, total packed request: 267 (default option, *tio_seq*)

NOREQ	RAND	SYNC	REL	READ	Fully packed
203	58	2	0	0	4

In order to investigate the reason of disutility, we trace 'the halt of packing'. Table 1 shows the reason of halt on packing sequential write. The number of halt from empty request queue (NOREQ), random write (RAND) is 203 and 58 respectively, and from the sync write (SYNC) is 2. The effect of sync write and reliable write (REL) and read operation (READ) are negligible. Focusing on halt of random write, we use Android Block IO Semantic Analyzer to find corresponding file of random write. Figure 9 shows that the identity of halting random write is all file system meta data. This result can be flexible depending on workload pattern, but implies that interference of file system's meta data is fundamental material of halting random write.

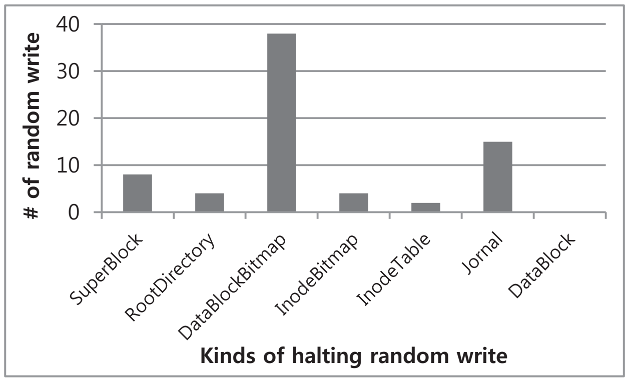

Figure 9. Disturbance of random write on packing sequential write, all of random writes are file system's meta data (default option, *tio_seq*).

Figure 10 represents the bandwidth on different IO schedulers, and the environment of experimentation is identical to the former. With default option, as shown in figure, cfq is the lowest, but deadline and noop scheduler appear similar performance. Figure 11 describes the number of packed IO request according to IO scheduler. On the figure, the number of packed IO request is much lower on cfq rather than deadline or noop. This result can be a reason of lower performance on cfq at Figure 10. And the reason of low number of packed IO request on cfq is proved on Table 2. Table 2 describes similar material to Table 1 depending on IO scheduler and shows that cfq has drastically large number of halt on packing because of empty request queue compared to others. It represents that cfq is much slower than deadline or noop on the aspect of queuing thus it harms application of packed command.

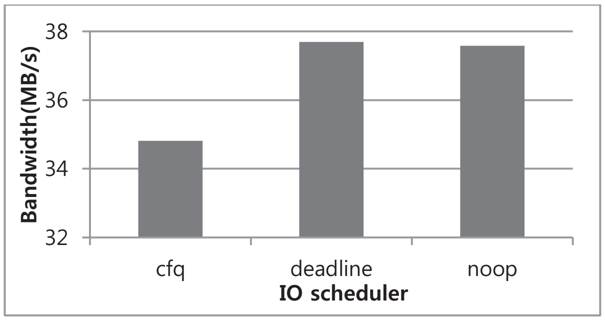

Figure 10. Bandwidth according to IO scheduler (default option, *tio_seq*).

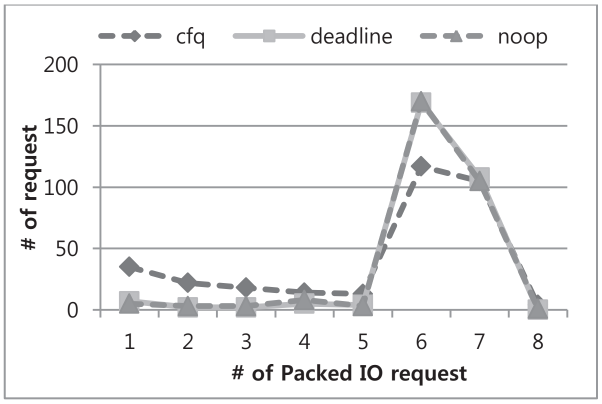

Figure 11. The number of packed IO request depending on IO scheduler (default option, *tio_seq*).

Table 2. Reason for halt on packing sequential write depending on IO scheduler (default option, *tio_seq*).

	cfq	deadline	noop
NOREQ	**203**	**7**	**19**
RAND	58	48	77
SYNC	2	3	0

From the observations, packed command on sequential write derives performance benefit but still needs optimization. For example, interference of random write on packing sequential write should be fixed by write operation reordering, and a compromise between cfq and deadline or noop is needed because still cfq scheduler is good scheduler to provide fair storage performance on burst jobs.

3.3.2 Packed Command on Random Write

On the current state, packed command is not used on random write. However packed command can solve the problem of interrupt latency on small random write as explained at Section 2.2.2. To prove this effect, we have experimentation about small random write with enabling packed command on random write. As mentioned above, because of revealing overhead, Driver-level Block IO Tracer is not used in this section.

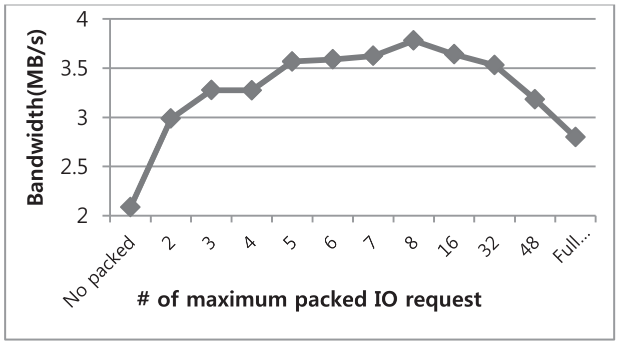

Figure 12. Bandwidth on packing random write depending on the number of maximum packed IO request (enable packed command on random write, *tio_rand*).

Figure 12 is bandwidth graph on packing random write according to the number of maximum packed IO request on small random write workload using tiobench. In case of small random write, the maximum number of packed IO request is 62 because the size limitation is meaningless. However, as shown in figure, limiting maximum number of packed IO request into 8 is much better than packing IO request with no limitation. Increase of bandwidth, until the number of maximum packed IO request is 8, can be considered as the reduction of interrupt latency. But in case of shrink in latter, its reason is unclear. Only reasons that can be presumed are packing overhead because of de-queuing and re-queuing IO requests, and eMMC device level overhead that can be larger when packed IO request is burst random writes. Because of unavailable tracer and black boxed internal details on eMMC device, it remains as future work.

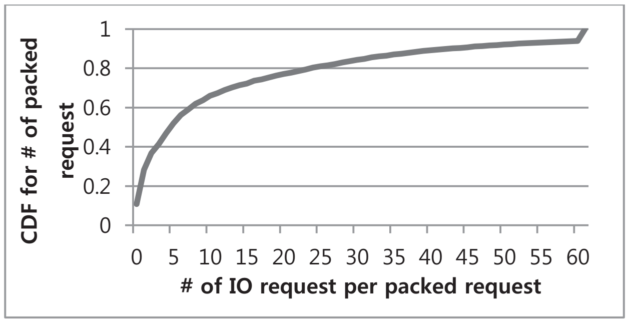

Figure 13. CDF graph on the number of IO request per packed request at packing random write (Full packed (62), *tio_rand*).

We measure actual number of packed IO request on small random write at no limitation for the number of packed IO request. Because this measurement doesn't care about latency, Driver-level Block IO Tracer is adopted. Figure 13 shows that actual number of packed IO requests is significantly different from maximum number of packed IO request, 62, and about 60% of packed requests are packing IO request under 8. According to the result of Figure 12 and Figure 13, we can at least conclude that limiting the maximum number of packed IO request is effective to achieve better performance on packing random write.

3.3.3 Usages of Packed command on Real Workload

In this section, we study about usages of packed command on real workload. We use Google drive and application installation workload. Google drive workload is made of synchronization to local storage of smartphone using Google drive and the synchronized data is music file and documentation, and application installation workload performs verbatim application installation that contains many large size applications. Table 3 directly shows that AppInstall, that has large size IO and utilizes packed command more, performs better bandwidth compared to GoogleDrive workload.

Table 3. Usage of packed command on real workload(default option, GoogleDrive, AppInstall).

	GoogleDrive	**AppInstall**
Write size(GB)	0.7	3.1
# of request	9912	10126
Bandwidth(MB/s)	24.146	36.179
Packed write/Total write	69.66%	94.47%

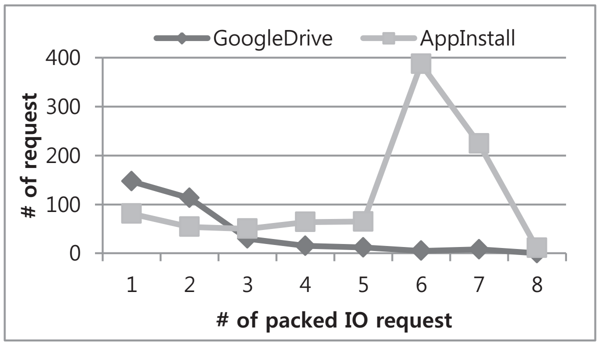

Figure 14. The number of packed IO request on real workload (GoogleDrive & AppInstall).

Figure 14 shows the number of packed IO request on real workload. In case of AppInstall, most of packed request contains 6 or 7 IO requests per single packed request. But GoogleDrive workload is not packed much compared to AppInstall. This result is reasonable based on workload pattern and shows that the more IO requests are packed, the better performance become.

3.4 Effects of Flex Group

In this section, we investigate the effect of flex group of ext4 file system that storing metadata with considering spatial locality. Evaluation is performed using GT-I9100 and Pandaboard as target devices, and used eMMC device in Pandaboard is same to GT-I9300. We use postmark as benchmark workload, and perform 10000 transactions within 5000 files, and each file size is 1MB. Experimentation is performed with increasing the number of block group per flex group.

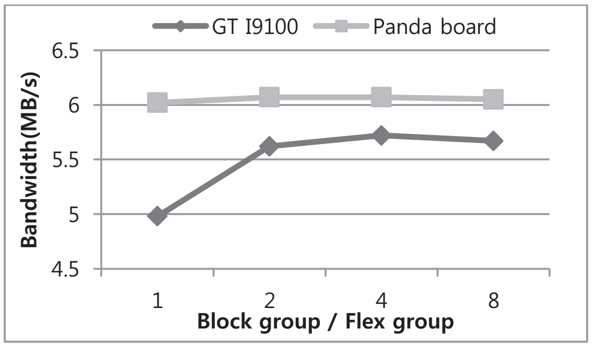

Figure 15. Bandwidth according to block group per flex group on GT-I9100 and Panda board (Postmark).

As Figure 15 shown, the performance of GT-I9100 is increased between 1 and 2 block groups per flex group. It is because if plural block groups are presented in single flex group, the meta data are stored into first block group on flex group, thus spatial locality of IO operation is improved. But in case of Pandaboard, improvement is not appeared. This is different result compared to previous research. As mentioned on Section 2.3, depending on FTL, considering spatial locality can be useless effort. Thus, we can assume that eMMC device in Pandaboard, and GT-I9300 uses page mapping FTL.

4. CONCLUSION & FUTURE WORKS

In this paper, we study about the relationship between Android and eMMC. Firstly, we investigate internal features of eMMC and the effects of LSB backup, and conclude that large size write operation can reduce IO latency, thus, IO pattern and application of packed command are important. Secondly, we evaluate the effects of packed command and find optimization points. We found that packing on sequential write can improve performance on large size IO, but still has optimization points, and packing on random write also can improve performance due to reducing interrupt latency. On the third, we re-evaluate the effect of flex group, but it is useless on new, high-end eMMC.

For the future works, we ultimately aim to optimize overall IO system of Android. Based on this paper, we'll optimize IO scheduler to pack more IO requests, and reduce interrupt latency on small random write pattern.

5. REFERENCES

[1] Samsung GT-I9300 open source kernel, http://opensource.samsung.com

[2] eMMC 4.5 specification, http://www.jedec.org

[3] Serial ATA revision 2.6, http://www.sata-io.org

[4] Google Drive, http://drive.google.com/

[5] uFLIP, http://uflip.inria.fr/~uFLIP/

[6] Panda board, http://pandaboard.org/

[7] Debugfs, http://linux.die.net/man/8/debugfs/

[8] Ext4 filesystem, http://kernel.org/doc/Documentation/filesystems/ext4.txt

[9] Feng Chen, Rubao Lee and Xiaodong Zhang, "Essential Roles of Exploiting Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing", HPCA'11, February 2011

[10] Jisoo Yang, Dave B. Minturn and Frank Hady, "When Poll is Better than Interrupt", FAST'12, February 2012

[11] Hyeong-Jun Kim and Jin-Soo Kim, "Tuning the Ext4 Filesystem Performance for Android-Based Smartphones", ICFCE 2 011, December 2011

[12] Sungjin Lee, Dongkun Shin, Young-Jin Kim, Jihong Kim, "LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems", SPEED2008, February 2008

[13] Jens Axboe and Alan D. Brunelle, "blktrace User Guide", February 2007

[14] Jens Axboe, "CFQ IO Scheduler", linux. conf. au, January 2007

[15] Marcus Dunn and A. L. Narasimha Reddy, "A New I/O Scheduler for Solid State Devices", Department of Electrical and Computer Engineering Texas A&M University, TR. TAMU-ECE-2009-02-3, 2009.

[16] Aayush Gupta, Youngjae Kim and Bhuvan Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings", ASPLOS'09, March 2009

[17] Sang-won Lee, Dong-joo Park, Tae-sun Chung, Dong-ho Lee, Sangwon Park and Ha-joo Song, "A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation", ACM Transactions on Embedded Computing Systems, Vol. 6, No. 3, July 2007

[18] Hyojun Kim, Nitin Agrawal, Cristion Ungureanu, "Revisiting Storage for Smartphones", Usenix FAST'12, February 2012

[19] Ki Yong Lee, Hyojun Kim, Kyoung-Gu Woo, Yon Dohn Chung, Myoung Ho Kim, "Design and implementation of MLC NAND flash-based DBMS for mobile devices", The Journal of Systems and Software, March 2009

[20] Jae-Sung Yu, Jin-Hyeok Choi, "Memory Systems Having a Multilevel Cell Flash Memory and Programming Methods Thereof", U.S. Patent 11/796,978, Ju. 24, 2008

[21] Tasha Frankie, Gordon Hughes, Ken Kreutz-delgado, "SSD Trim Commands Considerably Improve Overprovisioning", Flash Memory Summit 2011, August 2011

[22] Sungkyunkwan University Embedded Software Laboratory TR-10, "Replicant: Semantic Analyzer for Understanding the Storage I/O Behavior of Android Smartphone"

[23] Byeonggyu Park, Dongkun Shin, "Probing Internal Architecture of NAND Flash Storage Device", ITC-CSCC2011, June 2011

[24] Mika Kuoppala, "Tiobench – Threaded I/O bench for Linux", 2002