exF2FS: Transaction Support in Log-Structured Filesystem

Joontaek Oh, Sion Ji, Yongjin Kim, and Youjip Won, *KAIST*

https://www.usenix.org/conference/fast22/presentation/oh

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

978-1-939133-26-7

Open access to the Proceedings
of the 20th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

exF2FS: Transaction Support in Log-Structured Filesystem

Joontaek Oh Sion Ji Yongjin Kim Youjip Won
Department of Electrical Engineering, KAIST

Abstract

In this work, we present exF2FS, a transactional log-structured filesystem. The proposed filesystem consists of three key components: Membership-Oriented Transaction, Stealing-Enabled Transaction, and Shadow Garbage Collection. Membership-Oriented Transaction allows the transaction to span multiple files where the application can explicitly specify the files associated with a transaction. Stealing-Enabled Transaction allows the application to execute the transaction with a small amount of memory and to encapsulate many updates, e.g., hundreds of files with tens of GBs in total size, with a single transaction. Shadow Garbage Collection allows the log-structured filesystem to perform garbage collection without affecting the failure-atomicity of ongoing transactions. The transaction support in exF2FS is carefully trimmed to meet the critical needs of the application while minimizing the code complexity and avoiding any performance side effects. With exF2FS, SQLite multi-file transaction throughput increases by 24× against the multi-file transaction of stock SQLite. RocksDB throughput increases by 87% when it implements the compaction as a filesystem transaction.

1 Introduction

Modern applications strive to protect their data in a crash-consistent manner which is often split over multiple file abstractions. In the absence of proper transaction support from the underlying filesystem, the application employs complicated protocols to ensure the transactional updates that span multiple files, yielding long sequence of writes and `fsync()`'s. Text editors, such as `vim` and `emacs`, use atomic `rename()` to save the updated file atomically [1]. For a transaction that updates the multiple database files, the library-based embedded DBMS, SQLite, maintains the separate journal file for each database file [69], yielding excessive `fdatasync()` calls and a large write amplification [31]. Compaction operation of the modern LSM-based key-value store, such as RocksDB [22], maintains the state of the merge-sort at a separate journal file known as the *manifest file*. For the failure-atomicity of the compaction operation, the key-value storage engine flushes the output files separately and also flushes the global state of the compaction to the manifest file. With the transaction support from the filesystem, the application can replace the multiple `fsync()`'s for each output files and the manifest file with a single filesystem transaction, rendering higher performance by eliminating redundant IO's.

Despite the clear benefits of supporting transactions, it remains a challenge for the operating system and filesystem. To successfully deploy the transaction enabled system, the right balance must be found among the four requirements: easy to use, code complexity, degree of ACID support and performance. Unfortunately, achieving one of these is often at the cost of another. The system level supports for transaction can largely be categorized into four: native operating system support [57, 61, 61, 75], kernel level filesystem [14, 26, 27, 46, 55, 66, 72, 78], user level filesystem [24, 48, 54] and transactional block device [12, 28, 32, 52, 58, 65]. Supporting transaction as the first-class citizen of the operating system is ideal; however, it requires substantial change in the operating system. Transaction support from the user level filesystem exploits the user level DBMS to provide full ACID transaction [24, 48, 54]. ACID support comes at the cost of the performance. The transaction support from the kernel level filesystem can further be categorized with respect to the degree of ACID support: full ACID semantics [27, 66], ACD without isolation support [55, 72] or even AC without isolation and durability support [34]. An F2FS transaction [34] supports only the atomicity, neither isolation nor durability. The transaction in F2FS cannot span multiple files. Ironically, despite its barest minimum support for the transaction, F2FS is the only filesystem that successfully deploys its transaction support to the public. F2FS's transaction support has a specific target application: SQLite. With atomic write of F2FS, SQLite can implement the transaction without the rollback journal file and can eliminate the excessive flush overhead [31, 64].

In this work, we revisit the issue of providing the filesystem-level transaction support. In particular, we focus the domain of interests to the log-structured filesystem. Most of the preceding works on the transactional filesystems use the journaling filesystem as a baseline filesystem [27, 66, 72]. These works exploit the journaling layer of the filesystem to provide the transaction capability. F2FS, the log-structured filesystem designed for flash storage, recently gained wide popularity on smartphone platforms [56] and is beginning to expand into cloud platforms [6]. Few works have dealt with the transaction support in the log-structured filesystem. Seltzer et al. [62] is the nearest effort; however, their work is limited in terms of the transaction support. Their study does not support multi-

file transaction, stealing in the transaction, nor the conflict handling between a transaction and the garbage collection.

In this study, we present transaction support in the log-structured filesystem with three design objectives; (i) the transaction should be able to span multiple files, including the directory, (ii) the transaction should be able to handle large amounts of updates and (iii) the transaction should not be affected by the execution of garbage collection. Each of these requirements looks plain and essential from the application's point of view. Unfortunately, developing the transactional log-structured filesystem which satisfies these simple and plain requirements is a non-trivial exercise which calls for substantial changes in the underlying filesystem from the aspect of design as well as implementation; developing a new transaction model, redesigning the filesystem's page reclamation procedure and redesigning the garbage collection procedure. We find that few modern transactional filesystems address any of these essential requirements in its transaction management with sufficient maturity. To allow the transaction to span multiple files, we develop *Membership-Oriented Transaction Model*. To allow the transaction to handle large size transactions which may consist of hundreds of files with tens of GBs of data, we develop *Stealing* for the filesystem transaction. To prohibit the garbage collection from interfering with the ongoing transaction, we develop *Shadow Garbage Collection*. The main contributions of this work are as follows.

- **Membership-Oriented Transaction**. In Membership-Oriented Transaction, the filesystem maintains a kernel object, *Transaction File Group* that specifies the set of files, including directories, associated with the transaction. With Membership-Oriented Transaction, the application can explicitly specify the files that are subject to the transaction.
- **Stealing**. We allow dirty pages of uncommitted transactions to be evicted and yet guarantee the atomicity of the transaction. We develop *Delayed Invalidation* and *Relocation Record* to realize Stealing in the filesystem transaction. Delayed Invalidation prohibits the old disk locations of evicted pages from being garbage-collected until the transaction commits. Relocation Record maintains undo and redo information to abort and commit evicted pages, respectively.
- **Shadow Garbage Collection**. We develop *Shadow Garbage Collection* to prohibit the garbage collection module from making the dirty page of the uncommitted transaction prematurely durable and recoverable. Shadow Garbage Collection allows the filesystem to perform garbage collection transparently to the ongoing transactions.

We implement these features in F2FS. We call the newly developed filesystem extended F2FS (*exF2FS*). exF2FS improves the SQLite performance by 24× against stock SQLite and reduces the write volume to 1/6 compared to the PERSIST journal mode of SQLite. It improves RocksDB performance by 87% in the YCSB workload-A [13]. Special care has been taken not to change any on-disk structure of the existing F2FS so that exF2FS can mount the existing F2FS partition.

2 Background and Motivation

2.1 Multi-file Transaction

Multi-file transaction is an essential part of the modern software. The followings are a few examples of multi-file transaction method currently being used.

Maintaining the browsing history in the web browser. The Chrome browser maintains user browsing activity, such as visited `URL`'s, the list of downloaded files, an access history for each `URL` and the list of the most frequently visited `URL`'s. Chrome maintains each of these in a separate file and updates these files in failure-atomic fashion. For failure-atomicity, Chrome uses SQLite in updating these files [50] which renders excessive IO. The inefficiency of SQLite transactions will be explained later in this study.

Compaction in LSM-based key-value Store. Compaction is a process of merge-sorting several SSTables with overlapping intervals into a sequence of the output files with non-overlapping intervals [21]. The failure-atomicity of the compaction operation invokes `fsync()`'s for each output file and the parent directory and flushes the global state of the transaction to a special file called the *manifest file* [5, 23, 33]. In "load" workload of YCSB [13], a single compaction of RocksDB can create as many as 198 output files (over 200 `fsync()`'s) for a total of 13.3 GB.

Software Installation. Updating or installing a new software involves downloading and modifying hundreds of files and updating the associated directory in a failure-atomic manner. The partial completion of installation or update often leads to an unstable system [15, 42, 45, 73].

Mail client. `MAILDIR` IMAP format maintains the mailbox and the message as a directory and a file in the directory, respectively [2, 16, 20]. The email client updates the message files and the associated directory in transactional fashion. In the absence of transaction support from the underlying filesystem, mail clients use the expensive atomic rename to manage the mailbox and the message in transactional fashion [9, 70].

2.2 Multi-file Transaction and SQLite

SQLite is serverless embedded DBMS widely used in various applications: mobile applications such as Android Mail and Facebook App, desktop applications such as Gmail and Apple iWork [25, 27] and distributed filesystems such as Lustre [8] and Ceph [74]. These applications use SQLite to persistently manage the updates on the multiple files in failure-atomic fashion. To understand how the SQLite can benefit from the transaction support of the underlying filesystem, we instrument the IO behavior of the SQLite's multi-file transaction.

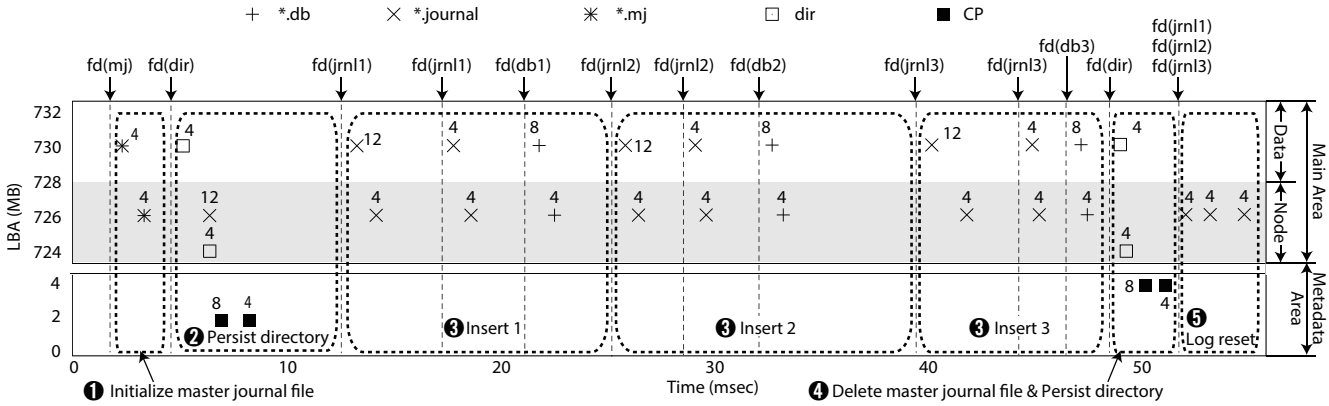


Figure 1: A multi-file transaction with three `insert()`'s in F2FS. Record size = 100 Byte, PERSIST mode. The number in each mark represents the number of KB written, Device: Samsung 850 PRO, `fd`: `fdatasync()`, `mj`: master journal file, `dir`: parent directory, `jrnl`: journal file, `db`: database file, `cp`: checkpoint

While the application can become simpler when using SQLite to persistently manage the data, it suffers from significant write amplification and excessive flush due to the page granularity physical logging and the file-backed journaling of SQLite [31, 64]. A single `insert()` of SQLite incurs five `fdatasync()`'s with 40 KB of `write()` [76]. A few studies have been dedicated to improving the extreme IO inefficiency of SQLite transaction [27, 31, 34, 36, 40, 53]. All these efforts are limited to improve the IO overhead in the transaction with a single database file.

SQLite constructs the multi-file transaction as a collection of the single file transactions and a few flushes to record the global state of the multi-file transaction at the *master journal file*. SQLite implements the multi-file transaction in the four steps listed below. Step 1 and Step 3 are for updating the master journal file. Step 2 and Step 4 are for executing the series of the single file transactions. Fig 1 shows how each of these steps is associated with the IO behavior through the physical experiment. Here, a transaction consists of three inserts to three different database files.

1. **Initializing the master journal file.** SQLite records the name of the journal files in the master journal file. Then, it flushes the master journal file (❶ in Fig. 1) and the updated directory to the disk (❷ in Fig. 1).

2. **Logging and Database Updates.** SQLite logs the undo records at the journal files and updates the database files. Each file is updated in the same way as in the single database transaction (❸ in Fig. 1). There are three ❸'s in Fig. 1 each of which corresponds to a single `insert()`.

3. **Deleting the master journal file.** As a mark of successful commit, SQLite deletes the master journal file and makes the associated directory durable (❹ in Fig. 1).

4. **Reset Logs.** SQLite resets the journal files and flushes them (❺ in Fig. 1).

In the Fig. 1, the X-axis and Y-axis denote the time and LBA, respectively. Here, we explicitly specify three regions of F2FS: the metadata area, data region of the main area, and node region of the main area. When SQLite flushes the dirty file block through `fdatasync()`, the underlying F2FS flushes not only data blocks but also the associated node block to the data region and the node region, respectively. In (❶), flushing the master journal file (`fd(mj)`) renders two separate 4 KB IO's to the disk: one for flushing the data block and the other for flushing the node block. The data block and the associated node block need to be made durable in order for guaranteeing the integrity of the filesystem. Each `insert()` has three `fdatasync()`'s (❸); the first and the second `fdatasync()` are for flushing the rollback journal file. The third one is for flushing the database file. In (❹), SQLite deletes the master journal file and persists the parent directory. When unlinking the master journal file becomes durable, the transaction is committed. In (❺), SQLite resets the rollback journal files of the transaction.

As in Fig. 1, the IO overhead of SQLite multi-file transaction is somewhat disastrous; inserting three 100 Byte records renders fifteen `fdatasync()`'s and 180 KBs write to the disk.

2.3 Log-structured Filesystem, F2FS and Atomic Write

We use F2FS [39] as a baseline log-structured filesystem. F2FS has a number of key design features that differentiate itself from the original log-structured filesystem designs [38, 60, 63]. Among them, the two features that we focus on in this work are *block allocation bitmap* and *dual log partition layout*. To realize *Stealing* and *Shadow Garbage Collection*, the way in which F2FS manipulates and updates the block allocation bitmap and the two logs must be overhauled.

The first is block allocation bitmap. In the original log-structured filesystem design [60, 63], there is no explicit data

structure that specifies whether a given block in the filesystem partition is allocated or not. The filesystem determines that a block in the filesystem partition is allocated if it is reachable through the file mapping. F2FS maintains the block allocation bitmap to denote whether a given block in the filesystem is valid or not. The second is dual log partition layout. Legacy log-structured filesystems treat the filesystem partition as a single log. They cluster the data block and the associated filemap[1] together and flush them in a single unit. F2FS organizes the filesystem partition with two separate logs: the data region and the node region. F2FS places the data block and the node block at the associated regions, respectively. Unlike the legacy log-structured filesystems, F2FS writes the data blocks and node blocks separately. To preserve the filesystem integrity against a system crash, F2FS ensures that the data blocks are made durable before the associated node blocks. Due to this ordering mechanism in F2FS, the block trace for writing the data block appears before the block trace for writing the node block in each pair of writes for the data block and the node block, as shown in Fig. 1.

F2FS provides the atomic write feature [34]; an application can write multiple blocks for a single file in a failure-atomic manner. This feature is primarily for addressing the excessive IO overhead of the SQLite's single file transaction.

```
start_atomic_write(fd) ;
write(fd, block1) ;
write(fd, block2) ;
commit_atomic_write(fd) ;
```

For atomic write, F2FS maintains the list of the dirty pages in the inode. When the transaction updates a file block, it inserts the dirty page to the per-inode dirty page list and pins the dirty page in memory. When the transaction commits, the filesystem unpins the dirty pages in the per-inode dirty page list and flushes the dirty pages and the associated node blocks that hold the updated file mapping to the disk. Since the atomic write pins the dirty pages in memory until it commits, F2FS, by design, cannot support Stealing in its atomic write transaction. When the transaction commits, F2FS sets the `FSYNC_BIT` flag at the node block. If more than one node blocks are flushed, atomic write places `FSYNC_BIT` flag at the last node block. F2FS sets the `FSYNC_BIT` flag at the node block to mark itself subject to the roll-forward recovery.

The log-structured filesystem periodically checkpoints its state, e.g. the updated file mapping, the updated bitmap (only for F2FS), and the disk location of the last block of each log. When the filesystem crashes, the recovery module recovers the state of the filesystem with respect to the most recent checkpoint information. After rollback recovery, the recovery module scans the logs from the last location, finds the node block with `FSYNC_BIT`, i.e. the transaction which has finished successfully after the most recent checkpoint, and recovers the associated file.

[1] F2FS calls blocks holding the file mapping information as a *node block*.

3 Design

We define three constraints which the transactional log-structured filesystem should satisfy: (i) Multi-File Transaction, (ii) Stealing and (iii) Transaction-aware Garbage Collection. We develop a transactional log-structured filesystem, exF2FS, that satisfies these constraints. The key technical components of exF2FS are Membership-Oriented Transaction, Stealing enabled Transaction, and Shadow Garbage Collection. Each component is summarized below.

Membership-Oriented Transaction (Section 3.1): The transaction of F2FS cannot span multiple files since it maintains the dirty pages of a transaction in a per-inode basis. In this study, we develop a new transaction model, called *Membership-Oriented Transaction*. In Membership-Oriented Transaction, the filesystem defines *Transaction File Group*, a set of files whose updates need to be handled as a transaction and maintains the dirty pages of a transaction for each transaction file group. In Membership-Oriented Transaction, a transaction can span multiple files and the application can explicitly specify the files that are subject to the transaction.

Stealing enabled Transaction (Section 4): For Stealing, the page reclamation procedure is overhauled so that the result of the page reclamation can be undone when the filesystem reclaims the dirty page of the uncommitted transaction. With Stealing enabled Transaction, the proposed filesystem can support large size transactions, e.g. hundreds of files with tens of GBs of data, with a small amount of memory.

Shadow Garbage Collection (Section 5): Garbage collection can make the dirty page associated with an uncommitted transaction durable and can checkpoint the updated file mapping prematurely before the transaction commits. We develop *Shadow Garbage Collection* to isolate the garbage collection from the uncommitted transaction.

3.1 Membership-Oriented Transaction Model

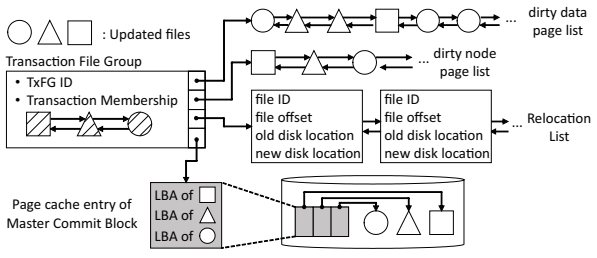


Figure 2: Concept of a Transaction: Transaction File Group, Dirty Page List, Relocation List and Master Commit block

In this work, we propose a new transaction model called *Membership-Oriented Transaction*. In this model, we define the new kernel entity, *Transaction File Group*. Transaction File Group is a set of files whose updates need to be treated

as a single transaction, and consists of Transaction Membership (a set of inodes), dirty page list, Relocation List, and Master Commit Block, as in Fig. 2. We use hash table for the namespace of Transaction File Group objects, which is widely used to organize the namespace for the kernel objects, e.g. semaphore and pipe [19].

With Transaction File Group, the application can specify the files that need to be included in the transaction. The dirty page list is a set of dirty pages for the transaction member files. There are two separate dirty page lists: the dirty data page list and the dirty node page list. A Relocation List is a set of Relocation Records. Relocation Record contains an information for the evicted page: file ID, file offset, old disk location, and new disk location. Master Commit Block holds the disk locations of the last node blocks for each file in the transaction membership. Transaction File Group, along with the Master Commit Block, allows the transaction to span multiple files. Relocation List is used for Stealing and Shadow Garbage Collection.

3.2 Transaction API's

```
1 id = create_tx_file_group();
2 for (i = 0; i<3; i++)
3     add_tx_file_group(db[i], id);
4 start_tx_file_group(id);
5 write(db[0], buf, 4096);
6 write(db[1], buf, 4096);
7 write(db[2], buf, 4096);
8 commit_tx_file_group(id);
9 delete_file_group(id);
```

Figure 3: Multi-file transaction in exF2FS

The application creates a Transaction File Group with an explicit call. When an application creates a Transaction File Group, and ID of the Transaction File Group is returned to the application. The application can add or remove a file to and from the Transaction File Group. To avoid a conflict between ongoing transactions, we forbid the application to add or to remove a file to and from the Transaction File Group in an ongoing transaction. When the transaction creates a file, the newly created file inherits the membership from the parent directory, which is called *Membership Inheritance*. Membership Inheritance saves the file created by the transaction from the transaction conflict since the newly created file is added to the Transaction File Group before it becomes externally visible. When the directory is removed from the Transaction File Group, child files who inherited the membership are also removed from the Transaction File Group.

The application specifies the ID of the Transaction File Group when it starts the transaction. When the transaction starts, the filesystem sets the flag at the inodes of the transaction member files denoting that the files are associated with the ongoing transaction. The application specifies the ID of the Transaction File Group when it calls for the transaction commit. exF2FS offers the API's for transaction abort and transaction delete. When the application calls for deleting a Transaction File Group, the Transaction File Group and the associated objects are deallocated if there is no ongoing transaction for the Transaction File Group, If there is an ongoing transaction when the application calls for deleting the transaction file group, exF2FS first aborts the transaction and then deletes the Transaction File Group. Table 1 illustrates the API's and pseudo-code of exF2FS, respectively.

In exF2FS, a transaction can include a directory update such as `rename()`, `unlink()`, and `create()`. The F2FS transaction does not support the directory update in the transaction.

3.3 Commit and Abort

When the transaction updates the file in the transaction file group, it inserts the updated page cache entry to the dirty data page list of the Transaction File Group.

In committing a transaction, the filesystem prepares the dirty data pages, the dirty node pages and the Master Commit Block for transaction commit. First, the filesystem inserts the dirty data pages in the dirty page list to the active data segment and obtains the disk location for each dirty data page. Second, the filesystem updates the associated node pages with the new disk location of each data page, inserts the updated node pages to the dirty node page list and determines the disk location for each dirty node page. Third, the filesystem allocates the Master Commit Block and stores the disk location of each node page in the dirty node page list at the Master Commit Block. The filesystem then sets `FSYNC_BIT` flag at the Master Commit Block.

Once these steps are complete, exF2FS flushes the dirty data pages, the dirty node pages and Master Commit Block. It ensures that the Master Commit Block becomes durable only after the data blocks and the node blocks become durable. *Master Commit Block* is the key component to fabricate the dirty pages of the multiple files into a single multi-file transaction. After the Master Commit Block becomes durable, the filesystem scans the Relocation List and invalidates the old disk locations of the Relocation Records. The details about the Relocation Record will be explained in Section 4.3.

If the transaction aborts, all entries in the dirty page list are discarded and the dirty page list becomes empty. When the aborted transaction has the evicted pages, the file mapping information is revoked to its original location based upon the Relocation Records.

When the system crashes, the recovery module performs rollback recovery and places the filesystem state to the most recent checkpoint. Then, exF2FS performs roll-forward recovery; it scans the log starting from the last logging offset recorded at the checkpoint. When it encounters Master Commit Block, the recovery module examines it and identifies the disk locations of the node blocks of the files in the transac-

API		Arguments	Return value	Description
Transaction File Group	`create_tx_file_group`	None	`int key`	Create a transaction file group
	`delete_tx_file_group`	`int key`	`int err`	Deallocate a transaction file group corresponding to the `key`
	`add_tx_file_group`	`int fd` `int key`	`int err`	Add a file `fd` to a transaction file group corresponding to the `key`
	`remove_tx_file_group`	`int fd` `int key`	`int err`	Remove a file `fd` from a transaction file group corresponding to the `key`
Transaction	`start_tx_file_group`	`int key`	`int err`	Start a transaction corresponding to the `key`
	`commit_tx_file_group`	`int key`	`int err`	Commit a transaction corresponding to the `key`
	`abort_tx_file_group`	`int key`	`int err`	Abort a transaction corresponding to the `key`

Table 1: API's in exF2FS

tion. Then, the recovery module of exF2FS uses roll-forward recovery routine of stock F2FS to recover the file associated with each node block. If the system crashes before the Master Commit Block becomes durable, the transient state of the transaction that was in-memory is completely lost. Through this recovery mechanism, exF2FS guarantees the atomicity and the durability of the transaction.

3.4 Concurrency Control and Isolation

Not being a full-fledged DBMS, we use coarse file-granularity concurrency control; a file can belong to only one Transaction File Group at a time. When adding a file to the Transaction File Group, the application checks if it is already in another Transaction File Group. If the file is already in another Transaction File Group, `add_tx_file_group` returns an error.

We leave the isolation support to the application as the other transactional filesystems do [11, 47, 72]. As the general purpose filesystem, it is difficult to meet all different levels of isolation requirements from a wide variety of applications at the same time. We carefully consider that the limited support of the filesystem for the isolation becomes redundant at best, unless the isolation level supported by the filesystem is well aligned with the isolation level required by the application. Text editor, application installer, git and the compaction of LSM-based key value store do not require the isolation [10]. SQLite and MySQL implement the multiple levels of isolation by themselves [49, 68]. In these applications, the limited support of filesystems for the isolation cannot be of much help. TxFS supports the isolation of "Repeatable Read" [4, 27]. It is overly strong for Text editor, and is too relaxed for some applications, such as "Serializable Read" in SQLite. SQLite must implement isolation of "Serializable Read" in its own database layer using the shared lock [67] even when using TxFS as the underlying filesystem. Filesystem support for the isolation has a cost. According to our experiments, the isolation support of TxFS renders 10% performance overhead due to the overhead of creating the shadow copies of the updated pages in the transaction. However, one limitation resulting from the absence of isolation support is that other processes cannot concurrently add, delete, or rename files in a directory that is included in another process's transaction. Supporting concurrent directory modifications is left for future work.

4 Stealing in the Filesystem Transaction

Stealing denotes the buffer management policy that allows the eviction of dirty pages of the uncommitted transaction [59]. The Steal policy in DBMS and the page reclamation of the Operating System (or the filesystem) [41] are the different manifestations of the same essential behavior: evicting a dirty page to the disk and freeing up the physical memory. While the two share the essential behavior, the two lie at the other end of extreme. For Stealing in the database transaction, DBMS prohibits the evicted dirty page from being externally visible (isolation) and/or undoes the Steal in case of transaction abort (atomicity). When the OS reclaims the file-backed dirty page, the result of the page eviction becomes externally visible and cannot be undone. In the journaling filesystem, the old file block is overwritten with the evicted page and in the log-structured filesystem, the old file block of the evicted page becomes unreachable due to the file mapping update.

4.1 Stealing and the Filesystem

The support for Stealing in the existing transactional filesystems bears substantial room for improvement. None of the TxFS [27], F2FS [34], Isotope [65], and Libnvmmio [11] support Stealing in the transaction. TxFS cannot support Stealing in a transaction due to its fundamental design limit. TxFS's support for transaction is built on top of EXT4 journaling. EXT4 journaling pins the log blocks in memory until the journal transaction commits. EXT4 limits the size of the journal transaction (256 MB by default). When the size of a journal transaction reaches its limit, the EXT4 journaling module commits the journal transaction. In EXT4, the dirty pages associated with a single system call can be split into two or more journal commits. TxFS must prohibit this from happening since it can make the transient state of the transaction durable prematurely, compromising the atomicity of the transaction. For atomicity guarantee, TxFS simply aborts the transaction when the transaction size exceeds its limit. F2FS pins the dirty pages of a transaction in memory until it commits. F2FS aborts *all* outstanding transactions [35] when the dirty pages of an uncommitted transaction exceeds a certain threshold (15% of the total physical page frames by default). CFS supports stealing [47]. However, CFS relies on a non-existent

transactional block device [32] for its support for Stealing. AdvFS [72] supports Stealing with the commodity hardware. AdvFS uses the writable file clone for the transactional updates. When the transaction commits, the filemap is updated to refer to the updated file blocks that are written in out-of-place manner. This nature allows AdvFS to freely support Stealing. However, a transaction in AdvFS can fragment the file since the filesystem deletes the old file blocks each time the transaction commits. The file defragmentation overhead of AdvFS is yet-to-be known. Our analysis on the AdvFS is limited since AdvFS is proprietary filesystem and the source code of its transaction module is not publicly available.

4.2 Delayed Invalidation and Node Page Pinning

In this study, we enable Stealing in the filesystem transaction. The log-structured filesystems [39,60,63] evict the dirty pages as follows: the evicted page is written to the new disk location, the old disk location of the evicted page is invalidated and the file mapping (node page in F2FS) is updated to refer to the new location of the associated file block. This page eviction routine cannot be used with Stealing for two critical reasons. First is the invalidation of the old disk location. Being invalidated, the old file block can be garbage collected and can be recycled before the transaction commits. If the old file block is recycled before the transaction commits, the transaction cannot be revoked when the transaction aborts. Second is the premature checkpoint of the updated node page. When the dirty page is evicted, the updated node page which contains the updated file mapping can be checkpointed if the filesystem runs the periodic checkpoint operation before the transaction commits. Then, the updated node page checkpointed to the disk refers to the new disk location of the evicted page of the uncommitted transaction. If the filesystem crashes before the transaction commits, the recovery module can recover the evicted page of the uncommitted transaction with respect to the most recent file mapping found on the disk. Subsequently, the filesystem can be recovered to the incorrect state.

There are two key issues that need to be addressed for supporting Stealing-enabled Transaction in the log-structured filesystem: (i) prohibit the old disk location from being garbage collected until the transaction commits and (ii) prohibit evicted pages of uncommitted transactions from being recovered after the system crash. To address the first issue, we propose *Delayed Invalidation*. In Delayed Invalidation, after evicting the dirty page from the uncommitted transaction, the filesystem postpones invalidating the old disk location until the transaction commits. To address the second issue, we propose *Node Page Pinning*. In Node Page Pinning, the filesystem pins the updated node page until the transaction commits to prohibit the updated node page from being checkpointed prematurely.

For Delayed Invalidation and Node Page Pinning, we introduce a new in-memory object, *Relocation Record*. Relocation Record holds the information associated with the page eviction. Relocation Record contains the file block ID (inode number and file offset), the old disk location, and the new disk location of the file block of the evicted page. With Relocation Record, the filesystem invalidates the old disk location asynchronously, not when it evicts dirty page but when it commits the transaction. Each transaction file group maintains a set of Relocation Records called the *Relocation List*. The filesystem creates the Relocation Record and appends it to the Relocation List when it evicts the dirty page in the transaction.

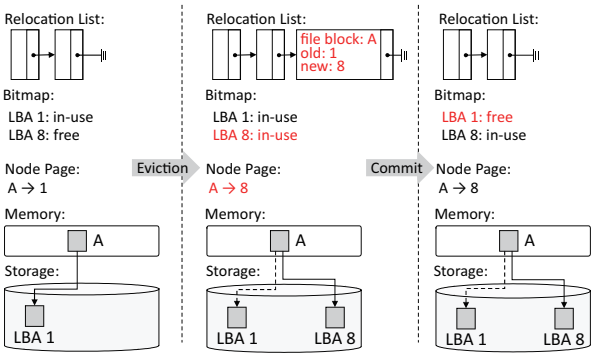


Figure 4: Delayed Invalidation: LBA 1 is invalidated not when the page is evicted but when the transaction commits.

Fig. 4 illustrates an example of stealing in exF2FS. The dirty page of the file block A is mapped to LBA 1 at the beginning. File block A is evicted to LBA 8. The node page in memory is updated to map file block A to LBA 8. The block bitmap for LBA 8 is set. The block bitmap for LBA 1 is *not* invalidated at the time of eviction due to Delayed Invalidation. The filesystem creates the Relocation Record and inserts the newly created record to the Relocation List. The newly created Relocation Record contains the file block ID (file block A), the old (LBA 1) and the new location (LBA 8) of the evicted block. Since LBA 1 is evicted to the disk, it is removed from the dirty page list of the associated Transaction File Group. When the transaction commits, LBA is invalidated and the updated node page is made durable.

4.3 Commit and Abort in Stealing

When the transaction commits, the filesystem makes the old location of the evicted page no longer reachable. Before it starts flushing the dirty pages, the filesystem scans the Relocation List in chronological order and invalidates the old disk locations of the evicted blocks (Delayed Invalidation). Once this finishes, it flushes the dirty data pages of the transaction. After the dirty pages become durable, the filesystem unpins the node page that has been updated in eviction and inserts it to the dirty node page list. Then, the filesystem flushes the

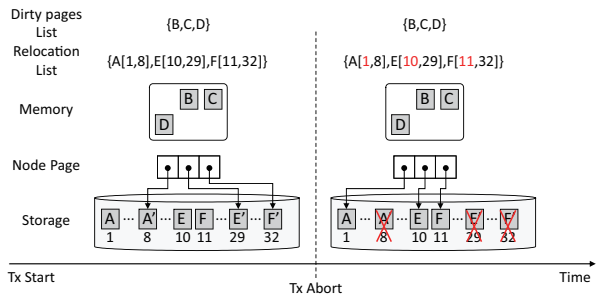


Figure 5: Stealing and Transaction Abort

dirty node pages. The transaction commits successfully if and only if the Master Commit Block becomes durable.

When the filesystem aborts the transaction, the filesystem scans the Relocation List in reverse chronological order. For each Relocation Record, the filesystem invalidates the new disk location and reverts the node page in memory to map the file block to the old disk location. After the node page is reverted, it is unpinned. Fig. 5 illustrates an example. At the time of abort, three pages have been evicted: A, E and F. The old location and the new location of page A corresponds to 1 and 8, respectively. In abort, the filesystem reverts the node pages for A, E and F to refer to page 1, 10 and 11, respectively, based upon the Relocation List. It also, invalidates the bitmap for the new disk locations, LBA 8, LBA 29 and LBA 32.

When the system crashes, Delayed Invalidation may leave the allocated but unreachable filesystem block. Delayed Invalidation temporarily leaves both the old and the new disk locations valid, from when the page is evicted until when the transaction commits. If the system crashes during this period, the filesystem can be recovered to the state where both old and new disk locations are valid but where only the old disk location is mapped to the file. If this happens, the new disk location needs to be collected through `fsck` [44] (offline) or through its online variant [17].

5 Transaction-aware Garbage Collection

We say that the garbage collection *conflicts* with the transaction if the garbage collection module selects a disk block which is associated with the dirty page of the uncommitted transaction as a victim for migration.

In this study, we develope a transaction-aware garbage collection technique called *Shadow Garbage Collection*. The Shadow Garbage Collection transparently migrates the victim block associated with the uncommitted transaction without any side effect to the transaction. F2FS performs the garbage collection in a transaction-aware manner but with substantial room for improvement; F2FS aborts *all* outstanding transactions when the garbage collection conflicts with any of the uncommitted transactions in the system [79].

5.1 Garbage Collection and the Transaction

The log-structured filesystem performs the garbage collection either in the foreground or in the background. Background garbage collection cannot conflict with the transactions since it runs only when the filesystem is idle. Here, the garbage collection implicitly denotes foreground garbage collection unless noted otherwise. The log-structured filesystem performs the garbage collection as follows. (i) First, the filesystem checkpoints the filesystem state (pre-GC checkpoint). (ii) The garbage collection module then selects the victim segment. (iii) Next, the garbage collection module migrates the valid blocks in the victim segment to the destination segment. This updates the associated file mapping to refer to the new disk location of the victim block. (iv) Finally, the filesystem checkpoints the updated state of the filesystem (post-GC checkpoint). The garbage collection module repeats step (ii) and step (iii) until it reclaims enough free segments. Pre-GC and post-GC checkpoints are essential in any log-structured filesystem to maintain its consistency against an unexpected filesystem failure.

In F2FS and a few other log-structured filesystems [38, 39, 77], the garbage collection module uses the page cache to migrate the victim disk block to the new location. In migrating the victim block, the garbage collection module first checks if the victim block exists in the page cache. There can be only one page cache entry for a single disk block. It is not possible to fetch the old data block into the page cache entry if the associated disk block already exists in the page cache. If the page cache entry for the victim block exists, the garbage collection module blindly writes the existing page cache entry to the destination without fetching the victim block from the disk. In this course, the garbage collection module may write the dirty page cache entry of the uncommitted transaction to the destination. After the garbage collection module migrates the victim disk block to the destination, the associated file mapping in the memory is updated to refer to the new disk location. Once the migration finishes, the garbage collection performs a checkpoint to make the state of the filesystem durable. As a result, the updated file mapping that refers to the new disk location of the victim block (dirty pages of the uncommitted transaction) becomes durable before the transaction commits. If the system crashes after the garbage collection finishes but before the transaction commits, the recovery module recovers the dirty pages of the uncommitted transaction. The atomicity of the transaction is then compromised.

5.2 Shadow Garbage Collection

In Shadow Garbage Collection, we reserve a set of page cache entries for garbage collection. We call this region *Shadow Page Cache*. When a victim block is associated with the uncommitted transaction, the garbage collection module uses Shadow Page Cache instead of generic page cache, to migrate

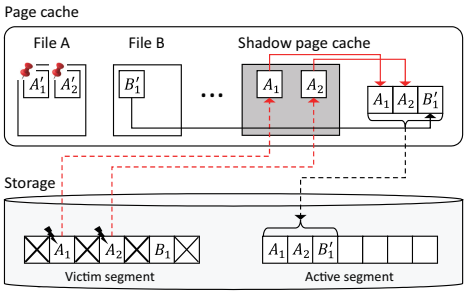


Figure 6: Shadow Garbage Collection: migrating A_1, A_2 and B_1. All are modified in memory to A'_1, A'_2 and B'_1, respectively. A_1 and A_2 are associated with an uncommitted transaction.

the victim block and the associated node block to the destination. Using the Shadow Page Cache in migrating the victim block to the destination, the filesystem prohibits the garbage collection from prematurely persisting the dirty pages of the uncommitted transaction. Fig. 6 illustrates an example of Shadow Garbage Collection. The disk block A_1, A_2 and B_1 are updated in the page cache to A'_1, A'_2 and B'_1, respectively. A_1 and A_2 are being modified by the transaction. The garbage collection module selects the disk block A_1, A_2 and B_1 as victims. In Shadow Garbage Collection, for migrating A_1 and A_2, the garbage collection module fetches A_1 and A_2 (the original version before the update) to Shadow Page Cache and flushes them to the destination. For migrating B_1, the garbage collection module uses the generic page cache since it is not associated with the transaction. Subsequently, it writes B'_1 (the updated version of B_1) to the destination segment.

The garbage collection can conflict with the uncommitted transaction in two ways; (i) the victim block can be associated with the evicted page by Stealing (type E, Evicted) and (ii) the victim block can be associated with the cached page (type C, Cached). When the victim block is associated with the evicted page, it can correspond to either the original file block before the update (type EO, Evicted and Old) or the updated file block (type EN, Evicted and New). When the victim block is associated with the cached page, the victim block corresponds to the original file block before the update (type CO, Cached and Old). Note that the victim block of type CN (Cached, New) cannot exist.

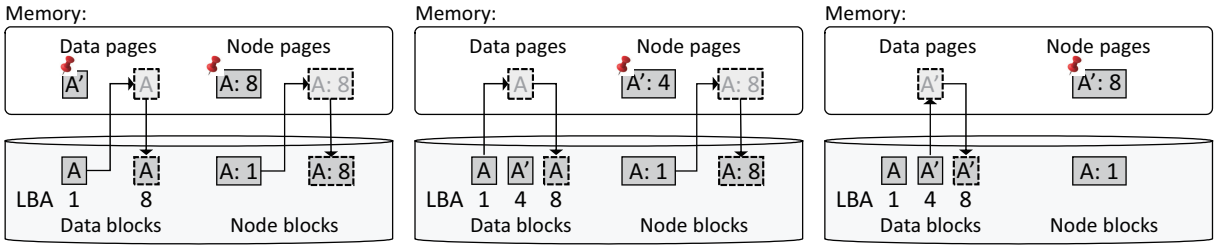
For each type of victim block, the Shadow Garbage Collection elaborately applies a different mechanism in migrating the victim block and the associated node block.

Type CO. When the victim block corresponds to old (O) version of the cached block (C) of the uncommitted transaction, we use the Shadow Page Cache in migrating the victim block and in storing the updated node block to the new disk location. In updating and storing the associated node block, the Shadow Garbage Collection updates the node block read from the disk, not the node block which has already been in the page cache. The node block in the page cache may have been updated since it is read from the disk and may contain transient file mapping that should not be made durable. After the garbage collection module finishes migrating both the victim block and the updated node block, it updates the node page in the page cache with the new file mapping. When the transaction aborts or the system crashes, the victim block at the migrated location can be recovered using the updated node block stored on the disk. An example of this can be seen in Fig. 7(a). File block A has been in LBA 1 and is updated in memory to A'. The disk block LBA 1 is selected as the victim. It is migrated to LBA 8 with shadow page caching. The associated node block is read into the Shadow Page Cache and is updated to map to LBA 8. Then, the updated node page is flushed to the disk. After both the victim block and the node block are flushed, the in-memory node block of file block A is updated to map to LBA 8.

Type EO. When the victim block corresponds to the old (O) version of the evicted page (E), we use the Shadow Page Cache in migrating the victim block and in storing the updated node block to the new disk location. Recall that the evicted page does not have the associated page cache entry (data page) and the associated node page is pinned in memory until the transaction commits due to Node Page Pinning. In migrating the victim block of type EO, the filesystem migrates the victim block using the Shadow Page Cache. For the node block update, we use the on-disk version of the node block as in the case of migrating the type EO victim block. After the Shadow Garbage Collection module finishes migrating the node block to the destination, it updates the node block pinned in memory with the updated file mapping. An example of this is illustrated in Fig. 7(b). The dirty page of the uncommitted transaction was evicted to LBA 4. File block A is migrated from LBA 1 to LBA 8. Shadow Page Cache is used to migrate the victim block and the associated node block. The node block that maps A is updated from "A:1" to "A:8" and flushed to the disk. The node page that maps the location of the dirty file block (A') of the evicted page remains unchanged in the page cache (A':4) and is pinned in memory.

Type EN. When the victim block corresponds to the new (N) version of the evicted page (E), we use the generic page cache in migrating the victim block to the new disk location. We can use the generic page cache, not Shadow Page Cache, in migrating the victim block since the victim block holds the most recent copy of the file block. The garbage collection module updates the node page in the page cache with the new file mapping after it migrates the victim block to the new location. An example is shown in Fig. 7(c). The updated file block of the evicted page A' is migrated from LBA 4 to LBA 8. Here, generic page cache (not Shadow Page Cache) is used to migrate the victim block. After the migration completes, the garbage collection module updates the associated node page in the page cache from "A':4" to "A':8".

When the garbage collection migrates the disk block associated with the evicted page, the garbage collection updates



(a) CO (LBA: 1→8) (b) EO (LBA: 1→8) (c) EN (LBA: 4→8)

Figure 7: Shadow Garbage Collection, A: original file block, A': updated file block. A rectangle with light grey background denotes the Shadow Page Cache.

the Relocation Record after the migration finishes. When the victim block is associated with the old disk location and the new disk location of the evicted page, it updates the old disk location field and new disk location field of the Relocation Record, respectively.

In implementing the Shadow Garbage Collection, we use an existing `META_MAPPING` object in Linux as the Shadow Page Cache. `META_MAPPING` is a special purpose `address_space` object, which is dedicated to cache the filesystem metadata [18]. Exploiting the existing `META_MAPPING` object as Shadow Page Cache, Shadow Garbage Collection does not require any new data structure for Shadow Page Cache in the kernel. Garbage collection of exF2FS (and also F2FS) reclaims the free blocks in a segment-granularity. Memory overhead for Shadow Garbage Collection corresponds to the size of a single segment, 2MB.

6 Applications with exF2FS

In this section, we explain how applications can exploit the transactional support from the underlying filesystem.

SQLite : Fig. 8(a) illustrates the implementation of the multi-file transaction in stock SQLite and in modified SQLite ported for exF2FS. In the stock SQLite's multidatabase transaction, the SQLite separately logs the updates to individual journal files and logs the global state of the transaction at the master journal file. In exF2FS, SQLite can implement its multi-file transaction with a single filesystem transaction eliminating the need for separately logging the individual database updates to the journal files.

Compaction in RocksDB: Fig. 8(b) illustrates the compaction in stock RocksDB and the compaction in RocksDB ported for exF2FS. In exF2FS, RocksDB can replace the multiple flushes of a compaction with a single filesystem transaction. In exF2FS, RocksDB can selectively exclude the `LOG` file from compaction transaction. It saves RocksDB from flushing the updates of the `LOG` file in making the result of compaction durable. The `LOG` file contains debugging information which is not an essential part of the compaction [21].

```
// without transaction support
write(/d/mj);
fdatasync(/d/mj);
fdatasync(/d);
while (/d/db[@]) {
  write(/d/log[@]);
  fdatasync(/d/log[@]);
  write(/dir/log[@]);
  fdatasync(/dir/log[@]);
  write(/dir/db[@]);
  fdatasync(/dir/db[@]);
}
unlink(/d/mj);
fdatasync(/d);
```

```
// with transaction support
while (/d/db[@])
  add_tx_file_group(tfg,
                    /d/db[@]);
start_tx_file_group(tfg);

while (/d/db[@]) {
  write(/d/db[@]);
}

commit_tx_file_group(tfg);
```

(a) Multi-database transaction in SQLite

```
// without transaction support
write(/d/LOG);
while (/d/sst[@]) {
  open(/d/newsst[@]);
  write(/d/newsst[@]);
  fsync(/d/newsst[@]);
  close(/d/sst [@]);
  write(/d/LOG);
}
fsync(/d);
write(/d/MANIFEST);
fsync(/d/MANIFEST);
write(/d/LOG);
```

```
// with transaction support
write(/d/LOG);
add_tx_file_group(tfg,/d);
start_tx_file_group(tfg);

while (/d/sst[@]) {
  create(/d/newsst [@]);
  write(/d/newsst[@]);
  write(/d/LOG);
}

commit_tx_file_group(tfg);
write(/d/LOG);
```

(b) Compaction in RocksDB

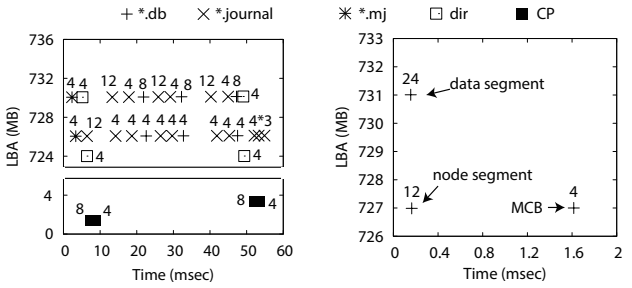
Figure 8: SQLite and RocksDB: with transaction support from the filesystem

7 Evaluation

Here, we evaluate the transaction feature of exF2FS. We implement exF2FS in Linux kernel 4.18. exF2FS is compared to three other filesystems: EXT4, F2FS, and TxFS [27]. TxFS [27] is the most recently published transactional filesystem based upon EXT4. TxFS was developed in Linux 3.18.22 and is not stable. For fair comparison, we re-implement only the atomicity and durability feature of TxFS on Linux 4.18.

Two storage devices were used in our experiment: Samsung 850 PRO [51] and Intel Optane 900P [29]. The 850 PRO and the Optane renders 1-2 msec and sub 10 μsec flush latency, respectively. We used a machine with an Intel CPU i7-9700K (3.60GHz, 4 core) and 64GB memory.

7.1 SQLite



(a) SQLite Transaction in F2FS (b) SQLite Transaction in exF2FS

Figure 9: IO trace: A multi-file transaction with three `insert()`'s in SQLite: F2FS vs. exF2FS. Record size: 100 Bytes. The number in each mark represents the number of KB written, Device: Samsung 850 PRO

Block level IO: We examine the raw IO behavior of the multi-file transaction in vanilla SQLite over F2FS and in SQLite with a multi-file transaction of exF2FS. Fig. 9(a) is the IO trace in vanilla SQLite over F2FS. A multi-file transaction consists of three `insert()`'s to three different database files. In vanilla SQLite, fifteen `fdatasync()` calls, two filesystem level checkpoints and a total of 32 write requests to the storage occur, taking 55 msec to complete a transaction. Fig. 9(b) illustrates the IO trace of SQLite's multi-file transaction when built with the multi-file transaction of exF2FS. There are three writes: one for the data blocks, one for the node blocks, and one for the master commit block, and takes 1.6 msec for a transaction. exF2FS resolves the excessive flush call problem.

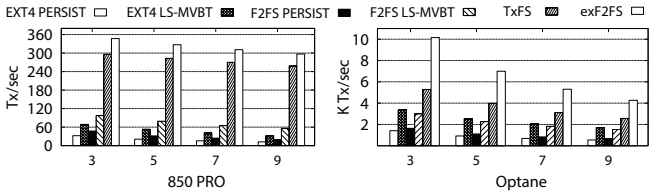


Figure 10: Transaction Throughput (Mobibench [3]-SQLite, `insert` operation), # of databases in a transaction = 3, 5, 7, 9

Throughput : We test the SQLite performance under the different SQLite journal modes and under different filesystems. For SQLite journal modes, we use PERSIST mode, and LS-MVBT [37]. PERSIST mode is the most popular journaling mode in SQLite. LS-MVBT [37] is the fastest SQLite journaling scheme known to the public. For the filesystem, F2FS, EXT4, exF2FS and TxFS are used, and Mobibench is used to generate the workload [3]. We port SQLite to use the transaction of exF2FS and TxFS. The results are shown in Fig. 10. In `insert` performance, exF2FS improves the throughput by as much as 24× against stock SQLite with PERSIST mode in F2FS (nine database files in a transaction, 850 PRO).

FS	Tput (KIOPS)	# of `fsync()`	# of cpt'n	compaction latency (sec)		
				Mean	99.9%	99.99%
F2FS	21.8	6135	892	18	153	373
exF2FS	40.8	622	622	7	50	51
EXT4	32.9	5873	862	9	48	88

Table 2: Throughput, total number of `fsync()`'s, total number of compactions, and compaction latency. cpt'n: Compaction

Let us compare the transaction performance of exF2FS against TxFS. As the storage gets faster, the performance benefit of exF2FS becomes more substantial than that of TxFS. In 850 PRO, SQLite exhibits 10% better performance in exF2FS than TxFS. In Optane, SQLite exhibits 100% better performance in exF2FS. The difference between exF2FS and TxFS are further elaborated in Section 7.4.

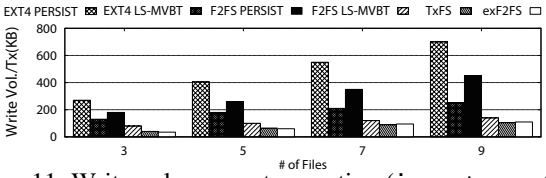


Figure 11: Write volume per transaction (`insert` operation), Number of database files in a transaction = 3,5,7,9

Write Volume: In all six transaction support methods, exF2FS creates the smallest amount of write (Fig. 11). Compared to F2FS with SQLite with PERSIST mode journaling, exF2FS with SQLite on the multi-file transaction generates 1/6 of the writes.

7.2 RocksDB Compaction

We found that using the transaction of exF2FS in RocksDB compaction produces two significant benefits: the performance improvement and the ability to handle the large size transaction. The YCSB benchmark (workload-A) is run for RocksDB. In this workload, a single compaction of RocksDB can create up to 13.3 GB of dirty pages with 198 SSTable files. Filesystems that pin the updated pages of the transaction in memory cannot perform RocksDB compaction as a transaction [27,39,65]. Here, the performance of transaction based RocksDB over exF2FS is compared with vanilla RocksDB over stock F2FS. The size of the memtable and the maximum size of the SSTable are both 64 MB. Key and value size are 23 Bytes and 1KB, respectively. Initially, RocksDB is populated with 50 M operations (55 GB). Then, YCSB-A is run with 50 M operations.

The performance results are summarized in Table 2. exF2FS improves YCSB performance by 87% against F2FS: 40.8 KIOPS vs. 21.8 KIOPS. On average, the compaction latency in exF2FS is 40% of the compaction latency in F2FS: 7 sec vs. 18 sec. The root cause for the performance and the latency difference is the number of `fsync()` calls. In F2FS, a single compaction creates seven `fsync()`'s on average, while

in exF2FS, a single compaction is executed with a single transaction which is equivalent to one `fsync()`.

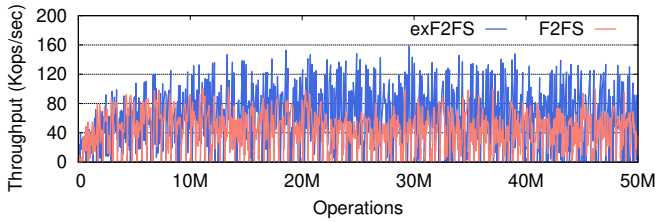
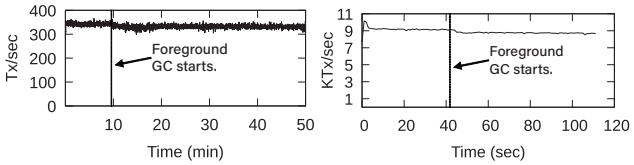


Figure 12: RocksDB Throughput in exF2FS vs. F2FS, YCSB workload A, a total of 50 M operations (read:write = 1:1), window size: 1 sec

We examine the throughput of RocksDB in exF2FS and F2FS (Fig. 12). The throughput is collected at one second intervals. Fig. 12 clearly shows that in RocksDB, exF2FS renders superior throughput behavior to F2FS. In this workload, 12% of the compactions are executed with stealing. On average, each compaction creates 100K dirty pages (400 MB) and 6K pages (24 MB) are evicted.

7.3 Garbage Collection



(a) 850 PRO (b) Optane

Figure 13: Throughput of multi-file transaction under foreground garbage collection in action (Mobibench [3]-SQLite, three `inserts` per transaction, record size = 100Byte)

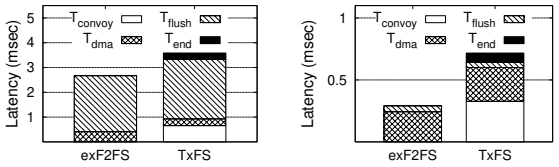
In F2FS, the transaction aborts when the garbage collection module selects one of its blocks as a victim block. In exF2FS, the transaction does not abort. However, the transaction is suspended until the garbage collection finishes when it encounters foreground garbage collection. Here, we examine how the garbage collection of exF2FS interferes with the throughput and latency of the foreground application. We also examine the throughput of the multi-file transaction (three inserts). The results are presented in Fig. 13. First, we mark the time when the foreground garbage collection is triggered. From then, the foreground garbage collection is triggered once every hundred transactions on average. With foreground garbage collection, the performance decreases by about 5%. Each foreground garbage collection reclaims a single free segment. With the foreground garbage collection, the tail latency (@99.9%) of the multi-file transaction has increased from 300 μsec to 470 μsec in Optane.

7.4 exF2FS vs. TxFS

We examine the detailed behavior of the transaction in exF2FS and TxFS. We use Mobibench [3] and generate the multi-file transaction in SQLite (`insert()`'s to three databases per transaction, record size: 100 Byte). While far from being complete, the analysis here provides a useful clue on how the log-structured filesystem and the journaling filesystem can fundamentally differ in supporting the transaction.

7.4.1 Convoy and Context Switch Overhead

In this section, we examine the latency of committing a transaction in exF2FS and TxFS. In 850 PRO and Optane, the commit latencies in exF2FS are 80% and 40% of those in TxFS, respectively. The latency difference between exF2FS and TxFS becomes more significant as the storage speed increases.

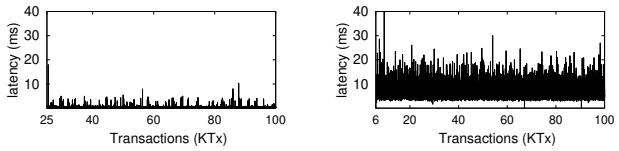


(a) Samsung 850 PRO (b) Intel Optane

Figure 14: Latency of multi-file transaction: exF2FS vs. TxFS (T_{convoy}: prepare for the commit, T_{dma}: time to transfer the blocks in the transaction, T_{flush}: time to make the blocks durable, T_{end}: wrap up the commit)

The latency to commit a transaction is partitioned into four components for detailed analysis: (i) prepare for commit (T_{convoy}), (ii) DMA transfer (T_{DMA}), (iii) flush (T_{flush}) and (iv) wrap up (T_{end}). The details of these are illustrated in Fig. 14. In exF2FS, the time for preparing a commit (T_{convoy}) includes preparing the Master Commit Block, constructing the IO commands and dispatching them to the storage. In TxFS, the time for preparing a commit (T_{convoy}) includes not only the time for preparing the journal descriptor block, constructing the IO commands and dispatching them to the storage, but also the time for writing *the unrelated data blocks to the disk*, the convoy [7]. T_{convoy} overhead is substantial in TxFS accounting for as much as 50% of the total commit latency (Optane). On the other hand, it is almost non-existent in exF2FS. This is due to the compound journaling of EXT4 [71]. EXT4 merges the updated metadata from multiple file operations into a single running transaction to increase the throughput of the filesystem journaling. Due to compound journaling, EXT4 can flush a large amount of unrelated dirty pages in an `fsync()` [30].

When the transaction is executed with the other metadata intensive applications, the convoy overhead of compound journaling becomes far more severe. Here, we examine the



(a) exF2FS, μ=169 usec (b) TxFS, μ=5.7 msec

Figure 15: SQLite: Latency of transaction with three inserts in SQLite, ten `varmail` threads [43] in the background, Optane, μ: average latency

Filesystems	Write Size	4KB	8KB	16KB	32KB
TxFS	Write	12GB	6GB	2.5GB	2.5GB
exF2FS	Volume	3GB	2GB	1.5GB	1.1GB

Table 3: Write Amplification of Transactional Write: Total Write Volume in writing 1 GB to a file (allocating write)

transaction latency of exF2FS and TxFS with a metadata intensive application, `varmail` benchmark [43], running in the background. Fig. 15 shows the result. The average transaction latency of TxFS is 34× that of exF2FS: 5.7 msec vs. 169μsec.

In exF2FS (or in F2FS), the filesystem commits the transaction in its own context. In TxFS (or in EXT4), the filesystem delegates the journal commit to the JBD thread, and the overhead of registering the committed blocks for the checkpoint and the context switch overhead, T_{end}, is non-negligible. T_{end} accounts for as much as 10% of the commit latency in TxFS while it does not exist in exF2FS. Due to the overhead of convoy and the context switch inherent in EXT4, exF2FS renders better transaction performance than TxFS.

7.4.2 Double Write and journal metadata overhead

We examine the write amplification of exF2FS and TxFS. The transactional write size varies from 4 KB to 32 KB and the total write volume is examined. Table 3 summarizes the result. In writing 1 GB with 4 KB atomic write, exF2FS writes 3 GB to the storage while TxFS creates 12 GB. In exF2FS, a 4 KB transactional write accompanies a 4 KB write for the node block and a 4 KB write for the Master Commit Block. In TxFS, a 4 KB transactional write (allocating write) journals four log blocks (superblock, inode table, data block, block bitmap), all of which are later checkpointed to their original locations. A double write overhead compound by the overhead of page granularity journaling renders a 12× write amplification in a 4 KB allocating write of TxFS. In exF2FS, the write amplification is 3× under the same workload. When the transaction size is 32 KB, exF2FS and TxFS render 1.1× and 2.5× write amplification, respectively. In this experiment, exF2FS does not perform any garbage collection. If it were included, it may render a larger write amplification. Unless the garbage collection amplifies the write volume by more than 2×, exF2FS renders less write volume than TxFS.

8 Related Work

Transaction support can be implemented in different layers of the software stack. TxOS [57] and QuickSilver [61] implement transaction support as a native kernel service. A transactional filesystem can readily be built using the interface offered by TxOS [26]. There are several kernel level filesystems that support transaction, such as AdvFS [72], TxFS [27], Valor [66], Transactional NTFS from Microsoft (TxF) [46], Failure-atomic msync() [55], and BTRFS [14]. OdeFS [24] and Inversion [54] are built as a user level filesystem and they rely on existing DBMS to realize an ACID property of the filesystem operation. CFS [47]'s crash consistency support is built on top of the transactional block device, X-FTL [32]. BVSSD [28], MARS [12], TxFlash [58], and Isotope [65] offer block device level transaction support. Libnvmmio [11] uses a user level log for its transaction support.

The degree of ACID support comes at the cost of the implementation complexity. Some works support full ACID (Atomicity, Consistency, Isolation and Durability) property [14, 27, 46, 66]. Some filesystems drop isolation support and support only ACD [47, 55, 72]. F2FS drops the durability and supports only AC in its atomic write [34]. By leaving the isolation support to the application, exF2FS limits the code changes to the local filesystem. TxOS requires a few 100K LOC [57]. Limiting the transaction support to the filesystem, TxFS reduces the required code changes to one tenth, 5K LOC. By exploiting the atomic write feature of F2FS and excluding the isolation support, exF2FS achieves its transaction support with 1.5K LOC.

9 Conclusion

In this work, we successfully address the three major issues of transaction support in log-structured filesystems: multi-file support, stealing and garbage collection. With the transactional log-structured filesystem proposed in this work, we can greatly simplify the application programming and can substantially improve the application performance in many popular applications including SQLite, RocksDB, and application installation.

Acknowledgements We are deeply indebted to our shepherd, Peter Macko, for helping shape the final version of this paper. We are also grateful to the anonymous reviewers for their comments that have greatly improved this paper. We also thank Seungyong Cheon, Jinsoo Yoo, Sundoo Kim, and Wonjong Lee for discussions and comments on earlier iterations of this work. This work was supported by IITP, Korea (grant No. 2018-0-00549 and No. 2018-0-00503), NRF, Korea (grant No. NRF-2020R1A2C3008525), and Samsung Electronics (IO201209-07867-01).

References

[1] http://vimdoc.sourceforge.net/htmldoc/recover.html.

[2] The manual of maildir. https://web.archive.org/web/19971012032244/http://www.qmail.org/qmail-manual-html/man5/maildir.html.

[3] Mobibench. https://github.com/ESOS-Lab/Mobibench.

[4] Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *Proc. of 16th International Conference on Data Engineering (ICDE)*, 2000.

[5] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2019.

[6] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, DL Moal, G Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proc. of the 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[7] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. The convoy phenomenon. *ACM SIGOPS Operating Systems Review*, 13(2):20–25, 1979.

[8] Jakob Blomer, Carlos Aguado-Sánchez, Predrag Buncic, and Artem Harutyunyan. Distributing LHC application software and conditions databases using the CernVM file system. *Journal of Physics: Conference Series*, 331(4), 2011.

[9] Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. Verifying concurrent software using movers in CSPEC. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[10] Deka Ganesh Chandra. BASE analysis of NoSQL database. *Future Generation Computer Systems*, 52:13–21, 2015.

[11] Jungsik Choi, Jaewan Hong, Youngjin Kwon, and Hwansoo Han. Libnvmmio: Reconstructing Software IO Path with Failure-Atomic Memory-Mapped Interface. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2020.

[12] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From ARIES to MARS: Transaction support for next-generation, solid-state drives. In *Proc. of 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[13] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of the 1st ACM Symposium On Cloud Computing (SOCC)*, 2010.

[14] Jonathan Corbet. Supporting transactions in btrfs, November 2009. https://lwn.net/Articles/361457/.

[15] Anton Kuijsten Cristiano Giuffrida, Călin Iorgulescu and Andrew S. Tanenbaum. Back to the Future: Fault-tolerant Live Update with Time-traveling State Transfer. In *Proc. of 27th Large Installation System Administration Conference (LISA)*, 2013.

[16] Pia Malkani Daniel Ellard, Jonathan Ledlie and Margo Seltzer. Passive NFS tracing of email and research workloads. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[17] David Domingo and Sudarsun Kannan. pFSCK: Accelerating File System Checking and Repair for Modern Storage. In *Proc. of 19th USENIX Conference on File and Storage Technologies (FAST '21)*.

[18] elixir.bootlin.com. Definition of `META_MAPPING`. https://elixir.bootlin.com/linux/v4.18/source/fs/f2fs/f2fs.h#L1476.

[19] elixir.bootlin.com. ipc/util.c. https://elixir.bootlin.com/linux/latest/source/ipc/util.c#L171.

[20] Nick Elprin and Bryan Parno. An Analysis of Database-Driven Mail Servers. In *Proc. of 17th Large Installation System Administration Conference (LISA)*, 2003.

[21] Facebook. Rocksdb Compaction. https://github.com/facebook/rocksdb/wiki/Compaction.

[22] Facebook. RocksDB homepage. http://rocksdb.org/.

[23] Facebook. Rocksdb MANIFEST. https://github.com/facebook/rocksdb/wiki/MANIFEST.

[24] Narain H Gehani, Hosagrahar V Jagadish, and William D Roome. Odefs: A file system interface to an object-oriented database. In *Proc. of 20th International Conference on Very Large Data Bases (VLDB)*, 1994.

[25] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of apple desktop applications. *Transactions on Computer Systems (TOCS)*, 30(3):10:1–10:39, August 2012.

[26] Yige Hu, Youngjin Kwon, Vijay Chidambaram, and Emmett Witchel. From Crash Consistency to Transactions. In *Proc. of ACM HotOS*, 2017.

[27] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. TxFS : Leveraging File-System Crash Consistency to Provide ACID Transactions. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2018.

[28] Ping Huang, Ke Zhou, Hua Wang, and Chun Hua Li. BVSSD: Build built-in versioning flash-based solid state drives. In *Proc. of the 5th Annual International Systems and Storage Conference (SYSTOR)*, 2012.

[29] intel.com. Intel optane ssd 900p series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-900p-series.html.

[30] Daeho Jeong, Youngjae Lee, and Jinsoo Kim. Boosting Quasi-asynchronous I/O for Better Responsiveness in Mobile Devices. In *Proc. of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[31] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. I/O Stack Optimization for Smartphones. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2013.

[32] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-FTL: transactional FTL for SQLite databases. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2013.

[33] Dongui Kim, Chanyeol Park, Sang-Won Lee, and Beomseok Nam. BoLT: Barrier-optimized LSM-Tree. In *Proc. of the 21st International Middleware Conference (MIDDLEWARE)*, 2020.

[34] Jaegeuk Kim. F2FS: support `atomic_write` feature for database. https://lkml.org/lkml/2014/9/26/19.

[35] Jaeguek Kim. f2fs: limit # of inmemory pages. https://patchwork.kernel.org/project/linux-fsdevel/patch/20171019021516.65627-1-jaegeuk@kernel.org/#21076797.

[36] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proc. of 21th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[37] Wook-Hee Kim, Beomseo Nam, Dongil Park, and Youjip Won. Resolving Journaling of Journal Anomaly in Android I/O: Multi-Version B-tree with Lazy Split. In *Proc. of 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[38] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[39] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *Proc. of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[40] Wongun Lee, Keonwoo Lee, Hankeun Son, Wookhee Kim, Beomseok Nam, and Youjip Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2015.

[41] Henry M Levy and Peter H Lipman. Virtual memory management in the VAX/VMS operating system. *Computer*, 15(03):35–41, 1982.

[42] Paul McDougall. Microsoft pulls buggy windows vista sp1 files. *Information Week*, 2008. https://www.informationweek.com/software/microsoft-pulls-buggy-windows-vista-sp1-files.

[43] Richard McDougall and Jim Mauro. Filebench, 2005.

[44] Marshall Kirk McKusick, Willian N Joy, Samuel J Leffler, and Robert S Fabry. Fsck-The UNIX File System Check Program. *Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version*, 1986.

[45] Rémy Evard Michail Gomberg and Craig Stacey. A Comparison of Large-Scale Software Installation Methods on NT and UNIX. In *Proc. of the Large Installation System Administration of Windows NT Conference*, 1998.

[46] Frederic Miller and Agnes Vandome. *NTFS*. Alpha Press, 2009.

[47] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proc. of USENIX Annual Technical Conference (ATC)*, 2015.

[48] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system, 2002.

[49] mysql.com. Transaction Isolation Levels. https://dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html.

[50] Rebecca Nelson, Atul Shukla, and Cory Smith. Web Browser Forensics in Google Chrome, Mozilla Firefox, and the Tor Browser Bundle. In *Digital Forensic Education*, pages 219–241. Springer, 2020.

[51] news.samsung.com. Samsung Electronics leads consumers into the new era of multi-terabyte SSDs with Launch of 2-TB 850 PRO and 850 EVO. https://news.samsung.com/us/samsung-electronics-leads-consumers-into-the-new-era-of-multi-terabyte-ssds-with-launch-of-2-tb-850-pro-and-850-evo/.

[52] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. SHARE interface in flash storage for relational and NoSQL databases. In *Proc. of 2016 International Conference on Management of Data (SIGMOD)*, 2016.

[53] Sehyeon Oh, Wook-Hee Kim, Jihye Seo, Hyeonho Song, Sam H Noh, and Beomseok Nam. Doubleheader Logging: Eliminating Journal Write Overhead for Mobile DBMS. In *Proc. of 2020 IEEE 36th International Conference on Data Engineering (ICDE)*, 2020.

[54] Michael A. Olson. The Design and Implementation of the Inversion File System. In *Proc. of USENIX Winter*, 1993.

[55] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync () a simple and efficient mechanism for preserving the integrity of durable data. In *Proc. of 8th ACM European Conference on Computer Systems (EUROSYS)*, 2013.

[56] Android Police. The Pixel 3 uses Samsung's super-fast F2FS file system, October 2018. https://www.androidpolice.com/2018/10/10/pixel-3-uses-samsungs-super-fast-f2fs-file-system/.

[57] Donald E Porter, Owen S Hofmann, Christopher J Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proc. of 22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[58] Vijayan Prabhakaran, Thomas L Rodeheffer, and Lidong Zhou. Transactional Flash. In *Proc. of 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[59] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems, Chapter 16.7.1 Stealing Frames and Forcing Pages*. McGraw-Hill, 2000.

[60] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[61] Frank Schmuck and Jim Wylie. Experience with transactions in QuickSilver. In *Proc. of 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[62] Margo I Seltzer. Transaction support in a log-structured file system. In *Proc. of IEEE 9th International Conference on Data Engineering (ICDE)*, 1993.

[63] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An Implementation of a Log-Structured File System for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[64] Kai Shen, Stan Park, and Meng Zhu. Journaling of Journal is (Almost) Free. In *Proc. of 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[65] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Isotope: Transactional Isolation for Block Storage. In *Proc. of 14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[66] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proc. of 7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[67] SQLite.org. Isolation in sqlite. https://www.sqlite.org/isolation.html.

[68] SQLite.org. Pragma read_uncommitted. https://www.sqlite.org/pragma.html#pragma_read_uncommitted.

[69] SQLite.org. Pragma statements, 2012. http://www.sqlite.org/pragma.html#pragma_journal_mode.

[70] Jan Stender, Björn Kolbeck, Mikael Högqvist, and Felix Hupfeld. BabuDB: Fast and Efficient File System Metadata Storage. In *Proc. of International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2010.

[71] Stephen C Tweedie et al. Journaling the Linux ext2fs filesystem. In *Proc. of Annual Linux Expo*, 1998.

[72] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya S Mannarswamy, Terence Kelly, and Charles B Morrey III. Failure-Atomic Updates of Application Data in a Linux File System. In *Proc. of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[73] Ingo Weber, Hiroshi Wada, Alan Fekete, Anna Liu, and Len Bass. Supporting Undoability in Systems Operations. In *27th Large Installation System Administration Conference (LISA)*, 2013.

[74] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[75] Matthew J Weinstein, Thomas W Page Jr, Brian K Livezey, and Gerald J Popek. Transactions and synchronization in a distributed operating system. *ACM SIGOPS Operating Systems Review*, 19(5):115–126, 1985.

[76] Youjip Won, Sundoo Kim, Juseong Yun, Dam Quang Tuan, and Jiwon Seo. Dash: Database shadowing for mobile dbms. *In Proc. of 45th International Conference on Very Large Data Bases (VLDB)*, 12(7):793–806, 2019.

[77] David Woodhouse. JFFS: The journalling flash file system. In *Proc. of Ottawa Linux Symposium*, 2001.

[78] Charles P Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID semantics to the file system. *ACM Transactions on Storage (TOS)*, 3(2), 2007.

[79] Chao Yu. f2fs: avoid stucking GC due to atomic write. https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1667312.html.