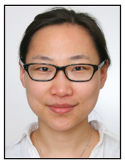
TxFS
Leveraging File-System Crash Consistency to Provide ACID Transactions

YIGE HU, ZHITING ZHU, IAN NEAL, YOUNGJIN KWON, TIANYU CHENG, VIJAY CHIDAMBARAM, AND EMMETT WITCHEL


Yige Hu is a PhD student at the University of Texas at Austin, under the supervision of Professor Emmett Witchel. Her research interests include operating systems, storage, and heterogeneous architecture. yige@cs.utexas.edu

Zhiting Zhu is a PhD student at the University of Texas at Austin, working with Emmett Witchel. He is interested in operating systems and security. zhitingz@cs.utexas.edu

Ian Neal received his computer science and electrical engineering degrees from the University of Texas at Austin in 2018. His undergraduate honors thesis was on transaction file systems, and he has also worked on other storage systems in non-volatile RAM. He will be starting his PhD program in the fall of 2018 at the University of Michigan at Ann Arbor. ian.glen.neal@utexas.edu

Youngjin Kwon is a PhD candidate at the University of Texas at Austin under the supervision of Professors Emmett Witchel and Simon Peter. His research interests lie in operating systems, including file systems, emerging storage and memory technologies, system support for security, and virtualization. His research has been recognized by VMware, and he contributed an initial version of his research work to VMware commercial hypervisor. yjkwon@cs.utexas.edu

We introduce TxFS, a novel transactional file system that builds upon a file system's atomic-update mechanism such as journaling. Although 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.

Modern applications store persistent state across multiple files. Some applications split their state among embedded databases, key-value stores, and file systems. 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.

Transactions present an intuitive way to atomically update persistent state. Unfortunately, building transactional systems is complex and error-prone, leading us to develop 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 userspace transactions. Our approach greatly reduces the development effort and complexity for building a transactional file system.

We introduce TxFS [4], a transactional file system that builds on the ext4 file system's journaling mechanism. We designed TxFS to be practical to implement and easy to use. TxFS has a unique set of properties. It has a small implementation (5200 lines of code) by building on the journal. It provides high performance, unlike various solutions that built a transactional file system over a userspace database [3, 12]. It has a simple API (just wrap code in `fs_tx_begin()` and `fs_tx_commit()`) compared to solutions like Valor [10] or TxF [8], 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 [5] and AdvFS [11], which only offer some of the guarantees, and it also provides transactions at the file level instead of at the block level, unlike Isotope [9], making several optimizations easier to implement. Finally, TxFS does not depend on specific properties of the underlying storage, unlike solutions such as MARS [2] and TxFlash [7].

We find that file system transactions lead naturally to a number of seemingly unrelated file-system optimizations. For example, one of the core techniques from our earlier work, separating ordering from durability [1], is easily accomplished in TxFS. Similarly, we find TxFS transactions allow us to identify and eliminate redundant application I/O 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 I/O to logs or temporary files, provided that the temporary files and logs are deleted inside the transaction. As a result, TxFS improves

Tianyu Cheng received an MS in computer science with high honors from the University of Texas at Austin in 2017. He is interested in a wide range of topics, including computer architecture and graphics. He is currently working on GPU architecture validation with Apple Inc. tianyu.cheng@utexas.edu

Vijay Chidambaram is an Assistant Professor in the Computer Science Department at the University of Texas Austin. He works on distributed systems, operating systems, and storage. His work has resulted in patent applications by VMware, Samsung, and Microsoft. His research has won the SIGOPS Dennis M. Ritchie Dissertation Award in 2016, Best Paper Awards at FAST 2017 and 2018, and a Best Poster at ApSys 2017. He was awarded the Microsoft Research Fellowship in 2014 and the University of Wisconsin-Madison Alumni Scholarship in 2009. vijay@cs.utexas.edu

Emmett Witchel is a Professor in Computer Science at the University of Texas at Austin. He received his doctorate from MIT in 2004. He and his group are interested in operating systems, security, performance, and concurrency. witchel@cs.utexas.edu

performance while simultaneously providing better crash-consistency semantics: a crash does not leave messy 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 1.6x, and a microbenchmark that mimics Android Mail obtains 2.3x 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.

We make 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 (see "TxFS Design and Implementation," below). We have made TxFS publicly available at https://github.com/ut-osa/txfs.
- We show that existing file system optimizations, such as separating ordering from durability, can be effectively implemented for TxFS transactions (see "Accelerating Programming Idioms with TxFS," below).
- We show that real applications can be easily modified to use TxFS, resulting in better crash semantics and significantly increased performance (see "Evaluation," below).

Why Use File-System Transactions?

We describe the complexity of current protocols used by applications to update persistent state and discuss a few case studies. We then describe the optimizations enabled by file-system transactions.

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 [6].

```
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

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. 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 [1]; since `fsync()` forces data to be durable, 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 [6].

We now describe two common techniques used by applications to consistently update storage, illustrated in Figure 1.

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. Protocol (a) illustrates the approach: 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. Finally, to ensure that the original file has been unlinked and deleted properly, the application calls `fsync()` on the parent directory.

Logging. Protocol (b) shows another popular technique for atomic updates, logging. In the write-ahead version of logging, the log file is written with new contents, and both the log file and the parent directory (with the new pointer to the log file) are persisted. The application then updates 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) [6]. It then follows a protocol similar to protocol (b). Android mail uses six `fsync()` calls to persist a single email with an attachment.

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 log file may disappear after a crash. If the application crashes in the middle of updating the original file, it will not be able to recover using the log. Many application developers avoid `fsync()` calls due to the resulting decrease in performance, leading to severe bugs that cause loss of data.

In summary, safe update protocols for stable storage are complex and low performance. System support for file-system transactions will enable high performance for these applications.

Application Case Studies

We present two examples of applications (in addition to the previously described Android Mail) that struggle to obtain crash consistency using primitives available today. Several applications store data across the file system, key-value stores, and embedded databases such as SQLite. 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.

Apple iWork and iLife. Analysis of the storage behavior of Apple's home-user desktop applications finds that applications use a combination of the file system, key-value stores, and SQLite to store data. iTunes uses SQLite to store metadata separately from songs similar to the Android Mail application. 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, these applications use `fsync()` to order updates correctly.

Version control systems. Git is a widely used version control system. 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 [6].

For these applications, transactional support would lead directly to more understandable and more efficient idioms (rather than approaches like atomic rename used today). It is difficult for a user-level program to efficiently provide crash-consistent transactional updates using the POSIX file-system interface.

Optimizations Enabled by File-System Transactions

A transactional file-system interface enables a number of interesting file-system optimizations:

Eliminate temporary durable files. A number of applications such as Vim, Emacs, Git, and LevelDB provide reasonable crash semantics using the atomic rename approach. But these applications can simply enclose writes inside a transaction and avoid making an entire copy of the file. For large files, the difference in performance can be significant. Additionally, transactions eliminate the clutter of temporary files orphaned by a crash.

Group commit. Transactions can buffer file-system updates in memory and submit updates to storage as a batch. Batching

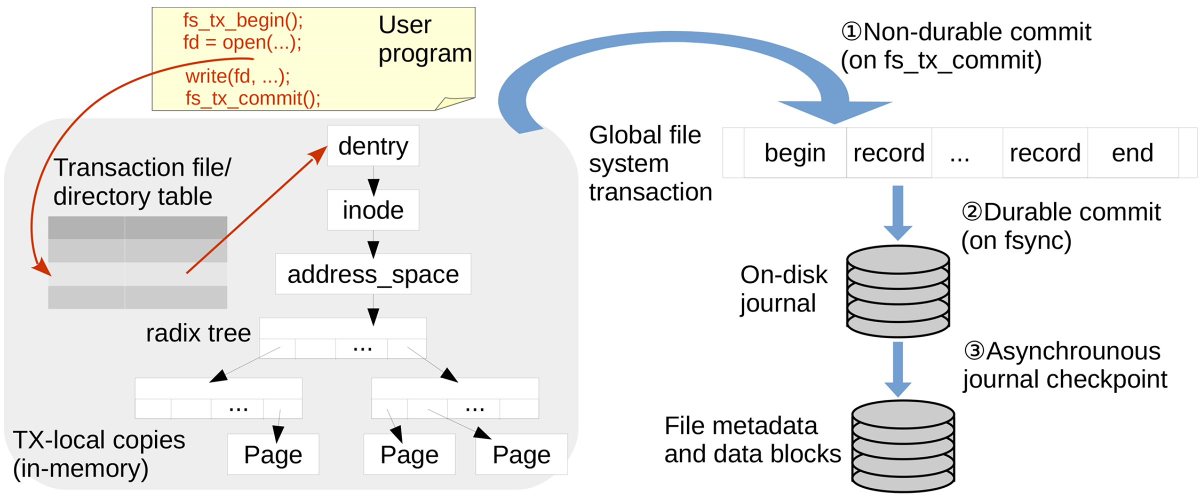

fs_tx_begin();
fd = open(...);
write(fd, ...);
fs_tx_commit();
User program
Transaction file/ directory table
dentry
inode
address_space
radix tree
TX-local copies (in-memory)
Page
Page
Page
①Non-durable commit (on fs_tx_commit)
Global file system transaction
begin
record
...
record
end
②Durable commit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

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

updates enables 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 I/O *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. Because the entire transaction is visible to the file system at commit time, it can eliminate redundant work.

Consolidate I/O *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. Optimizing multiple transactions, especially from different applications, is best done by the operating system, not by an individual application.

Separate ordering from durability. When ending a transaction, the programmer can specify whether 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. Thus, we can use transactions to gain the benefit of splitting sync into ordering sync (osync) and durability sync (dsync) [1].

TxFS Design and Implementation

TxFS avoids the pitfalls from earlier transactional file systems. 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.

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 the TxFS 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 several reasons, including a conflict with another transaction or not enough storage resources. Depending on the error code, the application can choose to retry the transaction.

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, such as Window's TxF and Valor, have far more complex, difficult-to-use interfaces.

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 that are difficult to coordinate without transactions.

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.

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. TxFS can be built upon any file system with a mechanism for atomic updates such as copy-on-write. TxFS guarantees atomicity by ensuring that all operations in a user transaction are added to a single local journal transaction, and it persists the journal transaction to ensure durability.

Isolation and Conflict Detection

Although the ext4 journal provides atomicity and durability, it does not provide isolation. 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. 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.

Split file-system functions. System calls such as `write()` and `open()` execute file-system functions that often result in allocation of file-system resources such as data blocks and inodes. TxFS splits such functions into two parts: file-system allocation and in-memory structures. TxFS moves file-system allocation to the commit point. In-memory changes execute as part of the system call, and they 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. For example, directory entries updated by the transaction are modified to point to a local inode that maintains a local radix tree with locally modified pages. 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.

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

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.

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 that follows the existing file-system conventions, so that deadlocks are avoided.

Conflict detection. Conflict detection is a key part of providing isolation. Since allocation-related 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 because many file-system data structures are modified all over the Linux kernel without a standard interface. TxFS eagerly detects conflicts on data pages, taking advantage of the structured kernel API for page management. It lazily detects conflicts on directory entries and file metadata structures, quickly detecting at commit time whether these structures have been updated.

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 locally 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.

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.

Accelerating Programming Idioms with TxFS

We explore a number of programming idioms where a transactional API can improve performance because transactions

Experiment	TxFS Benefit	Speed
Single-threaded SQLite	Faster I/O path, less sync	1.31x
TPC-C	Faster I/O path, less sync	1.61x
Android Mail	Cross abstraction tx	2.31x
Git	Better crash semantics	1.00x

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

provide the file system a sequence of operations that can be optimized as a group. 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 [1]) can be obtained easily with transactions.

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.

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 133x because the transaction completely eliminates the workload's I/O. 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.

Eliminating Logging I/O

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 I/O performed in half because the log file is never persisted. Rewriting the code increases performance by 55% (3.28 seconds, 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, 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. Just enclosing the logging activity of SQLite in its default mode (Rollback) within 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 system calls 2.5x.

Separating Ordering and Durability

Table 1 shows throughput for a workload that creates three 10 MB files and then updates 10 MB of a separate 40 MB 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. Our previous work that first distinguished ordering from persistence required adding modified sync system calls [1], but TxFS can achieve the same result with transactions.

Evaluation

We evaluate the performance and durability guarantees of TxFS on a variety of microbenchmarks and real workloads. The microbenchmarks help point out how TxFS achieves specific design goals. The larger benchmarks validate that transactions provide stronger crash semantics and improved performance for a variety of large applications with minimal porting effort. For example, we modified SQLite to use TxFS transactions and measured its performance improvement. 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 (no need for post-crash manual Git recovery) without degrading performance. Note that if not explicitly mentioned, all our baselines run on ext4 in its default ordered journaling mode. For more details please refer to the original publication [4].

Conclusion

We present TxFS, a transactional file system built with lower development effort than previous systems by leveraging the file-system journal. TxFS is easy to develop, is easy to use, and 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. We believe that file-system transactions, given their power and flexibility, 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.

References

[1] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic Crash Consistency," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pp. 228–243: http://research.cs.wisc.edu/adsl/Publications/optfs-sosp13.pdf.

[2] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From ARIES to MARS: Transaction Support for Next-Generation, Solid-State Drives," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, pp. 197–212: https://cseweb.ucsd.edu/~swanson/papers/SOSP2013-MARS.pdf.

[3] N. H. Gehani, H. V. Jagadish, and W. D. Roome, "OdeFS: A File System Interface to an Object-Oriented Database," in *Proceedings of the 20th Very Large Databases Conference (VLDB 1994)*, pp. 249–260: http://www.vldb.org/conf/1994/P249.pdf.

[4] Y. Hu, Z. Zhu, I. Neal, Y. Kwon, T. Cheng, V. Chidambaram, and E. Witchel, "TxFS: Leveraging File-System Crash Consistency to Provide ACID Transactions," 2018 USENIX Annual Technical Conference (USENIX ATC '18).

[5] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom, "Lightweight Application-Level Crash Consistency on Transactional Flash Storage," in *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15)*, pp. 221–234: https://www.usenix.org/system/files/conference/atc15/atc15-paper-min.pdf.

[6] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. 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)*, pp. 433–448: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-pillai.pdf.

[7] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional Flash," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 147–160: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/prabhakaran/prabhakaran.pdf.

[8] M. E. Russinovich, D. A. Solomon, and J. Allchin, *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, 4th edition (Microsoft Press, 2005).

[9] J.-Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon, "Isotope: Transactional Isolation for Block Storage," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pp. 23–37: https://www.usenix.org/system/files/conference/fast16/fast16-papers-shin.pdf.

[10] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright, "Enabling Transactional File Access via Lightweight Kernel Extensions," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)*, pp. 29–42: https://www.usenix.org/legacy/event/fast09/tech/full_papers/spillane/spillane.pdf.

[11] R. Verma, A. A. Mendez, S. Park, S. S. Mannarswamy, T. Kelly, and C. 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 '15)*, pp. 203–211: https://www.usenix.org/system/files/conference/fast15/fast15-paper-verma.pdf.

[12] C. P. Wright, R. Spillane, G. Sivathanu, E. Zadok, "Extending ACID Semantics to the File System," *ACM Transactions on Storage (TOS)*, vol. 3, no. 2 (May 2007), pp. 1–40: http://www.fsl.cs.stonybrook.edu/docs/amino-tos06/amino.pdf.