JOURNAL OF SEMICONDUCTOR TECHNOLOGY AND SCIENCE, VOL.15, NO.5, OCTOBER, 2015 ISSN(Print) 1598-1657
http://dx.doi.org/10.5573/JSTS.2015.15.5.570 ISSN(Online) 2233-4866

Optimizing Fsync Performance with Dynamic Queue Depth Adaptation

Daejun Park, Min Ji Kim, and Dongkun Shin*

Abstract—**Existing flash storage devices such as universal flash storage and solid state disk support command queuing to improve storage I/O bandwidth. Command queuing allows multiple read/write requests to be pending in a device queue. Because multi-channel and multi-way architecture of flash storage devices can handle multiple requests simultaneously, command queuing is an indispensable technique for utilizing parallel architecture. However, command queuing can be harmful to the latency of fsync system call, which is critical to application responsiveness. We propose a dynamic queue depth adaptation technique, which reduces the queue depth if user application is expected to send fsync calls. Experiments show that the proposed technique reduces the fsync latency by 79% on average compared to the original scheme.**

Index Terms—**fsync, NCQ, IO scheduler, SSD, latency, dynamic queue depth adaptation**

I. INTRODUCTION

In a buffered I/O, main memory is used as the cache of a file system, and the cached data are later written back to storage. However, due to the volatility of main memory, a buffered I/O can cause file system inconsistency at sudden system crashes.

To ensure the instant data durability, kernel provides synchronous operations such as the *fsync* system call. The latency of fsync call can affect the performance of user program because the corresponding process should be blocked until the operation is completed. Therefore, it is critical to minimize the fsync latency, which can be long significantly for several reasons described as follows.

The first reason concerns the compound transaction problem of file systems. Ext4 [1], which is a default file system in Linux, supports the journaling technique to ensure file system consistency. When a periodic journal flushing is invoked or a user program calls an fsync, the buffered file system changes are flushed into the journal area of the storage. In the ordered-mode journaling, only the modified metadata are logged at the journal area before the related data are flushed. Since the journaling scheme records all pending transactions to the journal area, the data blocks of other files as well as the fsync'ed file should be written during the fsync operation because of the compound transaction problem.

The second reason is that the write requests related to the fsync call can be delayed in the completely fair queuing (CFQ) [4] I/O scheduler. If a write request is generated by the fsync, it contains a synchronization (SYNC) flag with which the request can be issued earlier than asynchronous I/O requests in the CFQ. However, if the flush thread inserts the fsync-related I/O requests without the SYNC flags and before the fsync system call, they will be delayed in CFQ.

The last reason concerns the command queuing such as native command queuing (NCQ) [6], in which I/O requests can be queued in the device queue. Multiple I/O requests can be sent to storage under the command queuing. Current flash storage devices such as universal

Manuscript received Apr. 29, 2015; accepted Sep. 21, 2015
College of Information and Communication Engineering,
Sungkyunkwan University, 2066, Seobu-ro, Jangan-gu, Suwon-si, Gyeonggi-do, 16419, Korea
Corresponding Author: Dongkun Shin
E-mail : dongkun@skku.edu

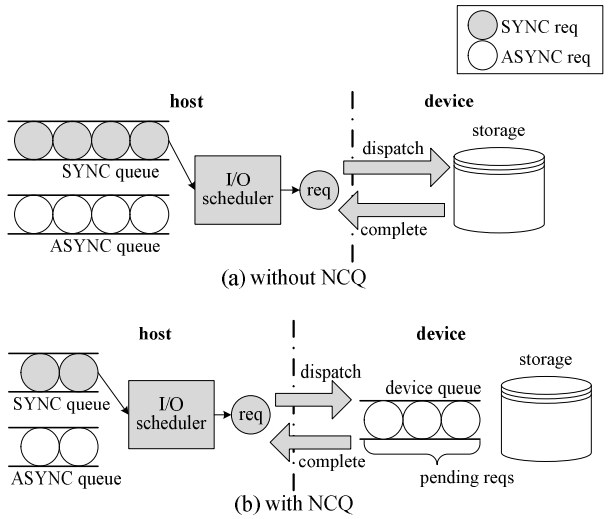

Fig. 1. I/O request handling in NCQ scheme

flash storage (UFS) and solid state disk (SSD) support command queuing in order to improve storage I/O bandwidth. This is because multi-channel and multi-way architectures of UFS and SSD can handle multiple I/O requests simultaneously. However, the command queuing can increase the latency of an urgent request such as that with a SYNC flag. Fig. 1(a) shows an I/O request handling without the support of NCQ. Only one request can be issued at a time and it is selected by the I/O scheduler based on the priorities of requests. However, as shown in Fig. 1(b), multiple I/O requests can be sent to the storage device simultaneously under command queuing. This can improve the I/O bandwidth by leveraging the parallel architecture of the flash storage device. However, if an urgent request with a SYNC flag arrives when the device queue is full of normal requests, the urgent request must wait until all the pending requests are completed. This is inevitable since the I/O scheduler cannot control the device queue. Despite the fsync latency problem, command queuing is definitely useful for increasing the I/O bandwidth. Therefore, we require a novel scheme for reducing the fsync latency under command queuing.

In this study, we propose a novel dynamic queue depth (QD) adaptation technique, which reduces the QD when fsync calls are expected to arrive. We profiled the file system behaviors and found several file types for which fsync calls are frequently used. The proposed technique monitors the opened files at runtime and changes the QD based on the probability of fsync calls to provide fast response. Experiments reveal that fsync latency can be reduced by 79% on average by means of the proposed technique when compared with the original scheme. The I/O bandwidth due to the technique is negligible.

II. BACKGROUNDS

Ext4 is currently the default file system of the Linux kernel. Because of its versatility and high performance, ext4 is widely used in mobile devices such as Android-based smartphones and desktop computers. Ext4 supports the journaling technique for file system consistency. A journaling block device [7] daemon manages the modifications on metadata and user data by a file operation as a transaction, and multiple transactions are grouped into a compound transaction for efficient handling.

The ext4 journaling supports three modes: write-back, ordered, and data journaling modes. The ordered mode, which is the default option, journals only metadata. However, it has an ordering constraint to guarantee file system consistency, in which the transaction-related data writes should be completed before the metadata journal writes. After the journal commit, the metadata can be written at its fixed location in the ext4 structures. The corresponding journal logs are then identified as *checkpointed* and the journal space can be reclaimed. At each step of the journaling, the *flush* command is sent to the storage device to guarantee that all data in the volatile buffer of the storage are flushed into the non-volatile storage media.

The Linux kernel uses the CFQ scheduler as a default I/O scheduler. The CFQ scheduler gives higher priorities to synchronous (sync) requests than to asynchronous (async) requests because the sync requests are usually more latency-sensitive than the async requests.

The NCQ is the command queuing protocol for the serial advanced technology attachment (SATA) interface. NCQ was originally proposed for a hard disk drive (HDD) and can reduce the head movement overhead of HDD. Although SSD has no head movement, NCQ remains useful for SSD because multiple pending requests can be handled concurrently in the multi-channel and multi-way architectures of SSD.

III. RELATED WORKS

IceFS [2] suggested the file system container, called *cube*, specified by the user-based directory structure. Since each cube has its own metadata, the compound transaction problem can be mitigated. However, since the cube configuration is based on the user directory structure, if there are multiple transactions within a cube, the compound transaction will be a problem. The per-block-group (PBG) journaling technique [3] also addressed the fsync latency problem caused by the compound transaction. To resolve this problem, the technique flushes only the transactions relevant to the block group of the fsynced file. Since each block group has its own metadata blocks, the PBG journaling technique can separate the transactions of the target block group from the compound transaction. Both IceFS and PBG journaling techniques have no consideration on the delay caused by the command queuing. Jeong *et al*. [5] defined the latency-sensitive quasi-async requests which are related to the fsync request. The technique handles the quasi-async requests with a higher priority in CFQ in order to boost them. Although the fsync latency problem can be alleviated by reducing the delay of the quasi-async requests under the boosting technique, it can manage only the requests in the host queue without controlling the requests on the device queue. OptFS [8] proposed a new system call, called *osync*, that assures only the ordering of the file operations. It can guarantee the durability of sync requests by receiving notifications from storage. However, this technique has no consideration for the pending requests in the device queue. In addition, it cannot reduce the latency of the fsync.

IV. ANATOMY OF FSYNC LATENCY

To observe the fsync latency problem under NCQ, a block I/O trace is analyzed. Fig. 2 shows the insert, issue, and complete events of the CFQ scheduler for handling several write requests. The device queue is initially empty and only one process generates the write requests without competing with other processes. Because the device queue can contain 32 requests, the inserted requests are issued (i.e., sent to the device queue) immediately.

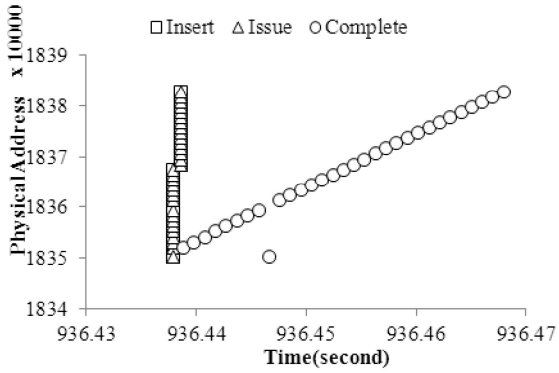

Fig. 2. Block I/O trace of a single I/O process

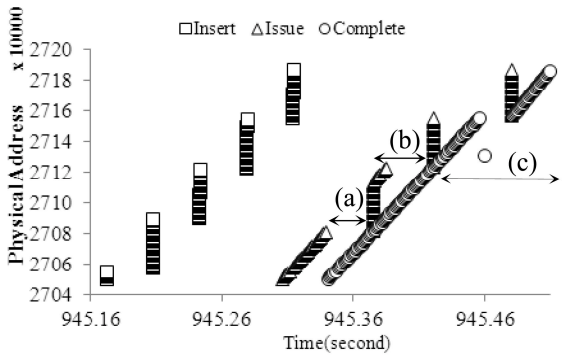

Fig. 3. Block I/O trace of multiple I/O processes

Fig. 3 shows the block I/O trace when multiple processes generate write requests concurrently. One process generates an fsync system call, and the other background processes generate several async write requests. The fsync call first generates sync write requests for the data blocks of an fsync'ed file. It then generates sync write requests for the journal blocks. Finally, it sends a flush command. When the sync requests on the data blocks arrive, the device queue is full with the async requests generated by the background processes. The time interval denoted by (a) in Fig. 3 is the waiting time of the sync request. Although the sync requests have a higher priority in CFQ, they cannot be issued because the command queue is filled with the async requests. The time interval denoted by (b) in Fig. 3 is the latency for writing the journal blocks. Before the journal write requests arrive, several async requests can interpose. Thus, because of the async requests in the device queue, the journal write requests are also delayed. After writing the journal blocks, the fsync call sends a flush command. At this point, interposed async requests can also be generated by the background processes, and

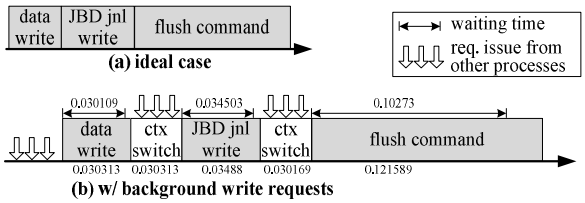

Fig. 4. Time diagram of fsync handling in an ext4 file system

they will increase the latency of the flush command. The time interval denoted by (c) in Fig. 3 shows the latency of the flush command. Because the device queue has several async write requests, the latency of the flush command is quite long.

Fig. 4 presents the fsync handling scenario using timing diagrams. As shown in Fig. 4(a), if no background write requests occur, the fsync latency is short, as the write requests of fsync call are generally small-sized. However, as shown in Fig. 4(b), if many background write requests occur, they can interpose between the fsync handling steps. Therefore, limiting the maximum number of issued requests irrelevant to the fsync call is required in order to minimize fsync latency.

V. DYNAMIC QUEUE DEPTH ADAPTATION

Fig. 5 shows performance changes with different QDs. The target storage device is a Samsung 840 pro SSD. While several concurrent threads run, fsync latency and I/O bandwidth are measured. One thread is an fsync-generating thread that writes 10 KB of data to a file and calls the fsync every five s. The remaining eight threads are write-intensive and each generates 4 KB of sequential write requests on its own file while increasing the file size. The measured fsync latency is shorter when the QD is low. However, the write bandwidths are similar when the QDs are greater than 4. This is because of the bandwidth limit of the SATA interface. Therefore, low QDs will not degrade the I/O bandwidth considerably. Considering this feature, the proposed dynamic QD adaptation technique decreases the QD if an fsync system call is expected to arrive.

To use fully the parallel architecture of SSD as well as minimize fsync latency, the QD should be decreased immediately when the sync requests of fsync arrive. However, considerable time is required to flush the device queue. Fig. 6 shows the change in fsync latency

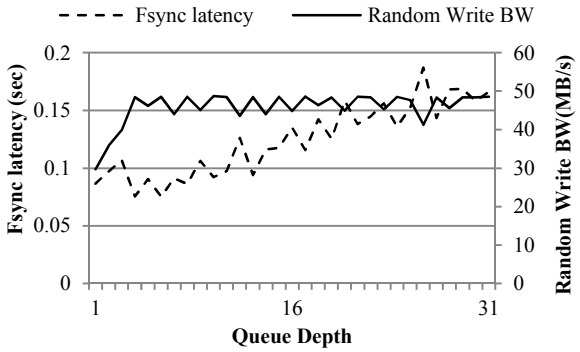

Fig. 5. Changes in fsync latency and I/O bandwidth while varying the queue depth of an NCQ-supported SSD

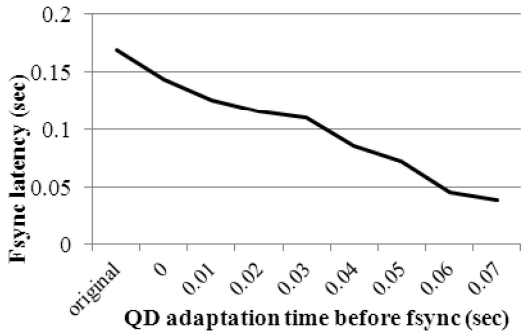

Fig. 6. Queue depth adaptation time and fsync latency

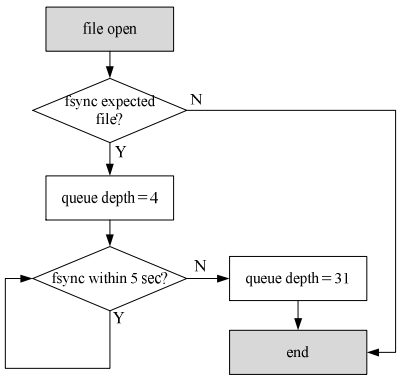

Fig. 7. Flow chart of dynamic QD adaptation algorithm

based on how early the QD is adjusted before the fsync-related sync requests arrive. The earlier the QD adaptation is invoked, the shorter is fsync latency. Therefore, we must predict arrivals of fsync-related requests in advance.

To predict the arrivals of fsync-related requests, we profiled the I/O traces of various workloads on a Linux-based desktop computer and discovered that several types of files exist for which there are frequent fsync calls. The discovered target files are database-related

Table 1. BG kernel source-copy elapsed time

Queue depth	BG kernel source copy elapsed time (s)
Queue depth 31	19.082
Dynamic QD adaptation	20.570

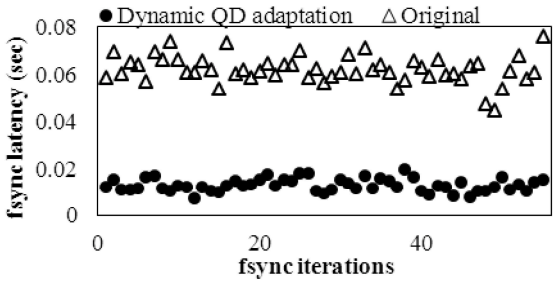

Fig. 8. Reduction in fsync latency by the dynamic QD adaptation scheme

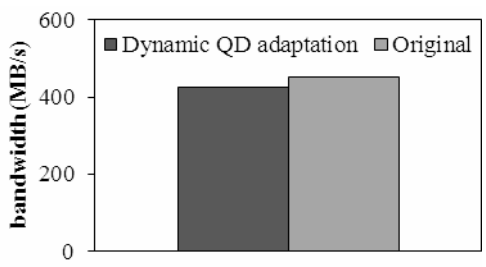

Fig. 9. I/O bandwidth reduction by the dynamic QD adaptation scheme

files such as .db, .db-journal, and .db-wal, and document files such as .doc. The proposed adaptation technique decreases QD if one of these files is opened and restores it if no fsync occurs during five s, as shown in Fig. 7.

VI. EXPERIMENTS

The effectiveness of the proposed dynamic QD adaptation scheme is evaluated on a Linux-based desktop computer, which is equipped with an Intel i5-4670 3.4 GHz CPU processor, 8 GB DRAM, and a Samsung 840 pro SSD. The Linux kernel version is 3.10.

Fig. 8 shows fsync latency while running two processes concurrently. One is a background process that generates 128 KB of random write requests, and another is an fsync-generating process that calls an fsync every two s. The dynamic QD adaptation scheme reduces the fsync latency by 70% on average compared to the original scheme. However, as shown in Fig. 9, dynamic QD adaptation reduces the write bandwidth of the

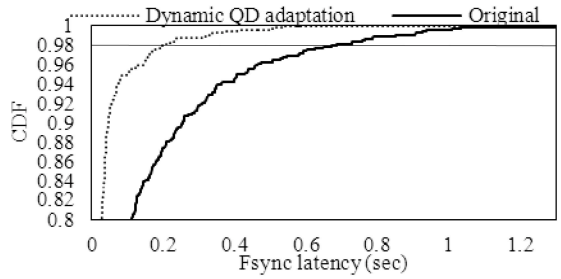

Fig. 10. CDF of fsync latencies on a Chrome browser

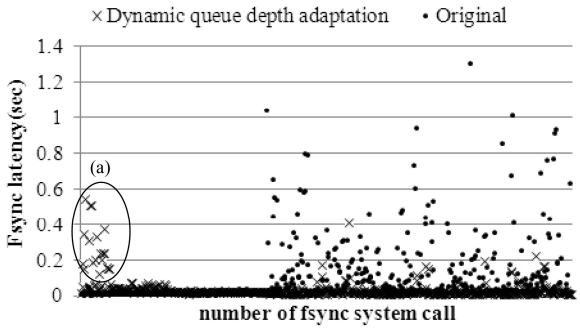

Fig. 11. Fsync latencies when running a Chrome browser

background process by only 5%.

For a real workload, we estimated fsync latency of a Chrome web browser while copying Linux kernel codes. The web browser generates many small configuration files and invokes fsync calls on the files frequently. Because the files are stored in a special directory, the dynamic QD adaptation scheme reduces the QD when a file of the target directory is opened. As shown in Fig. 10, the fsync latencies in a Chrome browser are significantly reduced. The result shows that 98% of fsync latencies on the dynamic QD adaptation are within 0.2 s, whereas those on the original scheme are within 0.6 s. Only a slight change occurs in the elapsed times of file copy operations, as shown in Table 1.

Fig. 11 shows the change in fsync latencies when running a Chrome browser. The overall latency values under the dynamic QD adaptation are less than 0.2 s. However, the time interval denoted by (a) in Fig. 11 shows rather long latencies because of missed predictions on fsync request arrivals. Our future research will involve developing a more exact prediction scheme.

We also examined the performance of Libre office word while running a background process that copies Linux kernel codes. The fsync latencies are measured while saving a 24 KB document file. As shown in Fig. 12,

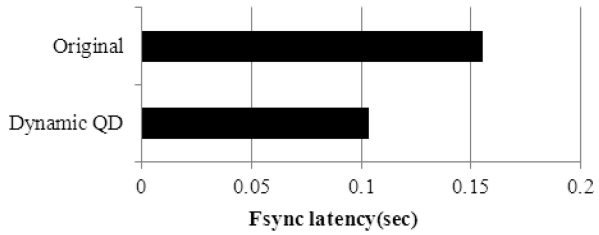

Fig. 12. Average fsync latency on Libre office word

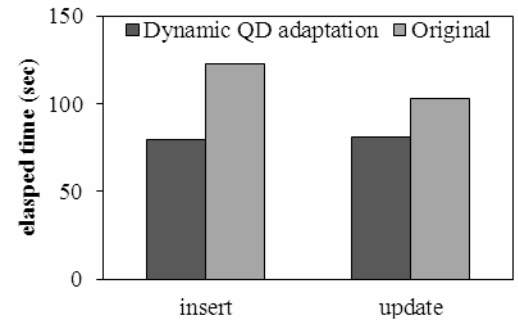

Fig. 13. Performance improvements based on the dynamic QD adaptation on the Mobibench workload

the proposed scheme reduces fsync latency by 33.5%.

The latency of an fsync call is related to throughput database applications as well as user responsiveness. Fig. 13 shows the elapsed times for the Mobibench [10] workload, which generates various types of SQLite requests. A background application that copies Linux kernel source codes is concurrently executed. Because each transaction requires several fsync calls, fsync latency affects database performance considerably.

VII. CONCLUSIONS

Command queuing is a useful technique for improving the I/O bandwidth of flash storage devices. However, command queuing can be harmful to the latency of fsync system call because the urgent requests of fsync calls can be delayed by pending I/O requests in the device queue. In this study, we examined reasons for fsync operation delays in the command queuing scheme. In addition, we proposed a dynamic queue depth adaptation for minimizing fsync latency without a considerable reduction in I/O bandwidth. Experiments with real devices showed that the dynamic queue depth adaptation improved fsync latency by approximately 79%.

ACKNOWLEDGEMENT

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2013R1A1A2A10013598).

REFERENCES

[1] Mathur, Avantika, et al. “The new ext4 filesystem: current status and future plans.” *The Linux Symposium.* Vol. 2. 2007.

[2] Lu, Lanyue, et al. “Physical disentanglement in a container-based file system.” *The 11th USENIX conference on Operating Systems Design and Implementation. USENIX Association*, 2014.

[3] Kang, Yunji, and Dongkun Shin. “Per-block-group journaling for improving fsync response time.” *Consumer Electronics (ISCE 2014), The 18th IEEE International Symposium on. IEEE*, 2014.

[4] J. Axboe, “Linux Block IO - Present and Future,” *The Ottawa Linux Symposium 2004*, pp. 51-61, Jul., 2004.

[5] Jeong, Daeho, Youngjae Lee, and Jin-Soo Kim. “Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices.” *The Conference on File and Storage Technology (FAST’15)*, pp. 191-202, Feb., 2003.

[6] Dees, Brian. “Native command queuing-advanced performance in desktop storage.” *Potentials, IEEE 24.4*, pp 4-7. 2005.

[7] Tweedie, Stephen C. “Journaling the Linux ext2fs filesystem.” *The Fourth Annual Linux Expo.* 1998.

[8] Chidambaram, Vijay, et al. “Optimistic crash consistency.” *The Twenty-Fourth ACM Symposium on Operating Systems Principles. ACM*, 2013.

[9] Axboe, J. FIO - flexible IO tester. http://freshmeat.net/projects/fio/.

[10] Jeong, S., Lee, K., Hwang, J., Lee, S., and Won, Y. AndroStep: Android storage performance analysis tool. *In Software Engineering (Workshops)’13*, pp. 327–340. 2013.

Daejun Park received a B.S degree in Computer Engineering from Sungkyunkwan University, Suwon, Korea, in 2013. He is currently a Ph.D. student in the School of Information and Communication Engineering, Sungkyunkwan University. His research interests include embedded software, file systems, journaling, and flash memory.

Min Ji Kim received B.S. and M.S. degrees in Computer Engineering from Sungkyunkwan University, Suwon, Korea, in 2013 and 2015, respectively. She is currently an Assistant Engineer at Samsung Electronics, Korea. Since 2015, her research interests have included embedded software, memory management, and flash memory.

Dongkun Shin received a B.S. degree in computer science and statistics, an M.S. degree in computer science, and a Ph.D. degree in computer science and engineering from Seoul National University, Korea, in 1994, 2000, and 2004, respectively. He is currently an associate professor in the Department of Computer Science and Engineering, Sungkyunkwan University (SKKU). Before joining SKKU in 2007, he was a senior engineer at Samsung Electronics Co., Korea. His research interests include embedded software, low-power systems, computer architecture, and real-time systems.