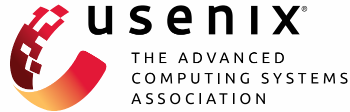


iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call

Daejun Park and Dongkun Shin, *Sungkyunkwan University, Korea*

https://www.usenix.org/conference/atc17/technical-sessions/presentation/park

This paper is included in the Proceedings of the
2017 USENIX Annual Technical Conference (USENIX ATC '17).

July 12–14, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-38-6

Open access to the Proceedings of the
2017 USENIX Annual Technical Conference
is sponsored by USENIX.

iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call

Daejun Park and Dongkun Shin
Sungkyunkwan University, Korea
pdaejun@skku.edu, dongkun@skku.edu

Abstract

For data durability, many applications rely on synchronous operations such as an `fsync()` system call. However, latency-sensitive synchronous operations can be delayed under the compound transaction scheme of the current journaling technique. Because a compound transaction includes irrelevant data and metadata, as well as the data and metadata of fsynced file, the latency of an fsync call can be unexpectedly long. In this paper, we first analyze various factors that may delay an fsync operation, and propose a novel hybrid journaling technique, called `ijournaling`, which journals only the corresponding file-level transaction for an fsync call, while recording a normal journal transaction during periodic journaling. The file-level transaction journal has only the related metadata updates of the fsynced file. By removing several factors detrimental to fsync latency, the proposed technique can reduce the fsync latency, mitigate the interference between fsync-intensive threads, and provide high manycore scalability. Experiments using a smartphone and a desktop computer showed significant improvements in fsync latency through the use of `ijournaling`.

1 Introduction

The buffered I/O is essential to a high-performance file system because data can be temporarily buffered in the main memory until being written back to storage. However, the buffered I/O cannot guarantee file-system consistency and data durability in the cases of unclean file-system shutdowns or hardware failures [22]. To ensure file-system consistency, many file systems have adopted a journaling technique, which can ensure the atomicity of a transaction. A transaction is a group of file system modifications that must be carried out atomically. For example, the `ext4` file system uses the journaling block device version 2 (JBD2) in the Linux kernel to support journaling [23]. All file system operations are logged in the journal area before updating the original file system. Therefore, by undoing any incomplete transactions and redoing all committed transactions, journaling can be used to maintain the file-system consistency despite sudden system crashes.

The `ext4` file system uses a physical logging scheme that records the modified blocks [14], rather than logical logs, which records operations. Because several of the metadata structures of `ext4`, such as `block bitmap` and `inode table`, are shared among multiple file operations, it is easier and more efficient to commit multiple transactions at once rather than commit each file operation-level transaction individually. For the purpose, `ext4` groups concurrent unrelated transactions into a single compound transaction [26], which is periodically flushed into a reserved storage area, called a journal area. The compound transactions are maintained in the journal transaction buffer of the main memory until being committed to the journal area. The compound transaction scheme provides a better performance, particularly when the same metadata structure is frequently updated within a short period of time.

`Ext4` supports three journaling modes: writeback, ordered, and data modes. Ordered mode, which is the default option, journals only the metadata. However, it enforces an ordering constraint to guarantee file-system consistency, in which the transaction-related data writes must be completed before the journal writes of the metadata. Therefore, the transaction commit latency will be lengthy if the size of the associated data is large. A long transaction commit latency may not be a serious problem, however, because the journal commit operations are periodically invoked by a background journaling thread.

Although the file-system consistency and data durability are supported using a journaling scheme, the data durability is not immediate. To ensure instant data durability, users must call a synchronous operation such as an `fsync()` or `fdatasync()`. Most database systems rely

on fsync system calls to be assured of immediate data durability. Recent mobile platforms such as Android also frequently use fsync system calls [17]. Because an fsync system call is a synchronous operation, the fsync latency affects the performance of the application.

When a file system uses journaling, all file-system changes are updated through the journaling layer. Therefore, when an application calls an fsync for a modified file, the journaling thread is awakened on demand, and the transactions in the transaction buffer are flushed into the storage immediately, irrespective of the journal commit interval. The fsync operation must wait until the journal commit operation is completed. In particular, a compound transaction in the transaction buffer may include data and metadata updates of other irrelevant files, as well as the target file of the fsync call (*fsynced* file). A long latency for committing a compound journal transaction will increase the latency of an fsync system call [12].

For a short fsync latency, a more fine-grained journaling scheme such as file-level transaction committing is required. However, under a physical logging scheme, fine-grained journaling is difficult to implement because several metadata blocks are shared by multiple file operations. In addition, fine-grained journaling imposes a high journaling overhead.

Another solution is the use of a logical logging scheme. For example, `XFS` [28] and `ZFS` [7] log the file operations rather the modified blocks for a synchronous request. All file system operations are logically-logged as transactions, which accumulate in memory until they are committed to the journal area for an fsync call. The logical logs are replayed during a crash recovery. However, logical logging requires a large sized transaction buffer in the memory compared with physical logging, particularly when the same metadata structure is frequently updated. For example, ZFS generates a 256 bytes of logical log in memory for each write operation.

To address this issue, we propose a hybrid approach that uses both the normal journaling by JBD2 and the file-level transaction journaling of our proposed `ijournaling` technique. Under a normal periodic journaling operation, the proposed scheme uses a legacy journaling scheme that flushes the compound transaction. However, if on-demand journaling is invoked by an fsync call, `ijournaling` commits only the transactions related to the fsynced file without flushing the compound transaction in the transaction buffer. The file-level transactions include only the minimum metadata, through which all relevant file-system metadata blocks can be recovered after a system crash. The `ijournaling` technique can eliminate the compound transaction problem for an fsync call without requiring an additional large amount of memory space for transaction management, unlike `ZFS`. We evaluated the performance improvements of the proposed journaling scheme on both a smartphone and a desktop system.

2 Background

`Ext4` is the default file system of Linux kernel, and is widely used on mobile devices such as Android-based smartphones and desktop computers. `Ext4` divides an entire storage space into several block groups. Two metadata structures, i.e., `superblock` and `group descriptor table` (`GDT`), describe the general information of the overall file system. Each block group has its own `block bitmap` and `inode bitmap` to manage the allocation status of the data blocks or inode entries. Each block group also maintains an `inode table`. Each inode entry of the `inode table` is 256 bytes in size and describes the attributes of a single file or directory. These metadata structures are allocated in a 4-KB block unit, and are shared by multiple files or directories. `Ext4` supports an extent-based block-mapping scheme. A single extent identifies a set of blocks that are logically contiguous within the file and also on the underlying block device. An inode entry can contain a maximum of four extent structures internally. If more extents are required, external extent structures are allocated in the data block area for indirect pointing.

`Ext4` uses a journaling technique. Information regarding pending file-system updates is first written to the journal to enable an efficient crash recovery. The journal space is treated as a circular buffer. Once the necessary information has been propagated to its fixed location in the `ext4` structures, the corresponding journal logs are identified as *checkpointed*, and the space can be reclaimed. All modified metadata blocks are recorded in a block unit at the journal area even though only a portion of the metadata blocks is modified. This feature makes it difficult to implement file-level journaling because a metadata block is shared by multiple files. One transaction log in the journal contains a journal header (JH), several journal descriptor blocks (JDs) to describe its contents, and a journal commit block (JC) to denote the end of the transaction.

`Ext4` manages the life cycle of each transaction. Each transaction has a metadata list and an inode list, which have the metadata blocks and pointers to the inodes modified by the transaction, respectively. First, a *running* transaction is created, and all file-system modifications are inserted into the running transaction. When the periodic JBD2 thread is invoked or an `fsync()` is called, the transaction state is changed to *committing*, and the transaction blocks are written into the journal area. After the completion of a transaction commit, the transaction is marked as *checkpoint*. After the transaction is checkpointed, it is removed from the transaction list.

3 Related Work

Prabhakaran *et al.* [26] observed the storage performance when a foreground asynchronous sequential stream and a background random synchronous stream compete to use the `ext3` file system. They showed that the more frequently the background process calls an fsync, the more traffic is sent to the journal owing to the compound transactions of `ext3`. The authors proposed an adaptive approach that selects the best journaling mode for each transaction according to its I/O pattern. However, this approach cannot solve the compound transaction problem completely, and may be unsafe [27].

Jeong *et al.* [17] revealed the journaling-of-journal (JoJ) problem on an Android-based smartphone, where the `ext4` file system uses a journaling scheme for data reliability, and SQLite [3] conducts additional journaling using its own journal file. Their study suggests using `fdatasync()` and write-ahead logging (WAL) in SQLite to reduce the number of journal commits. Here, `fdatasync()` does not commit a journal transaction unless the file-system metadata relevant to the target file are changed. However, WAL also generates frequent fsync calls, and `fdatasync()` can be effective only when there are no metadata updates.

To mitigate the JoJ overhead, Shen *et al.* [27] proposed using the data journaling mode of `ext4` adaptively. Data journaling writes both data and metadata in the journal area without generating page writes at the original file system locations during a journal commit operation. Because a journal commit operation sends only the sequential write requests to the storage, the journal commit latency can be reduced. However, this technique also flushes compound transactions and cannot completely avoid a long fsync latency.

There are several approaches that divide a file system space into several groups to localize the faults and transactions of the filesystem, or to avoid the lock contention on shared file-system data structures in memory. The per-block-group (PBG) journaling scheme [19] exploits the block groups of `ext4`. Because each block group has its own metadata blocks, PBG journaling extracts a block-group-level transaction including updates on the fsynced file from a compound transaction, and commits only the transaction of the target block group. PBG journaling shows significant improvements in terms of fsync latency when a fsynced file and other irrelevant files are allocated in different block groups. However, a long fsync latency occurs if irrelevant files share the same block group. The eager synching [8] also uses a similar technique as PBG journaling.

IceFS [21] proposed a new container abstraction, called *cube*, to provide more flexible and configurable isolations. SpanFS [18] distributes files and directories among the *domains*, which are the basic independent function units for file system services such as data allocation and journaling. IceFS and SpanFS also cannot avoid the compound transaction problem within a cube or domain. Moreover, IceFS is incompatible with legacy file systems, and the user should manage the cubes. SpanFS can generate a large compound transaction across multiple domains. Xsyncfs [25], NoFS [10], and OptFS [9] improved the fsync latency by delaying sync operations or changing the implementation of ordering constraint.

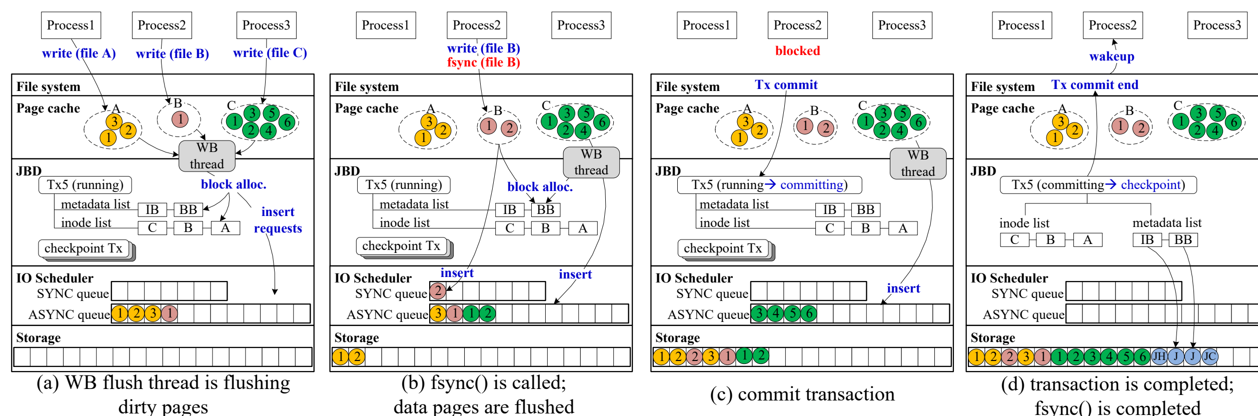
ScaleFS [13] uses a logical logging technique. Operation logs (OpLogs) are generated in its in-memory file system to record file-system changes. An OpLog consists of logical file-system operations, and is applied to the on-disk file system when an fsync is invoked. ScaleFS applies only dependent operations that are related to the file or directory being fsynced, which is a very similar approach to our proposed `ijournaling` technique. However, logical logging-based journaling scheme requires significant changes to the current `ext4` file systems. In addition, a performance overhead occurs because each file-system operation must record its own OpLog. Our proposed `ijournaling` follows the physical logging scheme of `ext4`, and has little overhead for managing file-level journals.

Jeong *et al.* [15] proposed an I/O scheduler technique that can detect asynchronous I/O requests related with latency-sensitive file operations such as an fsync call, and boost them over the other asynchronous I/Os. This technique improves the fsync latency and can be used along with our technique because they both handle the different underlying reasons for a long fsync latency problem. However, the number of latency-sensitive asynchronous I/Os can be minimized under our `ijournaling` scheme because only the relevant blocks are flushed by fsync calls.

Min *et al.* [24] investigated the performance of `fsync()` for a manycore architecture under five widely-deployed file systems. They showed that most of the file systems start to degrade in performance when more than ten cores compete for the file system. In our `ijournaling` scheme, a sync operation does not depend on a single journaling thread and each core has its own separate ijournal area. Therefore, our scheme provides a better manycore scalability, which is described in greater detail in Section 6.

4 Analysis of Fsync Latency in `Ext4`

When a user process calls an `fsync()` system call for a file, the process is blocked, and the system call service in the kernel performs the following operations, as shown in Figure 1. First, it updates the related metadata blocks for the file, inserts them into the running transaction man-



(a) WB flush thread is flushing dirty pages

(b) fsync() is called; data pages are flushed

(c) commit transaction

(d) transaction is completed; fsync() is completed

Figure 1: Dependency problems of a journal commit.

aged by JBD2, and flushes the data blocks of the fsynced file, as shown in Figure 1(b). For example, the `block bitmap` needs to be modified when an fsync call flushes newly allocated data blocks. `Ext4` uses a delayed block allocation scheme, and thus, the file-system location for a data block is determined just before the block is flushed into storage. The write requests on the data blocks of the fsynced file are transferred as synchronous requests because the user process is waiting for the completion of the system call.

Second, the system call service sends a commit request for the relevant transaction to JBD2 if the transaction state is still running, and waits for the completion of the commit operation, as shown in Figure 1(c). In this step, a commit operation cannot be issued immediately if there is another committing transaction because JBD2 can commit only one transaction at a time. During a commit operation, JBD2 awaits the completion of all data write requests relevant to the target transaction. In Figure 1(c), all data blocks of files **A**, **B**, and **C** must be flushed because the target transaction includes the inodes. Finally, JBD2 writes the journal blocks in the journal area after the completion of the data write operations, as shown in Figure 1(d). A journal block includes the modified metadata blocks. The final block written by JBD2 is the journal commit (JC) block, which is followed by a *flush* command. When the *flush* command is completed, the `fsync()` system call is completed, and the user process can continue with its operations.

Based on its operations, we can find several reasons for adverse effect on the latency of an fsync system call. The first reason is the inter-transaction (IT) dependency. Because `ext4` uses a single JBD2 thread, only one transaction (i.e., a committing transaction) can be committed at a time. Protecting concurrent journal commits is important for preventing multiple journals from being interleaved in the journal area. Furthermore, multiple transactions cannot be committed concurrently because they share several metadata blocks. Therefore, if the JBD2 thread is committing transaction Tx_{n-1}, the next transaction Tx_n relevant to the fsynced file cannot be changed into a committing transaction immediately. Such cases will occur frequently when multiple threads invoke fsync calls simultaneously. To solve this IT dependency problem, our `ijournaling` technique handles an fsync call at system call service rather than the journaling thread, and uses separated journal areas.

The second reason is the compound transaction (CTX) dependency, shown in Figure 1(c). When the JBD2 thread commits the transaction of an fsynced file, the inode list of the committing transaction includes irrelevant inodes. The JBD2 thread must wait for the completion of the data block write operations owing to the ordering constraint of ordered-mode journaling. The CTX dependency is severe when there are many processes generating file-system write operations. Even when only one process generates write operations, a CTX dependency problem can occur if the process updates multiple files. In some cases, a transaction can include *discard* commands [1], which have considerably long latencies.

The delayed block allocation technique of `ext4` aggravates the CTX problem. The delayed block allocation has many advantages because it postpones block allocations until the page flush time, rather than during a `write()` operation [23]. Therefore, the overall performance of the file system is higher when delayed allocation is enabled. However, if an fsync is called just after the flush kernel thread invocation, as shown in the example in Figure 1(a), the flush thread will allocate data blocks for dirty pages, and register several modified inodes in the running transaction during the delayed block allocation. Then, the commit operation of the journal transaction will generate many write requests into storage. If an fsync is called before the flush thread is invoked, the fsync latency will be short because there are few modifications to the file system. Therefore, fsync latencies will fluctuate in a delayed allocation scheme. On the contrary, if the delayed allocation is disabled, the

modified inodes will be distributed to different transactions, and the fsync latency will be unrelated with the flush thread invocation. Nevertheless, a delayed allocation can demonstrate a better performance and shorter average fsync latency, as described later in Section 6. Because our `ijournaling` scheme commits a file-level transaction rather than a compound transaction, it can always demonstrate a short fsync latency irrespective of the block allocation policy. Throughout our study, we used delayed allocation as the default scheme.

The last reason is the quasi-async request (QA) dependency revealed in [15]. In Figure 1(a), the writeback flush thread has sent a write request on data block 1 of file **B** before an fsync is called. Whereas the write requests generated by an fsync system call are sent along with a `SYNC` flag, the write requests generated by the flush thread are sent without the flag. The CFQ I/O scheduler in Linux gives lower priorities to requests without a `SYNC` flag. Although data block 1 is written by an async request, the request is latency-sensitive. Such a request is called a *quasi-async* request. A long latency will occur for completion of the quasi-async request, particularly when there are many competing async requests in the I/O queue. The QA dependency problem can be solved through the boosting technique proposed in [15], which changes a quasi-async request into a sync request. However, owing to the CTX dependency, the asynchronous write requests on **A** and **C** in Figure 1 must also be changed to sync requests in the boosting technique. The `ijournaling` can mitigate the QA dependency problem by removing unrelated dependencies. For example, the fsync call on **B** does not need to wait for the completion of write requests on **A** and **C**.

5 The iJournaling Scheme

5.1 Main Idea

The goal of `ijournaling` is to improve the performance of an `fsync()` call while exploiting the advantage of the legacy compound-transaction-based journaling scheme. Only when a process calls an `fsync()` system call, `ijournaling` is invoked. The `ijournaling` scheme generates `ijournal` transactions (i-transactions) and flushes them into a reserved `ijournal` area without committing the normal running transaction of an fsynced file. The i-transaction includes metadata modification logs, which are the minimum required information through which a crash recovery operation can recover the file-system metadata blocks modified through an fsync operation. Only file-level metadata such as an inode entry and the external extent structures of the target file, and any related directory entries (DEs), are recorded. Other modified metadata blocks shared by other files,

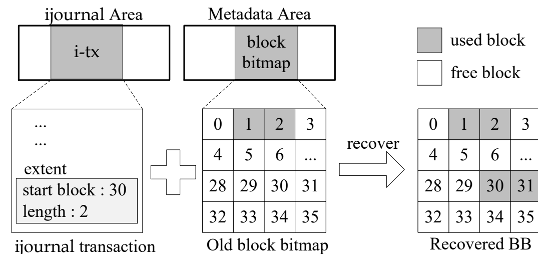


Figure 2: `Block bitmap` recovery with ijournal.

such as `GDT`, `block bitmap`, `inode bitmap`, or `inode table`, are not flushed into the `ijournal` area. They can be recovered during the crash recovery time using committed i-transactions. The `ijournaling` scheme does not change the normal running transaction used by the JBD2 thread. Therefore, the metadata blocks committed by `ijournaling` are again committed into the normal journal area through the following periodic JBD2 thread, which simplifies the crash recovery.

Figure 2 shows an example of a metadata recovery operation of `ijournaling`. When the file-system recovery module finds a committed i-transaction in the `ijournal` area, it can modify the old `block bitmap` in the file system using the extent allocation information, which can be found from the inode entry or the external extent structures in the i-transaction. Because two blocks from block number 30 are allocated for an extent, the 30-*th* and 31-*st* bits in the `block bitmap` must be set. The `inode table` and `inode bitmap` can also be easily recovered through a recorded inode entry. To implement `ijournaling`, no changes are required to the current JBD2 journaling scheme. Whereas a normal journaling thread flushes the transaction buffer periodically, `ijournaling` is performed in the `fsync()` system call service. Therefore, an `ijournaling` and a normal journaling can be performed simultaneously, and the inter-transaction dependency is removed. The file-system recovery module must be modified to handle `ijournal`.

5.2 iJournal Transaction

The `ijournal` area is separated from the normal journal area. In addition, each processor core uses a separate per-core `ijournal` area in order to support manycore scalability. Each `ijournal` area is managed as a circular buffer. This scheme needs to allocate space as many as the number of cores. If the existing normal journal area is shared by normal journal transactions and i-transactions, no additional space allocation is required. However, we should be carefully in allocating blocks in the journal area to prevent two different journal blocks from being mixed in the journal area in an interleaved manner, because a transaction must consist of consecutive blocks.

While a JBD2 thread is allocating blocks in the journal area, the `ijournaling` must wait until the block allocation is completed. Therefore, separating journal areas can improve the concurrency of journaling operations. The required storage space for per-core `ijournal` area is small because the the size of an i-transaction is smaller than that of a normal transaction, and i-transactions will be invalidated after its corresponding normal transaction is committed.

Figure 3 shows the structure of an i-transaction, of which there are two types: file i-transaction and directory i-transaction. Whereas the file i-transaction has the metadata information of an fsynced file, the directory i-transaction has the metadata information of any related parent directory.

A file i-transaction is composed of one header block, several external extent blocks (if they exist), and one commit block. The journal header in the header block has the same structure as a normal journal header. It includes the magic number and transaction ID. A file i-transaction has the same transaction ID as the running transaction of normal journaling, which includes the metadata updates of the corresponding fsynced file. Because the journal transactions are distributed among multiple journal areas, the crash recovery module must identify the order of each transaction based on its transaction IDs. Because there can be multiple fsync calls before the current running transaction of normal journaling is committed, several i-transactions will have the same transaction IDs. In particular, for the i-transactions recorded at different `ijournal` areas, it is impossible to know the order of them if they have a same transaction ID. To resolve this problem, `ijournaling` uses a sub-transaction ID, which is incremented by each fsync call and managed globally among multiple cores.

The inode number and inode structure in an i-transaction are used for recovering the `inode table`, `inode bitmap`, and `GDT`. Each block tag stores the mapping between an external extent block in the file i-transaction and its actual file-system block number. The crash recovery can update the `block bitmap` using the internal extent information in the inode structure, the block tags, and the external extent blocks. The file i-transaction collects only dirty external extent structures. To reduce the extent tree search overhead, we modified the file system to maintain a list of dirty extent blocks for each uncommitted file and update it during each extent allocation/free operation. Because only a 20 bytes of data structure is required for tracking one external extent, the memory overhead for external extent tracking is not significant. The commit block indicates whether an i-transaction has been completely committed.

The directory i-transaction is used to record any relevant directory updates. If a file is fsynced but its parent

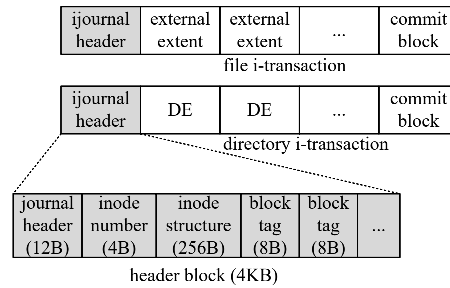


Figure 3: Structure of ijournal transaction.

directory entry is not committed before a system crash, the file will be unreachable after the system recovery. For example, if directory `A` and its subdirectory `B` are created, and an fsync call for file `/A/B/c` is called, `ijournaling` records all the changed directory information of the directories of `A` and `B`, as well as the changed file information of file `c`. The `ijournaling` identifies all directories that are related to the fsynced file and therefore must also be committed.

To track the uncommitted directories, we added the `uncommitted_DE` flag in the inode structure. When a new file is created, the flag is marked in the created file's inode to denote that its directory entry has not been recorded at the parent directory block. The flag is cleared when the parent directory block is committed by JBD2. The `ijournaling` first checks the flag of the fsynced inode. If the flag is marked, the parent directory is also examined recursively until no more uncommitted directory is found. At that time, the directory i-transaction of topmost uncommitted parent directory is first written in the `ijournal` area, and then the directory i-transactions of next-level directories are written in order. Finally, the file i-transaction of fsynced file is written. Therefore, even though there is a system crash during the ijournaling, the recovered file system can maintain its consistency (i.e., there is no unreachable file or directory.) Although only one directory entry in the DE blocks of an uncommitted directory is related with an fsync call, our scheme records the entire DE blocks of the uncommitted directory in the directory i-transaction for fast fsync handling, because it is time consuming to extract the modified directory entries from the DE blocks. Instead, the recovery process identifies the modified and valid DE entries to update the old DE blocks in the file system.

If there are no modified external extent blocks and DE blocks to be committed by an fsync call, it will be possible to write a single block i-transaction by recording all information in the ijournal header, which can reduce the write traffic on the ijournal area.

The ijournaling will show a slightly difference on crash recovery compared with the normal journaling scheme. While the normal journaling can recover all the other contemporary file operations as well as the fsynced

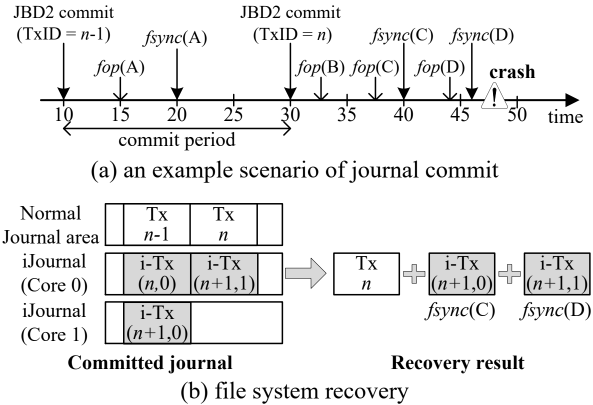
file operation, the proposed ijournaling can recover only the files and directories related to fsync operation. However, the file system consistency is guaranteed.

To simplify the `ijournaling` implementation, our scheme uses the normal journaling for some cases. For the fsync call for a directory itself, a normal transaction is committed instead of an ijournal to record all file-system changes in the subdirectories, as well as in the fsynced directory entry. This simplifies the journaling by removing the traversing of the subdirectories. When an inode is shared by multiple files using hard link and an `fsync()` is called for only one file, the file-system consistency can be broken if `ijournaling` records the parent directories of only the fsynced file. To eliminate the traversing of directories connected by hard links, a normal transaction is committed instead of an ijournal for the case. To track such a case, we added the `uncommitted_HL` flag in the inode structure. The flag of a file is marked if the `i_link_count` of its inode is incremented by a hard link operation. The flag is cleared when a running transaction is committed by the JBD2 thread. The fsync system call service checks the flag of the target inode, and calls normal journaling if the flag has been marked.

5.3 Crash Recovery

The ijournal crash recovery module replays only valid i-transactions. It first scans the normal journal area, replays the committed but not-yet-checkpointed journal transactions, and finds the last committed journal transaction ID (`Max_TxID`). Because valid i-transactions have the information on file-system changes after a valid normal journal transaction is committed, the normal journal transaction must be replayed before i-transactions. Then, the recovery module scans the `ijournal` areas. If an i-transaction has a transaction ID larger than `Max_TxID`, it is valid. Otherwise, the i-transaction is ignored since a normal committed journal transaction includes all the metadata modifications of the i-transaction. If there are multiple i-transactions on an inode, only the last i-transaction with the largest sub-transaction ID is valid since the last one includes all the metadata modifications of the previous i-transactions.

Figure 4(a) shows an example of journal commit. At a time of 30, the normal transaction with the transaction ID (TxID) n is committed and the TxID is incremented to $n+1$. Before the next periodic transaction with TxID = $n+1$ is committed, the files **B**, **C**, and **D** are modified, and `fsync()` calls are invoked for the files **C** and **D** by different processor cores. In Figure 4(b), the i-transactions with (TxID, sub-TxID) = $(n+1,0)$ and $(n+1,1)$ have the committed file information of the files **C** and **D**, respectively. The system is crashed before the periodic transaction commit (TxID = $n+1$). In Figure 4(b), the i-transaction with TxID = n is invalid because the normal transaction with TxID = n has been committed. Therefore, the recovery operation uses only the i-transactions with TxID = $n+1$. In Figure 4(a), there is a file operation on file **B** before a system crash, but the operation cannot be recovered by `ijournaling`. However, there is no problem in file-system consistency.



(a) an example scenario of journal commit

(b) file system recovery

Figure 4: Example of journal commit and recovery.

For each valid i-transaction, the recovery module modifies the corresponding inode entry and other metadata blocks in the file system. Because an fsync call can generate one file i-transaction and multiple directory i-transactions, the multiple i-transactions generated by an fsync call cannot be committed atomically if a system crash occurs during fsync handling. In addition, the DE blocks in directory i-transaction also contain information on irrelevant files. Instead of directly copying the DE blocks of a directory i-transaction into the file-system blocks during a crash recovery, the crash recovery operation first identifies the changed directory entries by comparing the two different DE blocks. If the inode pointed to by a changed directory entry is accessible, the entry is modified in the DE blocks in the file system.

Figure 5 shows an example of a file-system recovery under the `ijournaling` scheme. Initially, the file with inode number 3 has three external extents, which are used to access 24 blocks. Through some file operations, ten blocks (block numbers 50-59) and the corresponding external extent structure in block number 12 are freed. Then, six blocks (block numbers 74-79) are appended, and the external extent in block number 13 is modified. After the file operations, an fsync is called. Assume that there is a system crash before a normal journal is committed. The recovery module builds the inode structure including the external extent tree with the recorded i-transactions. By comparing the built inode with the corresponding inode in storage, the recovery module can identify the file-system changes by the logged fsync call, and can replay these changes. When the external extent block in block number 12 is freed, the original `ext4`

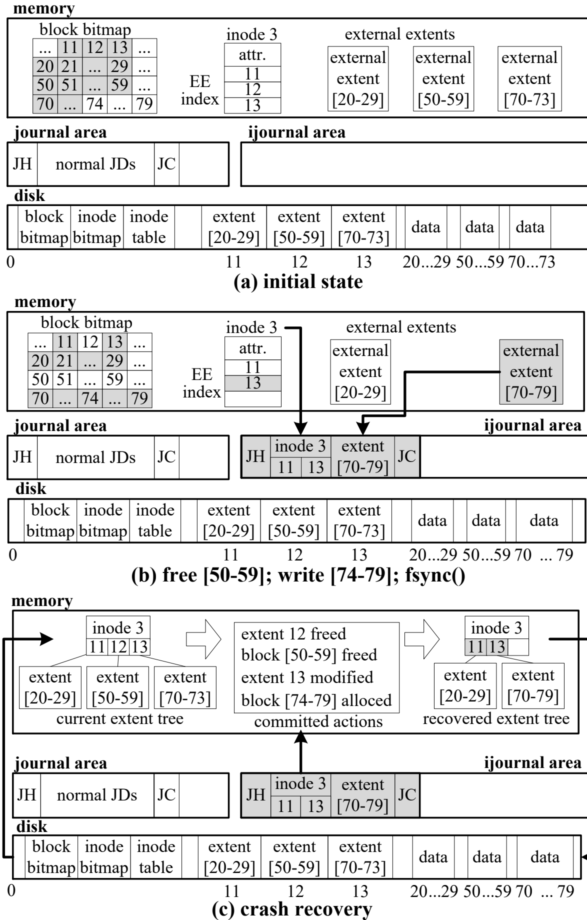


Figure 5: Example of a crash recovery.

journaling records a *revocation* block at the journal area to prevent an incorrect replay of the journal, which will cause a data corruption. The `ijournaling` scheme skips the writing of the revocation block because the following normal journaling will write it.

6 Experiments

6.1 Experiment Environments

To evaluate the effectiveness of `ijournaling`, an Android-based smartphone and a desktop computer were used. The smartphone was equipped with a Samsung Exynos 5410 (1.6-GHz Quad Cortex-A15 + 1.2-GHz Quad Cortex-A7) processor, 2 GB of DRAM, and 32 GB of eMMC. The Android OS version was 4.2.2 (Jelly Bean), and the Linux kernel version was 3.4.5. The desktop computer was equipped with an Intel i7-4790 3.6-GHz CPU, 16 GB of DRAM, and a Samsung 850 Pro SSD. The desktop Linux version was 4.7.3. The delayed allocation and ordered-mode journaling were used by default. The JBD2 thread conducts a journal commit operation at periodic 5-second intervals.

Linux kernel version 3.8 or later removes the ordering constraint of the ordered-mode journaling scheme [29]. Therefore, it is not necessary for an fsync call to wait until all data pages relevant to the journal transaction are flushed into the disk. However, the modified ordered-mode journaling scheme cannot guarantee file-system consistency similar to writeback-mode journaling. This flaw has been fixed at version 4.6.2 [20]. The Linux kernel versions used in our experiments (i.e., 3.4.5 and 4.7.3) keep a strict ordering constraint in ordered mode journaling.

6.2 Basic Comparison

We first measured the fsync latencies under different journaling schemes, normal and `ijournaling`, on the desktop and smartphone. The boosting technique [15] was optionally applied. We ran two programs for the experiments. One is an fsync-generating thread (*fsync tester*), which writes 80 KB of data in a file and calls an fsync repeatedly. We gave a delay of 0.1 second between `write()` and `fsync()` in order to generate many quasi-async requests. The other is the fio program [6], which generates 4 KB of sequential write requests for a file with a configurable write bandwidth of BG_{bw}. The fio program was used as a background process, which generated many data blocks to be flushed during the transaction commit operation. We determined the value of BG_{bw} at each experiment considering the storage bandwidth and the target foreground workload.

Figure 6(a) shows the results for the desktop when BG_{bw} = 400 MB/s. In the normal journaling scheme, the tail fsync latency at the 95th percentile is longer than 3.5 seconds. This is because the fsync must wait until a large number of dirty pages are flushed. In our measurement, 1.5 GB of data blocks at maximum were flushed during an fsync handling. However, `ijournaling` showed less than 0.2 seconds of fsync latency. The boosting technique was not very effective at reducing the fsync latency. Because SSD supports command queueing, most of the quasi-async requests were sent to storage without a long delay in the I/O scheduler. Once a request is sent to storage, the boosting cannot be applied because the host system cannot control the transferred requests.

Figure 6(b) shows the results for the smartphone when BG_{bw} = 50 MB/s. The `ijournaling` scheme also improved the fsync latency in the smartphone. Unlike with the desktop experiments, the boosting technique was effective because eMMC is slower than SSD, and does not support command queueing. By removing the CTX dependency, `ijournaling` significantly reduced the number of quasi-async requests and showed a shorter 95th percentile tail latency without boosting. We also implemented the logical logging scheme in the `ext4` file

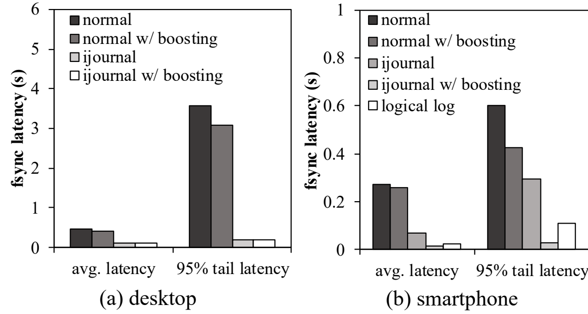


Figure 6: Fsync latency for different journaling schemes.

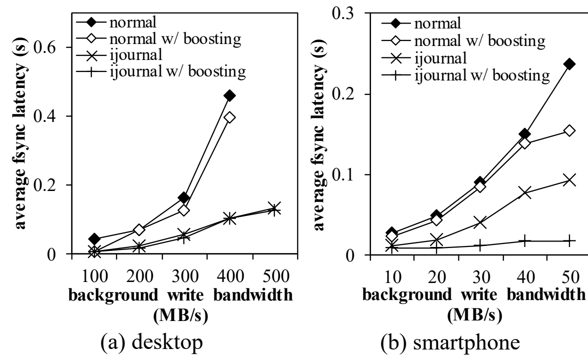


Figure 7: Changes in fsync latency when varying the number of concurrent file operations.

system. We followed the design of logical logging in ZFS. The delayed allocation was disabled in the logical logging experiments because the logical logging must generate an operation log for each file operation. The logical logging showed longer latencies compared with `ijournaling` using the boosting scheme. This is because the logical logging must flush a large size of logs.

To demonstrate the CTX dependency problem in legacy journaling, we measured the fsync latencies of *fsync tester* while varying the write bandwidth of the background process, i.e., BG_{bw} of fio. Figure 7 shows the average fsync latencies under four different journaling schemes. As the background write bandwidth increased, the fsync latency increased for the normal journaling scheme because more transactions were merged into a compound transaction. In particular, when BG_{bw} = 500 MB/s during the desktop experiment, the fsync system call was not completed until the background fio program was terminated. However, the `ijournaling` scheme showed short latencies even when BG_{bw} was high. The booting scheme was effective only when `ijournaling` is enabled.

Figure 8 compares the fsync latencies in legacy journaling under different block allocation policies. The experiment scenario is same as the scenario of Figure 6(a). When an `fsync()` was called while the flush thread was flushing dirty pages, the fsync latency became signifi-

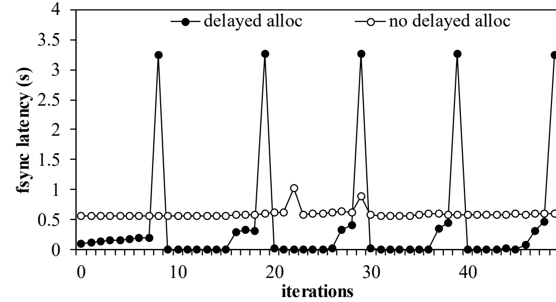


Figure 8: Fsync latency with and without delayed allocation at a desktop.

cantly high for the delayed allocation scheme. Otherwise, the latency was short. This is because data blocks are allocated when the flush thread is invoked. However, when the delayed allocation is disabled, there are no significant changes in the fsync latency. The average fsync latency is shorter when the delayed allocation is enabled. Because `ijournaling` can solve the CTX dependency problem, it can mitigate the fluctuating fsync latency problem of delayed allocation, and thus showed less than 0.2 seconds latencies as shown in Figure 6(a).

6.3 Manycore Scalability

A critical hurdle in implementing a manycore-scalable file system is the journaling contention, as reported in [24]. In particular, a single JBD2 thread handles all file-system transactions in `ext4`. Because `ijournaling` commits an fsync-related transaction in the system call service without calling the JBD2 thread, it improves the manycore scalability. In addition, each core has its own `ijournal` area, and thus, multiple fsync calls can be handled simultaneously at multiple processor cores.

In this experiment, we used a Xeon E5-2630 machine equipped with 2.4 GHz 8-core CPU, 64 GB of DRAM, and an Intel 750 NVMe SSD (400GB). The Linux kernel version was 4.7.3. Each core ran a process of sysbench [4], which generated 4 KB of sequential write requests on 128 files. Each `write()` operation was followed by an `fsync()` call. Figure 9 shows the changes in total bandwidth of the multiple sysbench processes while increasing the number of processor cores. Three different journal schemes were tested: normal journaling, `ijournaling` with one shared ijournal area, and `ijournaling` with a separate ijournal area per core.

The rate of increase in the total bandwidth decreased in normal journaling owing to its inter-transaction dependency problem. While JBD2 commits the transaction of a process, other processes must await the completion of the transaction commit. However, `ijournaling` improves the bandwidth significantly. In particular, when a separate `ijournal` area was allocated for each core

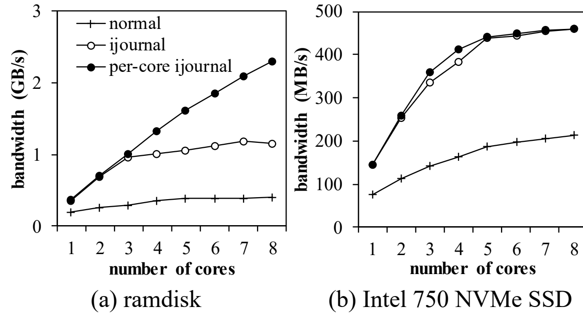


Figure 9: Multicore scalability.

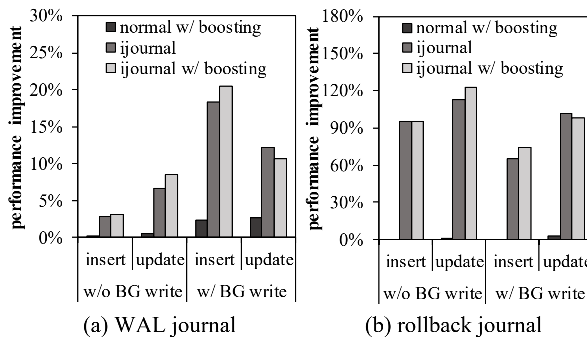


Figure 10: Mobibench results on a smartphone.

and the storage device used was a ramdisk, the total bandwidth increased linearly as the number of cores increased. When the storage device was an NVMe SSD, `ijournaling` showed a linear improvement in the total bandwidth at up to four cores. When more than four cores were used, however, the rate of bandwidth increase was reduced owing to the bandwidth limit of the SSD.

6.4 Benchmark Results

The fsync latency can affect the performance of an application if frequent fsync system calls are generated. To evaluate the performance gain from `ijournaling`, several benchmark programs were used. Figure 10 shows the results of Mobibench [16], which was designed for testing the SQLite performance on an Android-based smartphone. Because SQLite DBMS generates frequent fsync calls, its performance is closely related to the fsync latency. One-thousand DB transactions were generated, and two DB journaling modes, i.e., WAL journal and rollback journal modes, were used. The fio background application was optionally executed using BG_{bw} = 30 MB/s. We measured the performance improvement over the normal journaling scheme.

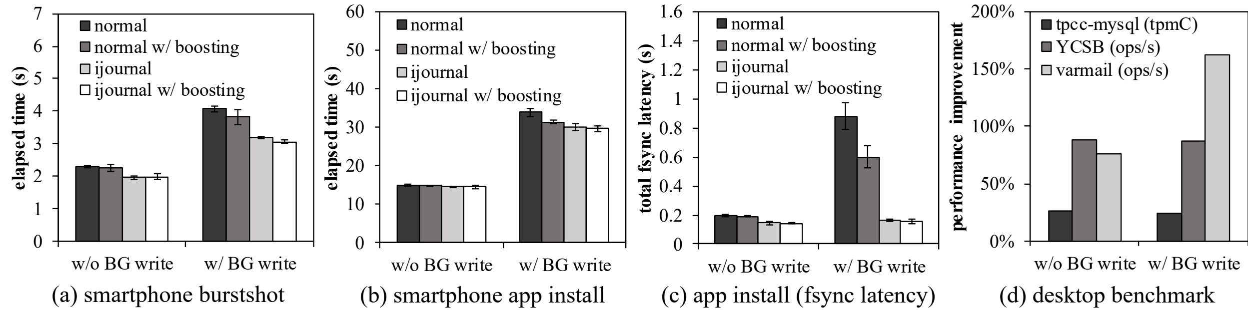
Even when no background process was used, and therefore no CTX dependency occurred, `ijournaling` improved the DB performance. The performance gain in WAL journal mode is due to the reduced journal write traffic of `ijournaling`. Whereas normal journaling must write multiple metadata blocks in a journal, `ijournaling` writes only two ijournal blocks for most cases because the modified inode entry is put into a 4 KB ijournal header block. The significant performance gain in rollback journal mode resulted from the CTX dependency problem. Although no background process was executed, the SQLite updated multiple files and the rollback journal file was truncated for every DB transaction. Owing to the truncated file, a *discard* command was included in the normal transaction. Therefore, the transaction commit was delayed owing to the handling of the *discard* command in normal journaling.

When a background process was executed, `ijournaling` showed significant performance improvements. In normal journaling, a journal commit invoked by an fsync call flushed about 25 MB of data blocks owing to the CTX dependency problem. The improvements achieved through boosting were poor because the SQLite application calls an `fsync()` immediately after a `write()` operation.

Figures 11(a) and (b) compares the performances of two smartphone applications under different journaling schemes. The camera burstshot program took 20 photos, and the application install program installed Angrybird. The fio background application was optionally executed using BG_{bw} = 30 MB/s. These applications also delete several files, and thus the transaction committed by an `fsync()` includes *discard* commands. Therefore, the `ijournaling` scheme reduced the execution times even when no background application was running. When a background application was executed, the performance improvements by `ijournaling` were more significant. Because the application install program is computing-intensive owing to the compilation work for java class files, its execution time is not significantly affected by the file-system performance. When we observed only the fsync latencies, however, there were significant performance gains by `ijournaling`, as shown in Figure 11(c).

Figure 11(d) shows the performance improvements by `ijournaling` for the desktop benchmarks. Three workloads were used: Percona's tpcc-mysql [5], YCSB [11], and FileBench's varmail [2]. In the tpcc-mysql workload, the DB page size was configured to 4 KB, ten warehouses were used, 16 connections were applied, and the running time was 100 seconds. In the case of the YCSB workload, the MySQL system and a update-heavy workload (i.e., Workload A), which has 50% reads and 50% updates, were used. The varmail workload was run with the default option. The fio background application was optionally executed using BG_{bw} = 200 MB/s.

Even when no background application was used, `ijournaling` improved the performance on the desktop benchmarks because these workloads generated multiple concurrent threads that called an `fsync()` simultane-



(a) smartphone burstshot (b) smartphone app install (c) app install (fsync latency) (d) desktop benchmark

Figure 11: Real application benchmarks.

ously, and thus the inter-transaction dependency was severe. For example, for the tpcc-mysql workload, 28.3% of all fsync calls were delayed owing to the IT dependency. In addition, `ijournaling` reduced the journal write traffic by 56% owing to its file-level journaling scheme for the YCSB workload. Most of the transactions committed by `fsync()` had *discard* commands in the varmail workload.

When a background application was executed, there were no further performance improvements compared with the case of no background process for the tpcc-mysql and YCSB workloads because these workloads had an excessive inter-transaction problem. The varmail workload is more fsync-intensive. In the case of the varmail workload, while a JBD2 was committing a normal transaction, many fsync calls were delayed owing to the IT dependency problem in normal journaling. Therefore, the performance gain by `ijournaling` was more significant.

6.5 Crash Recovery Tests

Finally, we conducted crash recovery tests under four file-system modification scenarios. During each test scenario, a crash was triggered and the system was restarted. The file-system operations generated during the tests were printed out, and recorded on a monitoring computer. The required file-system changes were derived from the logs, and we were able to check whether the file-system changes were correctly recovered. In addition, we also checked the file-system consistency using the `e2fsck` utility.

In the first scenario, one-thousand files were created sequentially, among which only odd-numbered files were fsynced. A system crash was triggered before normal periodic journaling was invoked. This scenario was able to test whether `ijournaling` can recover the inodes of fsynced files and whether the recovered directory entry of the parent directory has the entries of only committed files. In the second scenario, a file was created, and 4 KB of data were appended to the file repeatedly. After each 4 KB write, an fsync was called. To make external extent blocks, a crash was triggered after the file size reached larger than 1 GB. This test covered the correctness of external extent tracking. For the third scenario, more than two depths of directories were made, and an fsync for a file at leaf node was called. This scenario was able to check whether all related parent directories were recovered. For the last scenario, ten threads were generated, each of which executed file operations randomly selected among `mkdir`, `create`, `write`, `truncate`, `unlink`, and `fsync`. For each of these scenarios, we ascertained that `ijournaling` can correctly recover the fsynced files and their related directories without any file-system inconsistencies.

7 Conclusion

We rely on the journaling of data updates for file-system consistency, and synchronous writes for data durability. However, latency-sensitive synchronous operations such as an `fsync()` system call can be delayed under the compound transaction scheme of the current journaling technique. Because a compound transaction includes irrelevant data and metadata, as well as those of fsynced file, the fsync latency can be unexpectedly long. In this paper, we first analyzed the affecting factors that may delay an fsync operation, and proposed a novel hybrid journaling technique, called `ijournaling`, which journals only the related file-level transactions of an fsync call and recovers the file-system consistency through file-level journals upon a crash recovery. Experiments using real devices showed that there are significant improvements to the fsync latencies when using `ijournaling`, and that many synchronous applications can benefit from the proposed `ijournaling` technique.

Acknowledgements

We would like to thank Theodore Ts'o, who was our shepherd, and anonymous reviewers for their valuable comments and suggestions. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP). (No. 2016R1A2B2008672)

References

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

[2] Filebench. http://filebench.sourceforge.net/.

[3] SQLite. https://sqlite.org.

[4] Sysbench. https://github.com/akopytov/sysbench.

[5] tpcc-mysql. https://github.com/Percona-Lab/tpcc-mysql.

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

[7] Jeff Bonwick and Bill Moore. ZFS: The last word in file systems. http://wiki.illumos.org/download/attachments/1146951/zfs_last.pdf, 2007.

[8] Li-Pin Chang, Po-Han Sung, and Po-Hung Chen. Fast file synching for applications in flash-based android devices. In *Proceedings of the 3rd Non-Volatile Memory Systems and Applications Symposium*, pages 1–6, 2014.

[9] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP'13, pages 228–243, 2013.

[10] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, pages 101–116, 2012.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC'10, pages 143–154, 2010.

[12] Jonathan Corbet. Solving the ext3 latency problem. http://lwn.net/Articles/328363/.

[13] Rasha Eqbal. ScaleFS: A multicore-scalable file system. Master's thesis, Massachusetts Institute of Technology, August 2014.

[14] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, Inc., 1993.

[15] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. Boosting quasi-asynchronous I/O for better responsiveness in mobile devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 191–202, 2015.

[16] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. AndroStep: Android storage performance analysis tool. In *Software Engineering Workshops*, pages 327–340, 2013.

[17] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O stack optimization for smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference*, ATC'13, pages 309–320, 2013.

[18] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A scalable file system on fast storage devices. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ATC'15, pages 249–261, 2015.

[19] Yunji Kang and Dongkun Shin. Per-block-group journaling for improving fsync response time. In *Proceedings of the 18th IEEE International Symposium on Consumer Electronics*, pages 22–25, 2014.

[20] Jan Kara. ext4: fix data exposure after a crash. https://patchwork.kernel.org/patch/9156691/, 2016.

[21] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pages 81–96, 2014.

[22] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Marshall Kirk Mckusick. Ffsck: The fast file-system checker. *ACM Transactions on Storage*, 10(1):2:1–2:28, 2014.

[23] 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 Ottowa Linux Symposium*, 2007.

[24] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proceedings of the 2016 USENIX Annual Technical Conference*, ATC'16, pages 71–85, 2016.

[25] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Transactions on Computer Systems*, 26(3):6:1–6:26, 2008.

[26] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *Proceedings of the 2005 USENIX Annual Technical Conference*, ATC'05, pages 105–120, 2005.

[27] Kai Shen, Stan Park, and Meng Zhu. Journaling of journal is (almost) free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 287–293, 2014.

[28] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *Proceedings of the 1996 USENIX Annual Technical Conference*, ATC'96, pages 1–14, 1996.

[29] Theodore Ts'o. ext4: remove calls to ext4_jbd2_file_inode() from delalloc write path. http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=f3b59291a69d0b734be1fc8be489fef2dd846d3d, 2012.