Coarse-grained `mtime` Update for Better `fsync()` Performance

Hankeun Son Seongjin Lee Gyeongyeol Choi Youjip Won

Dept. of Computer Science
Hanyang University, Seoul, Korea

{hself|insight|chl4651|yjwon}@hanyang.ac.kr

ABSTRACT

This work is dedicated to improve the performance of the `fsync()`, which is one of the most expensive system calls in UNIX operating systems. Due to the recent advancement of the Flash storage based storage device, the storage device can flush the data blocks in order of magnitudes faster rate than the legacy HDD does. Often, the rate at which the storage device flushes the data blocks prevails the rate at which the CPU updates the kernel clock. The amount of the dirty blocks created in the system is dependent upon the timer interrupt interval. Frequently performed reads and writes update the `atime` and `mtime` metadata respectively. These timestamps are useful however, degradation occurred by overhead for frequent updates are significant. Now in the file system, `atime` has several options to mediate between usefulness and performance efficiency. Most of the Database Management Systems frequently perform `fsync()` depending synchronous writes to correct consistency of user data. The synchronous writes involve journaling overhead of `mtime` update metadata in ext4 file system. However, frequent updates of `mtime` on write intensive workload being negatively influenced efficiency has been overlooked. We introduce coarse-grained `mtime` update scheme to increase the `mtime/ctime` timestamp update interval while maintaining the same level of resolution for kernel time interrupts. As a result, coarse-grained update interval scheme reduces the journaling overhead with the least effort. The experiment results show that the I/O performance of random workload on mobile and PC increased about 7% and 107% against the default `mtime` update interval, respectively. The result of insert operations on PERSIST mode of SQLite on mobile and PC shows 8.4% and 45.1% of I/O performance increase, respectively. On MySQL OLTP workload, the performance increased by 7.9%.

SAC'17, April 3-7, 2017, Marrakesh, Morocco

http://dx.doi.org/xx.xxxx/xxxxxxx.xxxxxxx

Keywords

`mtime` update, `fsync()` performance, timer interrupt, journaling overhead

1. INTRODUCTION

As the volume of data stored, and processed by IoT devices, Big data and Cloud services, are increasing, the need for high performance and reliable storages system for mobile and server environment is also growing[15]. Hard disk drives which are thought to be the bottleneck of the system performance are slowly giving its place to more fast devices such as SSD, eMMC, UFS, and NVMe, and the rapid growth of the mobile devices [14] are expanding the bases for such high performing storage devices[37, 30, 8, 5]. Although database management systems and file systems are responsible for guaranteeing the consistency of data via journaling, they are known to induce significant journaling overhead in the system [17, 34, 24, 19, 27, 26]. Most of the storage systems and database management systems keep a log of I/O operations before writing the data to a storage device. To flush the data to the storage, databases exploit `fsync()` system call to synchronize all the updates to the storage, and file systems exploit journaling (ext4, xfs)[38, 29, 1, 36], Copy-on-Write(btrfs) [32], and log-structured write(F2FS) [13, 33, 22] mechanisms.

The current implementation of Linux synchronizes dirty pages in the cache to disk after an interval of 30 seconds, and file system journaling module flushes dirty pages to journaling area after an interval of 5 seconds. Regardless of these efforts to increase the reliability of the system, users explicitly call one of `sync()`, `fsync()`, and `fdatasync()` system calls to flush the dirty pages to the storage immediately. For example, an insert operation using PERSIST journal mode in SQLite, the default database for Android devices, issues four `fsync()` calls to persistently store the data and the journal files to the storage [25]. Since PERSIST mode reuses existing inodes and file blocks, time-related metadata are updated to the journal. And, an `fsync()` call in ordered mode of ext4 generates at least three I/Os. In ext4 ordered journaling mode, file system first flushes the data to the storage and then logs metadata to the journal area. Typical metadata logged on the journal are inode timestamps including accessed/modified/changed timestamps, inode bitmap, etc. As a result, single insert operation in the application can

create a few tens of Kbytes of data to the storage.

The combined efforts of database management system and file system in providing consistency and reliability of data are known to suffer from a phenomenon called Journaling of Journal problem [17]. This redundant effort by a file system to journal the data that is already journaled by database management system not only decreases the I/O performance but also decreases the life time of NAND Flash based storages. Existing solutions modify a great deal of file system and database implementations [20, 12, 7, 25]. In this work, our objective is to reduce the number of I/Os with much fewer endeavors while maintaining the same level of file system integrity. To this end, we have investigated the contents of file system journal logs and the PERSIST journaling mode in SQLite and found that timestamps recorded in file inode are frequently updated. For example, a read operation updates `atime` to record the file access time. Since the frequent update of `atime` is recognized as overhead to the system, there are a number of workarounds such as `noatime`, `nodiratime`, `relatime`, and `lazytime` to reduce the overhead. On the contrary, the other two timestamps (`mtime` for modification time and `ctime` for inode change time) which are also frequently updated have not received enough attention.

We analyzed the effect of kernel timer interrupt interval on journaling of ext4 file system to reduce the write amplification caused by metadata updates. The timer interrupt is responsible for deciding time in the kernel. As the timer interrupt interval increases, timestamp update interval also increases. As a result, the number and the amount of metadata logged to the journal area reduces.

In this work, we propose coarse-grained `mtime` update scheme to increase the performance of using `fsync()` comparable to that of `fdatasync()` while maintaining sufficient timestamps in file inode for the recovery. The proposed scheme is especially effective in writes to a pre-allocated block followed by `fsync()` and PERSIST journal mode in SQLite. The performance on Android mobile shows that there is about 7% performance increase in pre-allocated write I/O followed by `fsync()` workload and there is about 11% performance gain in insert operations with PERSIST journaling mode in SQLite than that of the default configuration. In PC, the performance gain is much dramatic; it shows that proposed scheme increases the performance of pre-allocated write followed by `fsync()` and inserts with PERSIST mode by 107% and 45%, respectively. Results also show that proposed scheme is also effective in MySQL OLTP workload; there is about 9% performance increase compared to that of the default configuration.

2. BACKGROUND

2.1 Timer Interrupt

The timer interrupt is periodically generated. This period is defined as `HZ` value in param.h file on the kernel. The value is set up by `CONFIG_HZ` option for configuration before kernel compile. The kernel has a volatile global variable called the jiffies which initialize as zero when system boot. The variable is a counter by the timer interrupt. The kernel recognizes the time stream by accumulating jiffies. The current time in kernel displays related time pass from the Unix epoch, 1970-01-01[6]. The time update uses Real-Time Clock (RTC) [2] as accuracy at the nsec unit. The current time is saved long type variable in xtime structure and is used by `current_kernel_time()`. While the file system does not use directly the function to record inode timestamp. The file system uses `current_kernel_time()` to gain current time. The `current_kernel_time()` returns truncated value which set up by defined timer interrupt.

Table 1: Timer interval in Android devices

Year	Device	Core #	OS	HZ	Interval
2010	Galaxy S	1	GB	256	3.9 ms
2011	Galaxy S2	2	JB	200	5 ms
2012	Galaxy S3	4	JB	200	5 ms
2013	Galaxy S4	8	JB	100	10 ms
2014	Galaxy S5	8	KK	100	10 ms
2015	Galaxy S6	8	LP	100	10 ms

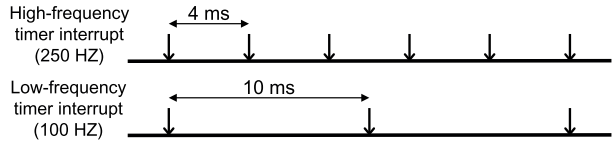


Figure 1: Timer interrupt interval and frequency (Downward arrows indicate the timer interrupt events)

With higher `HZ` the tick rate also gets higher. In return, timer interrupt resolution becomes much finer and allows precise handling of time-related events. The downside of having higher `HZ` is that it increases the number of execution of timer interrupt handler as well as the number of context switches. For example, the number of context switches at 1000 `HZ` is 10 times more than at 100 `HZ`. Fig. 1 illustrates the number of timer interrupts with respect to `HZ`, where each downward arrows indicate the timer interrupt events. For example, if the timer interrupt frequency is set to 100 `HZ`, the system makes a context switch at every 10 ms. Assuming that a job is completed in 6 ms before a given time slice, i.e. 10 msec, it may wait as long as 4 msec before making the context switch. In such cases, the delayed response time becomes a problem.

In Linux kernel, `HZ` is hardware architecture dependent, and with few exceptions (ia64, v850, m68knommu) the value is in between 100 to 1000 `HZ`.[35] The rate of timer interrupts has changed over the years but the recent systems are settled with 250 `HZ` that is 4 msec for a tick. Depending on the purpose of a system, the frequency may vary. In a system where fast response time is of importance, the frequency is set to 1000 `HZ`, and server and Non-Uniform Memory Access(NUMA) systems use frequency of 100 `HZ` to reduce the number of timer interrupts.[18] Table 1 shows that Android devices now use 100 `HZ`. There are works that try to minimize the overhead of having high rate timer interrupt while running applications in finer timer resolution and keeping the response time short [3]. However, the problem of manipulating the timer interrupt frequency is that it affects the

whole system.

2.2 Inode Timestamps

Inode keeps various information about a file such as access permission, ownership, group, size, the number of data blocks, addresses of data blocks, and timestamps. One that gets frequently updated in random workload is a timestamp. There are three timestamps stored in an inode, `atime`, `mtime`, and `ctime`, which can be retrieved using stat or istat. Although each timestamp represents seconds and nanoseconds since the Unix Epoch, it cannot have finer resolution than kernel timer interrupt interval because file system uses kernel time to update the inode timestamp. The description of each timestamp is as follows.

atime: It shows the last time the data from a file was accessed. There are many use cases of `atime`, e.g., the system can use it to manage log-in information of a user. Since `atime` changes not only in writing but also on opening and reading a file, the write overhead incurred by `atime` cannot be taken lightly.

mtime: It indicates the timestamp of the last change made to the contents of a file. It does not, however, track the changes made to metadata such as owner or permission of a file. One of use cases of `mtime` can be found in a mail server, it uses `mtime` and `atime` to track if a mail is read or not. It is also used in the recovery of a file system.

ctime: Along with `mtime`, it also tracks the time of the last change made to the contents of a file, however, it also tracks the time when the file's ownership or permission changes. Moving a file, for example, updates `ctime` but not `mtime` or `atime`. On the contrary, an update of `atime` or `mtime` makes `ctime` be updated.

2.3 ext4 Journaling and fsync()

ext4 uses journaling to guarantee the file system consistency. The default journaling mode in ext4 is ordered mode, where a data is always written first and then changes in metadata are stored in the journal area. The default commit interval for ext4 file system journal is 5 seconds, and explicit `fsync()` calls also write the changes to the journal area. Database management systems rely on `fsync()` to guarantee the persistency and the ordering of data. For example, when a user writes 4 KB data block, then file system writes 4 KB data block and marks a page that contains the timestamp, bitmap and other metadata for the file in page cache as dirty. With a call of `fsync()`, the data and dirty metadata page for the file is flushed to the storage. ext4 journal logs the updated metadata to the journal area along with a 4 KB journal header at the beginning of a transaction and a 4 KB journal commit mark at the end of a transaction. Thus, 4 KB write followed by an `fsync()` incurs at least 16 KB data to the storage, which is notable write amplification overhead. It is important to understand the journal behavior because a small write followed by an `fsync()` is prevalent in many applications such as SQLite and MySQL [31] to guarantee the data integrity and consistency of user data. For example, an insert in PERSIST SQLite journaling mode writes as much as 80 KB to the storage [17, 25].

2.4 fsync() and fdatasync()

`fsync()` and `fdatasync()` are alike in terms of providing the guarantee of persistency of updated contents of a file on the storage. `fdatasync()` is different from `fsync()` in handling the metadata, it does not synchronize the changes in timestamps. Thus, using `fdatasync()` may improve the file system performance by reducing the journaling overheads, but the file system may be unreliable when the system crashes. The Research on Android, Tizen, and PC platforms show that the ratio of synchronous writes over all write I/Os are higher than that of buffered writes [24, 17, 19, 27]. The works [27, 17] also show that `fsync()` are prevalent in database workloads and the number of `fsync()` calls outnumber the `fdatasync()` calls.

2.5 atime Update Policy

Since any read to a file updates `atime` that is the last access time, it generates a significant amount of I/Os and slows down the performance [10, 16]. However, not all environments and files have to have a precise `atime`. For example, NFS mounted with `noatime` option reduces the number of requests to the NFS server because the client does not have to fetch latest `atime` update from the server. There are a number of options such as `noatime`, `nodirtime`, `relatime` [11], and `lazytime` [9] to relax the constraints of `atime` updates. `noatime` and `nodirtime` disable update of `atime` for a file and a directory, respectively. However, permanently disabling `atime` from the system may cause some issues on general applications such as mail server. `relatime` is a more flexible solution for such problem in that `atime` gets updated when `mtime` and `ctime` of a file is newer than `atime` or updated `atime` is older than a defined interval, i.e., 1 day by default. Recently, ext4 introduced `lazytime` mount option to store the updates on `atime` on the in-memory inode, and non-time related changes on inode and `fsync()` call from user-space updates the on-disk timestamps. Since users have enough options to reduce the I/O overhead from using `atime`, we focus on optimizing updating policy of `mtime`/`ctime` timestamps.

3. EFFECT OF TIMER INTERRUPT ON FILE SYSTEM

Recent research works show that the portion of small random writes followed by `fsync()` in total I/Os is very high, yet synchronous operations in such workload slow down the storage performance [17, 19, 27, 16]. In such workloads, `atime` and `mtime` updates add to the overhead. Fig. 2 shows the result of 4 KB random write followed by `fsync()` while varying the timer interrupt intervals to 1 msec, 4 msec, and to 10 msec on PC and mobile (Nexus 5). Both platforms show the increase in IOPS because the number and the volume of writes, especially, to a journal device are reduced. Since all random writes overwrite the existing data blocks, the number of blocks and their addresses are not changed. Only metadata changed in this workload is timestamps of the file [10, 11, 9]. For example, one is trying to overwrite 400 KB of data using 4 KB random write followed by `fsync()`. In such workload `mtime` of the file gets updated 100 times; the total of 100×12 KB is written to the journal and 100×4 KB of data is written to the storage, which sums up to 1600

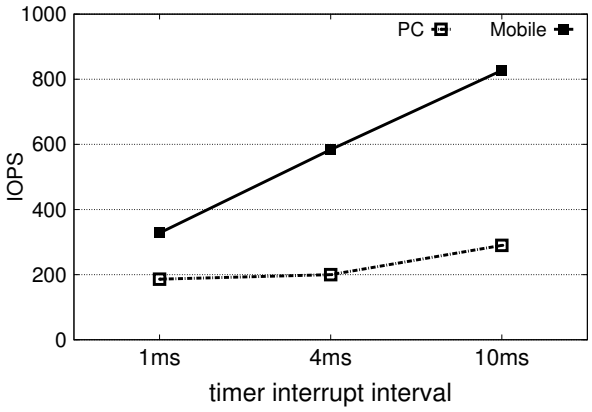


Figure 2: Effect of timer interrupt on file system

KB. However, when the timer interrupt interval is set to 10 msec, the write requests within the interval does not see the `mtime` changes; thus, the inode is not journaled as frequently as when the timer interrupt is set to 1 msec.

Upon receiving `write()` system call, `generic_file_aio_write()` writes data to the file. Then, `file_update_time()` is called to update `mtime` and `ctime` of an inode. The current time of file system acquired from `current_fs_time()` is compared to `mtime` and `ctime` on the inode, if they are different then it is marked for synchronization. Then, file system updates the current time to timestamps on the inode, and the page containing the updates is marked dirty. Finally, the dirty page is flushed with `fsync()`.

Fig. 3 illustrates the behavior of a write followed by `fsync()` under different timer interrupt conditions. Write request at t_1 updates data and metadata including the inode in the page cache and marks them dirty. Each downward arrows on the timer interrupt axis indicate the timer interrupt event, and arrows from t_1 to t_6 indicate the kernel time at each timer interrupt event. Circled D_n and t_n in memory axis indicates the updated data and timestamps at timer interrupt at t_n. If the write is for a new file, then data and various metadata including allocation information and timestamps get updated. On the other hand, if the write is for an existing file and all the blocks are already allocated, then only data and `mtime`/`ctime` gets updated. Since the mechanism to update `mtime` and ctime is the same, we describe the rest with only `mtime` unless otherwise noted. Since the kernel time and the timestamp on the inode is different at each timer interrupt in Fig. 3(a), `mtime` is marked dirty and `fsync()` call flushes the changes to the storage.

Since the kernel time increases discretely at each tick, if the storage system is fast enough to process several write requests within a tick, then the metadata updates are not journaled for the write requests. Fig. 3(b) illustrates coarse-grained timer interrupt where multiples of write followed by `fsync()` requests are processed within a tick. Upon completion of the second write request for updating D_2, file system compares the `current_fs_time()` with `mtime` which holds t1, and finds out that `mtime` is not changed. Thus, only the data is flushed to the storage reducing the journaling overhead with faster response time. However, the coarse-grained timer interrupt is a partial solution for lowering the journal overhead because manipulating the timer interrupt interval affects not only the file system but the entire system.

Algorithm 1 Coarse-grained `mtime` update

```
1: procedure COARSE-GRAINED MTIME(fs_time, interval)
2: // Update the mtime of a file
3:     now ← fs_time − (fs_time mod interval )
4: // current time 'now' is updated at coarse-grained fs_time interval
5:     if mtime ≠ now then
6:         dirty ← 1 // Set dirty to 1
7:     end if
8:     if dirty = 1 then
9:         mtime ← now // Set mtime to 'now'
10:    end if
11: end procedure
```

4. COARSE GRAIN TIMESTAMP UPDATE INTERVAL

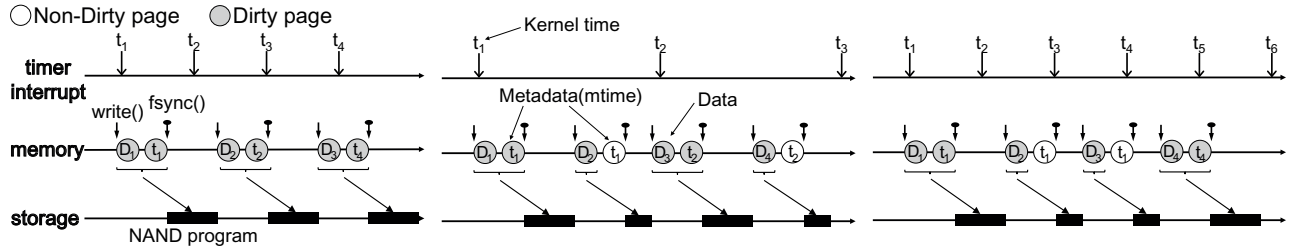
4.1 Coarse-grained `mtime` Update Policy

The conventional file system have introduced various options to mitigate a significant amount of disk I/Os generated by `atime` update; however, `mtime` and `ctime` timestamps have no other options but to rely on kernel timer interrupt interval. In this paper, we propose to use Fig. 3(c) to let the system use timer interrupt interval but relax the update constraints of `mtime` and `ctime`. We add the coarse-grained `mtime` update mount option called `mtime_update` to ext4 file system, which lets a user to choose the update interval to work with various workloads along with other file system mount options. If a user is already using a system with coarse-grained `mtime` update policy and decides not to use it, then there is `un_mtime_update` option to disable it. Since the storage of a system and the workload vary depending on the needs of a user, there is no one-fits-all solution for the right `mtime` interval value. Thus, we let the user choose the `mtime` update interval. It can have an integer value from 1 to 1000 where the unit is msec, and the default value for the `mtime` is 100 msec.

Even if a user chooses to have the finest `mtime` interval, the actual `mtime` interval cannot be finer than the timer interrupt interval of the kernel, which is decided at the kernel compile time with `HZ` value. For example, a user has timer interrupt interval of 10 msec and mounts a file system with `mtime` update interval of 5 msec, then the `mtime` update is bounded by the timer interrupt interval which is 10 msec. The described example case is same as the case shown in Fig. 3(b). The system does not return an error because the number of I/O operations in between the timer interrupt interval is dependent on the performance of the storage.

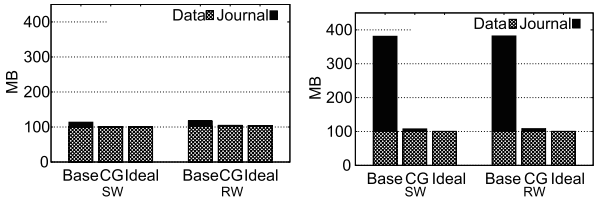
4.2 Coarse-grained `mtime` Algorithm

To enable coarse-grained `mtime` update feature, we modified `file_update_time()` function. Algorithm 1 shows the pseudo-code for coarse-grained `mtime` update scheme. `in-`



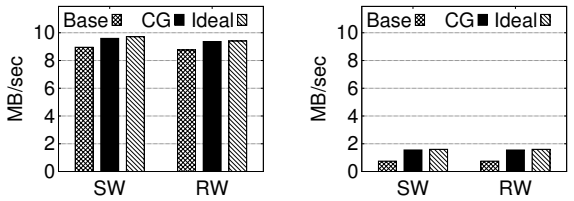
(a) Fine grained timer interrupt (b) Coarse-grained timer interrupt (c) Coarse-grained `mtime` update

Figure 3: Effect of timer interrupt on write followed by `fsync()`



(a) Mobile Device (b) Desktop PC

Figure 4: Volume (File size: 100 MByte, IO size: 4 KByte, Base: default `mtime` update interval, CG: Coarse-grained `mtime` update interval, Ideal: `fdatasync()` with default `mtime` update interval)



(a) Mobile Device (b) Desktop PC

Figure 5: Performance (File size: 100 MByte, IO size: 4 KByte, Base: default `mtime` update interval, CG: Coarse-grained `mtime` update interval, Ideal: `fdatasync()` with default `mtime` update interval)

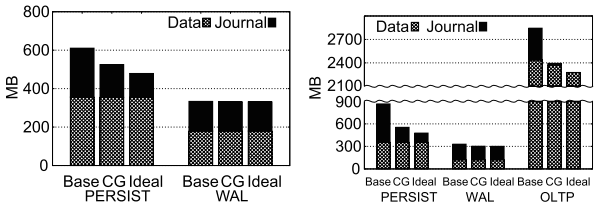
`terval` in line 3 represents timestamp update interval. To convert the unit of user-provided update interval value from msec to nsec, we multiply the value by 10^6. The current implementation of `mtime` update scheme exploits `fs_time`, return value of `current_fs_time()`, and sets the file system time to the current time variable `now`. In coarse-grained `mtime` update scheme, the result of *current timestamp - (current timestamp* mod *interval)* is stored to the variable `now`. For example, when a user sets the update interval to 100 msec, variable `interval` holds 100×10^6. Assuming that `fs_time` returns timestamp of 989,999,960, variable `now` stores 900,000,000 as result of deducting `fs_time` mod `interval` from the current time. `mtime` in line 5 represents the latest modification time in the inode. If `mtime` and `now` is not equal, then sets the dirty mark and updates the `mtime` to `now`. If `mtime` and `now` is the same, timestamp in the inode does not get updated. Since we are not changing the data type of `mtime`, enabling and disabling the coarse-grained mtime update policy has no side effects.

5. EVALUATION

We implement coarse-grained `mtime` update scheme on Linux kernel 3.10.61 and also ported to Galaxy s6 which uses Android 5.0.2 (Lollipop). Experiments on a PC are performed in the following environment: Intel Core i7-4790, DDR3 4 GB DRAM, and Samsung SSD 850 pro 128 GB. Experiments on mobile environment are performed Galaxy S6, and its specification is as follows: Samsung Exynos 7 Octa 7420 (4×Cortex-A57 2.1 GHz, 4×Cortex-A53 1.5GHz), LPDDR4 3 GB DRAM, and UFS 2.0 32 GB storage. Since the normal usage of the mobile device exploits on-demand policy with Dynamic Voltage and Frequency Scaling (DVFS) to save the power [28, 4], we explicitly made all the cores active and set the performance policy to let the cores run in maximum performance of 2.1 Ghz and 1.5Ghz for big and little cores, respectively. We ran separate runs of experiments to measure the effect of `mtime` update scheme on I/O volume and performance. All experiments are repeated ten times on both platforms and the results shown in this paper are the average. We set the `mtime` and `ctime` update interval to 100 msec. The default kernel timer interrupt for the mobile device is set to 100 HZ (10 msec), but PC has 250 HZ (4 msec). All experiments are compared with Base (the default `mtime` update interval that is 10 msec for mobile, 4 msec for Desktop) and the Ideal (using `fdatasync()` in default `mtime` update interval). The performance result of coarse-grained `mtime` update interval is denoted as CG in the figures.

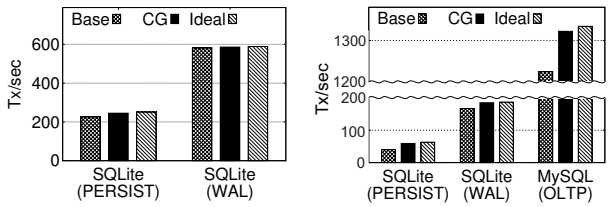
5.1 Micro Benchmark

After creating a 100 MB file with 4 KB I/Os, we performed 4 KB random write followed by `fsync()` on the file and measured the I/O volume. Fig. 4(a) shows the I/O volume observed on performing a sequential and random write on the mobile device. The result shows that the volume of the journal is about 12% of the total I/O volume for both sequential and random write workloads. When the timer interrupt interval is increased to 100 msec, 90.4% of the volume of metadata is decreased for sequential and 87.1% is decreased for the random workload. Compared to the I/O volume observed with `fdatasync()`, `fsync()` workload with 100 msec update interval has only 2% more overhead. Fig. 4(b) shows the result on PC. The I/O volume for file system journal is about 73.8% of the total I/O volume observed on sequen-



(a) Mobile Device (b) Desktop PC

Figure 6: Volume of DBMS Workloads (Base: default `mtime` update interval, CG: Coarse-grained `mtime` update interval, Ideal: `fdatasync()` with default `mtime` update interval)



(a) Mobile Device (b) Desktop PC

Figure 7: Average Performance of DBMS Workloads (Base: default `mtime` update interval, CG: Coarse-grained `mtime` update interval, Ideal: `fdatasync()` with default `mtime` update interval)

tial and random workload. When we set the `mtime` update interval to 100 msec, about 97% of the journal I/O volume is reduced. The journal I/O volume for the mobile device is much less than that of PC because the default kernel timer interrupt interval of mobiles is 10 msec and desktop is 4 msec.

Fig. 5 shows the performance on sequential and random workload. The performance gains from increasing the timer update interval to 100 msec in the mobile and PC is about 7% and 107% for both sequential and random workloads, respectively. The performance of using `fdatasync()` shows 1% and 6% better than that of the proposed scheme on the mobile and PC, respectively.

5.2 Application Benchmark

We use Mobibench [23] and Sysbench [21] to measure the performance of two widely used DBMS, SQLite and MySQL [31], respectively. We compare the transactions per second of different timer interrupt intervals on PERSIST and WAL SQLite journal mode while issuing 10,000 insert queries using Mobibench. Althought we measure the performance of insert, update, and delete workload on SQLite, we only show the performance of insert workload because they were similar to each other. We use Sysbench to test the performance of MySQL with OLTP workload [39] using eight threads to generate 200K queries on a table of 1 Million rows.

Fig. 6 shows the result of I/O volume measured while performing inserts in SQLite. The update interval for coarse-grained `mtime` update scheme is set to 100 msec for `fsync()` and 10 msec for `fdatasync()`. It shows that using coarse-grained `mtime` update in PERSIST mode on both of the platforms reduces the I/O volume significantly. The I/O volume for the journal area on mobile and PC is reduced by about 33.1% and 61.0% in using `fsync()` with coarse-grained update interval, respectively. Note that using `fdatasync()` in default `mtime` update interval decreases 51.3% the journal I/O volume. Since WAL mode uses less number of `fsync()`, the observed reduction in the I/O volume is small; it reduces 1.6% and 13.3% of the journal I/O volume on mobile and PC, respectively. As the effect of reduced journal I/O volume, the performance of coarse-grained `mtime` update, which is shown in Fig. 7, is about 8.4% and 4.1% better than the case with `fsync()` using the default `mtime` interval on mobile and PC, respectively.

Fig. 6(b) and Fig. 7(b) shows the I/O volume and the performance of using coarse-grained `mtime` update in OLTP workload, respectively. Using the proposed scheme reduces about 94.1% of the I/O volume and increases about 7.9% of the performance.

6. CONCLUSION

In this paper, we examined the effect of the kernel timer interrupt on the performance of pre-allocated write followed by `fsync()` to the file system which frequently updates timestamp metadata to journaling area. We present coarse-grain `mtime` update design with variable update interval to increase the synchronous write performance. With flexible `mtime` update constraints, the journaling overhead in the system is dramatically reduced in SQLite PERSIST mode. We find that the performance increased by 8.4% and the volume decreased by 33.1% compared to the default 10 msec `mtime` update interval in Android smartphone. In the case of OLTP workload in PC, coarse-grained `mtime` update interval increases the performance by 7.9% and decreases the journal I/O volume by more than 94.1%. We believe that there are two contributions of this study. First, our examine is applicable to explain root cause for unexpected performance and abnormal I/O pattern on file system. Second, the DBMS performance improvement on various devices at minimal modifying cost.

7. ACKNOWLEDGMENTS

This work is supported by IT R&D program MKE/KEIT (No. 10041608, Embedded System Software for New-memory based Smart Device), and by the MSIP(Ministry of Science, ICT&Future Planning), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2016-H8501-16-1006) supervised by the IITP(Institute for Information&communications Technology Promotion).

8. REFERENCES

[1] Ext4 Documentation. https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[2] Real time clock (rtc) drivers for linux.

[3] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of linux. In *Proc. of the IEEE RTAS*, San Jose, CA, USA, Sep 2002.

[4] Y. Ahn and K. S. Chung. User-centric Power Management for Embedded CPUs using CPU Bandwidth Control. *IEEE Transactions on Mobile Computing(TMC)*, PP(99):1–1, Oct 2015.

[5] P. Alcorn. Samsung Releases New 12 Gb/s SAS, M.2, AIC And 2.5" NVMe SSDs: 1 Million IOPS, Up To 15.63 TB, 2013. http://www.tomsitpro.com/articles/samsungsm953-pm1725-pm1633-pm1633a,1-2805.html.

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

[7] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proc. of the ACM SOSP '13*, Farmington, PA, USA, Nov 2013.

[8] D. Cobb and A. Huffman. NVM express and the PCI express SSD Revolution. In *Proc. of Intel Developer Forum*, San Francisco, CA, USA, 2012.

[9] J. Corbet. lazytime. https://lwn.net/Articles/621046/.

[10] J. Corbet. Once upon atime. https://lwn.net/Articles/244829/.

[11] J. Cortbet. relatime. http://lwn.net/Articles/326471/.

[12] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation Techniques for Main Memory Database Systems. In *Proc. of the ACM SIGMOD '84*, Boston, MA, USA, Jun 1984.

[13] F. Douglis and J. Ousterhout. Log-structured file systems. In *COMPCON Spring'89. Thirty-Fourth IEEE Computer Society International Conference: Intellectual Leverage, Digest of Papers.*, pages 124–129. IEEE, 1989.

[14] Gartner. Smartphone sales surpassed one billion units in 2014. http://www.gartner.com/newsroom/id/2996817.

[15] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of "big data" on cloud computing: Review and open research issues. *ELSEVIER Information Systems*, 47:98 – 115, 2015.

[16] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, et al. BetrFS: A right-optimized write-optimized file system. In *Proc. of the USENIX FAST '15*, Santa Clara, CA, USA, Feb. 2015.

[17] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In *Proc. of the USENIX ATC '13*, San Jose, CA, USA, Jun 2013.

[18] kernel/Kconfig.hz v3.0 rc7. Timer frequency.

[19] M. Kim and Y. Won. IO characteristics of modern smartphone platform: Android vs. Tizen. In *Proc. of the IEEE IWCMC '15*, Dubrovnik, Croatia, Aug 2015.

[20] W.-H. Kim, B. Nam, D. Park, and Y. Won. Resolving journaling of journal anomaly in android I/O: multi-version B-tree with lazy split. In *Proc. of the USENIX FAST '14*, Santa Clara, CA, USA, Feb 2014.

[21] A. Kopytov. SysBench manual, 2004. http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf.

[22] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A new file system for flash storage. In *Proc. of the USENIX FAST '15*, Santa Clara, CA, USA, Feb. 2015.

[23] K. Lee. Mobile Benchmark Tool (MOBIBENCH). https://github.com/ESOS-Lab/Mobibench.

[24] K. Lee and Y. Won. Smart Layers and Dumb Result: IO Characterization of an Android-Based Smartphone. In *Proc. of the ACM EMSOFT '12*, Tampere, Finland, Oct 2012.

[25] W. Lee, K. Lee, H. Son, W. Kim, B. Nam, and Y. Won. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proc. of the USENIX ATC '15*, Santa Clara, CA, USA, July 2015.

[26] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proc. of the USENIX ATC '08*, Boston, MA, USA, June 2008.

[27] E. Lim, S. Lee, and Y. Won. Androtrace: Framework for tracing and analyzing IOs on Android. In *Proc. of the ACM INFLOW'15*, Monterey, CA, USA, Oct 2015.

[28] Y. Liu, H. Yang, R. P. Dick, H. Wang, and L. Shang. Thermal vs Energy Optimization for DVFS-Enabled Processors in Embedded Systems. In *Proc. of the IEEE ISQED'07*, Santa Clara, CA, USA, March 2007.

[29] 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*, volume 2, pages 21–33. Citeseer, 2007.

[30] T. P. Morgan. Flashtec NVRAM Does 15 Million IOPS At Sub-Microsecond Latency, 2014. http://www.enterprisetech.com/2014/08/06/flashtecnvram-15-million-iops-sub-microsecond-latency/.

[31] A. MySQL. Mysql 5.1 reference manual. *Sun Microsystems*, 2007.

[32] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[33] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *ACM Transactions on Computer Systems (TOCS)*, volume 10, pages 26–52, 1992.

[34] K. Shen, S. Park, and M. Zhu. Journaling of journal is (almost) free. In *Proc. of the USENIX FAST '14*, Santa Clara, CA, USA, Feb. 2014.

[35] C. Shulyupin. The Tick Rate: HZ. http://www.makelinux.net/books/lkd2/ch10lev1sec2.

[36] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proc. of the USENIX ATC '96*, San Diego, CA, USA, Jan. 1996.

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

[38] S. C. Tweedie. Journaling the linux ext2fs filesystem. In *Proc. of The Fourth Annual Linux Expo*, 1998.

[39] W. Vogels. File system usage in Windows NT 4.0. *SIGOPS Operating System Review*, 33(5):93–109, Dec. 1999.