From Crash Consistency to Transactions

Yige Hu Youngjin Kwon Vijay Chidambaram Emmett Witchel
The University of Texas at Austin

Abstract

Modern applications use multiple storage abstractions such as the file system, key-value stores, and embedded databases such as SQLite. Maintaining consistency of data spread across multiple abstractions is complex and error-prone. Applications are forced to copy data unnecessarily and use long sequences of system calls to update state in a consistent manner. Not only does this create implementation complexity, it also introduces potential performance problems from redundant IO and `fsync()` calls, which fragment disk writes into small, random IOs. In this paper, we propose that the operating system should provide transactions across multiple storage abstractions; we can build such transactions with low development cost by taking advantage of a well-tested piece of software: the file-system journal. We present the design of our cross-abstraction transactions and some preliminary results, showing such transactions can increase performance by 31% in certain cases.

1 Introduction

The interface to the file system is a defining feature of operating systems. Early operating systems like the IBM 360 had indexed files to support database-like access to file data; early MacOS supported extended file metadata using resource forks; and UNIX beat all competitors by treating files as a simple sequence of bytes accessed via `read()`, `write()`, and `seek()` system calls.

Unfortunately, modern applications have certain requirements that do not fit well with the current interface: for example, storing structured data, and performing rich, complex queries on the data. Applications have turned to new abstractions such as embedded databases (*e.g.,* SQLite [28]) and key-value stores (*e.g.,* LevelDB [7]) to meet their needs. Unfortunately, since these abstractions operate on top of the file system, they have to work with or around the simplistic API.

Such attempts have led to either poor performance or loss of correctness in the face of a crash [18]. For example, databases and key-value stores often maintain an application-level log. The metadata for those logs is written twice: once to the kernel's file-system journal and once in-place in the file system (this is called the double journaling problem [11]). These additional IOs along with the `sync()` system call overheads severely degrade throughput on modern SSDs and the situation will be even worse for faster devices like non-volatile memory. As storage devices approach memory latency, software overheads such as the processing required for system calls begin to dominate latency (e.g., an `fsync()` in the Redis key-value store takes 137*us* [17]).

Even if a particular abstraction works well with the file system, modern applications often store *across* different abstractions. For example, the Android Mail client stores email messages in SQLite, attachments as files, and the backup of mailbox information in a key-value store [26]. For such applications, a single logical update from the user's point of view may require writes across different abstractions that all need to be performed atomically. Accomplishing this with user-level logs results in poor performance and added complexity (see §2).

We believe that file-system transactions can fix these performance and correctness problems while simplifying user-level code. Transactions have been attempted in file systems before, but have failed due to either low performance or hard-to-use interfaces (see §5). We believe both of these problems can be surmounted, and that they are not fundamental features of transactional systems. Another challenge in building transactional interfaces is the high implementation cost. We propose to build ACID transactions on top of the file-system journal, a novel approach that will greatly reduce implementation complexity by leveraging well-tested, mature kernel code.

We describe the design of **T2FS**, a file system which leverages the failure-atomic file-system journal to provide ACID transactions to user-space applications (§3). T2FS uses lazy version management and eager conflict detection, maintaining in-kernel logs to buffer transactional updates. T2FS will be the first system to provide transactions across different storage abstractions, enabling applications such as MediaWiki and Android Mail to transactionally update state distributed among different abstractions such as file systems, embedded databases, and key-value stores (§4).

T2FS will reduce application complexity and increase performance. Application performance will increase due to several factors: reduced IO (e.g., metadata is written only once and editors would not have to make

```
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.

copies of edited files to ensure logical updates), batching of updates in transactions (especially benefiting small writes [19]), fewer system calls (e.g., fewer calls to `fsync()`), and fine-grained conflict detection.

2 How Applications Update State Today

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

In this section, we show how complex it is to implement seemingly simple protocols for consistent updates to storage. There are many details that often get 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.

Currently, applications use the `fsync()` system call to order updates to storage [4]; 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 [18].

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

Atomic Rename. Protocol (a) shows how a file can be updated via atomic rename. 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 new pointer to log file) are persisted. The application then updates the original file, and persists the original file (since it already existed, the parent directory has not changed). Finally, the log is unlinked, and the parent directory is persisted.

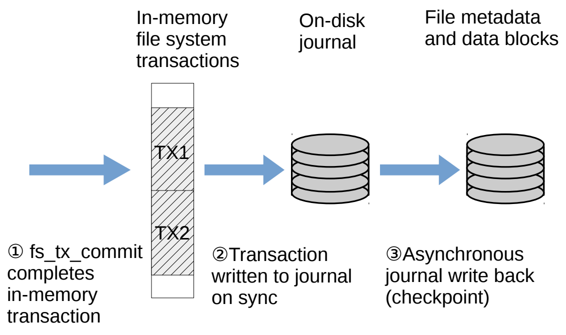
The situation becomes more complex when applications use more than one storage abstraction. Protocol (c) illustrates how the Android Mail application adds a new email with an attachment. The attachment is stored on the file system, while the email message (along with metadata) is stored in the database. 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, 18]. SQLite has been configured to use write-ahead-logging to then atomically update the database; it follows a protocol similar to Protocol (b).

Thus we see how complex the protocols are, and that they lead to low performance (e.g., Android Mail uses *six* `fsync()` calls to persist a single email with an attachment). System support for transactions over multiple storage abstractions is required to maintain high performance for these applications for current and emerging high-performance storage devices.

3 T2FS transactions

T2FS builds upon the atomic update mechanism of the file system (such as journaling [21]) to provide ACID transactions to applications. Although most file systems contain a well-tested, mature mechanism to atomically update multiple blocks on storage, this mechanism is not exposed to applications; rather it is used internally to guarantee the integrity of updates to file-system metadata.

We propose building on this mechanism to reduce the implementation complexity in building a transactional system. We use the running example of ext4 and its jour-



In-memory file system transactions
On-disk journal
File metadata and data blocks
TX1
TX2
① fs_tx_commit completes in-memory transaction
②Transaction written to journal on sync
③Asynchronous journal write back (checkpoint)

Figure 2: T2FS relies on ext4's own journal for atomic updates. At commit time, the transaction-local operations are made global, and are recorded into the global in-memory file system transaction. The following `sync()` then forces the writes of the transaction into the journal. Finally, the asynchronous journal checkpoint updates the in-place file data and metadata blocks.

naling mechanism, although similar mechanisms such as copy-on-write [10] in the ZFS file system [29] would also work. ext4 guarantees that all the updates within a journal transaction are applied to the file system atomically; T2FS guarantees that an application-level transaction is contained within a single journal transaction, thus ensuring it is atomically applied to the file system. While the file system by default writes only metadata to the journal, T2FS also journals data blocks. Journaling data can significantly degrade performance; to minimize this, T2FS employs selective data journaling [4], only journaling data blocks that were already allocated (*i.e.,* data blocks that are being updated), and avoiding journaling newly allocated data blocks. This optimization does not harm crash consistency because ext4's ordered mode forces data to be sync-ed before its corresponding metadata.

API. T2FS provides developers with three system call interfaces: `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 done as part of the preceding 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 transaction updates, or all of them. `fs_tx_commit()` returns an error code indicating whether the transaction was committed successfully (the commit may fail for various reasons, such as a conflict, the transaction is too big, or the file system has run out of space); the application can then choose to retry the transaction.

T2FS provides isolation for its transactions; transactional updates are not visible to other threads until commit. T2FS isolates file-system updates only, the application is responsible for synchronizing access to its own data structures.

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 makes it possible to bring the power of file-system transactions selectively to existing applications piecemeal and with little porting effort. This interface is easy for programmers to use and makes it simple to incrementally deploy file system transactions into existing applications. In contrast, some high-profile transactional file systems (namely Window's TxF [23]) had far more complex, difficult to use interfaces.

Figure 2 shows how T2FS transactions proceed from process-local in-memory modifications to durable, in-place file system updates. The kernel buffers a process' modifications to files, eventually making them globally visible when the transaction commits. When the transaction is made durable, it is first written to the file system journal. The journal contents are eventually, asynchronously written back to the file system (a process called check pointing the journal). The checkpoint latency does not affect the latency of committing a durable transaction.

Isolation. While ext4 journaling provides atomic, durable updates, it does not provide isolation. Thus the central challenge of T2FS is adding isolation. To support isolation, T2FS maintains an in-memory log to track local modifications to the kernel data structures. T2FS makes local copies of dentries and inodes (the in-memory data structures for directories and files) for a user transaction at lookup time. It does copy-on-write for pages and then inserts them into a transaction-local radix tree. File-system code that can potentially change global on-disk state is split into two parts: (1) that only changes the in-memory status of the data structures, which should be invoked immediately on the local copies when a system call is performed. This enables later operations inside the same transaction to read earlier writes; (2) one that changes the on-disk state, which should not be performed until commit time. T2FS monitors file access and guarantees that no other concurrent T2FS transactions can read the data from an uncommitted one and that any transaction is aborted if its data is written.

If `fs_tx_abort()` is called on a T2FS transaction, it simply aborts the recorded local copies of in-memory data structures, and recovers some potential side effects on the global data structures (which cannot be read by other transactions due to conflict detection). If `fs_tx_commit()` is called, T2FS first checks if there is any confict or if an error happened during the transaction. If so, it aborts the transaction. Otherwise, it starts the commit protocol. The protocol traverses all recorded data structures modified inside the local transaction, and performs two phase locking. It makes the transaction updates durable on the persistent journal (newly allocated blocks are made durable on the file system), then copies back the local dentries, inodes and pages to their global counterparts. We currently rely on application writers to call a flavor of the `sync()` system call to explicitly make transactions durable on disk. This adds flexibility to users to trade off between durability and performance. Future work will include a flag to `fs_tx_commit()` that achieves durability synchronously.

Conflict detection. We believe conflicts will be infrequent for the applications we target, therefore we keep conflict detection and resolution simple. Prior work has shown that multiple threads reading and writing the same file at the same time is rare [18]. Accordingly, T2FS uses a lightweight isolation mechanism that provides isolation at the level of repeatable reads [3]. T2FS stores transaction-local copies of written pages, and tracks pages that are read by the transaction.

T2FS performs eager conflict detection to provide isolation with respect to non-transactional writes: a non-transactional write will abort any active transaction that has read or written the same data. Nested T2FS transactions are flattened into a single transaction. We detect conflicts for file-system data structures, e.g. dentries, inodes and pages. We use fine-grained per-page conflict resolution to resolve page-level read and write conflicts. If two threads are modifying only data blocks from the same file, writes to different data blocks would not cause any conflict. By avoiding transactions being aborted by such kind of false conflicts, we expect better parallelism in applications using T2FS.

Limitations. Our current design has two main limitations. First, the maximum size of a T2FS transaction is limited by the size of the journal (similar to size limitations of transactional memory systems [12]). 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 Integrating T2FS with Applications

To demonstrate the power and usefulness of T2FS, we plan to modify several applications to use T2FS transactions. We describe how we modified SQLite, and how we plan to modify Cloudstone, MediaWiki, and a couple of version control systems.

SQLite. We modified SQLite to use T2FS transactions. Figure 3 illustrates the flow of updates when SQLite uses T2FS transactions. Data and metadata are first written safely to the journal, and then checkpointed in-place into

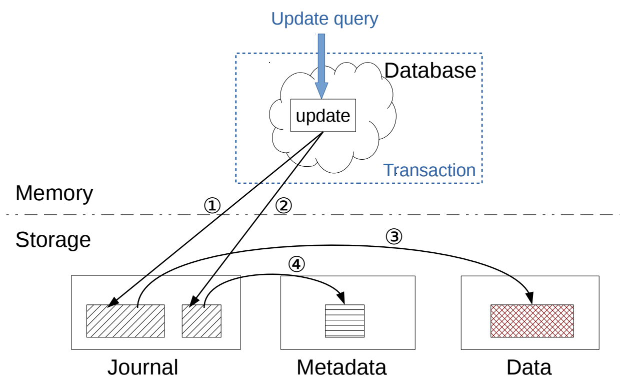


Figure 3: T2FS guarantees on-disk consistency for transactions by utilizing Ext4's own journal. The figure illustrates the memory and storage updates during a transaction with an update to a database row. Writes to persistent storage are listed as followed. (1) Data journal for the update; (2) Metadata journal for the update; (3) Asynchronous write-back for the database data; (4) Asynchronous write-back for the metadata.

the file system. Note that the metadata is written into the file system exactly once. With Write-ahead-logging (WAL) mode in SQLite, there would be 2× the metadata updates: once to the log, and once to the actual database file. When SQLite is run in its safest configuration, there are more metadata updates because it must update the parent directory whenever log files are created or deleted [28]. Furthermore, SQLite with T2FS neatly avoids having to clean up log files in the event of a crash: user-level sees only the database file, not messy in-flight data.

Preliminary Evaluation. We run SQLite modified to use T2FS transactions on a 6 core Intel Xeon E5-2620 CPU, with 8 GB DDR3 RAM, 250 GB Samsung SSD 850 and 512 GB Samsung SSD 850. We use Ubuntu 16.04 LTS and Linux kernel 3.18.22.

Our prototype can run limited-size single-threaded benchmarks. Table 1 reports a sequence of insert or update operations for a single database table grouped into a transaction. Performance is reported as operations per second (so larger is better) and compared with different SQLite journaling modes (where the default is rollback journaling).

Using T2FS is faster than any existing SQLite journaling scheme for updating, and it is competitive for inserting (with no mode dominating insert performance). The updating workload is 31% faster than the default. No journaling is an unsafe mode that would make SQLite vulnerable to corruption on a crash, but represents the best possible performance for the workload. Several choices for SQLite logging mode, including T2FS, re-

	Performance (Ops/s)		IO (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
T2FS	51398.5 (0.95×)	36695.8 (1.31×)	1970	3916	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 IO for SQLite executing 1.5M 1KB operations grouping 10K operations in a transaction using different journaling modes (including T2FS). The database is prepopulated with 15M rows. All experiments use SQLite's NORMAL synchronization mode (the most widely used mode by applications).

sult in similar levels of IO that resemble the no journal lower bound. Write-ahead logging mode (WAL) does write more data for the insert workload, which harms its performance. Note that T2FS 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 shares similarity with a WAL log, T2FS does not generate redundant IO on insert because of its selective data journaling. Newly allocated blocks in a T2FS transaction are not journaled.

T2FS's improved performance for the update workload is due to several factors. T2FS 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 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. T2FS does not suffer from the double journaling problem [24].

These preliminary measurements convince us that T2FS is a promising research direction. We anticipate higher performance gains for multi-threaded workloads.

PHP and MediaWiki. MediaWiki [2] is an open source Wiki written in PHP and originally used for Wikipedia. It stores binary files directly in a file system and metadata (such as file size and location) in a database. High-level actions such as uploading an image results in updates to both the file system and database; these updates must be done atomically to ensure the consistency of MediaWiki.

We plan to augment the PHP API with transactional calls (such as `tx_begin()` and `tx_end()`) and use these calls to transactionally update MediaWiki state. Using transactions reduces the number of `fsync()` calls and efficiently batches IO. Transactions also reduce complexity of supporting rollback of operations such as file updates, even in the presence of crashes.

Cloudstone. The Cloudstone WEB 2.0 benchmark simulates a social event website. Cloudstone models a LAMP stack, consisting of a web server (nginx), PHP, and MySQL. Cloudstone uses files (e.g., for storing images) and a database for storing information about events and people. Cloudstone provides a performance framework to evaluate T2FS in the context of a realistic web application. Making consistent updates to events requires transactions that update files and a database.

Version Control Systems. Git and Mercurial are widely-used version control systems that would benefit from file-system transactions. To enable high performance, git does not order its operations via `fsync()` [18] leaving it vulnerable to garbage files and outright data corruption on a system crash. 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 [18].

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 [18]. With simple changes, T2FS transactions will allow these programs to provide stronger failure guarantees more efficiently.

5 Related work

There have been a number of efforts over the years to provide systems support for file-system transactions. We now discuss different approaches and their problems.

Building file systems on top of user-space databases. One way to allow applications to update state in a transactional manner is to build a file system over a user-space transactional database. OdeFS [6], Inversion [15], and DBFS [14] use a database (such as Berkeley DB [16]) to provide ACID transactions to applications. Amino [31] tracks all user updates via `ptrace` and employs a user-level database to provide transactional updates. Such systems suffer from high performance degradation (*e.g.,* 5-7× on certain workloads).

In-kernel Transactional File Systems. An approach that leads to higher performance is adding transactions to in-kernel file systems. Valor [27] provides kernel support for file-system transactions. However, Valor relies on mandatory file locks held at user level and complex rules for write ordering due to its log being independent from the file system's. Its performance on several types of benchmarks is about 3× slower than ext3. T2FS features a simpler system call interface and more in-kernel work to maintain isolation. Its state management is simpler because it reuses the file system journal. T2FS has much higher performance.

Microsoft introduced Transactional NTFS (TxF), Transaction Registry (TxR), and the kernel transaction manager (KTM) in Windows Vista [23]. Using TxF requires all transactional operations be explicit (i.e., `read` does not work within 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. In contrast, T2FS allows the application to wrap unmodified file-system updates in a transaction. TxF also had restrictions on use, for example, no transactions for files on the boot disk. In contrast, in T2FS, the only restriction is that all operations inside a transaction have to be on the same file system (which is the same limitation as with hard links).

Transactional Operating Systems. A third, somewhat heavyweight, approach is modifying the entire operating system to provide transactions. Locus [30], QuickSilver [9], and TxOS [20] are operating systems that provide transactions. This approach adds significant complexity to the kernel; a large number of kernel data structures will have to rewritten to support transactions. T2FS mostly confines its modifications to the VFS and is designed to be extended to work for file systems other than ext4. Developing a correct transactional database is extremely complicated – retrofitting an existing kernel to do so is even more so.

Transactional Storage Systems. Similar to our work, CFS [13] provides a lightweight mechanism for atomic updates of multiple files. CFS builds on top of transactional flash storage. MARS [5] builds on hardware-provided atomicity to build a transactional system. TxFLash [22] uses the copy-on-write nature of Flash SSDs to provide transations at low cost. Isotope [25] uses the interal multi-versioning in block stores to provide isolation at the block level. In contrast to these systems, T2FS provides transactions without assuming any hardware support (beside device cache flush and atomic sector updates).

6 Conclusion

In this paper, we take the position that the operating system should provide a transactional interface for applications to efficiently update persistent state across storage abstractions. We propose a novel way to reduce the implementation complexity of building a transactional system, by leveraging the file-system journal. We show that our system, T2FS, increases performance in SQLite by up to 32% in insert and update workloads.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback and guidance. This work was supported in part by NSF (CNS-1618563, CCF-1333594) and by generous donations from VMware and Facebook. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of other institutions.

References

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

[2] MediaWiki.org. `https://www.mediawiki.org/wiki/MediaWiki`.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Google. LevelDB. `https://github.com/google/leveldb`.

[8] Robert Hagmann. *Reimplementing the Cedar file system using logging and group commit*, volume 21. ACM, 1987.

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

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

[12] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. Improving in-memory database index performance with intel® transactional synchronization extensions. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 476–487. IEEE, 2014.

[13] 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.

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

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

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

[17] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[18] 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.

[19] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alex Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

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

[30] 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.

[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.