Towards Efficient, Portable Application-Level Consistency

Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram,
Joo-Young Hwang†, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin, Madison † Samsung Electronics Co., LTD.

ABSTRACT

Applications employ complex protocols to ensure consistency after system crashes. Such protocols are affected by the exact behavior of file systems. However, modern file systems vary widely in such behavior, reducing the correctness and performance of applications. In this paper, we study application-level crash consistency. Through the detailed study of two popular database libraries (`SQLite`, `LevelDB`), we show that application performance and correctness heavily depend on file-system properties previously ignored in research. We define a number of such properties and show that they vary widely among file systems. We conclude with implications for future file-system and dependability research.

1. INTRODUCTION

Application complexity is increasing over the years: compared with simple, modular UNIX applications [7], modern applications are huge monoliths, offering users a rich set of features and managing a huge amount of user data [5]. In many cases, storing extra data allows the application to provide new features (e.g., Firefox stores browsing history to provide autocomplete).

Many applications require that certain invariants on the data hold across system crashes. For example, Firefox requires that auto-completed URLs match entries in the browsing history, while a photo-viewing application requires that thumbnails match photos [25]. An application is deemed *consistent* when its invariants hold.

Unfortunately, implementing crash invariants in an efficient manner often requires complex update (and recovery) protocols on top of the underlying file system. It is easy to make mistakes in these implementations [14]. Previous research has found bugs in even simple crash-invariance implementations [24].

It is known that many applications depend (for crash invariance) on how file systems behave when a file's contents are fully replaced [21]. Specifically, crash invariants are held only if the replacement is atomic with respect to a system crash. This dependency is sometimes a bug, and is sometimes intentional, to provide reasonable application performance over different file systems with widely varying performance characteristics. For the safe execution of applications, many modern file systems explicitly ensure atomic file replacement (during certain sequences of system calls), even though this is not a part of the POSIX standard.

However, application-level consistency is also affected by other undocumented (and unexplored) crash-related behavior that differs between file systems. This severely constrains the portability of applications. There is no consensus on what file-system behavior affects application-level consistency, and what behaviors file systems should hence guarantee. In addition, the lack of well-defined file-system behavior prevents careful applications from optimizing their crash-invariance protocols, thus reducing efficiency. Previous techniques for file-system internal consistency, such as journaling [22, 23], soft updates [4], copy-on-write [1, 6, 13], backpointer-based consistency [3], and optimistic crash consistency [2], have not investigated their (unintended) consequences on application-level consistency.

To better understand the problem described above, we take a two-pronged approach. First, we examine modern file systems, targeting their *post-crash properties*: properties that could potentially affect application-level consistency (§2). We carefully define these properties and show that they vary widely among file systems, and even among different versions of the same file system.

Second, we carefully study two database libraries, `SQLite` [20] and `LevelDB` [8], recording their exact relationship with differing file-system behavior. Both applications are widely-used, provide sufficiently-defined crash invariants, employ complex protocols to enforce such invariants, and are yet small enough for a complete study.

We find that both applications are affected by the post-crash behavior of the underlying file system. In the relatively younger (and less tested) `LevelDB`, correctness is affected (§3). We find and report a number of bugs that get exposed only under certain file-system behavior. In the more mature and better-tested `SQLite` library, performance is affected: `SQLite` provides many configuration options that allows it to be optimized for different file-system behaviors (§4). However, the developers note that determining the relevant properties of a particular storage stack is difficult. Some of the configuration options in `SQLite` and bugs in `LevelDB` are directly related to the behavior of common file systems.

HotDep'13, November 03-06 2013, Farmington, PA, USA

ACM 978-1-4503-2457-1/13/11 $15.00.
http://dx.doi.org/10.1145/2524224.2524229 .

Based on our study, we present lessons for application reliability and directions for future file-system research (§5). We discuss tools that could aid in understanding application-level consistency, and the requirements for building such tools. We also speculate on how file systems can be characterized more thoroughly, paving the way for file systems on which developers can easily build correct, efficient applications.

2. POST-CRASH PROPERTIES

Application-level consistency after a crash depends upon the contents of the file system that is visible to the application. We term this the *post-crash state*: directories, files, and file data that applications see after a crash. We define *post-crash properties* as the properties of flusters that affect the post-cash state. All post-crash properties that together define the complete post-crash state of a file system define the *post-crash behavior*.

Note that this definition of post-crash properties might include many characteristics that relate to a particular file system's implementation of crash recovery; we are only interested in properties that affect application-level dependability or performance. To illustrate, we define five properties that vary between ext3, ext4, and btrfs:

Ordered directory operations (P1): Consider three operations that can be performed on directories: `unlink()`, `creat()`, and `rename()`. Consider a sequence R_1,R_2,... of such operations performed on the file system, and a crash happening at any point in time. This property asserts that, if the post-crash state of the file system contains the effect corresponding to some R_i in the sequence, it will also contain the effects of $R_1, R_2, ...R_{i-1}$.

Safe appends (P2): Consider a file of size i bytes, and that an application appends the sequence of bytes $\langle b_1, b_2, b_3...b_j \rangle$ to the end of it using the `write()` call. Consider no other `write()` or `truncate()` operations happen on the file, and that a crash happens at any point in time. This property asserts that, if the size of the file in post-crash state is $i + x$, then the last x bytes in the file are $\langle b_1, b_2, b_3...b_x \rangle$. A slight variation is, for some $y < x$, the last x bytes are $\langle b_1, b_2, b_3...b_y, \{nullbytes\} \rangle$.

Ordered data appends (P3): Consider two files A and B. Consider that data $\langle p_1, p_2, p_3...p_x \rangle$ is appended to A, using the `write()` call or using memory-mapping interfaces; after appending to A, data $\langle q_1, q_2, q_3...q_y \rangle$ is then appended to B. If a crash happens at any point in time, this property asserts that, if $\langle q_1, q_2, q_3... \rangle$ is found appended to B in the post-crash state, then $\langle p_1, p_2, p_3...p_x \rangle$ will be found appended to A.

Safe new file flush (P4): Consider a file that has been created either using `open(O_CREAT)` or `creat()`, some amount of data (might be zero) is written to it, and a flushing operation (`fsync()`, `fdatasync()`, or `msync()`) is made on the file. This property asserts that, the post-crash state contains the directory entry corresponding to the created file. Some Linux manpages [10] for `fsync()` explicitly warn that this property should not be assumed, and that a `fsync()` on the directory itself is required for guaranteed existence of the directory entry.

Safe rename (P5): Consider that a file A already exists. Consider that a file B is created either using `open(O_CREAT)` or `creat()`, some amount of data is written to it, and `rename(`B`,`A`)` is called. On a crash, this property asserts that, if the effect of `rename()` is visible on the post-crash state, all data written to the file B before the crash now exists on file A.

	(P1)	(P2)	(P3)	(P4)	(P5)
ext3-ordered	✓	✓	✓	✓	✓
ext3-writeback	✓			✓	✓
ext4-original	✓	✓		✓	
ext4-current	✓	✓		✓	✓
btrfs		✓		✓	✓

Table 1: Post-crash properties of file systems. *The row header represents the post-crash properties. 'ext4-original' refers to the ext4 ordered mode using the mount option 'noauto_da_alloc', that corresponds to the original behavior of ext4. 'ext4-current' refers to the ext4 ordered mode with the current default behavior.*

Table 1 shows the relationship between these properties and various file systems. It should be noted that our understanding of which file systems confirm to each property is not based on any existing specification (a part of the problem is that clear specifications do not exist). Rather, it is based on our understanding of the file-system's source code, behavior reported by users, and our observation of the block-level activity of the file systems. To be more specific, for the values without '✓' in Table 1, we have repeatable test cases that disprove the property in the given file system. However, for the '✓' values, we were not able to find a disproving test case, and the file-system's implementation suggests that the property is present. Also, many values in Table 1 are sensitive to the source-code version and the file-system mount options; we used Linux kernel version 3.2.0, and only the explicitly specified mount options.

It should be noted that we have not completely described the properties of each file system. For example, we have not described any properties corresponding to the atomicity of writes. Completely describing all post-crash properties is impossible without studying applications in detail. Applications could be dependent on only specific properties: for atomicity, one property of interest might be that each `write()` call is atomic; another might be that, each set of writes to a file between two `fsync()` calls is atomic. File systems could provide specific properties; consider the *safe-rename* property. A related property might be that data written to any newly-created file will persist before a `rename()` of that file, even if the destination did not exist beforehand. This related property is not true for ext4-current, but is true for ext3-ordered.

We believe that there are three generic properties: the atomicity of file-system operations, the durability of operations, and the sequentiality (ordering) of operations. All other properties have a hierarchical relationship, with more specific properties subsumed by the three generic properties. For example, consider the property that all operations are persisted sequentially; such a property would essentially encompass four properties (all except P4) we described in this section. Guaranteeing more generic properties might affect file-system performance; more specific properties might not sufficiently describe the requirements for a broad range of applications. Thus, future research should discuss properties at the appropriate level. Consequently, though we find it convenient to use boolean post-crash properties in this paper, we believe future research would rather introduce more suitable formal

models to describe the post-crash behavior of file systems.

3. CASE STUDY: LevelDB

LevelDB [8] is a persistent key-value store originally developed in Google. LevelDB is widely deployed; the Chromium web browser uses it as an embedded storage library. We discovered four places where LevelDB was vulnerable to the post-crash behavior of file systems. Some of these were clearly bugs, causing bad behavior under commonly-used file systems we had already studied; the others might lead to bad behavior in the presence of (POSIX-compliant) file systems that we have not examined. Consequently, we refer to all of them as *vulnerabilities*. We have reported all four of our discovered vulnerabilities to the LevelDB bug database [9], and they are currently being investigated by the developers. We also found that there was one relevant bug reported previously (and subsequently fixed); this bug also happens only in select file systems.

Analysis methodology: We first understood the required crash invariants in LevelDB and studied the protocol that LevelDB uses to persist its data. We then used system-call traces to find the exact file-system calls issued, and decoded the vulnerabilities in the sequence of calls. On discovering a potential vulnerability, we verified it using some environmental changes to increase the window of vulnerability, running the corresponding workload, and inducing a system crash during the window. If the vulnerability does not correspond to any existing file-system configuration, we carefully reconstructed the post-crash state that would result from the vulnerability, and made LevelDB recover.

3.1 Vulnerabilities

We report only on the interesting parts of the protocol that lead to each bug, and the behavior required from the file system to expose the bug. We also provide the ticket number in the bug database corresponding to the vulnerability; the interested reader should refer to the database for details. The following are the vulnerabilities:

Bug #183: During the initial creation of a database, LevelDB writes initialization data to a newly-created file, then writes a pointer to the data in another newly-created file (without any `fsync()` in-between). If the system crashes during the creation, the pointer might get written to the disk first, before the data; after reboot, LevelDB would report an error if the user tries to recreate (or open) the database. Since all information is only appended to files, this bug would be hidden under the *Ordered append* property.

Bug #187: Another bug related to the *Ordered append* property. During `Put()` operations on the database, LevelDB keeps appending entries to the end of a log file. It switches from one log file to another when the first log file reaches a certain size. During this switch, LevelDB does not ensure that the first log is completely on disk. If the second log is persisted but the first one is not, LevelDB recovers the newer entries in the second log file without being aware of the loss of older entries, violating its guaranteed invariants.

Bug #189: This would have been hidden under a file system that possessed the *Ordered directory operations* property. LevelDB sometimes compacts a set of log files containing entries into a single file. The single file is first created with a temporary name, all data is written to it (and flushed), then atomically renamed; finally, the older files are unlinked. However, the rename is not explicitly persisted before unlinking the old files. On a crash, the old files might be permanently deleted, while the new file still exists under the temporary name (and is hence not recognized by LevelDB). After recovery, either corruption is reported, or some (previously existing) key-value pairs disappear without any indication of an error.

Bug #190: LevelDB always assumes the *Safe new file flush* property. If this property is not guaranteed, there are multiple chances of corruption within LevelDB.

Bug #68: This bug had been previously reported (not by us) and fixed. When opening a database, LevelDB updates a file by first creating a temporary file, writing the required contents to it, then renaming it over the original file. However, LevelDB does not flush the contents of the file to disk before issuing the `rename()`. On a file system not obeying *Safe rename*, a system crash might result in the (renamed) file not containing the desired contents; LevelDB reports corruption in this case.

3.2 Observations

Based on our study, we make the following observations:

- The complete absence of `fsync()` calls on parent directories, combined with the dependence of the recovery protocol on the *Ordered directory operations* and *Safe new file flush* properties (#189 and #190), suggest that the developers *assume* these properties.
- Although btrfs does not obey *Ordered directory operations*, it seems to re-order directory operations only in such a way that #189 does not get exposed. More specifically, given `{rename(); unlink()}`, both operations get persisted only in the issued order; re-ordering happens only when, for example, the issued sequence is `{unlink(); rename()}`.
- If all operations are persisted in-order, and *Safe new file flush* is guaranteed, all discussed vulnerabilities would be hidden (i.e., *tolerated*).
- The existence of these vulnerabilities, and their intricate relationship to post-crash behavior, suggests that this is an area for further exploration.

4. CASE STUDY: SQLite

SQLite [20] is a relational database that provides ACID guarantees, and is widely used as an embedded library in desktop and mobile applications. SQLite features an inbuilt crash-recovery test suite [18] that rigorously tests SQLite's crash invariants during power failures. This, combined with the long history of bug fixing in SQLite, seemingly leaves it portable across file systems: on analyzing it similarly to LevelDB, we found no evidence of vulnerabilities. A few past bugs are interesting, and are described further in Section 4.2.

Consequently, the protocol used by SQLite to ensure crash invariants is *pessimistic*: performance is sacrificed. The developers recognize this, and present a solution in the form of a set of configurable options, each of which slightly changes the protocol (thus improving performance). The developers suspect that, given different underlying storage and file systems, a subset of these configuration options can be switched on without sacrificing correctness. Thus, each

option depends on a specific post-crash property; however, the developers do not understand how existing file systems relate to the properties [15]. Thus, this directly motivates the need to document post-crash behaviors of file systems.

In this section, we describe the configuration options (related to post-crash properties) that `SQLite` provides, and show the performance improvements they provide. We note that `SQLite` has two implementations of the recovery protocol; while both can be considered ideal candidates for generalizing the types of protocols applications use, our study assumes *rollback journaling*.

Protocol: In *rollback journaling*, when the user modifies a database, `SQLite` first creates a temporary journal file, and appends a copy of some information in the database file to the journal file. The database file is then actually updated and the journal file deleted. If the update was interrupted due to a crash, the temporary journal file will be left on-disk; if `SQLite` finds a journal while opening the database, it recovers the contents from the journal.

4.1 Configurable post-crash expectations

We present here five relevant configuration options in `SQLite`. For brevity, we omit detailed explanations.

Safe append: This option corresponds to the post-crash property of the same name in §2. During recovery after a system crash, `SQLite` must find out if the journal file is valid, or if a crash happened while the journal file was being written to. Validity is checked using a header in the journal file; the header is marked valid only after the rest of the journal file is updated and flushed using `fsync()`. Since all writes to the journal file are appends, if *Safe append* is satisfied, the extra `fsync()` can be omitted.

Power-safe overwrite: This option assumes that no data in a given file, other than those explicitly over-written, are ever affected by a system crash. `SQLite` decides the *granularity* of information copied to the journal file based on this parameter. This property is probably true in most modern file systems (and storage stacks), and is switched on by default in the current version of `SQLite`.

Atomic writes: Given a granularity of atomicity, this option assumes that all `write()` calls lesser than that granularity are atomic (with respect to system crash). Thus, if a modification to the database is smaller than this granularity, `SQLite` can directly modify the database without using a journal file.

Sequential writes: This option assumes that, if a sequence of `write()` calls are issued, all calls only get persisted in-order. Thus, all `fsync()` operations required while creating the journal file and writing information to it, can be omitted.

Synchronous directory operations: When the temporary journal file is created, and also during a couple of other directory operations, `SQLite` normally flushes the directory after the operation. This configuration option omits the directory flushes. Thus, for safe operation, the file system should guarantee *Safe new file flush*, and also that `unlink()` gets persisted immediately.

We evaluated `SQLite`'s performance when each of these options are (separately) toggled, using two micro-benchmarks. The insert benchmark creates six tables of two columns each, and inserts a number of rows in each of them; each insert is a separate transaction. The update benchmark first inserts 300 rows into each table, then measures the performance of the SQL UPDATE statement on all rows in each table; each update (containing all rows in a table) is a transaction. We used a single core machine running Ubuntu 12.04, a 80 GB hard drive with ext4-current, and `SQLite` version 3.7.17.

Post-crash Expectations	**Throughput (X/s)**	
	Insert	*Update*
Default	10.25	9.23
Atomic writes	33.39 (+226%)	32.62 (+253%)
Safe append	14.47 (+41%)	12.62 (+37%)
Sequential write	33.88 (+231%)	32.52 (+252%)
Power-safe overwrite	10.43 (+2%)	10.33 (+12%)
Synchronous directory	11.24 (+10%)	9.98 (+8%)

Table 2: `SQLite` performance. *The table represents throughput obtained when different configuration options are toggled. "Default" represents running SQLite without changing any of the defaults (this switches on power-safe overwrite, and switches off the others). The "Power-safe overwrite" row represents the performance when the option is switched off. "Atomic writes" represent a configuration in which all writes are atomic.*

The observed performance is shown in Table 2. Each option seems to significantly affect throughput. Especially important are the performance effects of *safe append*: this is an option that can be safely switched on with ext4-current. Also, we modified `SQLite` to implement a new option corresponding to (only) *Safe new file flush*, which can again be safely switched on with ext4. The performance is the same as *synchronous directory operations*. We believe that *power-safe overwrite* improves performance because writes to the journal now happen at file-system-block granularity.

4.2 Past bugs

We examined all bugs from October 2009 to May 2013 in `SQLite`'s bug database; we also studied a few older bugs that seemed interesting. Among the studied bugs, three specifically affect system-crash invariants [16, 17, 19].

An early bug reported in `SQLite` [16] (before the introduction of the crash-test suite) concerns *Safe new file flush*; `SQLite` was modified to `fsync()` the parent directory after creating files. We should note that the developers were concerned about performance while fixing this issue.

A later bug [19] deals with recovery. During recovery, `SQLite` wrote the contents of the journal file to the database file, and then deleted the journal file; the database file was never flushed in-between. Thus, if a system crash happened again during recovery, the database file might be permanently left in a partially-recovered (corrupt) state. One post-crash property that would hide this bug is all write operations getting persisted before any directory operation. Note that the bug would not be hidden under any of the properties discussed in §2; specifically, no renames or data-appends are involved.

The third bug [17] deals with entries being appended to a log file, in `SQLite`'s other crash-invariance protocol (*write-ahead logging*). When the log of entries exceeds a certain threshold, `SQLite` wraps around, and starts writing entries again from the beginning of the log. However, without any flushes, such a wrap-around of the log could cause an older, invalid transaction to get replayed during recovery, causing corruption. If the file system had the *Sequential writes*

property described in §4.1, this would not happen.

4.3 Observations

Based on our study, we make the following observations:

- Except for the *Safe new file flush*, `SQLite` historically seems to have had few expectations from the file system. Specifically, two of the three bugs [17, 19] seem to only have occurred because the developers did not carefully consider the corresponding workload. Adding the workloads to the crash-test suite reveals the bugs.
- Similar to `LevelDB`, if all operations were persisted in-order and *Safe new file flush* is guaranteed, the three discussed bugs would be hidden.
- Many of the (specific) post-crash properties that are related to `SQLite` considering either performance or correctness, are different from `LevelDB` (or those illustrated in §2). However, they seem to be "high-level" properties. This emphasizes our previous observation (in §2) that boolean descriptions of specific post-crash properties is not a sufficient model.

5. FUTURE DIRECTIONS

Alternate file system interfaces: Transactional file systems and similar interfaces can reduce the complexity of writing crash-invariance protocols. However, such interfaces do not seem to be widely adopted; we believe the reasons are a few practical disadvantages. For example, the implementation might assume a specific file system (or device), resulting in unpredictable performance with a generic storage stack. Another problem is that the implementation can be complex by itself, thus not reducing the overall chance of bugs.

Designing safer file systems: Research is also needed on designing *efficient* file systems that safely tolerate application vulnerabilities. There seems to traditionally have been an inverse relationship between performance and its safety; ext3 ordered mode, while seemingly safer, is also rumored to be slower than many other file systems. Some previous research has partially focused on this direction. For example, *xsyncfs* [12] provides the illusion of sequential and durable file-system updates, but can also be affected by unrelated application behavior, a practical disadvantage.

Finding application vulnerabilities: Tools and techniques for discovering application vulnerabilities will be helpful both to make sure a certain application is portable, and to find common expectations of post-crash behavior (across applications) that help design more *useful* file systems. Previous work in this area, such as EXPLODE [24] and `SQLite`'s crash-test suite, use a given set of unit tests to find application vulnerabilities. Designing unit tests with a good *coverage* is hard, however: effective tests require understanding which aspects of the protocol affect post-crash behaviors of the file system, in addition to knowledge of the application's recovery protocol. Moreover, it is unclear whether current work will scale to abstract post-crash behaviors: EXPLODE targets a single given file system, and `SQLite`'s suite targets only a subset of the file-system interface.

Characterizing existing file systems: It is currently hard to verify if a file system satisfies a particular post-crash property, or to describe the post-crash behavior of a file system in sufficient detail. Research is needed on techniques for verification, on *verifiable* file systems, and on formal models of describing post-crash behavior. Such research should also concentrate on characterizing the performance of file systems as they relate to post-crash behavior, and on ways to export this to applications.

6. CONCLUSION

Consistency is an important requirement for modern applications. By chance, or on purpose, application consistency is dependent on certain properties of the underlying file systems. Although there seems to have been much discussion on this issue, and some attempt at designing file systems that provide related features to applications [11], the dependencies are not fully studied. Our paper demonstrates the dependencies; we hope future research could better address the full problem, such as modeling techniques to describe the post-crash behavior of file systems in sufficient detail, or tools that detect these dependencies in applications.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. We thank members of the ADSL lab, Sankaralingam Panneerselvam, and Venkatanathan Varadarajan for their feedback. We thank the developers and users of `SQLite` and `LevelDB` for helping us understand their software in detail. This material is based upon work supported by the NSF under CNS-1319405 and CNS-1218405 as well as donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Samsung, Sony, and VMware. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

[2] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *SOSP '13*, Farmington, Pennsylvania, November 2013.

[3] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *FAST '12*, San Jose, CA, February 2012.

[4] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *OSDI '94*, pages 49–60, Monterey, CA, November 1994.

[5] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *SOSP '11*, Cascais, Portugal, October 2011.

[6] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *USENIX Winter '94*, San Francisco, CA, January 1994.

[7] Butler Lampson. Computer Systems Research – Past and Present. SOSP 17 Keynote Lecture, December 1999.

[8] LevelDB. LevelDB: a fast and lightweight keyvalue database library by Google. http://code.google.com/p/leveldb/.

[9] LevelDB. LevelDB Issues List. http://code.google.com/p/leveldb/issues/list.

[10] Linux. fsync(2) - Linux Programmer's Manual. http://man7.org/linux/man-pages/man2/fsync.2.html.

[11] Microsoft. Alternatives to using Transactional NTFS. http://msdn.microsoft.com/en-us/library/windows/desktop/hh802690(v=vs.85).aspx.

[12] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the Sync. In *OSDI '06*, pages 1–16, Seattle, Washington, November 2006.

[13] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[14] Stewart Smith. Eat My Data: How everybody gets file I/O wrong. In *OSCON*, Portland, Oregon, July 2008.

[15] SQLite. Atomic Commit In SQLite. http://sqlite.org/atomiccommit.html.

[16] SQLite. Creation and deltions of files should fsync() directory. https://www2.sqlite.org/cvstrac/tktview?tn=410.

[17] SQLite. Database corruption following power-loss in WAL mode. http://www.sqlite.org/src/info/ff5be73dee.

[18] SQLite. How SQLite Is Tested. http://www.sqlite.org/testing.html.

[19] SQLite. Missing call to xSync() following rollback. http://www.sqlite.org/src/info/015d3820f2.

[20] SQLite. SQLite Documentations. http://www.sqlite.org/.

[21] Theodore Ts'o. Don't fear the fsync! http://thunk.org/tytso/blog/2009/03/15/dont-fear-the-fsync/.

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

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

[24] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06*, Seattle, Washington, November 2006.

[25] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services. In *HotStorage '13*, San Jose, California, June 2013.