EFFECT OF TIMER INTERRUPT INTERVAL ON FILE SYSTEM SYNCHRONIZATION OVERHEAD

Hankeun Son[1], Seongjin Lee[2], Youjip Won[3]

[1,2,3]Department of Computer Software, Hanyang University, Seoul, Korea
hself@hanyang.ac.kr, insight@hanyang.ac.kr , yjwon@hanyang.ac.kr

Abstract: File system metadata is indispensable in both describing the data and maintaining the file system. Despite the importance of metadata in the file system, the overhead of maintaining the metadata cannot be taken lightly. It is because the metadata also have to be persisted on the storage device and it consumes IO bandwidth as well as creates journaling overhead. In this paper, we find that the random write with synchronous performance of a storage is significantly affected by not only the hardware performance but also timer interrupt interval of the kernel. Extending the timer interrupt interval allows reducing the write volume and increasing the random write followed by fsync() performance of EXT4 file system. We propose intermittent mtime timestamp update on Coarse grain mtime interval instead of Fine grain mtime interval. The experiment results with mtime update interval of 1 second show that the total write volume is decreased by 75% and 28%, respectively compared to total write volume of 1 ms and 10 ms mtime interval, and the throughput increased 3.1× and 1.2× compared to 1 ms and 10 ms mtime interval. Coarse grain mtime update is resolve to the journaling overhead issues while still logging mtime timestamp.

Keywords: Journaling overhead; EXT4 file system; Random write; Kernel timer interrupt; mtime;

1 Introduction

In the era of NAND Flash memories and fast storage mediums, the storage device is no longer considered the bottleneck of the system [6]. SSDs can store a chunk of data within few hundreds of milliseconds, and for even faster storages can store in under a millisecond [3]. However the most of the kernel and the IO stack is built on top of the fact that the storage device is the bottleneck of the system and the software needs to provide the means to guarantee the reliability. EXT4 file system, for example, provides journaling mechanism to keep logs of changed data which has to be stored in the storage every time user writes some data [8]. The default unit of write in EXT4 journal area is 4 KByte but the smallest possible journal record size is not 4 Kbyte because journal itself has an overhead to mark the beginning and the end of the journal commit operation. The effect can be amortized when the write is buffered and many metadata is in a journal transaction. Typical metadata in the EXT4 file system that is frequently updated are inodes, inode bitmaps, and group descriptors. A mobile system, which also uses the EXT4 file system, is known to generate a lot of fsync() calls [7, 11]. This workload considered unfavorable to both file system and storage system because every call of fsync() has to not only immediately but also synchronously flush metadata of the file in page cache and write both journal and data to the storage. For example, let us consider 4 KByte random write followed by fsync(). Since this workload does not change the size of the file, only atime, mtime, and ctime field in the inode are updated, but the file system has to write metadata to journal area every time fsync() is called. When a file is accessed, data is modified, and inode is changed the file system takes note of the fact and keeps the timestamp in atime, mtime, ctime field in the file system metadata called inode. mtime which is also profusely used in the system, on the other hand, also can be rate limited to reduce the write volume; however, the use of mtime cannot be entirely disabled because it is essential in enabling features for sorting, finding and recovering.

2 Background

2.1 Kernel timer interrupt

Kernel timer interrupt is in charge of calculating time stream in the kernel. As the scheduling for context switching and the delay for various purpose, timer interrupt used frequently. For example, the scheduling allocates time slices on the priority list for the fairness of CPU utilization among the task in Linux system. Once one task uses up all time slice for oneself, then context switching occur to changing another task. Even though all time slices is consumed, context switching is not occurred immediately by an event of timer interrupt because time streams depend on kernel timer interrupt interval in the kernel.

In the kernel, system timer is jiffy counter, and we use jiffy counter to know time after running system. Jiffy is an increasing unit at every kernel timer tick. Since 1 tick, the time between two kernel timer interrupt, define 1 / Hertz (HZ). Therefore, a rate of increasing jiffies, called timer interrupt frequency, is determined by the parameter of HZ to depend on hardware architecture. In early Linux, tick rate is fixed 100 HZ and 1 tick is 10 ms, and in 2.5 version, 1000 HZ adapted as tick rate. After 2.6 version, however, users select tick rate at kernel compile time, while default tick rate is rollback to 100 HZ. On the other hands, on Windows OS, default timer resolution is 15.6 ms and adjust timer interrupt from 64 HZ to 1000 HZ [5].

Table I Timer interval in android smart devices

Year	Device	OS	HZ	Interval
2010	Galaxy S	GB	256	3.9 ms
2011	Galaxy S2	JB	200	5 ms
2012	Galaxy S3	JB	200	5 ms
2013	Galaxy S4	JB	100	10 ms
2014	Galaxy S5	KK	100	10 ms

High resolution of kernel timer by large HZ parameter is able to fast response for users request. However, it has disadvantage of overhead, power consumption and reducing battery lifetime by frequent interrupts. Table I show, early android mobile devices adapted high parameter e.g., 256 HZ tick rate for enhancing to slow response. However, up-to-date devices keep 100 HZ tick rate since tick rate was gradually decreased by increasing hardware performance of mobile devices. In the most of the latest devices, therefore, kernel time is changed for 10 milliseconds, 1 tick of 100 HZ.

2.2 Write mechanism of EXT4 file system

EXT4 [8] file system is most popular used to Linux storage system but also adapted default on the main partition in mobile devices after android 2.3. To guarantee of consistency from sudden power failure and system crash, it creates and stores journal file logging change data by a write-ahead logging. EXT4 is in-place file system and is adapted journaling mechanism, e.g., data on the way to store are damaged due to system crash, and then EXT4 implements redo recovery using the journal file. An ordered mode is not required to journaling except for metadata of file inode changed by the read and the write.

In ordered mode, contents of all updated metadata follow the descriptor, since a journal descriptor writes 4 KByte of the header including the original location of the changed metadata. After all the changes are persisted to the journal area, it marks the complete of a journal transaction by writing another 4 KByte commit block to the journal area. Therefore, minimum of one update in the file system write 4 KByte (data) + 2×4 KByte (descriptor, commit block) + 4 KByte (metadata) = 16 KByte. The effect can be amortized when the write is buffered and many metadata is in a journal transaction. After that, the dirty page is flushed by fsync() called upon applications.

2.3 Write mechanism of F2FS file system

The Flash Friendly File System (F2FS) [10] is an alternative file system for designed to avoid journaling overhead in NAND flash storage devices. Although it based on the log-structured file system(LFS) [9], It resolve wandering tree problem [2] to introducing pointer blocks called Node and Node Address Table (NAT).

In F2FS file system, performing 4 KByte write followed by fsync() is data and node block to in-place, respectively 4 KByte. Metadata of the validity of written block and the address table of node block only update to in-memory temporarily, by checkpoint. And then performing checkpoint write to in-place of storage. F2FS Therefore, one update in file system write 4 KByte (data) + 4 KByte (node) = 8 KByte except checkpoint overhead.

2.4 Timestamps update of file system

The File system support timestamps for record specific time information. The timestamps are composed of atime, mtime, ctime. Three timestamps are the time which is parts of metadata of file inode from Unix epoch time on 1 Jan 1970 00:00:00 UTC [1]. These timestamps are useful information as the file name when we find and sort files.

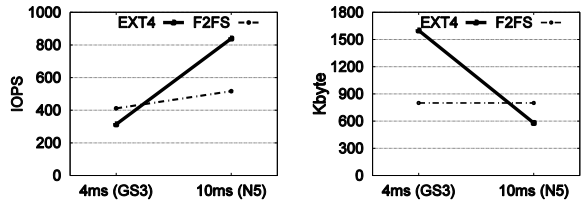
atime: This timestamp records the last time when anyone access to file. We find the last or specific time used the file for using atime. However, we focus on that atime is updated at write as well as read. In EXT4 file system, journaling for changing the metadata of file inode is used to persist data in ordered mode. atime triggers off overhead since read more frequently than write is always update atime. The overhead is amplified by journaling of EXT4 file system.

mtime: This timestamp means data modification time recorded when the last file was modified. After a file is created, file contents are modified and mtime is updated. However, mtime is not updated at copy and move because file contents remain constant. Since mtime is the last time information about a file modification, which is important for file recovery from the system crash. Rather than other timestamps, mtime is frequently used to find and sort. In mailing servers, mtime compared to atime used for a mark whether receiver already read the mail or not.

ctime: This timestamp shows change time of the file inode. When inode metadata e.g, mtime, ownership, and permission of the file change, ctime is updated. Although ctime is updated alone without mtime due to changing the ownership and permission, most of ctime is updated with the updating mtime. Therefore, the usability of ctime is lower than mtime for find and sort file, except for that case.

3 Effect of timer interrupt interval

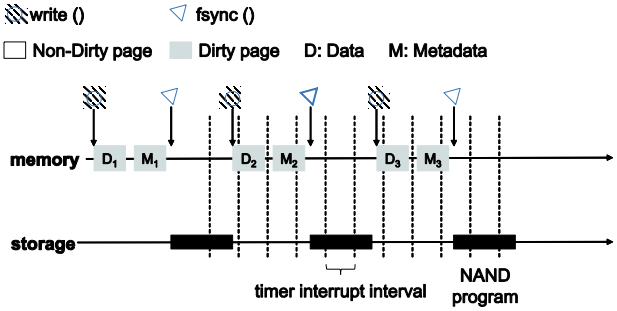
Figure 1 illustrates the motivating experiment with two android devices. It shows throughput and the total write volume generated while performing 4 KByte random write followed by fsync() on Galaxy S3 and Nexus 5. Although the specification of two devices is no exactly the same but it gives a rough idea of the behavior. The throughput of EXT4 file system on Galaxy S3 is lower than that of F2FS file system, but the behavior is reversed in Nexus 5. After the thorough analysis of the software stack of two devices, we find that the only difference was the time interrupt interval. The value in Galaxy S3 was set to 4 ms, whereas it was 10 ms in Nexus 5.



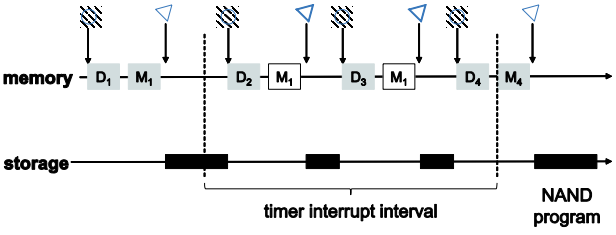
(a) Performance **(b)** Total write volume

Figure 1 Throughput and total write volume: Galaxy S3 Vs. Nexus 5

Kernel keeps a system timer to govern the timing of many policies and mechanisms. The unit of the time used in the system timer is called jiffies which set to 100 Hz that is 10 ms; the user can decide the value at a compile time. There is a trade-off in a decision of the value of jiffies; high resolution provides a better response but suffers from frequent context switches. Low resolutions, on the other hand, delays the context switch time until the rise of the timer interrupt, thus even if a process has used all its time slice, it is not context switched but to wait for the timer interrupt.



(a) Fine grained interrupt interval



(b) Coarse grained interrupt interval

Figure 2 The effect of interrupt interval in random write followed by fsync()

An interesting fact is that there is no update in mtime nor ctime for all write in between the interrupt intervals. It is because to update mtime and ctime, file system exploits file_update_time() function to check the time. Within the timer interrupt interval, file_update_time() return the same timestamps.

Figure 2 illustrates an overview of the mechanism that updating metadata; mtime in 4 KByte random write followed by fsync(). In After first write call, metadata of file inode is dirty independent on timer interrupt due to the metadata including file size, block allocation information above mtime. Let us consider 4 KByte random write followed by fsync() once again, before fsync() call, file_update_time() take kernel time to update to new mtime, and dirty inode page if it updated. Since random workload only operates on an existing file, the file size and block allocation information does not change and the only candidates of changes are mtime and ctime. In Figure 2(a), file_update_time() update mtime and ctime timestamps and dirty metadata page for every write, because file_update_time() called within the timer interrupt interval. In Figure 2(b), However, file system checks with file_update_time() to record the timestamps within the timer interrupt, but it only finds that the time not updated because the system uses the lower resolution. Thus mtime and ctime are updated only once every update in file_update_time() function within timer interrupt interval.

4 Enabling Coarse Grained mtime interval

Since modifying the jiffies or kernel timer interrupt interval causes many complications and creates many unknown side effects, we decided not to alter the interval. Instead, we extended the file system to update mtime interval once every 1 seconds. We simply modify only cut-off subsecond information for updating mtime and ctime in file_update_time() function. Even through time updating interval of file inode is 1 second, users still use to time calculating e.g., gettimeofday() in the file system.

5 Experiments

5.1 Experimental setup

For the experiment, we used Google's reference smartphone device, Nexus 5, which has SnapDragon MSM8974 2.26 GHz CPU, 2 GByte of DRAM, 16 GByte eMMC, and running Linux 3.4.0-g9eb14ba with Android 4.4 KitKat. To guarantee the consistency of CPU performance we set the CPU Governor to performance policy to manage the clock and also enabled all four cores in the device by setting them to online and fix at 2.26 GHz frequency which is maximum. We varied kernel timer interrupt interval from 1 ms to 10 ms with increments of 1 ms and used EXT4 [8] and F2FS [10] file system. We used Mobibench [4] to generate 4 Kbyte random write followed by fsync() on a 1 MByte file.

5.2 Total write volume

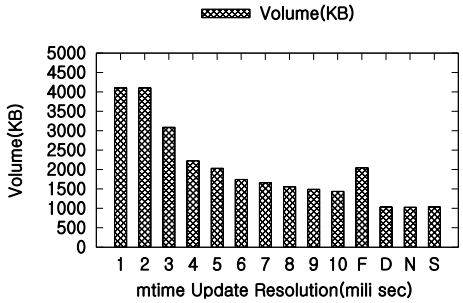


Figure 3 Total writevolume (F: F2FS, D: fdatasync(), S; 1 sec, N:None)

There is a total of 256×4 KByte IOs for 1 MByte file, and

we expect metadata update for 256×; the total write volume of data and metadata journal write will be 1024 KByte and 3072 KByte, respectively. Thus, the total write volume is at least 4096 KByte.

Figure 3 illustrates the effect of mtime on the total write volume. When kernel timer interrupt interval is set to 1 ms, the total write volume is 4104 KByte, which as much as the theoretical volume. As the interval becomes longer, the write volume decreased; in 10 ms case, the total write volume becomes about 65 % lower than that of 1 ms case. When we change data synchronous function call from fsync() to fdatasync(), which updates metadata when the size of a file is changed, the write volume was reduced as much as the case where mtime is updated once every second. The volume is 1036 KByte, and it is comparable to the case that the update of mtime is disabled which has a volume of 1028 KByte.

5.3 Throughput

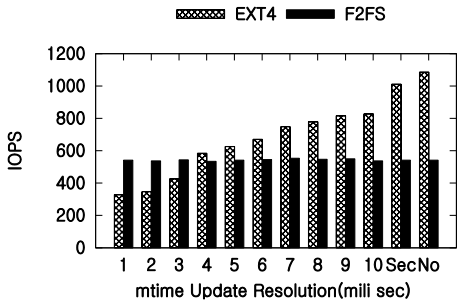


Figure 4 Total performance (S: 1 sec, N: None)

Figure 4 shows the effect of the interval on the throughput of the system. As we have observed in Figure 4, we can expect the throughput to increase as the total volume decreases. The result shows that 10 ms timer interval is 2.5× faster than that of 1 ms interval. F2FS, on the other hand, did not benefit much from the changes of resolutions. It shows that the throughput of F2FS is as high as the resolution of 4 ms case. Since F2FS is a log-structured file system, it has to update the node block to in-place regardless of the update the changes in the mtime and ctime to in-memory. When the resolution is set to 1 sec, the proposed coarse grain mtime update shows about 3.1× better throughputs than that of 1 ms case, and it is only 7 % lower than that of no update case.

5.4 Discussion

Our experiments on smartphone show that relationship between the kernel timer interrupt and the performance of file system. Unexpectedly, F2FS shows lower performance than EXT4 in high tick rate. In the micro benchmark, there are significant performance gains by coarse grained mtime update. We expect that this performance improvement can apply most widely used applications such as mailing server and DBMS. Because commonly DBMS is I/O optimized application and has intensive synchronous write by fsync() [7, 11].

6 Conclusions

In this paper, we introduce that there is a direct relationship between kernel timer interrupt interval and IO performance, especially on Android mobile devices which frequently updates file system metadata to the journal area. Then, we compared two existing smartphones that exhibit totally different IO performance and the reason for the difference is update frequency of mtime and ctime. We reduce the total write volume and increase the throughput of the system by modifying the timer interrupt interval to 1sec. The experiment shows that the throughput increased by 3.1 times and the volume decreased by 75 % compared to 1 ms interval case.

Acknowledgements

This work is supported by IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for New-memory based Smart Device). This work was supported by the ICT R&D program of MSIP/IITP.[R0601-16-1063, Software Platform for ICT Equipments] This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) [R7117-16-0232, Development of extreme I/O storage technology for 32Gbps data services]

References

[1] Unix Programmer' Manual(1st ed) https://www.bell-labs.com/usr/dmr/www/pdfs/man22.pdf

[2] Bityutskiy, A. 2005. JFFS3 design issues. http://www.linux-mtd.infradead.org/

[3] B. Tallis. Intel Announces SSD DC P3608 Series, 2015. http://www.anandtech.com/show/9646/intelannounces-ssd-dc-p3608-series.

[4] LEE, K. Mobile Benchmark Tool (MOBIBENCH). https://github.com/ESOS-Lab/Mobibench

[5] Windows Timestamp Project http://www.windowstimestamp.com/description

[6] H. Kim, N. Agrawal, and C. Ungureanu. Revisiting storage for smartphones. ACM Transactions on Storage (TOS) 8, 4 (2012), 14.

[7] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In Proceedings of the USENIX Annual Technical Conference (ATC), San Jose, CA, USA, Jun 2013.

[8] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In Proceedings of the Linux Symposium, 2007.

[9] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. ACM Trans. Comput. Syst., 10(1):26–52, Feb. 1992. ISSN 0734-2071.

[10] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In Proceedings of the USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, Feb. 2015.

[11] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In Proceedings of the USENIX Annual Technical Conference (ATC), Santa Clara, CA, USA, July 2015.