Cross-layered View on Android Storage IO System

Relationship between Android and eMMC

Hyukjoong Kim
College of Information & Communication Engineering
Sungkyunkwan University
Suwon, Korea
wangmir@skku.edu

Dongkun Shin
College of Information & Communication Engineering
Sungkyunkwan University
Suwon, Korea
dongkun@skku.edu

***Abstract*— Recently, Smart devices and its functions and applications have flourished. And this development has brought one important requirement, the performance. Storage IO performance is especially important because applications and data are all stored in storage. Smart devices often use NAND-based storage because of its high performance and small size. However, storage IO subsystem on operating system is still not optimized to NAND-based storage. And more, from high performance on storage device itself, management overhead in operating system is revealed. On the other hand, Android, one of most popular platform for smart devices, uses eMMC as main storage and eMMC has several features that can achieve much higher performance than before. In this paper, we measure the overhead of each storage IO layer on Linux, the kernel of Android. And also we study about extended features of eMMC.**

***Index Terms*— Smartphone, Android, Storage, NAND flash, eMMC, IO system, file system**

I. Introduction

Android is one of the most popular smart-device platforms. An increase of uses and interests on Android and its applications makes people require more improved performance on smart-devices. Especially, storage IO performance is very important factor on end-user's performance because every data and applications is stored on storage. Previous work also verified that storage IO performance is negligible compared to many other performance issues[8].

Smart device, especially Android device often adopts NAND Flash based storage as main storage because of its advantages, high performance, small size, low heat, silence and etc. compared to HDD. However, storage IO system in Linux, low-level kernel of Android, is still optimized based on HDD, thus it cannot fully utilize high performance of NAND-based storage. And more, because of high performance of NAND-based storage, the overhead from storage IO subsystem of Linux kernel, is revealed. This overhead was not a problem with HDD because it can be covered by much long latency of HDD. Previous research also indicated this problem, and predicted that this overhead will be bigger as soon because of improvement of NAND performance [7].

On the other hand, NAND-based storage that is often used by Android smart-device is embedded Multi-Media Card (eMMC). This is different from pure NAND storage because it has own controller, block management policy, Flash Translation Layer (FTL) and Error Correction Code (ECC). Consequently, eMMC reduces the responsibility of host operating system on handling storage IO operation thus improves IO performance. Figure 1 compares pure NAND and eMMC storage. Although host OS can get out from the overhead of managing NAND storage by using eMMC, however, eMMC has significant weakness. When the host OS manages block management, the data will be stored more flexibly based on what data they are. But in case of eMMC, because the management is separated from host OS, it is impossible to handle data depending on their types. In order to solve this problem, eMMC Standard [1] provides several extended interface for host operating system.

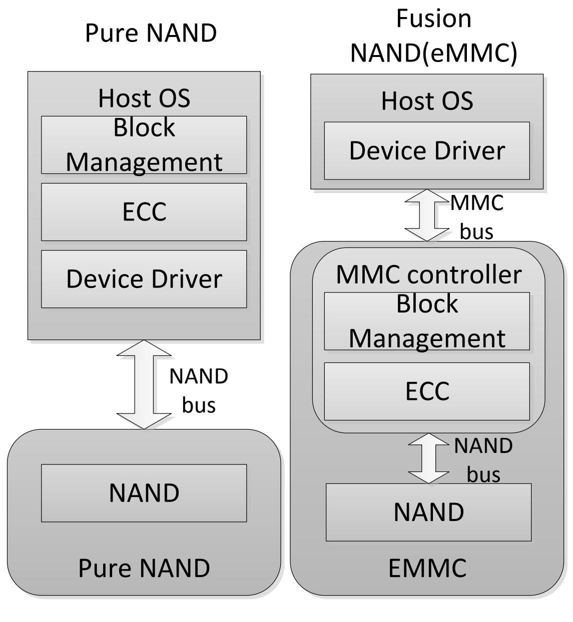

Figure 1 Difference between Pure NAND and eMMC

In this paper, we study about relationship between Android and eMMC storage. To do this, firstly, we investigate the overhead of each IO subsystem layer. We developed synthetic IO android application that can perform

IO operations with various sizes and patterns. And we use ftrace [6] and blktrace [5] to measure exact latency of each layer. In result, the overhead of operating system is not negligible especially on small size random write. Secondly, we study about eMMC Standard interfaces. Especially, we measure effects of packed command. Packed command is a function of eMMC Standard that can 'pack' plural write operations thus improve IO performance. Basically, packed is used only for sequential write operation, we enable to pack random write and measure performance.

II. Cross Layered View On Android

In this section, we measure unit latency of each IO subsystem and evaluate the overhead and bandwidth of each layer. It is proved on experimentation that OS overhead is negligible and the reason of this overhead is additional write operation performed by OS.

Table 1 Specification of target android device

Platform	Android 4.0.4 Ice cream Sandwich
Kernel version	Linux kernel 3.0.15-808555
CPU	Exynos 4412 Quad Core 1.6GHz
Memory	2048MB RAM
Internal storage	eMMC 16GB

A. Experimental Setups and methods

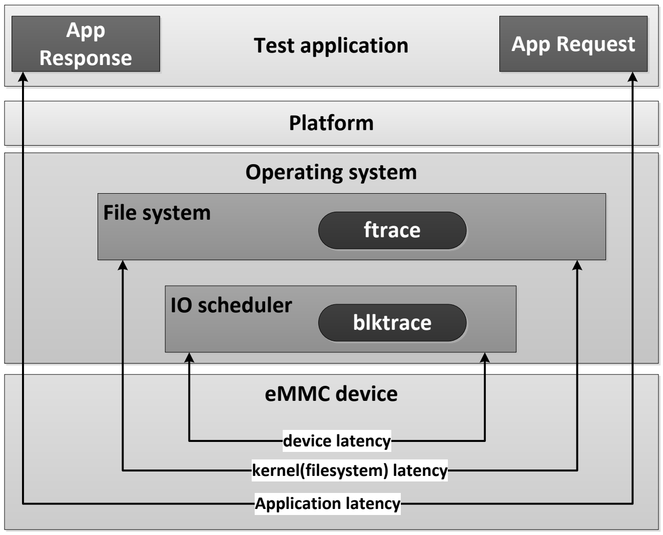

Figure 2 Measurement method, we evaluate the overhead of each layer, Platform, kernel, device.

We use commercial Android smartphone as target device and its specifications are on Table 1. To measure the overhead of each layer, we use synthetic IO benchmark as Android application, ftrace and blktrace. Ftrace is function tracing tool provided by Linux kernel system. It can trace all of kernel functions if target Linux kernel is built with certain configuration. Thus, we trace the functions that start and response latency of IO operation on ext4 file system. Through this, we are able to measure IO latency on the layer of file system. On the other hand, blktrace is block IO operation tracing tool that can investigate behavior of IO scheduler, especially request queue. With use of blktrace, en-queuing, de-queuing, dispatching, completing, etc. are all visible behaviors. We specify dispatching and completion behavior to evaluate device level latency because dispatching is just before device driver and completing is just after device driver. Figure 2 summarizes the measurement method using benchmark application, ftrace and blktrace.

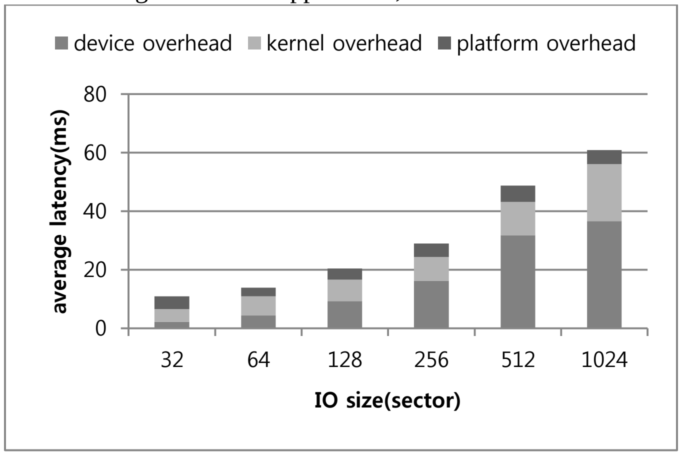

Figure 3 Overhead of each layer on random write operation with various IO size, we can find that OS overhead is more negligible on small random write.

Figure 3 shows overhead of each layer on random write operation with various IO size. This experimentation is performed with Android benchmark application that is our own development. Total write size is 256MB and random write is performed uniformly. From the figure, OS overhead (Platform + Kernel overhead) is negligible portion, especially on small write like 32 sectors, OS overhead is much higher than own device latency.

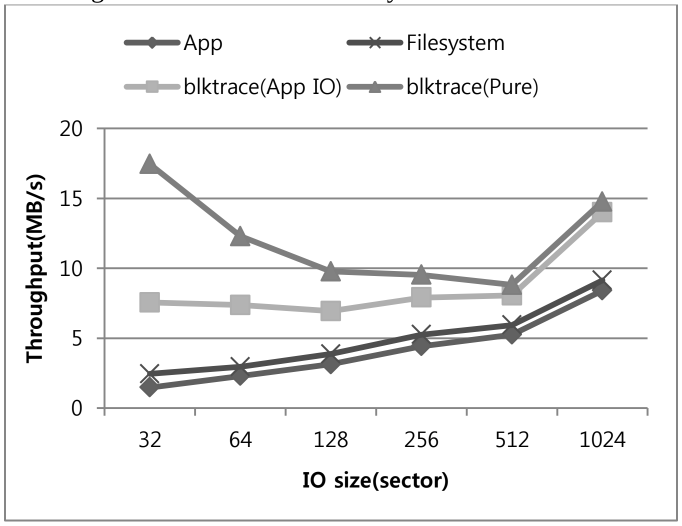

Figure 4 Throughput measured on each layer, all measured latencies are divided by application IO except blktrace (Pure), blktrace (Pure) is pure bandwidth on device. It

means that the gap between blktrace (App IO) and blktrace (Pure) is additional write overhead.

Figure 4 is throughput calculated by response time of each layer on same experimentation to above. All other throughput is the value that is total latency divided the number of IO operations on Application. But in case of blktrace (Pure), it is blktrace's total latency divided by the number of its own IO operations. The remarkable thing is the truth that the bandwidth of blktrace (App IO) is much slower than blktrace (Pure). It implies that OS performs additional write operations and its latency hams user-level, or application-level IO performance.

From the experimentation, we can find the striking existence of 'Operation System Overhead' when storage IO operation is performed through Android application. To solve this problem, we should trace and prove the additional write operation, and if can, reordering this write operation to back to user-level IO operation.

III. STUDY ON EMMC

eMMC standard has several extended interfaces for host OS to communicate with storage device. In this section, we investigate the eMMC functions especially packed command.

A. Study on Standard Device driver

Standard driver, released by eMMC standard, has management code for several extended interface. We review Standard driver code on Open source android kernel [2] to investigate management of extended features.

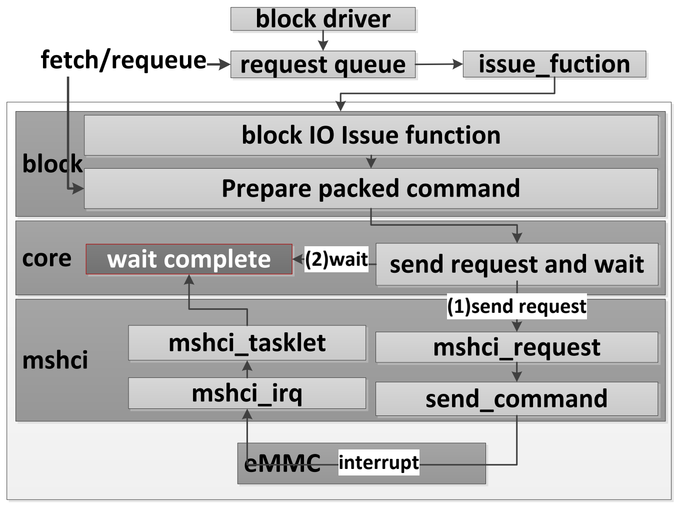

Figure 5 Architecture of MMC Standard Driver, block interface driver, core driver and host (mshci) driver

Figure 5 describes the architecture of MMC Standard driver. It is located under block IO layer and directly operated by issue function of request queue. MMC standard driver is composed with three particular drivers, block interface driver, core driver, and host (mshci) driver. Block interface driver translates block IO request into mmc request, and also, prepares packed command. Core driver performs mmc operations like read, write, discard, high priority interrupt (HPI) and etc., and then waits for rescheduling. Host (mhsci) driver actually operates mmc request by setting command, preparing DMA and etc.

B. Study on Packed Command

Packed command is extended interface of eMMC standard. It can 'pack' plural IO requests into single packed request. Using this feature, scattered sequential write operations are merged into single write operation. It is similar behavior to NCQ [3] that is often used by HDD and Solid State Drive (SSD). Merge a number of small sequential write into large sequential write is important role because large size write operation can be interleaved [9].

In this section, we evaluate packed command's effects on large sequential write and small random write. Packed command is used on sequential write only in current state, but we also measure random write performance and re-evaluate the value of packing random small write. All experimentations are performed using tiobench [4]. Sequential write uses 8MB record size, single thread and total 1GB writes and random write uses 8KB record size, 4 threads and also total 1GB writes.

Figure 6 shows the bandwidth of packing sequential write depending on the number of maximum packed IO request. Benchmark performance is measured at tiobench and blktrace performance is measured by calculating latency trace of blktrace. The maximum number of packed IO request can be changed on MMC standard driver. Because eMMC can write maximum 4MB by single write operation, and IO scheduler splits write request into 512KB, the maximum number of packed IO request is 8. Based on maximum value, we can see that packed command is effective than non-packing any request. And also, increase of maximum value can improve performance until the value is 4, but after that, performance is saturated. The gap between benchmark performance and blktrace performance represents the overhead of operating system.

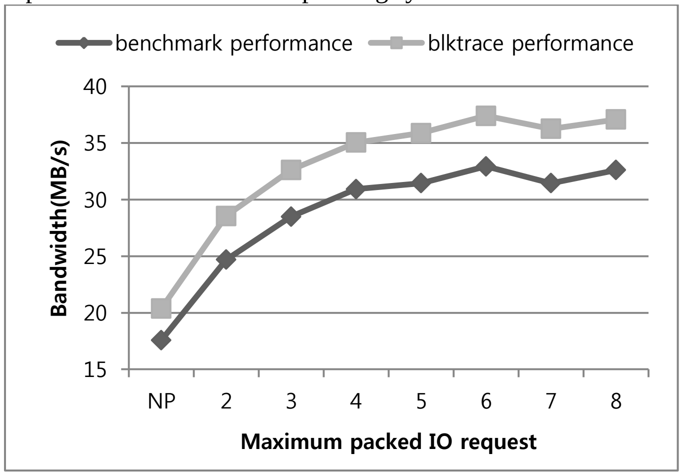

Figure 6 Bandwidth depending on maximum packed IO request, benchmark performance is bandwidth measured on tiobench, blktrace performance is measured on blktrace. The gap represents the OS overhead (Packing sequential write only, NP: No packed)

Figure 7 is experimentation on packing random write. The performance of packing random write is also improved despite the effect of interleaving cannot be implemented on random write. It is remarkable result, and should be investigated more. On the other hand, based on MMC standard driver, packed command function can pack 62 IO requests when size limitation is not obstacle. But packing 62 IO requests is not efficient than packing only 8 IO requests based on figure.

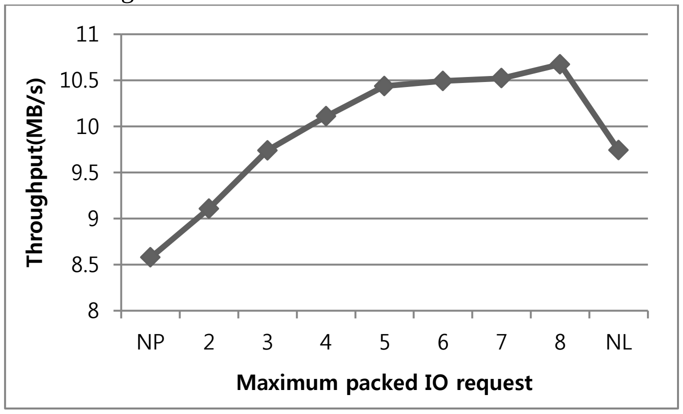

Figure 7 Bandwidth depending on maximum packed IO request (Packing sequential and random write, NP: No packed, NL: No limitations on maximum)

Blktrace is not appropriate tool to investigate packed command because blktrace trace IO behavior on IO scheduler layer. From blktrace, we only can watch scattered IO operations rather than packed IO operation. As mentioned above, ftrace can trace all functions on kernel. However, in case of ftrace, the internal parameter and behavior is not visible. The only result that ftrace can produce is function event occurs. Thus, in order to trace and estimate MMC standard driver's behavior, new tracer is required.

IV. Conclusions & Future Works

In this paper, we studied about relationship between Android and eMMC device. First of investigation, we evaluated the overhead of operating system, and result implied that the overhead of operating system is not negligible and the reason of this overhead is additional write operation that is not created by user or application. Secondly, we review eMMC specification and standard driver, and then evaluate the effects of packed command that is extended interface produced by eMMC standard.

On the future, we plan to make new tracer that can investigate the behaviors of eMMC standard like trim, discard, read, write and especially, packed command. And we will also estimate relationship between IO scheduler and packed command because we consider that optimizing IO scheduler can improve packing more IO request.

Acknowledgment

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2012-006417)

References

[1] eMMC 4.5 Specification, http://www.jedec.org

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

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

[4] M. Kuoppala, "Tiobench – Threaded I/O bench for Linux", 2002

[5] J. Axboe and A. D. Brunelle, "blktrace User Guide", 2007

[6] S. Rostedt, "Ftrace, Linux Kernel Tracing", Linux Conference Japan, 2010

[7] J. Yang, D. B. Minturn and F. Hady, "When Poll is Better than Interrupt", FAST'12, February 2012

[8] H. Kim, N. Agrawal, C. Ungureanu, "Revisiting Storage for Smartphones", Usenix FAST'12, February 2012

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