TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions

Yige Hu[1], Zhiting Zhu[1], Ian Neal[1], Youngjin Kwon[1], Tianyu Cheng[1]
Vijay Chidambaram[1,2], Emmett Witchel[1]
[1]*University of Texas at Austin* [2]*VMware Research*

Abstract

We introduce TxFS, a novel transactional file system that builds upon a file system's atomic-update mechanism such as journaling. Though prior work has explored a number of transactional file systems, TxFS has a unique set of properties: a simple API, portability across different hardware, high performance, low complexity (by building on the journal), and full ACID transactions. We port SQLite and Git to use TxFS, and experimentally show that TxFS provides strong crash consistency while providing equal or better performance.

1 Introduction

Modern applications store persistent state across multiple files [21]. Some applications split their state among embedded databases, key-value stores, and file systems [27]. Such applications need to ensure that their data is not corrupted or lost in the event of a crash. Unfortunately, existing techniques for crash consistency, such as logging or using atomic rename, result in complex protocols and subtle bugs [21].

Transactions present an intuitive way to atomically update persistent state [6]. Unfortunately, building transactional systems is complex and error-prone. In this paper, we introduce a novel approach to building a transactional file system. We take advantage of a mature, well-tested piece of functionality in the operating system: the file-system journal, which is used to ensure atomic updates to the internal state of the file system. We use the atomicity and durability provided by journal transactions and leverage it to build ACID transactions available to user-space transactions. Our approach greatly reduces the development effort and complexity for building a transactional file system.

We introduce TxFS, a transactional file system that builds on the ext4 file system's journaling mechanism. We designed TxFS to be practical to implement and to use. TxFS has a unique set of properties: it has a small implementation (5,200 LOC) by building on the journal (for example, TxFS has 25% the LOC of the TxOS transactional operating system [22]); it provides high performance unlike various solutions which built a transactional file system over a user-space database [5, 16, 18, 31]; it has a simple API (just wrap code in `fs_tx_begin()` and `fs_tx_commit()`) compared to solutions like Valor [28] or TxF [24] which require multiple system calls per transaction and can require the developer to understand implementation details like logging; it provides all ACID guarantees unlike solutions such as CFS [15] and AdvFS [30] which only offer some of the guarantees; it provides transactions at the file level instead of at the block level unlike Isotope [26], making several optimizations easier to implement; finally, TxFS does not depend upon specific properties of the underlying storage unlike solutions such as MARS [3] and TxFlash [23].

The advantage to building TxFS on the file-system journal is that TxFS transactions obtain atomicity, consistency, and durability by placing each one entirely within a single file-system journal transaction (which is applied atomically to the file system). Using well-tested journal code to obtain ACD reduces the implementation complexity of TxFS, while limiting the maximum size of transactions to the size of the journal.

The main challenge of building TxFS is providing isolation. Isolation for TxFS transactions requires that in-progress TxFS transactions are not visible to other processes until the transaction commits. At a high level, TxFS achieves isolation by making private copies of all data that is read or written, and updating global data during commit. However, the naive implementation of this approach would be extremely inefficient: global data structures such as bitmaps would cause conflicts for every transaction, causing high abort rates and excessive transaction retries. TxFS makes concurrent transactions efficient by collecting logical updates to global structures, and applying the updates at commit time. TxFS includes a number of other optimizations such as eager conflict detection that are tailored to the current implementation of file-system data structures in ext4.

We find that the transactional framework allows us to easily implement a number of file-system optimizations. For example, one of the core techniques from our earlier work, separating ordering from durability [2], is easily accomplished in TxFS. Similarly, we find TxFS transactions allow us to identify and eliminate redundant application IO where temporary files or logs are used to atomically update a file: when the sequence is simply enclosed in a transaction (and without any other changes), TxFS atomically updates the file (maintaining functionality) while eliminating the IO to logs or temporary files (provided the temporary files and logs are deleted inside the transaction). As a result, TxFS improves performance while simultaneously providing better crash-consistency semantics: a crash does not leave ugly temporary files or logs that need to be cleaned up.

To demonstrate the power and ease of use of TxFS transactions, we modify SQLite and Git to incorporate TxFS transactions. We show that when using TxFS transactions, SQLite performance on the TPC-C benchmark improves by 1.6× and a micro-benchmark which mimics Android Mail obtains 2.3× better throughput. Using TxFS transactions greatly simplifies Git's code while providing crash consistency without performance overhead. Thus, TxFS transactions increase performance, reduce complexity, and provide crash consistency.

Our paper makes the following contributions.

- We present the design and implementation of TxFS, a transactional file system for modern applications built by leveraging the file-system journal (§3). We have made TxFS publicly available at `https://github.com/ut-osa/txfs`.
- We show that existing file systems optimizations, such as separating ordering from durability, can be effectively implemented for TxFS transactions (§4).
- We show that real applications can be easily modified to use TxFS, resulting in better crash semantics and significantly increased performance (§5).

2 Background and motivation

We first describe the protocols used by current applications to update state in a crash-consistent manner. We then present a study of different applications and the challenges they face in maintaining crash consistency across persistent state stored in different abstractions. We describe the file-system optimizations enabled by transactions and finally summarize why we think transactional file systems should be revisited.

2.1 How applications update state today

Given that applications today do not have access to transactions, how do they consistently update state to multiple storage locations? Even if the system crashes or power fails, applications need to maintain invariants across state in different files (*e.g.,* an image file should match the thumbnail in a picture Gallery). Applications achieve this by using ad hoc protocols that are complex and error-prone [21].

In this section, we show how difficult it is to implement seemingly simple protocols for consistent updates to storage. There are many details that are often overlooked, like the persistence of directory contents. These protocols are complex, error prone, and inefficient. With current storage technologies, these protocols must sacrifice performance to be correct because there is no efficient way to order storage updates.

```
open(/dir/tmp)
write(/dir/tmp)
fsync(/dir/tmp)
fsync(/dir)
rename(/dir/tmp, /dir/orig)
fsync(/dir/)
```

(a) Atomic Update via Rename

```
open(/dir/log)
write(/dir/log)
fsync(/dir/log)
fsync(/dir/)
write(/dir/orig)
fsync(/dir/orig)
unlink(/dir/log)
fsync(/dir/)
```

(b) Atomic Update via Logging

```
// Write attachment
open(/dir/attachment)
write(/dir/attachment)
fsync(/dir/attachment)
fsync(/dir/)

// Writing SQLite Database
open(/dir/journal)
write(/dir/journal)
fsync(/dir/journal)
fsync(/dir/)
write(/dir/db)
fsync(/dir/db)
unlink(/dir/journal)
fsync(/dir/)
```

(c) Atomically adding a email message with attachments in Android Mail

Figure 1: Different protocols used by applications to make consistent updates to persistent data.

Currently, applications use the `fsync()` system call to order updates to storage [2]; since `fsync()` forces durability of data, the latency of a `fsync()` call varies from a few milliseconds to several seconds. As a result, applications do not call `fsync()` at all the places in the update protocol where it is necessary, leading to severe data loss and corruption bugs [21].

We now describe two common techniques used by applications to consistently update stable storage. Figure 1 illustrates these protocols.

Atomic rename. Protocol (a) shows how a file can be updated via atomic rename. The atomic rename approach is widely used by editors, such as Emacs and Vim, and by GNOME applications that need to atomically update dot configuration files. The application writes new data to a temporary file, persists it with an `fsync()` call, updates the parent directory with another `fsync()` call, and then renames the temporary file *over* the original file, effectively causing the directory entry of the original file to point to the temporary file instead. The old contents of the original file are unlinked and deleted. Finally, to ensure that the temporary file has been unlinked properly, the application calls `fsync()` on the parent directory.

Logging. Protocol (b) shows another popular technique for atomic updates, logging [8] (either write-ahead-logging or undo logging). The log file is written with new contents, and both the log file and the parent directory (with the new pointer to log file) are persisted. The application then updates the original file and persists the original file; the parent directory does not change during this step. Finally, the log is unlinked, and the parent directory is persisted.

The situation becomes more complex when applications store state across multiple files. Protocol (c) illustrates how the Android Mail application adds a new email with an attachment. The attachment is stored on the file system, while the email message (along with metadata) is stored in the database (which for SQLite, also resides on the file system). Since the database has a pointer to the attachment (i.e., a file name), the attachment must be persisted first. Persisting the attachment requires two `fsync()` calls (to the file and its containing directory) [1, 21]. SQLite's most performant mode uses write-ahead-logging to atomically update the database. It then follows a protocol similar to Protocol (b).

Removing `fsync()` calls in any of the presented protocols will lead to data loss or corruption. For instance, in Protocol (b), if the parent directory is not persisted with an `fsync()` call, the following scenario could occur: the application writes the log file, and then starts overwriting the original file in place. The system crashes at this point. Upon reboot, the log file does not exist, since the directory entry pointing to the log file was not persisted. Thus, the application file has been irreversibly partially edited, and cannot be restored to a consistent version. Many application developers avoid `fsync()` calls due to the resulting decrease in performance, leading to severe bugs that cause loss of data.

Safe update protocols for stable storage are complex and low performance (*e.g.,* Android Mail uses *six* `fsync()` calls to persist a single email with an attachment). System support for transactions will provide high performance for these applications.

2.2 Application case studies

We now present four examples of applications that struggle with obtaining crash consistency using primitives available today. Several applications store data across the file system, key-value stores, and embedded databases such as SQLite [27]. While all of this data ultimately resides in the file system, their APIs and performance constraints are different and consistently updating state across these systems is complex and error-prone.

Android mail. Android's default mail application stores mail messages using the SQLite embedded database [29]. Mail attachments are stored separately as a file, and the database stores a pointer to the file. The user requires both the file and the database to be updated atomically; SQLite only ensures the database is updated correctly. For example, a crash could leave the database consistent, but with a dangling pointer to a missing attachment file. The mail application handles this by first persisting the attachment (via `fsync()`), and then persisting a database transaction. Clearly, this harms performance – a transaction that spans both the database and the file system would need to persist data only at a single commit point.

Apple iWork and iLife. Analysis of the storage behavior of Apple's home-user, desktop applications [9] finds that applications use a combination of the file system, key-value stores, and SQLite to store data. iTunes uses SQLite to store metadata similar to the Android Mail application. When you download a new song via iTunes, the sound file is transferred and the database updated with the song's metadata. Apple's Pages application uses a combination of SQLite and key-value stores for user preferences and other metadata (two SQLite databases and 128 `.plist` key-value store files). Similar to Android Mail, Apple uses `fsync()` to order updates correctly.

Browsers. Mozilla Firefox stores user data in multiple SQLite databases. For example, addons, cookies, and download history are each stored in their separate SQLite database. Since downloads and other files are stored on the file system, a crash could leave a database with a dangling pointer to a missing file.

Version control systems. Git and Mercurial are widely-used version control systems. The `git commit` command requires two file-system operations to be atomic: a file append (`logs/HEAD`) and a file rename (to a lock file). Failure to achieve atomicity results in data loss and a corrupted repository [21]. Mercurial uses a combination of different files (`journal, filelog, manifest`) to consistently update state. Mercurial's `commit` command requires a long sequence of file-system operations including file creations, appends, and renames be atomic; if not, the repository is corrupted [21].

For these applications, transactional support would lead directly to more understandable and more efficient idioms. It is difficult for a user-level program to provide crash-consistent transactional updates using the POSIX file-system interface. A transactional file-system interface will also enable high-performance idioms like editors grouping updates into transactions rather than the less efficient process they currently use of making temporary file copies that are committed via `rename`.

Note that applications that use temporary files and techniques like atomic rename do achieve crash consistency; however, after a crash there may be temporary files which need to be cleaned up. After a crash, the application runs a recovery procedure and returns to a consistent state. Often, the "recovery procedure" forces a human user to look for and manually delete stale files. A transactional file system does not provide *new* crash-consistency guarantees for these applications; rather, transactional file systems remove the burden of recovery and cleanup, simplifying the application and eliminating bugs [21].

2.3 Optimizing transactions

A transactional file-system interface enables a number of interesting file-system optimizations. We now describe a few of them.

Eliminate temporary durable files. A number of applications such as Vim, Emacs, Git, and LevelDB provide reasonable crash semantics (*i.e.,* the user sees either the old version or the new version after an update) by making a temporary copy of a file, editing it, then renaming it atomically to the permanent name when the user updates data. The application can simply enclose its writes inside a transaction, avoiding the copy. For large files, the difference in performance can be significant. In addition, the file system will not be cluttered with temporary files in the event of a crash.

Group commit. Transactions buffer related file-system updates in memory, which can all be sent to the storage device at once. Batching updates is often more efficient, enabling efficient allocation of file-system data structures and better device-level scheduling. Without user-provided transaction boundaries, the file system provides uniform, best-effort persistence for all updates.

Eliminate redundant IO *within* transactions. Workloads often contain redundancy; for example, files are often updated several times at the same offset, or a file is created, written, read, and unlinked. Transaction boundaries allow the file system to eliminate some of this redundant work because the entire transaction is visible to the file system at commit time, which enables global optimization.

Consolidate IO *across* transactions. Transactions often update data written by prior transactions. When a workload anticipates data in its transaction will be updated by another transaction shortly, it can prioritize throughput over latency. Committing a transaction with a special flag allows the system to delay a transaction commit, anticipating that the data will be overwritten, and then it can be persisted once instead of twice. Note that consolidating IO in this manner is different from eliminating redundant IO within a transaction; this optimization operates across multiple transactions. Optimizing multiple transactions, especially from different applications, is best done by the operating system, not by an individual application. This non-work conserving strategy is similar to the anticipatory disk scheduler [12].

Separate ordering from durability. When ending a transaction, the programmer can specify if the transaction should commit durably. If so, the call blocks until all updates specified by the transaction have been written to a persistent journal. If we commit non-durable transaction A and then start non-durable transaction B, then A is ordered before B, but neither is durable. A subsequent transaction (*e.g.,* C), can specify that it and all previous transactions should be made durable. In this way we can use transactions to gain much of the benefit of splitting `sync` into ordering sync (`osync`), and durability sync (`dsync`) [2].

In summary, we believe transactional file systems should be revisited for two reasons. First, applications routinely store persistent state in multiple files and across different storage systems such as databases and key-value stores, and maintaining crash consistency of this state using techniques such as atomic rename results in complexity and bugs. Second, using a transactional API enables the file system to provide a number of optimizations that would be significantly harder to introduce in a non-transactional file system.

3 TxFS Design and implementation

We now present the design and implementation of TxFS. TxFS avoids the pitfalls from earlier transactional file systems (§6): it has a simple API; provides complete ACID guarantees; does not depend on specific hardware; and takes advantage of the file-system journal and how the kernel is implemented to achieve a small implementation (≈5,200 LOC).

3.1 API

A simple API was one of the key goals of TxFS. Thus, TxFS provides developers with only three system calls: `fs_tx_begin()`, which begins a transaction; `fs_tx_commit()`, which ends a transaction and attempts to commit it; and `fs_tx_abort()`, which discards all file-system updates contained in the current transaction. On commit, all file-system updates in an application-level transaction are persisted in an atomic fashion – after a crash, users see all of the transaction updates, or none of them. This API significantly simplifies application code and provides clean crash semantics, since temporary files or partially written logs will not need to be cleaned up after a crash.

`fs_tx_commit()` returns a value indicating whether the transaction was committed successfully, or if it failed, why it failed. A transaction can fail for three reasons: there was a conflict with another concurrent transaction, there is no journal space for the transaction, or the file system does not have enough resources for the transaction to complete (no space or inodes). Depending upon the error code, the application can choose to retry the transaction. Nested TxFS transactions are flattened into a single transaction, which succeed or fail as a unit. Flat nesting is a common choice in transactional systems [22, 28].

A user can surround any sequence of file-system related system calls with `fs_tx_begin()` and `fs_tx_commit()` and the system will execute those system calls in a single transaction. This interface is easy for programmers to use and makes it simple to incrementally deploy file-system transactions into existing applications. In contrast, some transactional file systems (*e.g.,* Window's TxF [24] and Valor [28]) have far more complex, difficult-to-use interfaces. TxF assigns a handle to each transaction, and requires users to explicitly call the transactional APIs with the handle. Valor exposes operations on the kernel log to user-level code.

TxFS isolates file-system updates only. The application is still responsible for synchronizing access to its own user-level data structures. A transactional file system is not intended to be an application's sole concurrency control mechanism; it only coordinates file-system updates which are difficult to coordinate without transactions.

3.2 Atomicity and durability

Most modern Linux file systems have an internal mechanism for atomically updating multiple blocks on storage. These mechanisms are crucial for maintaining file-system crash consistency, and thus have well-tested and mature implementations. TxFS takes advantage of these mechanisms to obtain three of the ACID properties: atomicity, consistency, and durability. This is the key insight which allows TxFS to have a small implementation.

TxFS builds upon the ext4 file system's journal. The journal provides the guarantee that each journal transaction is applied to the file system in an atomic fashion. We could have instead used a different mechanism such as copy-on-write [10] which provides the same guarantee in btrfs and F2FS. TxFS can be built upon any file system with a mechanism for atomic updates.

For each TxFS transaction, TxFS maintains a private `jbd2` transaction, and at commit, merges the private transaction into the global `jbd2` transaction. While the global `jbd2` transaction contains only metadata by default, TxFS also adds data blocks to the transaction to ensure atomic updates. If, by chance, a block added to the private `jbd2` transaction is also being committed by a previous global `jbd2` transaction, TxFS creates a shadow block. Ext4 also creates a shadow block when a block is shared between a running and a committing transaction. TxFS employs selective data journaling [2], only journaling data blocks that were already allocated (*i.e.,* data blocks that are being updated), and avoids journaling newly allocated data blocks (because it can write them directly). Selective data journaling provides the same guarantees as full data journaling at a fraction of the cost.

TxFS ensures that an entire transaction can be merged into a *single* journal transaction; otherwise, an error is returned to the user. As long as a TxFS transaction is added to a single journal transaction, the journal will ensure it is applied to the file system atomically. After merging a user's transaction into the journal transaction, TxFS persists the journal transaction, ensuring the durability of the TxFS transaction.

3.3 Isolation and conflict detection

Although the ext4 journal provides atomicity and durability, it does not provide isolation. Adding isolation for file-system data structures in the Linux kernel is challenging because a large number of functions all over the kernel modify file-system data structures without using a common interface. In TxFS, we tailor our approach to isolation for each data structure to simplify the implementation.

To provide isolation, TxFS has to ensure that all operations performed inside a transaction are not visible to other transactions or the rest of the system until commit time. TxFS achieves the isolation level of *repeatable reads* [7] using a combination of different techniques.

Split file-system functions. System calls such as `write()` and `open()` execute file-system functions which often result in allocation of file-system resources such as data blocks and inodes. TxFS splits such functions into two parts: one part which does file-system allocation, and one part which operates on in-memory structures. The part doing file-system allocation is moved to the commit point. The other part is executed as part of the system call, and the in-memory changes are kept private to the transaction.

Transaction-private copies. TxFS makes transaction-private copies of all kernel data structures modified during the transaction. File-system related system calls inside a transaction operate on these private copies, allowing transactions to read their own writes. In case of abort, these private copies are discarded; in case of commit, these private copies are carefully applied to the global state of the file system in an atomic fashion. During a transaction, file-system operations are redirected to the local in-memory versions of the data structures. For example, dentries updated by the transaction are modified to point to a local inode which maintains a local radix tree which has locally modified pages.

Two phase commit. TxFS transactions are committed using a two-phase commit protocol. TxFS first obtains a lock on all relevant file-system data structures using a total order. The following order prevents deadlock: inode mutexes, page locks, inode buffer head locks, the global `inode_hash_lock`, the global `inode_sb_list_lock`, inode locks, and dentry locks. The Linux kernel orders the acquiring of inode mutexes based on the pointer addresses of their inodes; we adopt this locking discipline in TxFS. Similarly, page locks are acquired in order of the address of the page. Acquiring the locks for directory data block buffers and inode metadata buffers is ordered by inode number.

After obtaining the locks, all allocation decisions are checked to see if they would succeed; for example, if the transaction creates inodes, TxFS checks if there are enough free inodes. Next, TxFS checks the journal to ensure there is enough space in the global `jbd2` transaction to allow the transaction to be merged. Finally, TxFS

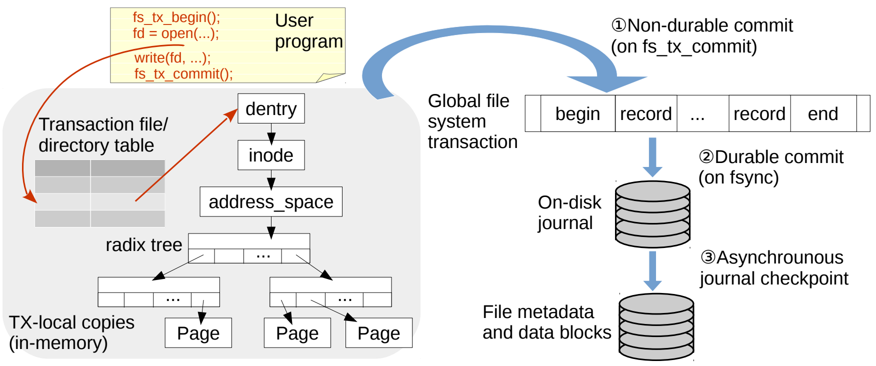
fs_tx_begin();
fd = open(...);
write(fd, ...);
fs_tx_commit();
User program
①Non-durable commit (on fs_tx_commit)
Global file system transaction
begin
record
...
record
end
②Durable commit (on fsync)
On-disk journal
③Asynchrounous journal checkpoint
File metadata and data blocks
Transaction file/ directory table
dentry
inode
address_space
radix tree
TX-local copies (in-memory)
Page
Page
Page

Figure 2: TxFS relies on ext4's own journal for atomic updates and maintains local copies of in-memory data structures, such as inodes, dentries, and pages to provide isolation guarantees. At commit time, the local operations are made global and durable.

checks for conflicts with other transactions (as described below). If any of these checks fail, all locks are released, and the commit returns an error to the user. Otherwise, the in-memory data structures are updated, all file-system allocation is performed, and the private `jbd2` transaction is merged with the global `jbd2` transaction. At this point, the transaction is committed, locks are released and the changes are persisted to storage in a crash-consistent manner.

Conflict detection. Conflict detection is a key part of providing isolation. Since allocation structures such as bitmaps are not modified until commit time, they cannot be modified by multiple transactions at the same time, and do not give rise to conflicts; as a result, TxFS avoids false conflicts involving global allocation structures.

Conflict detection is challenging as file-system data structures are modified all over the Linux kernel without a standard interface. TxFS takes advantage of how file-system data structures are implemented to detect conflicts efficiently.

Conflict detection for pages. The `struct page` data structure holds the data for cached files. TxFS adds two fields to this structure: `write_flag` and `reader_count`. The `write_flag` indicates if there is another transaction that has written this page. The `reader_count` field indicates the number of other transaction that have read this page. Non-transactional threads will never see the in-flight un-committed data in transactions, and thus can always safely read data. TxFS does eager conflict detection for pages since there is a single interface to read and write pages that TxFS interposes. The following rules are followed on a page read or write:

1. When a transaction reads a page, it increments `reader_count` by one. If the page has the `write_flag` set, the transaction aborts.
2. If a transaction attempts to write a page that has either the `write_flag` set or `reader_count` greater than zero, it aborts. Otherwise, it sets the `write_flag`.
3. If a non-transactional thread attempts to write to a page with `reader_count` or `write_flag` set, it is put to sleep until the transaction commits or aborts.
4. When the transaction commits or aborts, `write_flag` is reset and `reader_count` is decremented.

Aborting transactions in this manner can lead to livelock, but we have not found it a problem with our benchmarks and the policy can be easily changed to resolve conflicts in favor of the oldest transaction (which does not livelock). TxFS favors transactional throughput, but for greater fairness between transactional and non-transactional threads, TxFS could allow a non-transactional thread to proceed by aborting all transactions conflicted by its operation [22].

Conflict detection for dentries and inodes. Apart from pages, TxFS must detect conflicts on two other data structures: dentries (directory entries) and inodes. Unfortunately, unlike pages, inodes and dentries do not have a standard interface and are modified throughout kernel code. Therefore, TxFS uses lazy conflict detection for inodes and dentries, detecting conflicts at commit time. At commit time, TxFS needs to detect if the global copy of the data structure has changed since it was copied into the local transaction. Doing a byte-by-byte comparison of each modified data structure would significantly increase commit latency; instead, TxFS takes advantage of the inode's `i_ctime` field that is changed whenever the inode is changed; TxFS simply has to check that the `i_ctime` has not changed for each inode that TxFS has read or written (writes are performed to a transaction-local copy of the inode). TxFS similarly adds a new `d_ctime` field to the dentry data structure to track its last modified time. We added kernel code in a number of places to update `d_ctime` whenever a dentry is changed. Creating different named entries within a directory does not create a conflict because the names are checked at commit time. By taking advantage of `i_ctime` and `d_ctime`, TxFS is able to perform conflict detection for these structures without radically changing the Linux kernel.

Summary. Figure 2 shows how TxFS uses ext4's journal for atomically updating operations inside a transaction, and maintaining local state to provide isolation guarantees. File operations inside a TxFS transaction are redirected to the transaction's local copied data structures, hence they do not affect the file system's global state, while being observable by subsequent operations in the same transaction. Only after a TxFS transaction finishes its commit (by calling `fs_tx_commit()`) will its modifications be globally visible.

3.4 Implementation

We modified Linux version 3.18 and the ext4 file system. The implementation requires a total of 5,200 lines of code, with 1,300 in TxFS internal bookkeeping, 1,600 in the VFS layer, 900 in the journal (JBD2) layer, 1,200 for ext4 and 200 for memory management (all measurements with SLOCCount [4]). Except for the ext4 and jbd2 extensions, all other code could be reused to port TxFS to other file systems, such as ZFS, in the future.

3.5 Limitations

TxFS has two main limitations. First, the maximum size of a TxFS transaction is limited to one fourth the size of the journal (the maximum journal transaction size allowed by ext4). We note that the journal can be configured to be as large as required. Multi-gigabyte journals are common today. Second, although parallel transactions can proceed with ACID guarantees, each transaction can only contain operations from a single process. Transactions spanning multiple processes are future work.

4 Accelerating program idioms with TxFS

We now explore a number of programming idioms where a transactional API can improve performance because transactions provide the file system a sequence of operations which can be optimized as a group (§2). Whole transaction optimization can result in dramatic performance gains because the file system can eliminate temporary durable writes (such as the creation, use and deletion of a log file). In some cases, we show that benefits previously obtained by new interfaces (such as `osync` [2]) can be obtained easily with transactions.

4.1 Eliminating file creation

When an application creates a temporary file, syncs it, uses it, and then unlinks it (*e.g.,* logging shown in Figure 1b), enclosing the entire sequence in a transaction allows the file system to optimize out the file creation and all writes while maintaining crash consistency.

Workload	FS	TX
Create/unlink/sync	37.35s	0.28s (133×)
Logging	5.09s	4.23s (1.20×)
Ordering work	2.86 it/s	3.96 it/s (1.38×)

Table 1: Programming idioms sped up by TxFS transactions. Performance is measured in seconds (s), and iterations per second (it/s). Speedups for the transaction case are reported in parentheses.

The create/unlink/sync workload spawns six threads (one per core) where each thread repeatedly creates a file, unlinks it, and syncs the parent directory. Table 1 shows that placing the operation within a transaction increases performance by 133× because the transaction completely eliminates the workload's IO. While this test is an extreme case, we next look at using transactions to automatically convert a logging protocol into a more efficient update protocol.

4.2 Eliminating logging IO

Figure 1b shows the logging idiom used by modern applications to achieve crash consistency. Enclosing the entire protocol within a transaction allows the file system to transparently optimize this protocol into a more efficient direct modification. During a TxFS transaction, all sync-family calls are functional nops. Because the log file is created and deleted within the transaction, it does not need to be made persistent on transaction commit. Eliminating the persistence of the log file greatly reduces the amount of user data but also file system metadata (*e.g.,* block and inode bitmaps) that must be persisted.

Table 1 shows execution time for a microbenchmark that writes and syncs a log, and a version that encloses the entire protocol in a single TxFS transaction. Enclosing the logging protocol within a transaction increases performance by 20% and cuts the amount of IO performed in half because the log file is never persisted. Rewriting the code increases performance by 55% (3.28s, not shown in the table). In this case getting the most performance out of transactions requires rewriting the code to eliminate work that transactions make redundant. But even without a programmer rewrite, by just adding two lines of code to wrap a protocol in a transaction achieves 47% of the performance of doing a complete rewrite.

Optimizing SQLite logging with TxFS. Table 3 reports results for SQLite. "Rollback with TxFS" represents SQLite's default logging mode encased within a TxFS transaction. Just enclosing the logging activity with a transaction increases performance for updates by 14%. Modifying the code to eliminate the logging work that transactions make redundant increases the performance for updates to 31%, in part by reducing the number of

Experiment	TxFS benefit	Speed
Single-threaded SQLite	Faster IO path, Less sync	1.31×
TPC-C	Faster IO path, Less sync	1.61×
Android Mail	Cross abstraction	2.31×
Git	Crash consistency	1×

Table 2: The table summarizes the micro- and macro-benchmarks used to evaluate TxFS, and the speedup obtained in each experiment.

system calls by 2.5×.

4.3 Separating ordering and durability

Table 1 shows throughput for a workload that creates three 10MB files and then updates 10MB of a separate 40MB file. The user would like to create the files first, then update the data file. This type of ordering constraint often occurs in systems like Git that create log files and other files that hold intermediate state.

The first version uses `fsync()` to order the operations, while the second uses transactions that allow the first three file create operations to execute in any order, but they are all serialized behind the final data update transaction (using flags to `fs_tx_begin()` and `fs_tx_commit()`). The transactional approach has 38% higher throughput because the ordering constraints are decoupled from the persistence constraints. The work that first distinguished ordering from persistence suggests adding different flavor sync system calls [2], but TxFS can achieve the same result with transactions.

5 Evaluation

We evaluate the performance and durability guarantees of TxFS on a variety of micro-benchmarks and real workloads. The micro-benchmarks help point out how TxFS achieves specific design goals while the larger benchmarks validate that transactions provide stronger crash semantics and improved performance to a variety of large applications with minimal porting effort.

Testbed. Our experimental testbed consists of a machine with a 4 core Intel Xeon E3-1220 CPU and 32 GB DDR3 RAM and a machine with a 6 core Intel Xeon E5-2620 CPU and 8 GB DDR3 RAM. All experiments are performed on Ubuntu 16.04 LTS (Linux kernel 3.18.22). The kernel is installed on a Samsung 850 (512 GB) SSD and all experiments are done on a Samsung 850 (250 GB) SSD. The experimental SSD is run at low utilization (around 20%) to prevent confounding factors from wear leveling firmware.

Table 2 presents a summary of the different experiments used to evaluate TxFS and the speedup obtained in each experiment. In the Git experiment, TxFS provides strong crash-consistency guarantees without degrading performance. Note that if not explicitly mentioned, all our baselines run on ext4 with its default journaling mode, the ordered journaling mode.

5.1 Crash consistency

TxFS's ACID transactions should be recoverable after a system crash. In order to verify this crucial correctness property, we boot a virtual machine and run a script that creates many types of transactions in multiple threads with random amounts of contained work and conflict probabilities. We crash the VM at a random time and make sure the file system journal is recoverable and that the file system passes all fsck checks. We have run over 100 random crashes and can recover the file system in all cases. An alternate way to test crash consistency would use a testing framework such as CrashMonkey [13].

5.2 Stress testing TxFS

We performed stress testing on TxFS to ensure its correctness in the face of conflicts and multi-threaded operations. Our stress tests had two main workloads. Our first workload was a micro-benchmark with six threads starting TxFS transactions and performing file-system operations picked at random across two files before committing. These threads generate a lot of conflicts, stressing TxFS conflict detection and isolation mechanisms. Our second workload uses the SQLite embedded database, performing a number of database operations with multiple threads. We were able to run both workloads for over 24 hours on TxFS without a kernel crash or our unit tests failing, giving us a measure of confidence in the correctness and stability of the codebase.

5.3 SQLite

We modified SQLite to use TxFS transactions. Data and metadata are first written safely to the journal and then checkpointed in-place into the file system. Note that all metadata is written into the file system exactly once. With SQLite in write-ahead-logging (WAL) mode, metadata is written twice: once to SQLite's log and once to the actual database file. The size and frequency of metadata updates for SQLite is significant because in order to be recoverable, it must update the parent directory whenever log files are created or deleted [29]. We use `PRAGMA synchronous=NORMAL` (default) for all modes, and `PRAGMA wal_checkpoint(FULL)` for WAL mode to guarantee all ACID properties.

When SQLite uses TxFS transactions, crashes do not leave any residual files on storage. Currently, users often must remove these residual files by hand which is tedious and error-prone. TxFS transactions eliminate user-visible log files; user-level code sees only the before and after state of the database, not messy in-flight data.

Single-threaded SQLite. Table 3 shows that TxFS is the best performing option for SQLite updates. Data is

	Performance (kOps/s)		IO (GB)		Sync/tx	
Journal mode	Insert	Update	Insert	Update	Insert	Update
Rollback (default)	53.9	28.0	1.9	3.9	4	10
Truncate	53.5 (0.99×)	28.9 (1.03×)	1.9	3.9	4	10
WAL	39.8 (0.74×)	34.6 (1.23×)	3.9	3.8	3	3
TxFS	51.4 (0.95×)	36.7 (1.31×)	1.9	3.8	1	1
Rollback with TxFS	52.1 (0.97×)	31.9 (1.14×)	1.9	3.8	1	1
No journal (**unsafe**)	54.9 (1.02×)	50.6 (1.81×)	1.9	1.9	1	1

Table 3: The table compares operations per second (larger is better) and total amount of IO for SQLite executing 1.5M 1KB operations grouping 10K operations in a transaction using different journaling modes (including TxFS). The database is pre-populated with 15M rows. All experiments use SQLite's synchronous mode (its default).

	Rollback (default)	Truncate	WAL	TxFS	No journal (**unsafe**)
Delivery	110.52	123.33	157.01	188	300.4
New Order	142.38	165.15	216.8	240.34	445.14
Order Status	1998.53	2067.29	3317.1	2489.94	3141.13
Payment	198.45	240.21	367.26	300.61	909.91
Stock levl	575.03	602.33	765.41	684.06	1079.85
Total	172.97	203.3 (1.18×)	280.01 (1.62×)	278.97 (1.61×)	600.15 (3.47×)
Syscall/tx	208.0	207.95	138.26	100.35	146.9
Sync/tx	2.76	2.75	2.76	0.92	0.92
R MB/tx	0.018	0.016	0.013	0.013	0.007
W MB/tx	0.17	0.158	0.131	0.129	0.066
T MB/tx	0.187	0.174 (0.93×)	0.144 (0.77×)	0.142 (0.76×)	0.073 (0.39×)

Table 4: Rates (in transactions per second) for the TPC-C workload using different SQLite journaling modes. Each workload runs continuously for a fixed amount of time.

the average of five trials with standard deviations below 2.2% of the mean. For the update workload, TxFS is 31% faster than the default. We report IO totals as part of our validation that TxFS correctly writes all data in a crash-consistent manner. Several choices for SQLite logging mode, including TxFS, result in similar levels of IO that resemble the no-journal lower bound. Write-ahead logging mode (WAL) writes more data for the insert workload, which harms its performance. Note that TxFS does not suffer WAL's performance shortfall on insert, and TxFS surpasses WAL's performance on update, making it a better alternative. Although the file system journal shares similarity with a WAL log, TxFS does not generate redundant IO on insert because of its selective data journaling.

We run similar experiments with small updates (16 bytes) and find that there is little difference in performance between SQLite's different modes and TxFS. This shows that small transactions do not have significant overhead in TxFS.

TxFS's improves performance for the update workload is due to several factors. TxFS reduces the number of data syncs from 10 (in Rollback and Truncate mode) or 3 (in WAL mode) to only 1, which leads to better batching and re-ordering of writes inside a single transaction. It performs half of its IO to the journal, which is written sequentially. The remaining IO is done asynchronously via a periodic file-system checkpoint that writes the journaled blocks to in-place files. Since TxFS uses the file-system journal instead of an application-level journal for logging the transaction, it avoids the *journaling on journal* problem [25], where the journaling of the application-level log causes a significant slowdown. Even in realistic settings where performance is at a premium, transactions provide a simple, clean interface to get significantly increased file-system performance, while maintaining crash safety.

5.4 TPC-C

We run a version of the TPC-C benchmark [17], ported to use single-threaded SQLite[1]. TPC-C is a standard online transaction processing benchmark for an order-entry environment. R MB/tx is the amount of read IO per transaction, W is written IO and T is total.

Table 4 shows that TxFS outperforms SQLite's default mode by 1.61×. The performance advantage comes from two sources. First, TxFS writes less data and batches its writes. TxFS writes much of its data sequentially to the file system journal on `fs_tx_commit()` and writes back the journal data asynchronously. SQLite's default mode must write data to the SQLite journal and to the database file on `fsync()`. Therefore, TxFS writes only once in the critical path (to the journal), while SQLite (as configured in Section 5.3) must write to the journal plus database in the critical path. Second, TxFS decreases the number of system calls, especially sync-family calls. Table 4 shows that TxFS reduces the number of sync-family calls per transaction by 3×. By reducing the sync-familly calls, TxFS can batch writes in a transaction, reducing the amount of writes by 31.7% compared to default mode.

The performance of TxFS and WAL is similar. When transactions contain writes, TxFS has better performance than WAL, but it has worse performance for read-only

[1]`https://github.com/apavlo/py-tpcc/wiki/SQLite-Driver`

Journal mode	Throughput	IO(MB)
Rollback (default)	45.73	3269
Truncate	45.48 (0.99×)	3154
WAL	53.43 (1.17×)	3539
TxFS	105.68(2.31×)	6797
TxFS Small tx	60.85 (1.33×)	4052
No journal (**unsafe**)	61.88 (1.35×)	3995

Table 5: TxFS supports transactions across storage abstractions. Performance is measured in iterations per second.

transactions: WAL is 28% faster than TxFS for read-only transactions. "Order status" and "Stock level" consist of 3 select queries and 2 select queries respectively, resulting in lower throughput for TxFS compared with WAL. However, "Delivery" consists of 3 select, 3 update, and 1 delete queries, so TxFS outperforms WAL by 20%.

5.5 Abstractions built on files

Modern file systems support storage of not only files but databases (*e.g.,* SQLite) and key-value stores (*e.g.,* LevelDB and RocksDB). These abstractions are built on the file system and generally are a lower-performing, but easier to set up and maintain alternative to their dedicated counterparts.

TxFS supports transactions that span storage abstractions. Table 5 shows the throughput for a workload that models the core activity of Android mail, storing an image file and recording the path to that file in a SQLite database along with other metadata. The database is prepopulated with 100,000 1KB rows, image files are 1 MB. The workload creates the database record in one transaction, creates a uniquely named file where it stores the file data, syncs the data, and then updates the database record in a second transaction.

TxFS outperforms default SQLite by 2.31× and the best alternative (WAL mode) by 1.98×. It is essential to TxFS's performance that both database transactions as well as the file system operation are all contained in a single transaction. When they are separate transactions (TxFS Small tx), performance is bounded by SQLite (i.e., it is close to no journaling). IO is not a bottleneck for this workload. The amount of IO performed is proportional to the amount of work done: TxFS has higher throughput, so it performs more IO.

5.6 Git

Git is a widely-used version control system. Git commands such as `git add` and `git commit` result in a large number of file-system operations. Git updates files by creating a temporary file, writing the desired

Category	System	Isolation	Durability	Easy-to-use API	Hardware independence	Performance	Complexity
In-kernel transactional FS	**TXFS**	✓	✓	✓	✓	H	L
	Valor	✓	✓	✗	✓	H	L
	TxF	✓	✓	✗	✓	H	H
Transactional OS	TxOS	✓	✓	✓	✓	H	H
FS over userspace databases	OdeFS Inversion DBFS Amino	Relying on databases		✗	✓	L	L
Transactional storage	CFS	✗	✓	✓	✗	H	L
	MARS	✓	✓	✗	✗	H	H
	Isotope	✓	✓	✓	✓	H	H
Failure atomicity	msync	✗	✓	✓	✓	H	L
	AdvFS	✗	✓	✓	✓	H	L

Table 6: The table compares prior work providing ACID transactions or failure atomicity in a local file system. Legend: ✓- supported, ✗- unsupported, L - Low, H - High. Note that only TxFS provides isolation and durability with high performance and low implementation complexity without restrictions or hardware modifications.

data to it, and renaming it over the old file. To enable high performance, Git does not order its operations via `fsync()` [21], leaving it vulnerable to garbage files and outright data corruption on a system crash.

In our experiment, we run Git inside a virtual machine. We instrument the Git code to crash the VM at vulnerable points (such as after the temp file rename, but before the file is persistent). The workload first initializes a Git repository, populates it with 20,000 empty files, then adds all files at once.

After a VM restart, we find that the `.git/index` file has been truncated to zero bytes, resulting in a loss of the working tree. Running the Git recovery command `git fsck` simply reports a fatal error. Recovery is not possible unless the data has been backed up in another location. In contrast, when we change Git to use TxFS transactions, we find that crashes no longer produce such catastrophic errors. Furthermore, we do not find a significant difference in performance between the code that use TxFS transactions, and the code that does not. Thus, using TxFS transactions provides crash consistency for Git without any performance overhead.

6 Related work

There have been a number of efforts over the years to provide systems support for file-system transactions. Each of these systems failed to gain adoption due to one of the following reasons: they had severe restrictions on what could be placed inside a transaction, they were complicated to use, they added complexity to the kernel, or they caused significant performance degradation. Learning from prior systems, TxFS avoids all of these mistakes. Table 6 summarizes related work and demonstrates that TxFS is unique among transactional file systems.

Building file systems on top of user-space databases. One way to provide transactional updates for applications is to build a file system over a user-space transactional database. OdeFS [5], Inversion [18], and DBFS [16] use a database (such as Berkeley DB [19]) to provide ACID transactions to applications via NFS. Amino [31] tracks all user updates via `ptrace` and employs a user-level database to provide transactional updates. Such systems come with significant performance cost (*e.g.,* 50-80% for large operations in DBFS [16]).

In-kernel transactional file systems. An approach that leads to higher performance is adding transactions to in-kernel file systems. Valor [28] provides kernel support for file-system transactions. However, Valor does not provide a simple begin/end transaction interface, and it forces programmers to use seven new system calls to manage the transaction log.

Microsoft introduced Transactional NTFS (TxF), Transaction Registry (TxR), and the kernel transaction manager (KTM) in Windows Vista [24]. Using TxF requires all transactional operations be explicit (i.e., instead of using `read()` in a transaction, the programmer must add an explicit transactional read). Therefore TxF had a high barrier to entry and code that used it required separate maintenance. TxF also had significant limitations, like no transactions on the root file system.

Transactional operating systems. A third, somewhat heavyweight, approach is modifying the entire operating system to provide transactions. Our prior work, TxOS [22], is an operating system that provides transactions. This approach adds significant complexity to the kernel. For example, TxOS modified tens of thousands of lines of code and changed core OS data structures like the inode. Maintaining such a kernel will be tricky – Windows abandoned its transactional file system and kernel transaction manager [14].

The transactional capabilities of the file system supported by TxOS is similar in approach to TxFS. It also uses the file-system journal and modifies the virtual file system (VFS) code to provide isolation. One could view TxFS as specializing TxOS to the file system, achieving a transactional file system at significantly lower cost, while adding file-system specific optimizations like selective journaling and eliminating redundant work within transactions.

Transactional storage systems. Similar to our work, CFS [15] provides a lightweight mechanism for atomic updates of multiple files, building on top of transactional flash storage. MARS [3] builds on hardware-provided atomicity to build a transactional system. TxFlash [23] uses the copy-on-write nature of Flash SSDs to provide transactions at low cost. In contrast to these systems, TxFS provides transactions without assuming any hardware support (beside device cache flush and atomic sector updates). Isotope [26] uses multi-version concurrency control to provide isolation, significantly increasing its complexity. Isotope builds a user-space transactional file system using FUSE, which limits its performance for certain workloads. The higher abstraction level of TxFS makes implementing transactional optimizations and tailored isolation significantly easier than the lower level of Isotope.

Failure atomicity. Failure-atomic msync [20] is similar to TxFS in that it re-uses the journal for providing atomicity to application updates; in contrast, TxFS provides full ACID transactions at significantly higher complexity. AdvFS [30] is also limited in the same way, is specific to the Tru64 file system, and is not available as open-source (latest version available was from 2008). The principles behind TxFS could be used in any file system that has an internal mechanism for atomic updates.

We previewed the ideas behind TxFS at HotOS [11], but this paper reports on the completed system with comprehensive evaluation.

7 Conclusion

We present TxFS, a transactional file system built with less development effort than previous systems by leveraging the file-system journal. TxFS is easy to develop, it is easy to use, and it does not have significant overhead for transactions. We show that using TxFS transactions increases performance significantly for a number of different workloads.

Transactional file systems have not been successful for a variety of reasons. TxFS shows that it is possible to avoid the mistakes of the past, and build a transactional file system with low complexity. Given the power and flexibility of file-system transactions, we believe they should be examined again by file-system researchers and developers. Adopting a transactional interface would allow us to borrow decades of research on optimizations from the database community while greatly simplifying the development of crash-consistent applications.

Acknowledgement

We would like to thank the anonymous reviewers for their feedback and guidance. This work is supported in part by NSF (CNS-1618563, CCF-1333594) and by generous donations from VMware, Facebook, and Google.

References

[1] Fsync man page. `http://man7.org/linux/man-pages/man2/fdatasync.2.html`.

[2] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[3] Coburn, Joel and Bunker, Trevor and Schwarz, Meir and Gupta, Rajesh and Swanson, Steven. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, New York, NY, USA, 2013. ACM.

[4] David A. Wheeler. Sloccount. `https://www.dwheeler.com/sloccount/`.

[5] Gehani, Narain H and Jagadish, HV and Roome, William D. OdeFS: A File System Interface to an Object-Oriented Database. In *VLDB*, pages 249–260. Citeseer, 1994.

[6] Jim Gray. The Transaction Concept: Virtues and Limitations. In *VLDB*, volume 81, pages 144–154, 1981.

[7] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In G. M. Nijssen, editor, *Modelling in Data Base Management Systems, Proceeding of the IFIP Working Conference on Modelling in Data Base Management Systems, Freudenstadt, Germany, January 5-8, 1976*, pages 365–394. North-Holland, 1976.

[8] Robert Hagmann. *Reimplementing the Cedar File System Using Logging and Group Commit*, volume 21. ACM, 1987.

[9] 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. *ACM Transactions on Computer Systems (TOCS)*, 30(3):10, 2012.

[10] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[11] Yige Hu, Younjin Kwon, Vijay Chidambaram, and Emmett Witchel. From Crash Consistency to Transactions. In *16th Workshop on Hot Topics in Operating Systems (HotOS 17)*, Whistler, Canada, 2017.

[12] Sitaram Iyer and Peter Druschel. Anticipatory Scheduling: A Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, 2001.

[13] Ashlie Martinez and Vijay Chidambaram. CrashMonkey: A Framework to Systematically Test File-System Crash Consistency. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[14] Microsoft. Alternatives to using transactional ntfs. "`https://msdn.microsoft.com/en-us/en-%20us/library/hh802690.aspx`".

[15] Min, Changwoo and Kang, Woon-Hak and Kim, Taesoo and Lee, Sang-Won and Eom, Young Ik. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 221–234, 2015.

[16] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system. `https://goo.gl/3Gj328`, 2002.

[17] Raghunath Nambiar, Meikel Poess, Andrew Masland, H. Reza Taheri, Andrew Bond, Forrest Carman, and Michael Majdalany. TPC state of the council 2013. In Raghunath Nambiar and Meikel Poess, editors, *Performance Characterization and Benchmarking - 5th TPC Technology Conference, TPCTC 2013, Trento, Italy, August 26, 2013, Revised Selected Papers*, volume 8391 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2013.

[18] Michael A Olson. The Design and Implementation of the Inversion File System. In *USENIX Winter*, pages 205–218, 1993.

[19] Michael A Olson, Keith Bostic, and Margo I Seltzer. Berkeley DB. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191, 1999.

[20] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 225–238. ACM, 2013.

[21] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[22] Donald E Porter, Owen S Hofmann, Christopher J Rossbach, Alexander Benn, and Emmett Witchel. Operating System Transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 161–176. ACM, 2009.

[23] Prabhakaran, Vijayan and Rodeheffer, Thomas L and Zhou, Lidong. Transactional Flash. In *OSDI*, pages 147–160, 2008.

[24] Russinovich, Mark E and Solomon, David A and Allchin, Jim. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, volume 4. Microsoft Press Redmond, 2005.

[25] Kai Shen, Stan Park, and Men 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.

[26] Shin, Ji-Yong and Balakrishnan, Mahesh and Marian, Tudor and Weatherspoon, Hakim. Isotope: Transactional Isolation for Block Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016.

[27] Riley Spahn, Jonathan Bell, Michael Lee, Sravan Bhamidipati, Roxana Geambasu, and Gail Kaiser. Pebbles: Fine-Grained Data Management Abstractions for Modern Operating Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 113–129, 2014.

[28] Richard P Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *FAST*, volume 9, pages 29–42, 2009.

[29] SQLite. SQLite transactional SQL database engine. `http://www.sqlite.org/`.

[30] 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, pages 203–211, 2015.

[31] Wright, Charles P and Spillane, Richard and Sivathanu, Gopalan and Zadok, Erez. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, 3(2):4, 2007.