The Advantages of a Transactional Interface: Porting Applications to TxFS

Ian Glen Neal

May 10, 2017

Department of Computer Science

The University of Texas at Austin

Committee: Emmett Witchel (Advisor), Vijay Chidambaram, and Robert van de Geijn.

Acknowledgements

I would first like to thank Dr. Emmett Witchel, an excellent advisor who always pushed me to become a better researcher and helped me improve myself and my work. I would also like to thank Yige Hu and Zhiting Zhu for letting me help on the TxFS project, as well as for their guidance on this project.

I would like to thank Graham Mitchell, who started me down this path and without whom I would not have experienced the amazing opportunties of this program and of my undergraduate education.

I would also like to thank my family for their constant support throughout this project and throughout my education.

1 Abstract

In this paper I explore the value of transactional file systems by showing how such systems can benefit existing applications while not adding additional complexity to the codebase. I first discuss the concept of transactions in computing and how transactional semantics are used to provide consistency and durability to an application's state. I examine a new work developed at the University of Texas, TxFS, which provides a very simple and powerful transactional interface. I then introduce how existing systems can be modified to take advantage of TxFS by modifying SQLite, a widely used embedded database, and by modifying OpenLDAP, a widely used implementation of the Lightweight Directory Access Protocol. These modified systems benefit from running on TxFS by having a simplified transactional system, reduced locking, no user-level logging, and enhanced support for multithreaded operations. Additionally, I show how simple it is to port existing systems to TxFS, and demonstrate how easy it would be for other systems to adopt TxFS to ensure durability and consistency for their users.

2 Introduction

Outside of small projects which read and write binary data directly to files, most applications rely on a database of some form to manage their data. Databases are scalable abstractions on top of direct `read()` and `write()` system calls that provide application developers with powerful APIs for data management. One of the most important requirements for a database system is durability. Users expect their data to remain valid upon power loss, system crash, or when multiple applications are attempting to modify the same data.

Any system that performs any sort of data manipulation needs to provide its users with some form of crash consistency and concurrent-modification protection. These guarantees are best described in the form of transactional semantics. Neither modern commodity file systems nor modern commodity operating systems expose a standard way for users to update arbitrary amounts of data atomically, which forces application writers to manufacture their

own methods for atomic updates at user-level [23, 8]. Writing applications in this way can be difficult and can introduce bugs, data races, data loss, and data corruption [23, 38]. Additionally, this practice leads to a large amount of redundant software being developed, as many different systems attempt to reimplement the same concepts in different ways.

In order to address the lack of a standard atomic update interface, a new file system has been developed to provide applications with a transactional interface. This Transactional Filesystem, or TxFS, modifies the `ext4` file system journal to add a simple set of new systems calls for applications to begin, commit, and abort transactions that are composed of an arbitrary number of complex file operations. This elegant solution only modifies the file system, adding no major modifications to any other part of the kernel. By providing this standardized interface, multiple applications can easily use transactions without having to reimplement the logic.

In this paper I discuss the value of transactional file systems and address whether or not the guarantees provided by the system are worth the cost of modification to the operating system and existing applications. I introduce a modified version of SQLite that replaces the regular implementation of SQLite transactions with file system transactions that TxFS provides. I then describe the modifications done to SQLite in order for it to use TxFS transactions and the simplifications to the SQLite codebase that were gained because of TxFS. I also explain how I modified OpenLDAP to take advantage of TxFS transactions, and discuss how certain kinds of operations have to be altered in order to accommodate for transactional semantics. I then evaluate the performance of both of these systems by analyzing a number of benchmarks and show the performance difference between the TxFS implementations and the standard Linux versions. Finally, I discuss related work.

3 Motivation

We live in the age of "Big Data" — we are able to collect large amounts of data, and now we face the problems associated with processing and making use of such data [7]. Of particular interest as it relates to this project is the ability to store this information. As we demand

more from our storage systems, issues of data consistency become essential. Journaling File Systems are one method of addressing this problem at the operating system level. However, this is a solution that solves consistency issues for primitive operations only, such as `read()`, `write()`, or atomic `rename()` [1]. Applications can build upon these primitives, but in order for them to guarantee consistency for higher level user transactions, they must be written with particular care to ensure data consistency, and even then they are prone to faults and data races.

3.1 Data Consistency

One major issue surrounding writing data to permanent storage is the possibility of system failure (*e.g.* power outage or fatal system error) or user error (*e.g.* allowing undergraduates sudo access on research machines) during the middle of a series of updates. This could leave the data in an indeterminate state, requiring tedious and error-prone manual data recovery. Transactional semantics have been the solution for many databasing systems — either an entire set of updates occur and are stable on disk, or none of the updates occur, meaning that they can be reapplied at system recovery. However, the generally complex implementation of user-level transactions causes them to be inefficient. By moving transactional logic into the kernel, user applications would be greatly simplified.

3.2 Performance

A quick search on your favorite or least-favorite search engine will show that database performance is a well-researched topic and is important for writers of I/O bound applications. SQLite, as well as other databases, use a separate file for journaling. However, using a file for the journal causes SQLite to suffer from double journaling, causing 73% slowdown in some cases [31]. Using TxFS eliminates double journaling, and therefore increases the performance of systems that suffer from this problem. Other applications, such as OpenLDAP, protect their state by way of reader-writer locks, so that concurrent processes cannot accidentally modify the same data. This strategy limits concurrency, and is often unnecessary,

such as in OpenLDAP, where separate entries are located in separate files and can be safely modified concurrently. By protecting against concurrency using TxFS transactions, the only operations that get aborted are ones that try to concurrently modify the same data set as another running operations. This leads to higher average operational throughput.

3.3 SQLite

SQLite is a highly portable database system, with the database residing on a single file and an additional journal file for crash consistency. With this design, simply moving database files allows SQLite to be run on different file systems and operating systems [35]. SQLite is an embedded database, which means it is designed to run without a dedicated server process. SQLite databases simply exist as files on disk, and all an application needs to do to access a SQLite database is link with `sqlite.h` and perform queries with `sqlite3_exec()`. SQLite's popularity [35], along with its relative simplicity, made it an ideal candidate for modification to run on TxFS.

3.4 OpenLDAP

OpenLDAP [11] is a widely used and well supported implementation of the Lightweight Directory Access Protocol. This system is commonly used to provide authentication and organize authentication information about users. OpenLDAP can be configured to use a variety of different storage backends, from MDB, Berkeley DB, to simple ldif (LDAP Data Interchange Format) file backend [11]. In this paper, I focus on the ldif backend, as it is the easy-to-use backend for OpenLDAP, but has lower performance compared to the other backends [11]. The ldif backend uses plaintext ldif files and directory hierarchies to organize the LDAP data, instead of using a database construct. This backend uses atomic file and directory renaming in order to provide its durability and consistency guarantees. This can result in temporary files and directories being left on disk during the event of unexpected system shutdown, which requires lengthy manual cleanup by a system administrator. By modifying OpenLDAP, I hope to improve performance over the standard Linux implementation, as well

```
1  copy(file, tmpfile)
2  write(tmpfile, data)
3  rename(tmpfile, file)
```

Listing 1: *Simple technique for providing atomic updates to a file. By copying the file's contents into a temporary file, making edits to the temporary file, then atomically renaming the temporary file to reference the original file, the user either sees the old file or the completely updated file, effectively making the updates atomic.*

as eliminate the undesirable consequences of atomic renaming logic.

4 Background

I first describe the crash consistency problem, how journaling solves the problem, how transactions can be built on top of journaling to solve the problem, and how it is implemented both in unmodified SQLite and TxFS. I also discuss how durability and consistency is achieved in OpenLDAP.

4.1 Atomic Renaming

Listing 1 shows an example of atomic renaming, a simple method for providing atomic updates to a file [27]. This technique is simple to implement, but ends up being costly in terms of additional storage space and time (potentially taking 2× space on disk). It can also end up leaving temporary files on disk in the event of a crash, which requires manual recovery. This technique is used by the ldif backend to OpenLDAP for entry creation and entry modification, since entry files tend to be small on average (approximately 0.5 KB).

4.2 Journaling

A single file system update typically involves updating multiple structures on storage [1, 26]. An example would be the creation of a new directory. Not only does the parent directory

have to update its data, but the file system metadata structures (such are the inode bitmap, data bitmap, and superblock) must also be updated. In file systems without journaling support (such as `ext2`) [4], if the system crashes or power is lost in the middle of updating storage, the file system could be left in an inconsistent state.

A variety of techniques have been developed to ensure file system crash-consistency, such as `fsck` [6, 24], copy-on-write [16, 18, 36, 29], soft updates [30, 12], backpointer-based consistency [9], and journaling [26, 8].

Journaling is employed by a number of file systems such as Windows NTFS [33], SGI XFS [37], IBM JFS [3], `ext3` [39, 40, 4], and `ext4` [19]. At a high level, the technique works as follows: changes to the file system are grouped into "transactions", and first written to a special location on storage called the journal. Ensuring that the journal writes are safely persisted is called *committing*. After committing a transaction, the file system is then updated in-place in a process called *checkpointing*. If the system crashes before the journal is fully written, the interrupted operation will be aborted. If the system crashes after the journal is written, the file system re-reads the journal and performs checkpointing again. As a result, the file system updates in each transaction are atomically applied to the file system, and consistency is ensured.

Example — `ext4`: In `ext4` journaling, when a thread updates part of the file system, it becomes part of the currently running transaction. `ext4` does physical journaling, so the data and metadata blocks that are affected by the update are copied entirely into the journal. When any thread calls `fsync()`, or at the end of a user-defined time period (the default being 5 seconds), the running transaction is committed, and a new running transaction is started. Note that there is only one transaction running or being committed at any given point — concurrent transactions are not supported. The `ext4` journal transactions provide atomicity and durability, and ensure that a minimal amount of work is lost upon system crash (with data loss being bound by the last call to `fsync()` or the last automatic checkpoint).

`ext4` has three journal modes: `ordered` (default), `data`, and `writeback`. In `data` mode, all data and metadata is first copied into the journal and then checkpointed. `data` mode

provides the strongest consistency guarantees of `ext4` — files cannot end up with garbage data after a crash. Since copying data into the journal then updating in place is expensive (sequential writes achieve half the bandwidth [26]), by default only metadata (*e.g.* file inodes) are journaled in the default `ordered` mode. `ordered` mode ensures that the data associated with a transaction is persisted before the metadata. While `ordered` mode performs better than `data` mode, `ordered` mode allows files to contain garbage data after a crash. Finally, `writeback` mode has the highest performance, but does not provide any consistency guarantees about data — only metadata.

4.3 Database Transactions

Databases have long been implementing transactions as their method of ACID (Atomicity, Consistency, Isolation, Durability) properties [5]. A transaction is a method of performing an operation in a way that either happens completely (atomicity), obeys set rules or formats (consistency), does not interfere with other concurrent transactions (isolation), and persists on system power down or crash (durability) [14]. Databases implement transactions with a combination of `fsync()` calls, atomic operations like `rename()`, and the use of temporary files and locking.

4.3.1 Bugs

Unsurprisingly, application-specific solutions for persistent atomic updates (even in stable and widely-used applications like git) contain many bugs that lead to data loss and corruption [23]. Furthermore, such solutions are not portable across file systems, introducing subtle bugs that can linger for years [23].

4.3.2 Double Journaling

Another issue that occurs by implementing transactions at user level is double journaling [31]. This problem occurs because databases use a separate file for journaling. As shown in Figure 1, double journaling occurs when a database runs on top of a journaling file system (such

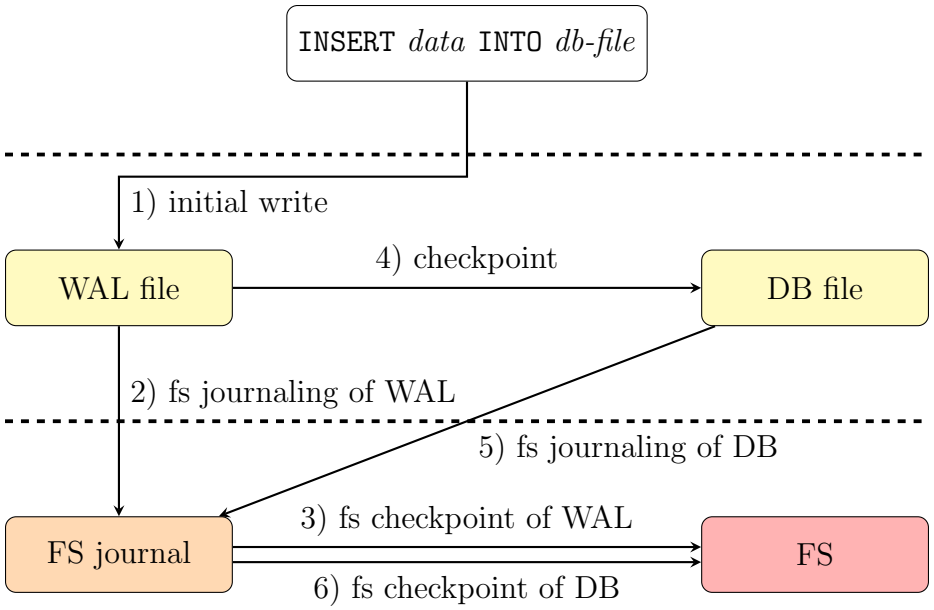

Figure 1: *The double journaling problem occurs when a database uses a separate journal file. When changes are written to the journal, they are also journaled by the file system. Subsequent checkpoints back to the database file are journaled as well. This results in a single change being journaled twice, hence the name double journaling. Note that these writes also occur for metadata changes as well as data changes. Only one generic set of arrows are shown here.*

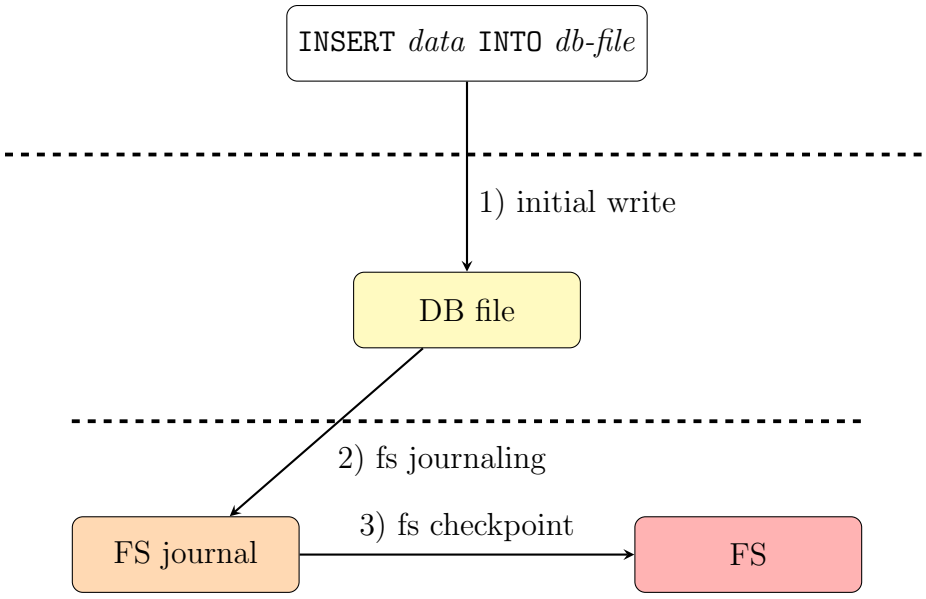

Figure 2: *How SQLite changes are journaled when using TxFS. TxFS eliminates double journaling.*

as `ext4`). Double journaling is a major cause of write amplification, which is the amplification of I/O due to journaling. Since the same data is journaled in different places, a single write can turn into multiple writes of the same data to different locations, which increases latency and reduces the lifespan of the underlying storage medium (and can lead to even more write amplification due to wear-leveling if the writes are being performed on an SSD). Figure 2 shows how TxFS eliminates double journaling. By using file system transactions in place of a user-level journal file, TxFS eliminates the double journaling problem, and significantly reduces the amount of I/O performed, which then results in higher throughput.

4.3.3 Concurrency Problems

In order to provide proper isolation at the user-level, applications implementing transactions have to use some form of locking to prevent concurrent modification. OpenLDAP uses thread-level reader/writer locks to block concurrent modifications, whereas SQLite mainly uses file-level locking. Figure 3 shows how locking is implemented in unaltered SQLite. Any number of database connections can hold a read lock on a particular database file.

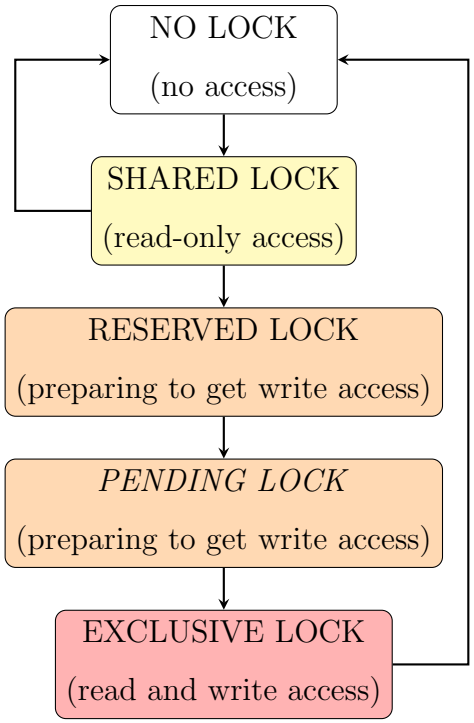

Figure 3: *Process of file locking as performed in SQLite. RESERVED and PENDING locks are used by a thread to claim priority on a write lock while waiting for current readers (threads with SHARED locks) to drop their locks. While a single process holds a RESERVED lock, other processes can still acquire new SHARED locks. While holding a PENDING lock (which is never acquired directly — only acquired through a request for an EXCLUSIVE lock), no new SHARED locks can be acquired. Once no other threads are holding locks on the database, a EXCLUSIVE lock is finally granted to the writer. Note that these locks are coarse-grain — these locks apply to the entire file, not just a particular page or a particular database table.*

```
open(/journal, O_CREATE)
write(/journal, "old")
fsync(/journal)
fsync(/)
write(/db, "new")
fsync(/db)
unlink(/journal)
fsync(/)
```

Listing 2: *System-call sequence in SQLite for updating database in crash-consistent manner.*

However, only one connection can hold a write lock. The write lock is exclusive, and no other connection can hold read locks while there is a write lock on the file. Additionally, SQLite uses coarse in-memory locking to protect entire database files, so individual tables within a file cannot be updated concurrently. Achieving multithreaded performance scalability for SQLite requires multiple database files. However, because SQLite has no concurrency control for separate processes, writes to a database file lock the file and another process must wait for the database file to be unlocked before the file can be opened for reading or writing. Multiple readers are allowed. When running with TxFS transactions, multiple processes can update the same SQLite database, and transactions that update disjoint regions of the file will succeed.

4.4 Journaling in SQLite

Current file systems do not expose a standard mechanism that applications can use to atomically update arbitrary amounts of data on storage. Many applications build ad-hoc solutions on top of system calls such as `fsync()` and `rename()` to atomically update state [23, 20]. Such solutions are complex, often involving multiple files and a long sequence of system calls. SQLite can maintain three files to atomically update a single byte in a database: a rollback-journal file, a database file, and possibly a master journal file [35, 20]. SQLite performs a

sequence of `write()`, `fsync()`, and `unlink()` calls on these files to ensure crash consistency.

`DELETE` mode (Rollback mode): Listing 2 illustrates the system calls issued by SQLite to update a single row in the database in its default crash-consistency mode, which is known as `DELETE` mode. First, SQLite creates a `journal`, and the old contents are copied to it. It flushes the `journal` to storage with `fsync()`. To ensure that the new directory entry pointing to `journal` is also persisted, it flushes the root directory with `fsync()`. It then writes database file `db` and flushes it to storage. It unlinks the `journal` file and makes the deletion persistent with another `fsync()` on the root directory. Truncate mode is another similar mode, however the `journal` file is truncated to 0 length rather than being unlinked — this tends to be faster on most systems.

`WAL` mode: `WAL` mode is the newest journaling mode offered by SQLite and is the most performant under most circumstances [35]. The `WAL` approach inverts traditional journaling. In `WAL` mode, all of the original content is stored in the database file and any changes are added to a separate journal file. A transaction commit occurs when a special record indicating a commit is appended to the `WAL` journal — commits can therefore occur without writing to the original database. This allows readers to continue operating from the original unaltered database while changes are simultaneously being committed into the `WAL` journal.

4.5 TxFS

TxFS is a file system that was designed to offer primitives that allow applications to easily and efficiently achieve crash consistency. While other systems have provided applications with transactions for crash consistency, they have required that either the whole operating system be modified [41, 15, 25], use specialized hardware [20, 10, 26, 32], induce significant performance degradation [13, 22, 21, 42, 34] or create a large amount of work on the part of the developer [28]. TxFS offers a simple system call interface for handling file system transactions which does not require major modifications to an application's source code in order for it to adopt TxFS transactions, as will be shown in section 6.

```
1  fs_tx_begin()
2  write(/db, "new")
3  fs_tx_commit()
4  fsync(/db)
```

Listing 3: *Using TxFS Transactions to update a database in a crash-consistent manner. The call to* `fsync()` *ensures the TxFS transaction is durable.*

4.5.1 TxFS API

TxFS provides developers with three system calls: `fs_tx_begin()`, which begins a transaction; `fs_tx_end()`, which ends a transaction and attempts to commit it; and `fs_tx_abort()`, which discards all file system updates done as part of 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 none of the file system updates, or all of them. `fs_tx_end()` returns an error code indicating whether the transaction was committed successfully (the commit may fail for various reasons, such as the transaction being too big, or the file system has run out of space); the application can then choose to retry the transaction. Application-level transactions provide isolation at the level of repeatable reads [2]. Nested application-level transactions are flattened into a single transaction.

Listing 3 shows how application-level transactions can be used to perform the same SQLite database update as shown in Listing 2. This code is much simpler, and allows application writers to avoid bugs, simplify internal transactional APIs, and eliminate performance bottlenecks like file locking.

5 Design

The main goals of this project were to port OpenLDAP and SQLite to using TxFS transactions. This required two main modifications to the standard logic of these existing systems: the removal of the current method of concurrency control and means of providing ACID

```
retry_loop:
  fs_tx_begin()
  write(file, data)
  if (CONFLICT)
    fs_tx_abort()
    goto retry_loop
  fs_tx_end()
  if (CONFLICT) goto retry_loop
```

Listing 4: *Basic sequence for performing a write transaction using TxFS.*

properties, and the addition of transactional abort-and-retry logic in its place.

5.1 Removal of File and Thread Locking

SQLite maintains coarse-grained file locks for read and write transactions, and only allows one writer at a time in order to maintain file consistency. Additionally, OpenLDAP maintains reader-writer locks in order to prevent concurrent modifications, at the cost of parallelism. TxFS provides isolation by only allowing one transaction to modify a particular block at a time, with all other competing transactions receiving a `ECONFLICT` error. By allowing TxFS to handle isolation, there is no need to lock files that are modified transactionally. Because of this, SQLite can be modified to eliminate all file locking, and OpenLDAP can have all of its locking removed as well. This modification allows for higher levels of concurrency, which improves performance for multithreaded applications.

5.2 Transactional Aborts and Retries

Listing 4 shows the basic flow for a thread in TxFS that is attempting to commit a write transaction. In a multithreaded environment with a small working set of files, write conflicts are a very real possibility. Therefore, threads must check to see if their transactions are successful — on conflict, TxFS will set `errno` to `ECONFLICT`. If the process still wants to

attempt its update, it has to start a new transaction and replay all file operations. This abort-and-retry style of logic is implemented in many places throughout both of these systems under TxFS.

6 Implementation

In this section, I describe the modifications performed on both SQLite and OpenLDAP, and how the modified versions of these two applications end up differing from the standard implementations.

6.1 SQLite

The TxFS project team selected SQLite (version 3.12.1), which was the current version when SQLite was starting to be used as a benchmark for TxFS (April 2016). We introduced a new journaling mode, `FS`, similar to SQLite's `WAL` mode, to implement TxFS transactions in SQLite. SQLite requires two guarantees from `FS` mode: atomic commits and repeatable reads, both of which are provided by TxFS transactions. These modifications amounted to approximately 600 LOC.

The majority of SQLite changes were made to the `pager`, the interface responsible for reading and writing database pages to disk. The other changes involved correctly propagating error messages if the TxFS transaction failed (*e.g.* if the transaction was too big) and plumbing changes to make sure that `FS` was a valid journaling mode for SQLite.

One major goal of the project was to export the user-level implementation of transactions as done in SQLite with calls to `fs_tx_begin()` and `fs_tx_end()`. The modifications made to SQLite can be grouped into two main categories: removal of file locking and removal of `fsync()` and `fdatasync()` calls. Additionally, using SQLite in `FS` mode requires slight modifications to existing applications that use large multi-line SQLite transactions.

6.1.1 File Locking Modifications

The nature of TxFS transactions is that locking is unnecessary — any number of threads can attempt to write a file, and as long as the writes are not conflicting (*e.g.* are to different parts of the file), no thread is any the wiser. Should there be a conflict during an operation, the SQLite interface will return a `SQLITE_CONFLICT` error code to the user, which will indicate that the current thread of execution should abort the current transaction (make a `fs_tx_abort()` call) and retry/perform a different operation.

SQLite's `FS` journaling mode does not acquire file locks — it depends upon TxFS to enforce isolation. This allows the implementation to be simpler and increases performance. This serves as an example where applications become simpler by depending upon the properties provided by system-level transactions. This modification was also very simple to make, as only a few lines of code had to be disabled in `FS` mode in order to disable all file locking performed by SQLite.

6.1.2 Removal of `fsync()` and `fdatasync()` Calls

Unaltered SQLite has to make many calls to `fsync()` and `fdatasync()` while committing a transaction in order for the transaction to provide durability. However, these sync calls are redundant during the middle of a TxFS transaction, and can hurt performance — only a call to `fsync()` after a `fs_tx_end()` will help transactional durability. SQLite databases running in `FS` mode disable SQLite's usual calls to `fsync()` and `fdatasync()`— this will reduce the number of file operations that occur during a transaction, which will increase throughput.

6.1.3 Changes to SQLite Applications

SQLite handles transactions in two different modes: autocommit mode, used for single-line transactions, and non-autocommit mode, used for multi-line transactions where the user makes explicit calls to "BEGIN TRANSACTION;" and "END TRANSACTION;". When SQLite is in `FS` mode and is in autocommit mode, it automatically triggers `fs_tx_end()`. Any error during the SQLite transaction results in triggering `fs_tx_abort()` and immediate

```
1   rc = sqlite3_exec(SQL query)
2   if (rc != SQLITE_OK) return FAILURE
3   return SUCCESS
```

Listing 5: *Example workflow for unaltered SQLite*

```
1   while (success == 0)
2     fs_tx_begin()
3     rc = sqlite3_exec(SQL query)
4     if (rc == SQLITE_CONFLICT)
5       fs_tx_abort()
6       continue
7     success = fs_tx_end()
8   if (rc != SQLITE_OK) return FAILURE
9   return SUCCESS
```

Listing 6: *Example workflow for TxFS*

retry. If SQLite is not in autocommit mode while in `FS` mode, then SQLite will not call `fs_tx_begin()` and `fs_tx_end()` automatically, and the user will be responsible for handling the abort-retry logic. This difference in SQLite API usage is shown in Listings 5 and 6. Running without autocommit mode on is only slightly more cumbersome to the user, but allows the user more control on the contents of a transaction, and allows users to create transactions that cross multiple databases or even across abstractions (*e.g.* a transaction involving raw files and a SQLite database).

6.1.4 Consequences of SQLite Modification

Porting SQLite to TxFS prompted the modification of TxFS to allow a limited set of operations to fail without aborting the transaction. In the original SQLite code it was common to call `unlink()` on files without first checking to see if the file existed. While an operation like this has no negative side effects on a normal file system, when running inside a TxFS transaction an `unlink()` failure aborts the transaction. Many other codebases contain similar code because on traditional file systems such code is essentially turned into a harmless no-op. The team developing TxFS believes that TxFS will have to adapted in similar ways for other legacy software.

6.2 OpenLDAP

I modified OpenLDAP version 2.4.44, the current version as of April 2017. The number of changes to the ldif backend totaled to approximately 500 LOC, with an additional 250 LOC for caching search results. This caching logic was adapted from a similar port of OpenLDAP as reported in the TxOS study [25].

6.2.1 Wrapping Read and Write Accesses

In the standard implementation of OpenLDAP, read and write accesses are protected by a read-writer thread lock, in order to protect against concurrent access and modification. However, in the case that writes are happening to different locations (*e.g.* two different entries, which are stored in two different files in the ldif backend), this limits concurrency. I altered the ldif backend to wrap transactions around the main backend calls (the `ldif_back_*` and `ldif_tool_entry_*` family of functions), and replaced the reader-writer locks with a TxFS abort-and-retry loop.

6.2.2 Caching Search Results

A big part of the functionality of OpenLDAP is to search for records that have been stored in the system. Search operations emit search results to the client process while the search is being conducted, so if TxFS abort-and-retry loops had been naively added, the transaction retry would have caused some of the search results to be sent to the client multiple times. Therefore, in the TxFS modified version, search results are cached and sent them to the client only after completing the entire search operation.

7 Evaluation

The following evaluations were performed by myself and the graduate students who were working on the TxFS project. This section discusses how the TxFS implementations of SQLite and OpenLDAP compare to the Linux implementations, and also evaluates why

	Performance (Ops/s)		I/O (MB)		Sync/tx	
Journal mode	Insert	Update	Insert	Update	Insert	Update
Rollback (default)	53899.7	28001	1977	3946	4	10
Truncate	53496.3 (0.99×)	28907 (1.03×)	1976	3944	4	10
WAL	39774.5 (0.74×)	34551.9 (1.23×)	3944	3928	3	3
TxFS	51398.5 (0.95×)	36695.8 (1.31×)	1970	3916	1	1
Rollback with TxFS	52148.1 (0.97×)	31924.2 (1.14×)	1970	3915	1	1
No journal (**unsafe**)	54888.4 (1.02×)	50608 (1.81×)	1966	1956	1	1

Table 1: *The table compares operations per second (larger is better) and total amount of I/O for SQLite executing 1.5M 1KB operations grouping 10K operations in a transaction using different journaling modes (including TxFS). The database is prepopulated with 15M rows. All experiments use SQLite's synchronous mode (its default), which ensures that unmodified SQLite's transactions are durable.*

TxFS performs better or worse in our benchmarks.

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

7.1 SQLite

Single-threaded SQLite: Table 1 shows that TxFS is the best performing option for SQLite updates. Data is 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 I/O totals as part of our validation that TxFS correctly writes all data in a crash consistent

Journal mode	Time (s)	Stall (s)	Read (MB)	Write (MB)
Rollback (default)	17.07	0.24	6.73	404
Truncate	16.99	0.27	6.70	409
WAL	10.67	0.05	6.74	202
TxFS	5.20 (3.3×)	1.24	6.71	122
No journal (**unsafe**)	10.67	0.22	6.73	406

Table 2: *Six threads read and update twelve database files, picking a random file, reading a random row, hashing it and writing the hash to another randomly chosen database. Each database is prepopulated with 100,000 1KB entries. Each database file is 131 MB.*

manner. Several choices for SQLite logging mode, including TxFS, result in similar levels of I/O that resemble the no-journal lower bound. `WAL` mode does write more data for the insert workload, which harms its performance. Note that TxFS does not suffer `WAL`'s performance shortfall on insert, and it surpasses its performance on update, making it a better alternative. Although the file system journal is similar to a `WAL` log, TxFS does not generate redundant I/O on insert because of its selective data journaling.

We ran similar experiments with small updates (16 bytes), where we found 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 improved 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. TxFS performs half its I/O to the journal, which is written sequentially. The remaining I/O is done asynchronously via a periodic file system checkpoint that writes the journaled blocks to in-place files. TxFS does not suffer from the double journaling problem [31]. 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.

Multithreaded SQLite: The results of the multithreaded SQLite benchmark are presented in Table 2. We populated twelve databases, each with a single table with 100,000 1KB rows. We ran six threads (one for each physical core of the machine), which pick a random database file and a random row and then read the row, compute the SHA-256 hash and insert the result to a second randomly chosen database. The workload does small appending writes to the database files, with an estimated amount of 424 KB data written in each run. We then compared the final state of the database for all cases to verify that all data is properly written. For TxFS, stall time is time spent backing off after a transactional conflict. After a conflict, one process waits for a random time before restarting in order to avoid excessive conflicts (the retry interval is capped at 0.05s). For all other SQLite journaling modes, stall time refers to the time spent waiting to gain exclusive access to the file.

TxFS outperforms the default configuration by 3.3× and the best alternative (`WAL` mode) by 2.1×. All journaling modes read about the same amount of data. They all have heavy write amplification caused by small and fragmented insertions (the data size is only 424 KB). Both `WAL` mode and TxFS have a much smaller write amplification compared to the other modes. This is because the `WAL` log and the file system journal are all write-ahead-style logs, which can batch fragmented writes into the same data block.

Because most SQLite journaling modes gain exclusive access to the database file for writes, they spend about the same amount of time waiting for access (about 0.24s). `WAL` mode spends significantly less time waiting, because it only needs to grab the database read-write lock while writing to the `WAL` log. The data checkpoint from the `WAL` log to the database file happens asynchronously, without blocking other writers. TxFS spends more time backing off than the exclusive access wait time. The threads are busy waiting for the database files, which occupies a CPU core while doing no work and therefore minimizes wait latency. Backoff frees up a CPU, but wastes more time. However, the write concurrency enabled by TxFS transactions more than compensates for backoff time.

TxFS mode outperforms no-journal mode because this mode still opens database files

	1-thread		6-thread	
	Linux	TxFS	Linux	TxFS
AddJob	3517	2305 (0.66×)	6703	9836 (1.47×)
SearchJob	19	23 (1.21×)	97	107 (1.10×)
ModifyJob	2932	2445 (0.83×)	5101	9722 (1.90×)

Table 3: *Operations per second for OpenLDAP workloads of different operation types. The database contains 1,000 entries, with each entry being approximately 0.5 KB in size. SearchJob runs 1,000 random searches, AddJob adds 1,000 new entries, and ModifyJob modifies each of the 1,000 entries in a random way.*

	1-AddJob, 1-SearchJob		1-AddJob, 5-SearchJob	
	Linux	TxFS	Linux	TxFS
Average Throughput	375	1479 (3.94×)	808	989 (1.22×)

Table 4: *Operations per second for the MultiJob OpenLDAP benchmark.*

with an exclusive lock, preventing concurrency. While "real" databases like Oracle and MySQL have good concurrent scalability, SQLite has found it difficult to achieve due to file system limitations. TxFS removes these limitations and makes it easy to scale performance for access to non-conflicting data. Additionally, TxFS has a smaller amount of write amplification.

7.2 OpenLDAP

Table 3 compares the unaltered version (Linux) and the version that uses TxFS transactions (TxFS). We gathered data using a modified a version of `lb`[1], which is an open-source tool for benchmarking OpenLDAP. Search operations are read-heavy, but they show moderate speedups (21% for single thread and 10% for six threads). The performance of TxFS for write-heavy workloads trails for the single threaded case where the overhead of system calls

[1] `https://github.com/hamano/lb`

to begin and end transactions, along with the small amount of work in the transactions, makes it impossible to amortize the overheads. The multithreaded scalability shows the appeal of TxFS transactions, which increase performance by 47% and 90% for adding users and modifying their records. The Linux implementation of OpenLDAP uses reader-writer locks which limit concurrency. There are cases where concurrent writes are safe (*e.g.* to two separate entries in two separate files), and TxFS allows these writes to occur concurrently, giving TxFS higher multithreaded write performance.

Independent from performance, the TxFS port of OpenLDAP has crash consistency advantages. When we killed server processes during a series of ModifyJobs, 96% of the crashes resulted in temporary files being left in the base LDAP directory. These files must be deleted by hand, which is time consuming and error-prone. On average, the number of temporary files left behind was equal to 0.5% of the size of the modification job in entries (*e.g.* for an operation of adding 10,000 entries, we found around 50 temporary files). TxFS eliminates this problem completely; after a crash there are no orphaned temporary files.

Table 4 shows a mix of OpenLDAP activities with a single AddJob running concurrently with either 1 or 5 SearchJobs. The workload models the common case use of OpenLDAP where reads dominate writes. The database contained 100 entries, with each entry being approximately 0.5 kB in size. The single-threaded version runs 3.9× faster with TxFS transactions and the 6 core version runs 22% faster. The TxFS version has higher throughput, but transaction abort-and-retry penalizes adds, since searches are slow and read from the directory, and since adds modify the directory by adding a new entry, they get aborted multiple times waiting on the same search, hindering AddJob throughput. The Linux implementation uses a read-writer lock for coordination, which is more fair, but has lower overall throughput.

8 Related Work

TxOS [25] is operating system that was also designed to provide transactions. Unlike TxFS, the approach taken adds significant complexity to the kernel; a large number of kernel data

structures will have to rewritten to support transactions. Developing a correct transactional database is extremely complicated, and retrofitting an existing kernel to do so is even more so.

SQLite has previously been ported to suit many projects. One similar to this project is X-FTL [17]. X-FTL modifies SQLite to use transactional semantics, as provided by the flash translation layer, in order to more efficiently use flash memory and take advantage of the firmware of the FTL, rather than trying to use user-level transaction implementations. The modifications applied to SQLite in the X-FTL study are similar to the modifications implemented in the course of this project, since both bypass the standard implementation of SQLite transactions and instead use the transactional interface of the underlying system.

9 Conclusion

In this paper I demonstrated how TxFS's simple interface is easy to use and simple for existing systems to adopt by porting both SQLite and OpenLDAP to use TxFS without having to rewrite major parts of their existing codebase. I showed how both of these systems benefit from using TxFS instead of traditional techniques for atomic updating. TxFS signficantly increases the operational throughput of these systems while simulatenously simplifying parts of their internal structure. These improvements make a good case for the benefit and future use of transactional file systems, and for their reconsideration for mainstream adoption.

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[2] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *ACM SIGMOD Record*, volume 24, pages 1–10. ACM, 1995.

[3] Steve Best. JFS Overview. `http://jfs.sourceforge.net/project/pub/jfs.pdf`, 2000.

[4] Daniel P. (Daniel Pierre) Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly, Beijing;Sebastopol, CA;, 3rd;3; edition, 2006;2005;2007;.

[5] Michael L. Brodie and Dzenan Ridjanovic. *On the Design and Specification of Database Transactions*. Springer New York, New York, NY, 1984.

[6] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[7] Min Chen, Shiwen Mao, and Yunhao Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, 2014.

[8] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Farmington, PA, November 2013.

[9] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, California, February 2012.

[10] 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.

[11] The OpenLDAP Foundation. Openldap. `https://www.openldap.org/`.

[12] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[13] 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.

[14] Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the 7th International Conference on Very Large Data Bases (VLDB 7)*, Cannes, France, September 1981.

[15] Rober Haskin, Yoni Malachi, and Gregory Chan. Recovery management in quicksilver. *ACM Transactions on Computer Systems (TOCS)*, 6(1):82–108, 1988.

[16] 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.

[17] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Gi-Hwan Oh, and Changwoo Min. X-ftl: Transactional ftl for sqlite databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 97–108, New York, NY, USA, 2013. ACM.

[18] Chris Mason. The Btrfs Filesystem. `oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf`, September 2007.

[19] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Alex Tomas Andreas Dilge and, and Laurent Vivier. The New Ext4 filesystem: Current Status and Future Plans. In *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, July 2007.

[20] 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.

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

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

[23] 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.

[24] Dave Poirier. Second Extended File System. http://www.nongnu.org/ext2-doc, August 2002.

[25] 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.

[26] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, April 2005.

[27] M. (.) Raynal. *Concurrent programming: algorithms, principles, and foundations*. Springer-Verlag, New York;Heidelberg;, 2013.

[28] 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.

[29] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '93)*, pages 307–326, San Diego, California, January 1993.

[30] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 71–84, San Diego, California, June 2000.

[31] 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, Santa Clara, CA, 2014. USENIX.

[32] 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.

[33] David A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.

[34] 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.

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

[36] Sun Microsystems. ZFS: The last word in file systems. `www.sun.com/2004-0914/feature/`, 2006.

[37] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[38] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Joo-young Hwang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Towards Efficient, Portable Application-Level Consistency. In

Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep '13), Farmington, PA, November 2013.

[39] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[40] Stephen C. Tweedie. EXT3, Journaling File System. `olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`, July 2000.

[41] Matthew J Weinstein, Thomas W Page Jr, Brian K Livezey, and Gerald J Popek. Transactions and synchronization in a distributed operating system. In *ACM SIGOPS Operating Systems Review*, volume 19, pages 115–126. ACM, 1985.

[42] 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.