FAST SYNCHRONOUS FILE WRITING FOR FLASH STORAGE IN ANDROID DEVICES

Author

Po-Han Sung

Supervisor

Li-Pin Chang

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT
NATIONAL CHIAO TUNG UNIVERSITY
HSINCHU, TAIWAN
JULY 2013

Table of Contents

List of Tables

List of Figures

Abstract

Using hand-held devices is now a part of peoples' lives. Android is one among the most popular operating systems for hand-held devices. Recent research pointed out that application data manipulation in Android systems involves a great deal of file synching operations, which poses a severe negative impact to the write performance. This work introduces an efficient implementation of file synching operations, called *eager synching*, for Android devices equipped with flash storage. The basic idea is simple: when a file is being synched, eager synching writes only the dirty data associated with the file in a sequential log space, reducing the randomness in the write pattern of file synching. Because sequential write is significantly faster than random write in flash storage, this design can effectively reduce the latency of file synching operations. We implemented our eager synching in the ext4 file system, and the experimental results show that the write throughput of real Android applications and storage benchmarks were improved by up to 50%, and the largest improvement in write sequentiality was 55%.

Acknowledgements

I will never forget the years of my college life. Without the help of my advisor, Prof. Li-Pin Chang, I would not accomplish my research. In spite of going astray on research many times, Prof. Chang still patiently gave me a lot of advices and helped me polish my thesis, so I would like to express my greatest appreciation to Prof. Chang, thank you.

These years are filled with happiness, sorrow, hesitation, and gratefulness. Thanks Ying-Jie Li, Yi-Cheng Wu, Wen-Hui Lin, and Wei-Han Wang, all of you are my learning example on research, and thank all your concerning. Thanks Yi-Kang Chang, Tung-Yang Chou, Wen-Ping Li, and Chen-Yi Wen, my past lab mate, without your encourage, discussions with you, my research life would be colorless. Thanks Sheng-Min Huang, Chau-Yuan Mao, Chen-Yu Hung, and Ting-Chieh Huang, It's nice to work with you after my postponed graduation, it would be desperate if I were to fighting alone. Wish you all the best. Thanks Yu-Chia Chang for your accommodation since the last summer vacation. It is a good experience to discuss with you. Don't be frustrated, be positive. You will graduate soon. Thanks Ming-Chi Yan, Pin-Xue Lai, Guan-Chun Chen, Po-Hong Chen, Chun-Yu Lin, Yu-Syun Liou, and Chia-Hsiang Cheng, it is happy with your accompany when doing research, making me not boring too much. Wish you all have a happy and fulfilling research life.

Finally, but most importantly, I want to thanks my family, without their support, I could not step this far. Sorry for not going home for several months in order to keep my research progress on track. Now I'm to going home!

Chapter 1

Introduction

Hand-held devices, including smartphones, tablets, and wearable computers, have been a great success in the recent years. They are taking more and more of time in people's daily life. Android, the Google's open-source approach, is based on the Linux kernel and has been one among the most popular operating systems for hand-held devices. According to Google, by April 2013, there were 1.5 millions Android devices being activated per day [1].

The nature of hand-held devices imposes many challenges on the design of their storage sub-system. The storage must be resistent to shock, efficient in read and write, and conservative in power consumption. Flash storage edges over the conventional hard disks and other non-volatile memories because it better fits the design considerations described above. Since Android version 2.3, standard Android systems adopt the ext4 file system (ext4 for short) over a flash storage device. Because ext4 is a file system for generic block devices not raw flash memory, the underlying flash storage devices adopt a firmware layer, called Flash-Translation Layer (FTL), to emulate block devices using flash memory [2, 3].

Most of the Android applications do not directly interact with the file system. Instead, Android adds a software layer, i.e., SQLite, between the applications and the file system. SQLite is a light-weight, self-contained, and transactional database engine. It enables simple and reliable data manipulation at the application level. To guarantee atomicity and durability of database operations, SQLite frequently calls fsync() to ensure that all the modified data have been written to the storage device. We observed that, during a five-minute interval of using the Google's Gmail client to browse emails, 78% out of all the block-write requests were synchronous, i.e., originated from fsync() calls. Recent research had also report similar observations [4, 5].

File synching operations (i.e., fsync() calls) have many negative performance impacts: First, a call to fsync() is blocked until the file system completely writes all the modified user data and metadata associated with the file being synched to the flash storage [6]. Thus, the file system cannot hide the storage write latency from applications. On Google's Nexus 7 tablet, we observed that writing and then synching a file can be up to 60 times slower than writing the file asynchronously. Second, fsync() calls are synchronous, the page cache and the I/O scheduler cannot deliberately delay write requests for write buffering and request merging. In other words, frequent fsync() calls will neutralize the efficacy of page caching and I/O scheduling, resulting in many tiny write requests and a highly random write pattern.

Even though flash memory is random access, sequential write in flash storage is significantly faster than random write [2, 3, 7, 8]. Flash memory is a kind of erase-before-write medium, and thus garbage collection is necessary to recycle available space. The smallest unit for flash erasure is called a block. Under random write, alive (valid) and dead (invalidated) data are mixed inside of blocks, and the overhead of moving valid data out of a block before erasing the block for garbage collection is very high. Because the write pattern under frequent calls to fsync() is highly random, the block I/O operations originated from the fsync() calls suffers from long write latencies. In contrast, sequential writes can sequentially invalidate all the alive data in blocks, relieving garbage collection of a high overhead of data copying. On the Android tablet mentioned earlier, we observed that the average throughput of sequential write was 27 times higher than that of random write.

To alleviate the performance impacts of file synching operations, prior work suggested storing the SQLite database files in non-volatile RAM like PRAM [4]. However, as non-volatile memory is not yet a cost-efficient option for the main memory, we decided to take a different approach. This work investigates an efficient implementation of file synching operations for flash storage, called *eager synching*. Our basic idea is simple: the file system writes as sequential as possible when synching a file. Specifically, when a file is being synched, the file system writes the user data and the metadata associated with the file to a sequential log space in the flash storage. As soon as the fast sequential writing is done, all changes on the file being synched becomes permeant. The rationale behind this design are twofold: First, because sequential write in flash storage is very fast, the sequential-writing strategy can significantly reduce the latencies of fsync() calls. Second, numerous fsync() calls produce many tiny and random write requests. Collecting the tiny write requests in a long and sequential write burst can not only reduce the frequency of down-calls to the block layer but also reduce the randomness in the write pattern.

There are two design challenges for our eager synching. First, even though eager synching writes data in the sequential log space for synching files, the logged data must later be copied back to the file-system image at proper timings. Thus, it is important to control the copy-back overhead by reducing the amount of data written to the log space. Second, the copy-back operations between the log space and the file-system image must not corrupt the file system. It is non-trivial because the modification operations on different files may write to the same metadata. For example, in the ext4 file system, the creations of two files may allocate two new inodes and update the same inode-allocation bitmap. When either one of the two files is being synched, including the bitmap, eager synching must write both the two new inodes to the log space, because the bitmap indicates that two new inodes have been allocated.

The design of our eager synching is generic and not specific to any file system. All that eager synching requires is a sequential log space in the flash storage. We implemented eager synching in the ext4 file system, and conducted a series of experiments on Google's Nexus 7 tablet. The Android version of the device was 4.2, the file system was ext4, and the flash storage was an embedded MultiMediaCard (eMMC). We compared our eager synching against the original ext4 synching method. Our experiments used the workloads of several typical Android applications, including the Gmail client, the Facebook client, the Chrome browser, and the file-accessing benchmark and the SQLite benchmark in AndroBench[1]. Our results show that eager synching improved the average write throughput by 25%, and the

[1]`http://www.androbench.org/wiki/AndroBench`

best improvement even reached 100%. The performance improvement is mainly attributed to the sequential write pattern produced by eager synching and the properly controlled copy-back overhead.

The rest of this paper is organized as follows: Section 2 summarizes related work. Section 3 explains the file and storage access behaviors in Android systems. Section 4 presents the design and implementation of our eager syncing. Section 6 reports our experimental results, and Section 7 concludes this paper.

Chapter 2

Related Work

As mentioned before, SQLite are pervasively utilized over Android application, this leads to the consequence that file synchronization happens more often. Lee et al. [5] found that there are many small and synchrnous writes on particular partition (/data partition) of Android storage. Kim et al. [4] also found that aside from SQLite doing synchronous I/O, ext4 file system implements file synchronization by forcing transaction committing, writing all dirty data down to flash storage. These phenomenon causes system performance low, because flash storage are not well at handling small and random writes.

To alleviate the impact caused by frequent synchronous writes, there are many solutions proposed. ZFS [9] adopts an external storage for accommodating journaled data to speed up performance of file synchronization. However, this does not solve the problem of small and random writes, since those journaled data are to be replayed after. Wang et al. [10] proposed a method for fast file synching by writing small and synchronous data to free space nearest to the current disk head position. This is an implementation of eager writing, nevertheless it targeted traditional hard disk, not suited for flash storage, and flash storage has no seek time, but is sensitive to write patterns. Chiueh et al. [11] also proposed a method for fast synching by using external disk to write synchronous data, and take into consideration the performance model of traditional hard disk. Again this is not suitable for flash storage. Margaret [12] propsed mechanisms used for databases recovery, and one of the mechanism are also suitable for speeding up databases performance by logging data specially, whose concept is similar to eager logging. Still, this is designed for database systems, not for general file system.

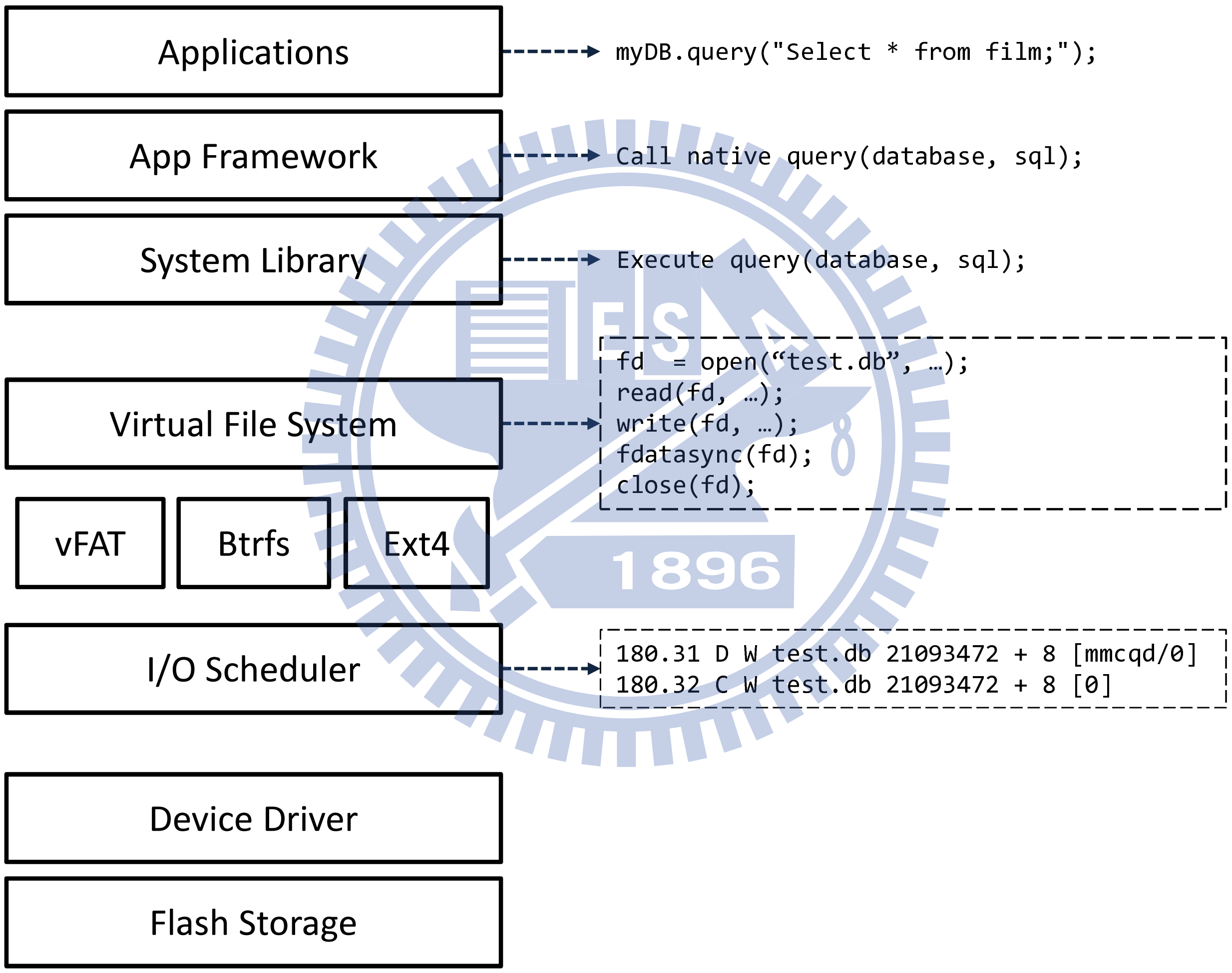
Various implementation of eager writing are discussed in [13], those who are interested in eager writing can refer to this paper.

Chapter 3

File-Accessing Behaviors In Android Systems

3.1 Related Software Components

Figure 3.1 shows the hierarchy of Android system components involved in data access. Android applications do not directly use the file system services. Instead, there is a lightweight database layer that lies between applications and file systems and provides transactional data manipulation. Android applications invoke the query-based database Java methods, the calls are mapped to the corresponding application interfaces (APIs) in the Android application framework and then the system library. The system library then translates the query string into a series of calls to the file-system services. The down-calls are then handled by the kernel file-system drivers like ext4 and vFAT. The data accessed during the file operations are cached in the page cache layer in terms of virtual-memory pages. Later on, at proper timing, the file system or the kernel page flushing thread will write the dirty (i.e., modified) pages to the underlying block device. The page writes are encapsulated as block I/O requests (bio), which will then be merged and re-ordered by the I/O scheduler for starvation prevention and seek-overhead optimization. Finally, the requests arrive at the flash storage, and the FTL inside of the flash storage fulfills the requests using flash



```
myDB.query("Select * from film;");
```

```
Call native query(database, sql);
```

```
Execute query(database, sql);
```

```
fd  = open("test.db", …);
read(fd, …);
write(fd, …);
fdatasync(fd);
close(fd);
```

```
180.31 D W test.db 21093472 + 8 [mmcqd/0]
180.32 C W test.db 21093472 + 8 [0]
```

Figure 3.1: Software hierarchy of data accessing in Android applications

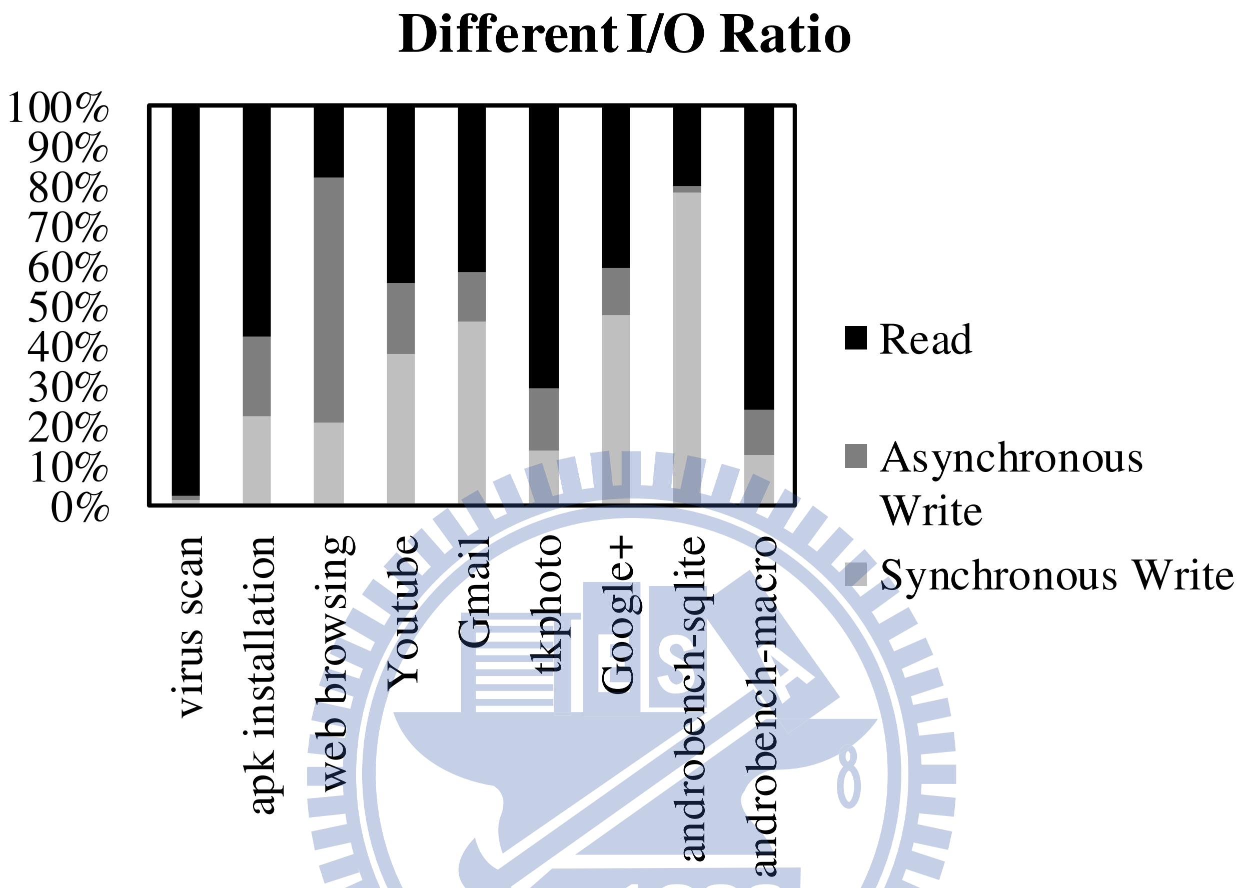


Figure 3.2: The distribution of block I/Os produced by file reading, asynchronous file writing, and synchronous file writing under various Android applications.

operations.

Applications write files asynchronously to take advantage of page caching and I/O scheduling. We say that a piece of data becomes *durable* if it has been written the storage. The completion of a asynchronous file writing operation does not guarantee the durability of the written data. A series of asynchronous file writing operations must be followed by a file synching operations to make all the dirty data associated with the file durable. The Linux kernel supports two kinds of file synching operations, i.e., fsync() and fdatasync(). While fsync() writes the user data and metadata associated with the file being synched to the storage, fdatasync() writes only the user data. We do not explicitly distinguish the two operations as they are handed by the file system in similar ways.

An asynchronous fwrite() call returns as soon as the new data arrive at the page cache.

On Nexus 7, we measured that the average latency of writing 4 KB of data using asynchronous fwrite() is 0.032 ms. In contrast, a fsync() call is blocked until all the dirty data associated with the file being synched have been written to the storage. The latency of a fsync() call that immediately follows a fwrite() call the same as above is 14.446 milli-seconds in average and 0.44 seconds in the worst case. Such large delays can easily be noticed by users. A block I/O request is synchronous if it is originated from a file synching operation. We used the *blktrace* tool to collect the block-layer traces of several typical Android applications. As Figure 3.2 shows, a large portion of the block write requests is synchronous. In particular, for the Gmail case, 78% out of all the block writes were synchronous, and 60% among the synchronous writes were contributed by file synching operations on the SQLite database files. Summing up, speeding up file synching operations can greatly benefit the data manipulation performance and user experience in Android systems.

3.2 File Synching in Ext4 File System

This section describes the original implementation of fsync() in ext4, and points out the possible performance issues. Ext4 is a journaling file system, and it flushes dirty pages to the block device in terms of *transactions*. A transaction is a collection of dirty pages modified by a set of completed file operations. Because none of the dirty pages in a transaction is associated with an undergoing file operation, writing transactions to the block device as a whole can preserve the integrity of the file-system image in the storage. Ext4 periodically writes (appends) transactions to a sequential log space in the storage, called the *journal*. The journal is not the final destination of transactions. At proper timings, ext4 will copy the data encapsulated in transactions from the journal back to the file-system image. If power failures occur during the copy-back operation, after power restores, ext4 will scan the journal and re-do the interrupted copy-back operation.

When an application calls fsync() to synch a file, ext4 checks the status of the transaction that the file being synched is associated with. If the status of the transaction is locked, i.e., the transaction is not associated with any ongoing file operations, then ext4 writes

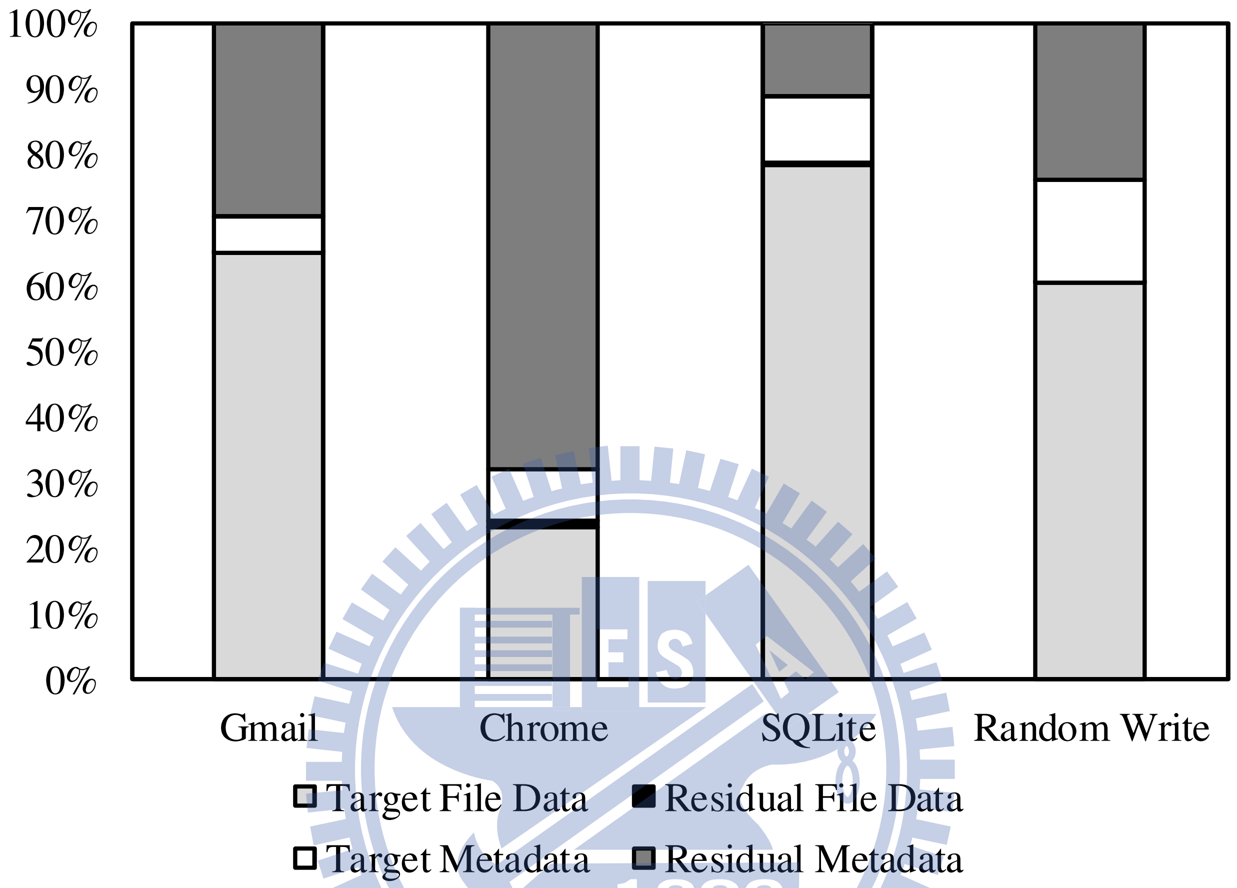


Figure 3.3: The distribution of block write requests contributed by the file being synched and the other files. Ext4 wrote many metadata not associated with the files being synched.

the entire transaction to the journal, waits until the transaction is copied back to the file-system image, and then returns to the application. If the status of the transaction is not locked and is still associated with ongoing file operations, ext4 waits until the ongoing file operations complete, marks the transaction locked, and then perform the the procedure for a locked transaction mentioned above.

The implementation of fsync() in ext4 has two potential performance issues. First, a transaction contains dirty data from many files, and writing the entire transaction to synch one file is not efficient. We analyzed the association between block write requests and files for different Android applications. Figure 3.3 shows that many of the block writes went to the metadata not associated with the files being synched. In particular, xx% among the metadata block writes of the Chrome browser are not associated with the files being synched. Ruling out the dirty data not associated with the files being synched can reduce the count of I/O requests to the flash storage.

Second, ext4 flushes all dirty data in a transaction upon every fsync() call. Thus, with frequent fsync() calls, write operations cannot be deliberately delayed for write buffering and/or request merging. The resultant write pattern can be highly random and contain many tiny requests. To quantify the sequentiality of a write pattern, let the *sequential ratio* be the percentage of the total sector writes contributed by write bursts longer than 128 sectors. Adjacent write requests form a long write burst. The sequentiality of the Chrome browser test in Figure 3.3 was xx% and the average write size was yy sectors. If we modify fsync() in ext4 to be an empty handler, then the sequentiality increased to yy% and the write size grew to zz sectors. Writing sequentially for synching files can reduce the write latency in flash storage.

3.3 Flash Management and Performance Characteristics

Flash storage implements a flash translation layer (FTL) in the firmware to emulate flash memory as a standard block device. Flash memory reads/writes and erases in terms of pages and blocks, respectively. A flash page cannot be overwritten unless the block that encompasses the page is erased first. To avoid erasing a block every time when updating a page, the FTL updates page data in a out-of-place fashion, and marks old copies of page data invalid. The FTL maintains a mapping table to translate logical sector numbers into flash memory addresses. Because handling write requests from the host consumes free space in flash memory, when the free space is running low, the FTL must initiate garbage collection to recycle the flash pages occupied by invalid data.

The write latency in flash storage is largely affected by the garbage collection overhead, which is sensitive to the write pattern. Figure 3.4 depicts how garbage collection works. In Figure 3.4(a), the host sequentially writes disk sectors a, b, and c, which are all mapped to flash block B_1. The FTL can simply erase block B_1 to reclaim the flash pages occupied by invalid data. Now consider that the host randomly writes disk sectors a, e, and f, as

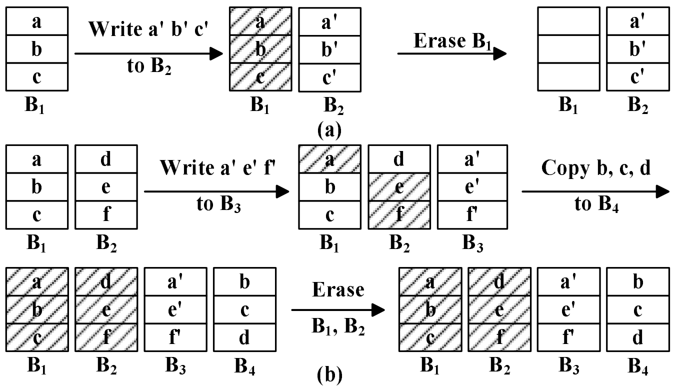


Figure 3.4: The overhead of garbage collection under (a) sequential write and (b) random write. Garbage collection under random write requires extra page copy operations (i.e., copy b, c, and d to B_4).

shown in Figure 3.4(b). Now, before erasing the two blocks B_1 and B_2 to recycle the three pages of stale data, the FTL must copy the three pages of valid data to a new block B_3. Garbage collection uses extra page copy operations under random write compared to under sequential write. On Nexus 7, we tested sequential writing and random writing inside of a 16 MB file, which was sequentially allocated in the flash storage. The file chunk sizes for fwrite() was 4 KB for random writing and 256 KB for sequential writing. Each fwrite() call was followed by a fsync() call. The sequential write throughput was about 27 times higher than that of random write (13.2 MB/s vs. 0.5 MB/s). The result supports our design concept of writing sequentially for fast file synching operations.

Chapter 4

Eager Synching: An Efficient Implementation of File Synching Operations

4.1 Overview

Eager synching is an implementation of fsync(), and eager synching can co-exist with the original fsync() implementation of a file system. When a file is being synched, eager synching sequentially writes the dirty user data and dirty metadata associated with the file to a log space in the flash storage. Eager synching reports complete as soon as the dirty data have been logged. Because the logged data have not been written to the file-system image, they remain dirty in the page cache. Later, at proper timings, the file system or the page cache will submit the dirty data to the block driver of the flash storage. Writing the logged data to the file-system image is carried out in the most efficient manner because write buffering (in the page cache) and request merging (in the I/O scheduler) are still effective.

There are several design challenges behind this simple idea. First, eager synching should reduce the total amount of data written to the log space, because the file system will write the logged data to the file-system image later in background. Because metadata are shared among files, it is not easy to know whether not to write a piece of metadata to the log space can preserve the integrity of the file-system image. Second, eager synching requires a sequential log space for writing, and integrating this logging mechanism with the existing ext4 journaling mechanism avoids creating multiple sequential writing streams in the flash storage. Third, eager synching must correctly recover the all previously synched files after a system crash. The rest of this section will be focused on the three subjects.

4.2 Indivisible Metadata Sets (isets)

Figure 3.3 showed that, when a file is being synched, only a portion of the dirty data are modified by the prior file operations on the file, and eager synching needs not write all the dirty data to the log space. It is then a question how to rule out the dirty data not necessary to make the file being synched durable. File system manage two kinds of data: user data are the contents of files, and metadata are the data structures of file systems. A piece of user data is exclusively owned by a file (specifically, an inode in Linux), and it is easy to identify whether a piece of dirty user data is associated with the file being synched or not. However, many metadata, such as the inode allocation bitmap in ext4, are shared among files. Writing only the dirty metadata modified by the prior operations on the file being synched can possibly corrupt the file-system image in the flash storage.

Figure 4.1 illustrates how the two sets of dirty metadata modified by two different files overlap. Suppose that, to create a new file A, the file system allocates a new inode and updates the inode allocation bitmap. Later on, another new file B is created, and the file system again updates the allocation bitmap and write a new inode for the file. The creations of the two files both modify the allocation bitmap. Now consider that file A is being synched and the file system writes only the allocation bitmap and the inode of file A to the storage. The bitmap in the storage indicates that two inodes have been allocated, but only the inode of file A presents in the storage but the inode of file B is missing. The synching operation corrupts the file system in the storage.

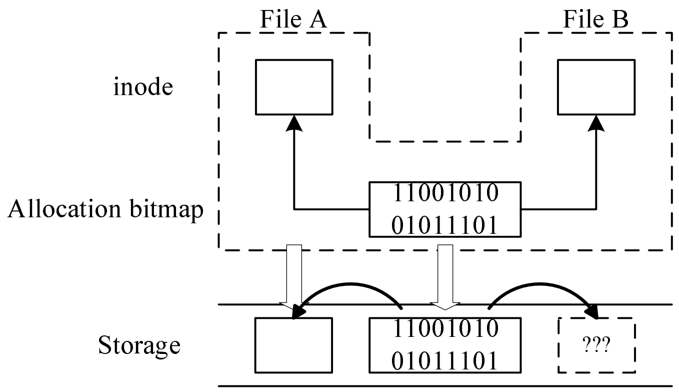


Figure 4.1: Two file creation operations both modify the inode allocation bitmap. Synching file A by writing only its inode and the allocation bitmap causes file-system inconsistency in the storage.

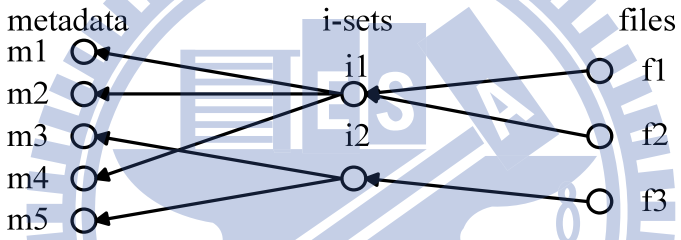


Figure 4.2: The relationships among metadta, isets, and files. The mapping from files to isets is many-to-one, while the mapping from isets to metadata is one-to-many.

To deal with this problem, we propose a new data structure called *indivisible metadata set* (iset for short). A file is associated with only one iset, which contains a collection of dirty metadata that are required to make the changes to the file durable subject to the consistency of the file system in the storage. A piece of dirty metadata is exclusively owned by an iset. Isets describe the relation between files and metadata, and they stay in the main memory and need not to be written to the storage. When a file operation on a file modifies a piece of metadata, then the dirty metadata is added to the iset associated with the file. A new iset is created for the file if the file is not currently associated with one. If a piece of dirty metadata is written back to the storage and marked clean, then the metadata is removed from its iset. If an iset is empty, then it is de-allocated from memory. If two isets have an overlap because the operations on the two files modify the same metadata, then the two isets are merged into a new iset. Figure 4.2 shows the relationships among dirty metadata, isets, and files. For example, the two files in Figure 4.1 are associated with the same iset, which contains a dirty inode allocation bitmap and two dirty inodes.

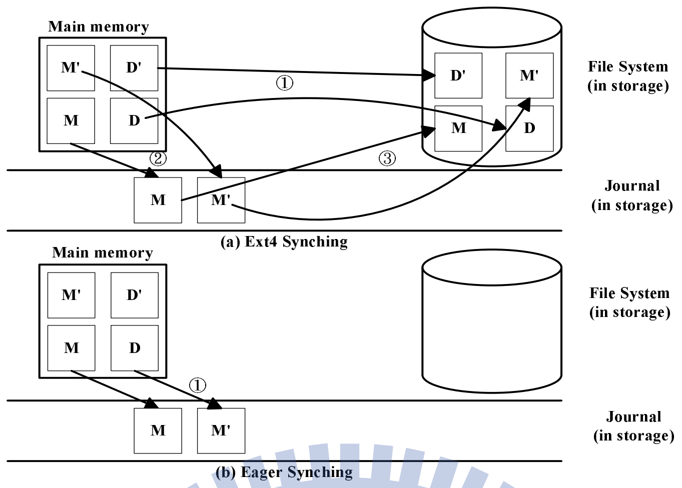


Figure 4.3: The procedures of synching a file using (a) the original ext4 synching and (b) eager synching. M and D stand for the metadata and user data associated with the synched file, respectively, while M' and D' are the metadata and user data not relevant to the synched file, respectively.

To perform fsync() on a file, eager synching will first identifies the iset associated with the file, and then finds all the the dirty metadata in the iset. In Figure 4.2, to sync file f_1, eager synching first finds iset i_1 and the associated metadata m_1, m_2, and m_4. Eager synching will then finds all the dirty user data associated with file f_1. Finally, eager synching writes the three pieces of dirty metadata and the dirty user data to the sequential log space in the flash storage, and then reports completion of the file synching operation.

4.3 Integrating Eager Synching with Ext4 Journaling

Eager synching requires a sequential log space in flash storage for fast file synching. As described in Section 3.2, ext4 also uses a sequential log space for journaling. We propose combining the two log spaces as one to form a sequential write stream in the flash storage. The ext4 journal is a hidden file that is sequentially allocated in the storage space upon disk formatting. Ext4 writes a in-memory transaction into the in-storage journal in terms of *segments*. A segment consists of a mapping table and a series of data blocks. The mapping table indicates the disk sector numbers where the data blocks are mapped to. When ext4 finishes writing the last segment of a transaction, it appends a *commit record* to the last segment to indicate the transaction complete. Because writing the segment data to the journal is out-of-place update, later at proper timings a kernel thread will write the segment data in the oldest transaction to the file-system image. After the write, the oldest transaction is then deleted from the journal. If the system crashes before the segment data are written to the file-system image, for crash recovery, then the file system uses the complete transactions in the journal to redo the writing. Incomplete transactions will be discarded.

Like ext4 journaling, eager synching writes in terms of segments. Figure 4.3 compares the procedures of synching a file using the original ext4 synching and our eager synching. Let the dirty metadata and user data necessary to synch the file be M and D, respectively. Let the dirty user data and metadata not necessary to make the changes on the file durable be M' and D', respectively. Notice that how M and D can be separated from M' and D' has been described in Section 4.2. In Figure 4.3(a)[1], ext4 first writes all the dirty user data D and D' to the file system image, logs all the dirty metadata M and M' in the journal, and then writes the metadata M and M' to the file-system image. In Figure 4.3(b), eager synching writes only M and D to the journal, and then reports completion of the synching operation.

Unlike the original ext4 synching, eager synching does not forcibly commit a transaction on every file synching operation. Thus all the dirty data can stay in the page cache for write buffering and request merging. To be transparent to the ext4 journaling, eager synching does not affect the dirtiness of the data in the page cache. Instead, eager synching maintains its own dirty bit to indicate whether a piece of dirty data has been written to the storage by eager synching (*sync-clean*) or not (*sync-dirty*). When a piece of data is modified, it becomes dirty and sync-dirty. Eager synching writes only the sync-dirty data, while ext4 journaling writes the dirty data. This design prevents eager synching from repeatedly writing the same data to the journal.

[1]This example is based on the default journaling mode, the ordered mode.

4.4 File-Synching Cost Analysis

This section compares the write costs of the original ext4 synching and our eager synching. Let us first focus on the total amount of data written to the flash storage. Consider the example in Figure 4.3. Let the file system perform N file synching operations on the same file within a journal committing interval, and M, M', D, and D' all be dirty before every file synching operation. Now, because the original ext4 synching performs a full transaction committing operation every time when the file is being synched, the total amount of data written to the storage is

$$N \times \{ \underbrace{(M + M')}_{\text{ext4 journaling}} + \underbrace{(M + M' + D + D')}_{\text{ext4 file-system writing}} \}. \tag{4.1}$$

On the other hand, eager synching writes only M and D in the journal to synch the file. When

However, ext4 still commits a transaction after the N synching operations, because eager synching is transparent to the ext4 journaling. There are two cases here. First, if the N synching operations belong to the same file and all the write operations on the file modify the same set of data, then the page cache can absorb the multiple writes to the data. Thus, eager synching writes the following amount of data:

$$\underbrace{N(M + D)}_{\text{eager synching}} + \underbrace{(M + M')}_{\text{ext4 journaling}} + \underbrace{(M + M' + D + D')}_{\text{ext4 file-system writing}}. \tag{4.2}$$

Comparing Equations 4.1 and 4.2, we can see that eager synching can effectively reduce the amount of data written if there are temporal localities in the file writing operations. The second case is that the N file synching operations belong to different files and none of the data writes can be absorbed by the page cache. The total amount of data written is:

$$\underbrace{N(M + D)}_{\text{eager synching}} + \underbrace{N(M + M')}_{\text{ext4 journaling}} + \underbrace{N(M + M' + D + D')}_{\text{ext4 file-system writing}}. \tag{4.3}$$

In this case, because there is no temporal locality in the write pattern, eager synching adds an extra amount of data $N(M + D)$ compared to the original ext4 synching. However, eager synching writes these extra data in the ext4 journal, and sequential write in flash storage is very fast.

The write latency of flash storage is very sensitive to the randomness in the write pattern. Our analysis is then focused on the count of write requests, because a large I/O is a sign of a high write randomness. Let us assume that the I/O scheduler can always merge adjacent write requests going to the ext4 journal into one request. Therefore, the total request count of ext4 corresponding to Equation 4.1 is $N \times (1 + 4)$. For the case corresponding to Equation 4.2, i.e., there are temporal localities in the write pattern, the write request count significantly reduces to $N + 1 + 4$, as ext4 commits a transaction after all the N synching operations. For the case corresponding to Equation 4.3, the request count is $N + 1 + 4N$, which is only one larger than that of the original ext4 synching.

Let us recall that, compared to the original ext4 synching, eager synching will produce an extra amount of data $N(M + D)$ in the worst case. We have introduced isets for reducing the size of M in Section 4.2, and the new dirty bit for reducing the size of D in Section 4.3. We also observed that if the amount of dirty user data D is large, then there is a good chance that the dirty data are sequential. Because eager synching can hardly optimize sequential writes, we propose using a threshold on the size of D: If the size of D is below the threshold, then the file system uses eager synching to handle a fsync() call. Otherwise, the file system uses the original ext4 synching to handle the call. The threshold is 100 pages (400 KB) in the current design. Various threshold settings will be evaluated in the experimental section.

4.5 Crash Recovery for Eager Synching

Hand-held devices are prone to unexpected power interruptions. Upon power restoration, as a journaling file system, ext4 will scan the journal in the storage for crash recovery. If ext4 finds a complete transaction (i.e., multiple segments followed by a commit record), it copies back the segment data to the file-system image in the storage and then delete the transaction from the journal. The last transaction in the journal could be partially written because of a power failure. Ext4 simply discards the incomplete transaction because copying the segment data in the transaction to the file-system image might corrupt the file system.

Now consider how ext4 performs crash recovery if it is equipped with eager synching. As described earlier, eager synching writes new segments to the current transaction in the journal, so the segments in transactions are from both eager synching and the ext4 journaling. Because eager synching is transparent to the ext4 journaling, a complete transaction will include the latest versions of all the dirty data associated with the files being synched in the lifetime of the transaction. On crash recovery, if ext4 finds a complete transaction, then it will copy back the segment data from the ext4 journaling but skip the segment data from eager synching. However, if ext4 does the same thing for the segments in the incomplete (and the last) transaction, it will lost the recently synched files. Therefore, ext4 examines the segments in the incomplete transaction, and copies the segment data contributed eager synching back to the file-system image, because the segments are contributed by successful synching operations. Copying the segment data back will not corrupt the file system, because eager synching uses isets to guarantee that the segments include all or none of the dirty metadata of a file operation.

Chapter 5

Experimental Results

5.1 Experimental Setup and Performance Metrics

We conducted a series of experiments on an Android tablet, Nexus 7, to evaluate our eager synching approach. The hardware and software specifications of Nexus 7 can be found in Table 5.1. The flash storage (i.e., an eMMC) of Nexus 7 is partitioned into a 639 MB system partition and a 27.6 GB data partition, and our experiments were all conducted on the data partition. We implemented the proposed eager synching in ext4 for performance evaluation.

Our experiments adopt both micro-benchmarks and macro-benchmarks. For the micro-benchmarks, we used AndroBench version 3.4, which is a popular storage benchmarking tool developed at Sungkyunkwan University for Android devices. For the macro-benchmarks, we used systemtap[1] to collect the file-level traces of three typical Android applications, including the Gamil client, the Facebook client, and the Chrome browser. The collected traces were replayed on the file system for performance evaluation. For performance comparison, we ran the tests on the ext4 file system with the original file synching method or with our eager synching.

[1]http://sourceware.org/systemtap/

Item	Description
Hand-held device	Google Nexus 7
CPU	ARM Cortex-A9 Nvidia Tegra 3 1.2GHz quad-core
Main memory	1GB DRAM DDR3
Storage	32GB eMMC flash storage
Android version	4.2.2 (CyanogenMod 10.1)
Linux kernel version	3.1.10
File system	Ext4

Table 5.1: The experimental platform

There are several performance metrics in our experiments. The primary metric is write performance. It was measured in terms of write throughput (MB/s) for most of the tests, or transaction per section (TPS) for the SQLite benchmark in AndroBench. Because there are a large number of fsync() calls in all the tests, a higher value of the throughput or the TPS means that the file system can handle fsync() calls more efficiently. The second metric is the sequential ratio of the write pattern. The sequential ratio is the percentage of the total sector writes contributed by write bursts longer than 128 sectors. A long write burst can consist of multiple adjacent write requests. The higher the sequential ratio is, the more sequential the write pattern is. The third metric is the total amount of data written to the flash storage. As discussed in Section 4.4, eager synching tries to reduce the amount of data written to the journal. However, we must emphasize that a small amount of random writes can trigger a considerable garbage collection overhead, as pointed out in Section 3.3. Thus, what we expected is, compared to the original ext4 synching method, eager synching has a comparable amount of data written but a much better write sequentiality.

5.2 Micro-Benchmark

5.2.1 Workload Characteristics

This experiment adopted the SQLite benchmark and the file-access benchmark in AndroBench. The SQLite benchmark has three tests, i.e., insert, update, and delete. Each of the test performed 300 transactions. The TPS index is separately computed for each test

	SQLite	File access
File data read (MB)	25.5 (59.6%)	191.8 (81.4%)
File data written (MB)	17.3 (40.4%)	43.8 (18.6%)
Avg. fread() record size (KB)	3.1	7.9
Avg. fwrite() record size (KB)	1.6	28.4
fwrite() call counts	11,329	1,579
fsync() call counts	4,533	1,519

Table 5.2: File access characteristics of the SQLite and file-access benchmarks

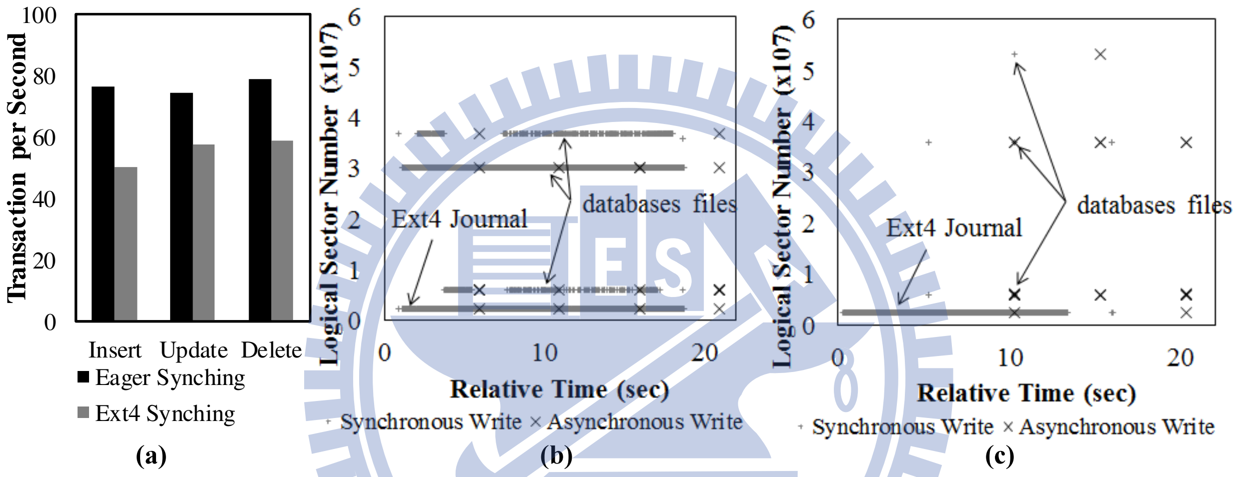


Figure 5.1: Evaluation results of the original ext4 synching and the proposed eager synching in the SQLite benchmark. (a) Eager synching improved the Transaction per Second by up to 50%. (b) The original ext4 synching randomly distributed write requests among the ext4 journal and the database files. (c) Eager synching re-directed the write requests of the synched file to the ext4 journal, making the disk write pattern more sequential.

by dividing 300 by the time spent on each test. The file-access benchmark is composed of four tests on file accessing: random read, sequential read, random write, and sequential write. The two read tests and the two write tests accessed total amounts of 191 MB and 43 MB of data, respectively. The file record sizes of the sequential tests and the random tests were 256 KB and 4 KB, respectively. The throughput of the four tests are separately computed by dividing the transfer size of a test by the time spent on the test.

We captured the file operations produced by the two benchmarks, and the characteristics of their file access behaviors are shown in Table 5.2. The SQLite benchmark wrote as much as it read, while the file-accessing benchmark read more. The average write size of the SQLite benchmark is small, while that of the file-accessing benchmark is larger. Even

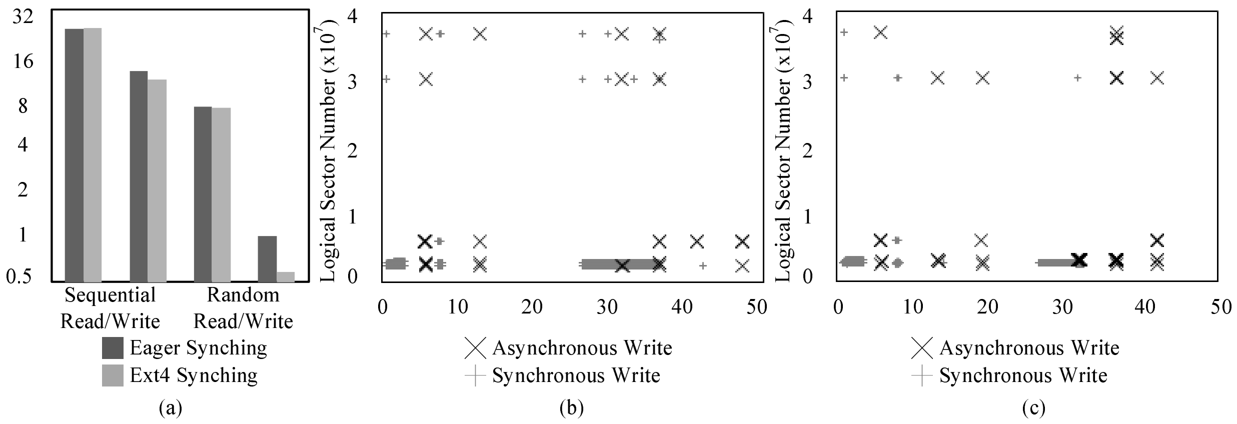


Figure 5.2: Evaluation results of the original ext4 synching and the proposed eager synching in the file-accessing benchmark. (a) Eager synching performed much better than ext4 synching in the random write test. (b) The original ext4 synching produced many write requests to both the journal and the test files. (c) Eager synching wrote data to the journal when synching files and reduced the write count to the test files.

though the write size of the latter is larger, it used small record size (4 KB) in the random write test. Both the SQLite benchmark and file-accessing benchmark performed fsync() at very high frequencies: the former issued an fsync() for almost every two fwrite() operations, while the latter called fsync() after almost every fwrite() operation.

5.2.2 Evaluation Results of SQLite Benchmark

Figure 5.1 shows the evaluation results of the SQLite benchmark using the original ext4 synching and our eager synching. Figure 5.1(a) indicates that eager synching noticeably improved upon ext4 synching in terms of TPS by up to 50%. There are two reasons for the significant improvement: a higher write sequentiality and a larger average write request size. First, when a file is being synched, the ext4 synching method wrote the dirty data directly to the file-system image, causing random in-place updates to the flash storage. Differently, eager synching redirected the dirty data to the sequential journal, producing sequential out-of-place updates in the journal. The file system generated in-place writes to the database files by the end of every journal committing interval. Figure 5.1(b) depicts the write pattern of the ext4 synching method, and the sequential ratio was 0%. Eager synching reshaped the write pattern to be more sequential, as shown in Figures 5.1(b), and the sequential ratio was boosted to 65.03%.

Second, to synch a file, the ext4 synching method flushed all the dirty data to the storage, and thus dirty data could only stay in the page cache for a short period of time. As a result, the ext4 synching method produced many small write requests. On the other hand, eager synching wrote only the data associated with the files being synched, and dirty data accumulated in the page cache for a longer period of time. Later on, when the file system submitted the dirty data to the block driver of the flash storage, the I/O scheduler merged many adjacent requests into larger ones. We found that the average request size of block write of the ext4 synching method was 10.1 sectors, while that size of eager synching significantly increased to 23.0 sectors. Writing the flash storage with large requests reduces the I/O path overhead and increases the write sequentiality.

In this experiment, eager synching slightly increased the total amount of data written to the flash storage (38.7 MB vs. 41.1 MB). We observed that the SQLite benchmark did not frequently overwrite existing data in the database files. In other words, there were not many temporal localities in the file-writing operations. As discussed in Section 4.4, if the page cache does not much reduce the write traffic, then eager synching can write slightly more data than the ext4 synching method. However, because eager synching still boosted the write performance, the cost of writing the extra amount of data was completely compensated by the benefit of making the write pattern sequential.

5.2.3 Evaluation Results of File-Accessing Benchmark

Figure 5.2 shows that 1) eager synching achieved a remarkable performance improvement in the random-write test, 2) eager synching did not suffer from performance degradation in the sequential-write test, and 3) eager synching does not affect the performance of the two read tests. For the random-write test, like the results of the SQLite benchmark, the write patterns in Figures 5.2(b) and 5.2(c) show that eager synching successfully prevented from producing random write to the journal and test files. In addition, compared to the block-level average write request size of using ext4 synching, using eager synching increased the size from 18.6 sectors to 46.4 sectors. As mentioned above, writing with large requests

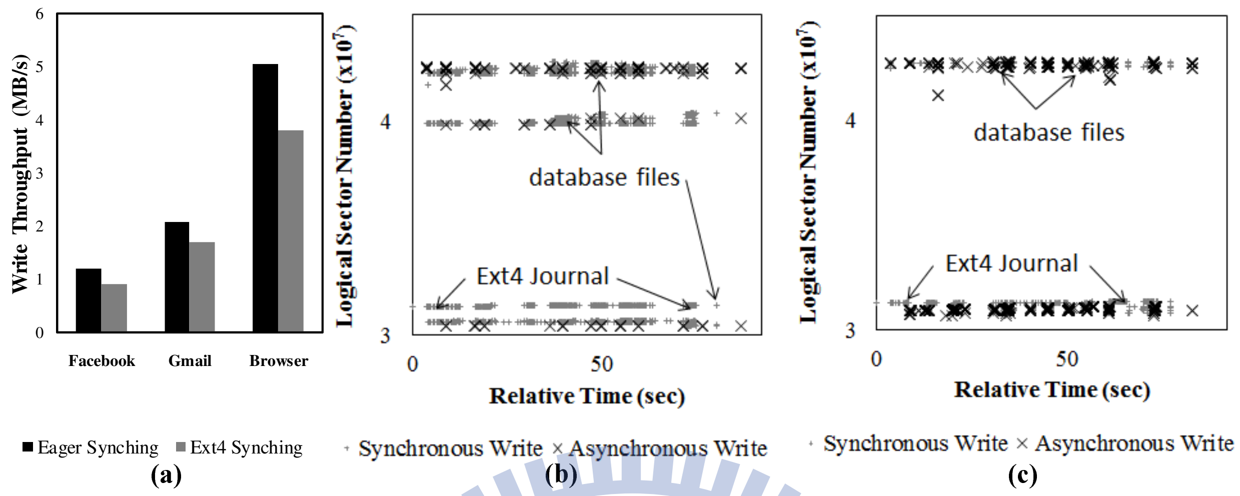


Figure 5.3: Evaluation results of the macro-benchmark. (a) Eager synching noticeably improved the write throughput under the workloads of the three Android applications. (b) Write pattern using ext4 synching is full of many small and random writes. (c) Write pattern using eager synching becomes less requests and more sequential.

is beneficial to the write performance. Another factor contributing to the performance improvement is that using eager synching decreased the total amounts of data written to the flash storage from 39.4 MB to 34.9 MB. As mentioned in Section 5.2.2, this comes from ignoring commit record writing.

Eager synching did not degrade the performance of the sequential-write test because, as illustrated in Section 4.3, the file system performs the original ext4 synching if it finds the amount of data to be synched larger than a threshold. This design prevents the file system from re-directing a sequential stream to the journal. As to the two read tests, eager synching did not affect their performance since the file system never reads the journal on cache misses. This is because a piece of cached file data can be released from the main memory only after the file system allocates disk space for the data and writes the data back to the allocated space.

This may stem from the fact that Eager Log tends to roll-back on detecting large requests. In some cases, requests size may vary across the roll-back threshold back and forth, thus making part of requests written in journal by Eager Log are forced to be written again by journaling mechanism. This reduces hot data to accomodate more writes, and thus the amount written is enlarged. Even so, Eager Log still benefits from sequential writes as

	Facebook	Gmail	Chrome
File data read (MB)	39 (43.7%)	198 (44.1%)	13 (5.3%)
File data written (MB)	50 (56.3%)	251 (55.9%)	238 (94.7%)
Avg. fread() size (KB)	1.55	3.49	0.42
Avg. fwrite() size (KB)	3.36	2.12	2.69
fwrite() calls	15,369	120,971	90,638
fsync() calls	2,875	6,385	575
Different files accessed	370	135	721

Table 5.3: File access characteristics of the three applications in our macro-benchmark.

flash memory is good at it.

5.3 Macro-Benchmark

5.3.1 Workload Characteristics

The second experiment adopted file-operation workloads from three typical Android applications, i.e., the Facebook client, the Gmail client, and the Chrome browser. We captured the traces of file operations from the three applications using the following procedure: We cleared the application storage cache of the applications, and started systemtap for file-operation trace collecting. We then started the applications, and used the application to emulate users' daily activities, like checking the new feeds on Facebook, checking and sending e-mails, and surfing the web, until the applications wrote sufficiently many file data to the flash storage. The collected traces, consisting of calls to fopen(), fclose(), fread(), fwrite(), rename(), unlink(), fdatasync(), and fsync(), were then replayed on the original ext4 file system and the modified ext4 file system with our eager synching for performance evaluation.

Table 5.3 shows the characteristics of the file-accessing behaviors of the three applications. The read-write ratios of Facebook and Gmail were very close, while Chrome performed file writing for most of the time. Their file-writing sizes were small, between 2 KB and 3KB, which is a sign of random file writing as most of the random writes carry out with small

	Write Performance	Amount of data written (MB)	Avg. block write req. size (sectors)	Sequential ratio
SQLite benchmark†	78 (45) TPS	41,1 (38.7)	23.0 (10.1)	65.03% (0%)
File-accessing benchmark‡	1 (0.6) MB/s	34.9 (39.4)	46.4 (18.6)	99.02% (0%)
Facebook	1.19 (0.91) MB/s	91.8 (85.0)	91.15 (45.73)	59.02% (1.23%)
Gmail	2.08 (1.69) MB/s	292.2 (306.2)	70.99 (25.74)	17.15% (0.13%)
Chrome	5.05 (3.80) MB/s	146.0 (149.8)	126.05 (63.02)	14.44% (3.49%)

Table 5.4: Summary of the experimental results. †The insert test. ‡The random write test.

size. In particular, Chrome touched the largest amount of different files among the three applications, and these files belong to the cache of web pages and image files. Facebook and Gmail made a lot of fsync() calls, which were mainly contributed by operations on the SQLite database files. Differently, Chrome produced a smaller amount of fsync() calls, and the calls were contributed by synching the web-page caching files.

5.3.2 Evaluation Results

Figure 5.3(a) shows that all the write throughput significantly benefited from eager synching under the workloads of the three applications. The improvement was the largest for the workload of Chrome, archiving almost 30%. Like in the micro-benchmark, the performance improvements was contributed by the reduced write randomness and the enlarged write request size.

As shwon in Table 5.3, even though Chrome produced the smallest amount of fsync() calls, it accessed the largest number of different files. Most of the files were the local copy of web pages and image files. Figure 5.3(b) shows the disk write pattern of using the original ext4 synching, and the write requests to the small files increase the randomness of the write pattern. Figure 5.3(c) shows that using eager synching re-directed many of the synchronous write requests to the journal, and thus the write pattern appears much more sequential compared to Figure 5.3(b). As to the average block-level write request size, we found that the size increased from 63 sectors to 126 sectors, almost being doubled. Using eager synching also reduced the total amount of data written by about 5% under the workloads of Gmail and Chrome, while increased the amount by 6% under the Facebook workload. Nevertheless, the slight increase did not much affect the benefit of using eager synching, as flash storage is very fast in terms of sequential write.

Chapter 6

Conclusion

Our research proposes a fast file syncing mechanism called eager syncing which tries to solve the problem that frequent file synchronization causes small and random write pattern to underlying flash storage, and this kind of write is weak point to flash memory, thus results in poor performance.

Syncing a file through eager syncing do not write data back directly, which is small and random, instead, those data are gathered up, written sequentially into a log space. This creates a bigger chance where I/O scheduler can do better jobs of sorting and merging I/O requests, therefore presenting to flash storage more sequential and bigger requests. Data to be synced are also considered so that none of critical data are missing where inconsistency raised.

Eager syncing are implemented in ext4, and some experiments conducted shows that not only write pattern are more sequential than before, thus performance gain are obvious, but the amount of data written are also decreased in some cases.

Bibliography

[1] D. Melanson. (2013) Eric schmidt: Google now at 1.5 million android activations per day. [Online]. Available: http://www.engadget.com/2013/04/16/eric-schmidt-google-now-at-1-5-million-android-activations-per/

[2] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 3, p. 18, 2007.

[3] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," in *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, 2009.

[4] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proc. of the USENIX conf. on File and stroage technologies*, 2012.

[5] K. Lee and Y. Won, "Smart layers and dumb result: Io characterization of an android-based smartphone," in *Proceedings of the tenth ACM international conference on Embedded software*, ser. EMSOFT '12. New York, NY, USA: ACM, 2012, pp. 23–32. [Online]. Available: http://doi.acm.org/10.1145/2380356.2380367

[6] T. O. G. B. Specifications. (2004) fsync. [Online]. Available: http://pubs.opengroup.org/onlinepubs/009695399/functions/fsync.html

[7] W.-H. Lin and L.-P. Chang, "Dual greedy: Adaptive garbage collection for page-mapping solid-state disks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. IEEE, 2012, pp. 117–122.

[8] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim, "A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 7, no. 4, p. 38, 2008.

[9] A. Leventhal, "Flash storage memory," *Commun. ACM*, vol. 51, no. 7, pp. 47–51, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1364782.1364796

[10] R. Y. Wang, T. E. Anderson, and D. A. Patterson, "Virtual log based file systems for a programmable disk," in *Proceedings of the third symposium on Operating systems design and implementation*, ser. OSDI '99. Berkeley, CA, USA: USENIX Association, 1999, pp. 29–43. [Online]. Available: http://dl.acm.org/citation.cfm?id=296806.296809

[11] T.-c. Chiueh and L. Huang, "Trail: A fast synchronous write disk subsystem using track-based logging," 1999.

[12] M. H. Eich, "Main memory database recovery," in *Proceedings of 1986 ACM Fall joint computer conference*, ser. ACM '86. Los Alamitos, CA, USA: IEEE Computer Society Press, 1986, pp. 1226–1232. [Online]. Available: http://dl.acm.org/citation.cfm?id=324493.325092

[13] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang, "Configuring and scheduling an eager-writing disk array for a transaction processing workload," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083323.1083355