Application Crash Consistency and Performance with CCFS

THANUMALAYAN SANKARANARAYANA PILLAI, RAMNATTHAN ALAGAPPAN, and LANYUE LU, University of Wisconsin – Madison
VIJAY CHIDAMBARAM, The University of Texas at Austin
ANDREA C. ARPACI-DUSSEAU and REMZI H. ARPACI-DUSSEAU, University of Wisconsin – Madison

Recent research has shown that applications often incorrectly implement crash consistency. We present the Crash-Consistent File System (ccfs), a file system that improves the correctness of application-level crash consistency protocols while maintaining high performance. A key idea in ccfs is the abstraction of a *stream*. Within a stream, updates are committed in program order, improving correctness; across streams, there are no ordering restrictions, enabling scheduling flexibility and high performance. We empirically demonstrate that applications running atop ccfs achieve high levels of crash consistency. Further, we show that ccfs performance under standard file-system benchmarks is excellent, in the worst case on par with the highest performing modes of Linux ext4, and in some cases notably better. Overall, we demonstrate that both application correctness and high performance can be realized in a modern file system.

CCS Concepts: • **General and reference** → **Reliability**; **Performance**; • **Information systems** → *File systems*; • **Software and its engineering** → *File systems management*;

Additional Key Words and Phrases: File systems, crash consistency, reordering, performance

ACM Reference format:
Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application Crash Consistency and Performance with CCFS. *ACM Trans. Storage* 13, 3, Article 19 (September 2017), 29 pages.
https://doi.org/10.1145/3119897

The research herein was supported by funding from NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS-1218405, DOE grant DE-SC0014935, as well as donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Seagate, Veritas, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions. This article is an extended version of a FAST '17 paper by T.S. Pillai et al. [37]. The additional material here includes a detailed description of the stream API, detailed discussion of the limitations of the stream abstraction and the ccfs file system, more thorough discussion of developer effort required to make applications crash consistent atop ccfs, more figures and code snippets to illustrate usage of streams, and several other minor edits and updates.
Authors' addresses: T. S. Pillai, R. Alagappan, and L. Lu, 1210 W. Dayton St., Madison, WI 53706; V. Chidambaram, 2317 Speedway, Austin, TX 78712; A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, 1210 W. Dayton St., Madison, WI 53706; emails: {madthanu, ra, ll}@cs.wisc.edu, vijay@cs.utexas.edu, {dusseau, remzi}@cs.wisc.edu.

ACM 1553-3077/2017/09-ART19 $15.00
https://doi.org/10.1145/3119897

1 INTRODUCTION

"Filesystem people should aim to make 'badly written' code 'just work.'"

—Linus Torvalds [56]

The constraint of ordering is a common technique applied throughout all levels of computer systems to ease the construction of correct programs. For example, locks and condition variables limit how multi-threaded programs run, making concurrent programming simpler [3]; memory consistency models with stricter constraints (e.g., sequential consistency) generally make reasoning about program behavior easier [51]; `fsync` calls in data management applications ensure preceding I/O operations complete before later operations [6, 38].

Unfortunately, constraining ordering imposes a fundamental cost: poor performance. Adding synchronization primitives to concurrent programs adds overhead and reduces performance [20, 22]; stronger multiprocessor memory models are known to yield lower throughput [17]; forcing writes to a disk or SSD can radically reduce I/O performance [6, 7]. While in rare cases we can achieve both correctness and performance [43], in most cases we must make an unsavory choice to sacrifice one.

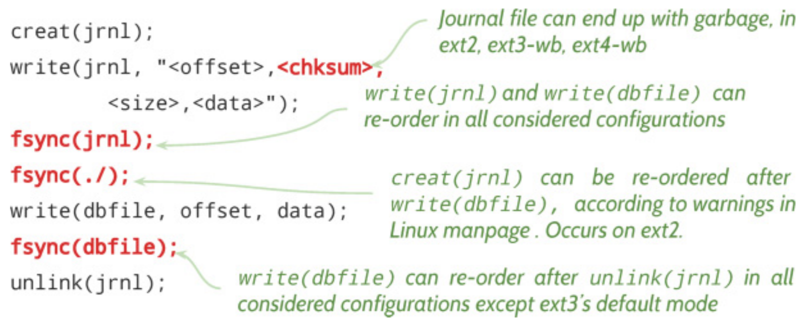
Within modern storage systems, this same tension arises. A file system, for example, could commit all updates in order, adding constraints to ease the construction of applications (and their crash-recovery protocols) atop them [4, 38]. Many file system developers have determined that such ordering is performance prohibitive; as a result, most modern file systems reduce internal ordering constraints. For example, many file systems (including ext4, xfs, btrfs, and the 4.4BSD fast file system) re-order application writes [2], and some file systems commit directory operations out of order (e.g., btrfs [38]). Lower levels of the storage stack also re-order aggressively, to reduce seeks and obtain grouping benefits [23, 24, 45, 47].

However, research has shown that user-level applications are often incorrect because of re-ordering [1, 38, 60]. Many applications use a specialized write protocol to maintain crash consistency of their persistent data structures. The protocols, by design or accident, frequently require all file-system updates to commit in their issued order [39].

The main hypothesis in this article is that a carefully designed and implemented file system can achieve *both* ordering and high performance. We explore this hypothesis in the context of the Crash-Consistent File System (ccfs), a new file system that enables crash-consistent applications while delivering excellent performance.

The key new abstraction provided by ccfs, which enables the goals of high performance and correctness to be met simultaneously, is the *stream*. Each application's file-system updates are logically grouped into a stream; updates within a stream, including file data writes, are guaranteed to commit to disk in order. Streams thus enable an application to ensure that commits are ordered (making recovery simple); separating updates between streams prevents *false write dependencies* and enables the file system to re-order sufficiently for performance.

Underneath this abstraction, ccfs contains numerous mechanisms for high performance. Critically, while ordering updates would seem to overly restrict file-system implementations, we show that the journaling machinery found in many modern systems can be adapted to yield high performance while maintaining order. More specifically, ccfs uses a novel hybrid-granularity journaling approach that separately preserves the order of each stream; hybrid granularity further enables other needed optimizations, including delta journaling and pointer-less metadata structures. Ccfs takes enough care to retain optimizations in modern file systems (like ext4) that appear at first to be incompatible with strict ordering, with new techniques such as order-preserving delayed allocation.



```
creat(jrnl);
write(jrnl, "<offset>,<chksum>,
        <size>,<data>");
fsync(jrnl);
fsync(./);
write(dbfile, offset, data);
fsync(dbfile);
unlink(jrnl);
```

Journal file can end up with garbage, in ext2, ext3-wb, ext4-wb

write(jrnl) and write(dbfile) can re-order in all considered configurations

creat(jrnl) can be re-ordered after write(dbfile), according to warnings in Linux manpage . Occurs on ext2.

write(dbfile) can re-order after unlink(jrnl) in all considered configurations except ext3's default mode

Fig. 1. Journaling Update Protocol. *Pseudo-code for a simple version of write-ahead journaling; each statement is a system call. The normal text corresponds directly to the protocol's logic, while the bold parts are additional measures needed for portability. Italicized comments show which measures are needed under the default modes of ext2, ext3, ext4, xfs, and btrfs, and the writeback mode of ext3/4 (ext3-wb, ext4-wb).*

We show that the ordering maintained by ccfs improves correctness by testing five widely used applications (Git, LevelDB, ZooKeeper, Mercurial, and SQLite), four of which (all except SQLite) are inconsistent on ext4 and other modern file systems [38]. We also show that the performance of standard benchmarks on ccfs are on par with ext4. We compare the performance of three real-world applications (Git, LevelDB, and SQLite) on ccfs and ext4; SQLite and LevelDB perform similarly on both file systems and Git has a 80× performance improvement on ccfs, when modified on ext4 to achieve the same correctness as on ccfs. Furthermore, with additional developer effort, the performance of SQLite and LevelDB on ccfs can be improved by factors of 685× and 5×, respectively. We analyze in detail the developer effort required for applications to use the stream abstraction and conclude it is trivial; for instance, we added a single line of code to Git to achieve both crash consistency and efficiency atop ccfs, while many complicated changes to many source files are required to make Git crash-consistent atop re-ordering file systems.

The article is structured as follows. We provide motivation and background (Section 2), present ccfs (Section 3), and evaluate it (Section 4). We then discuss the limitations of streams and ccfs (Section 5). Finally, we discuss related work (Section 6) and conclude (Section 7).

2 MOTIVATION AND BACKGROUND

In this section, we first explain the extent to which current data-intensive applications are vulnerable during a crash. We then describe why a file system that preserves the order of application updates will automatically improve the state of application-level crash consistency. Finally, we discuss the performance overheads of preserving order and how the overheads can be addressed.

2.1 State of Crash Consistency

To maintain the consistency of their user-level data structures in the event of a crash, many applications [21, 26, 34] modify the data they store in the file system via a carefully implemented *update protocol*. The update protocol is a sequence of system calls (such as file writes and renames) that updates underlying files and directories in a recoverable way. As an example, consider a simple DBMS that stores its user data in a single database file. To maintain transactional atomicity across a system crash, the DBMS can use an update protocol called *journaling* (or *write-ahead logging*): before updating the database file, the DBMS records the updates in a separate journal file. The pseudocode for the update protocol is shown in Figure 1. If a crash happens, then the

DBMS executes a *recovery protocol* when restarted: if the database file was only partially updated, the full update from the journal is replayed.

Correctly implementing crash-consistency protocols has proven to be difficult for a variety of reasons. First, their correctness inherently depends on the exact semantics of the system calls in the update protocol with respect to a system crash. Because file systems buffer writes in memory and send them to disk later, from the perspective of an application the effects of system calls can get re-ordered before they are persisted on disk. For example, in a naive version of the journaling update protocol, the unlink of the journal file can be re-ordered before the update of the database file. In Figure 1, an explicit `fsync` system call is used to force the update to disk before issuing the `unlink`. Also, the semantics of system calls can differ between file systems; for example, the aforementioned re-ordering occurs in the default configurations of ext2, ext4, xfs, and btrfs, but not in ext3.

Second, the recovery protocol must correctly consider and recover from the multitude of states that are possible when a crash happens during the update protocol. Application developers strive for update protocols to be efficient, since the protocols are invoked during each modification to the data store; more efficient update protocols often result in more possible states to be reasoned about during recovery. For example, the journal protocol in Figure 1 is often extended to batch multiple transactions onto the journal before the actual update to the database file, to avoid performance-intensive `fsync` calls.

Finally, crash-consistency protocols are hard to test, much like concurrency mechanisms, because the states that might occur on a crash are non-deterministic. Since efficient protocol implementations are inherently tied to the format used by the application's data structures and concurrency mechanisms, it is impractical to re-use a single, verified implementation across applications.

Unsurprisingly, past research [38, 60, 61] has found many vulnerabilities in the implementations of crash consistency protocols in widely used applications written by experienced developers, such as Google's LevelDB and Linus Torvalds's Git. However, in this article, we argue that it is practical to construct a file system that automatically improves application crash consistency. We base our arguments on the following hypotheses:

The Ordering Hypothesis: Existing update and recovery protocols (mostly) work correctly on an ordered and weakly atomic file system (we define these terms subsequently).

The Efficiency Hypothesis: An ordered and weakly atomic file system can be as efficient as a file system that does not provide these properties, with the proper design, implementation, and realistic application workloads.

2.2 Weak Atomicity and Order

We hypothesize that most vulnerabilities that exist in application-level update protocols are caused because the related application code depends on two specific file-system guarantees. File systems that provide these guarantees, therefore, automatically mask application vulnerabilities. The first guarantee, and the major focus of our work, is that the effect of system calls should be persisted on disk in the order they were issued by applications; a system crash should not produce a state where the system calls appear re-ordered. The second (minor) guarantee is that, when an application issues certain types of system calls, the effect of the system call should be atomic across a system crash. The second guarantee, which we term *weak atomicity*, is specifically required for system calls that perform directory operations (including the creation, deletion, and renaming of files and hard links). Weak atomicity also includes stipulations about writes to files, but only at sector granularity (i.e., there is generally no need to guarantee that arbitrarily large writes are atomic). If a system call appends data to the end of a file, then both increasing the file size and the writing of data to the newly appended portion of the file should be atomic together.

Table 1. Seeks and Order

	Time (s)	Seeks	Median seek distance (sectors)
Re-ordered	25.82	23,762	120
FIFO	192.56	38,201	2002112

The table shows the number of disk seeks incurred and the total time taken when 25,600 writes are issued to random positions within a 2GB file on a Toshiba MK1665GSX 160GB HDD (our experimental setup is described in more detail in Section 4). Two different settings are investigated: the writes can be re-ordered or the order of writes is maintained using the FIFO strategy. The number of seeks incurred in each setting and the LBA seek distance shown are determined from a block-level I/O trace.

The fundamental reason that order simplifies the creation of update protocols is that it drastically reduces the number of possible states that can arise in the event of a crash, that is, the number of states that the recovery protocol has to handle. For example, consider an update protocol that simply overwrites n sectors in a file; if the file system maintains order and weak atomicity, only n crash states are possible, whereas 2^n states are possible if the file system can re-order. Maintaining order makes it easier to reason about the correctness of recovery for both humans and automated tools [38].

The effectiveness of maintaining weak atomicity and order can be understood by considering the application-level crash vulnerabilities discovered in our previous study [38]. Among 60 vulnerabilities, we found that 16 are masked by maintaining weak atomicity alone. We also found that 27 vulnerabilities are masked by guaranteeing order. Of the remaining vulnerabilities, 12 are attributed to durability; however, we observe that 8 of these 12 will be masked if the file system guarantees order. Thus, in all, 50 of the 60 vulnerabilities are addressed by maintaining order and weak atomicity; the remaining 10 have minor consequences and are readily masked or fixed [39].

2.3 Order: Bad for Performance

Most real-world deployed file systems (such as btrfs) already maintain the weak atomicity required to mask application-level crash-consistency vulnerabilities. However, all commonly deployed file-system configurations (including ext4 in metadata-journaling mode, btrfs, and xfs) re-order updates, and the re-ordering only seems to increase with each new version of a file system (e.g., ext4 re-orders more than ext3 [38]; newer versions of ext4 re-order even more [57], as do newer systems like btrfs [38]). While maintaining update order is important for application crash consistency, it has traditionally been considered bad for performance, as we now discuss.

At low levels in the storage stack, re-ordering is a fundamental technique that improves performance. To make this case concrete, we created a workload that issues writes to random locations over a disk. Forcing these writes to commit in issue order takes roughly eight times longer than a seek-optimized order (Table 1). Re-ordering is important for hard drives [47] and SSDs [24]; approaches that constrict write ordering are insufficient.

Higher up the stack, ordering can induce negative (and sometimes surprising) performance degradations. Consider the following code sequence:

`write`(f_1); `write`(f_2); `fsync`(f_2); `truncate`(f_1);

In this code, without mandated order, the forced writes to f_2 can move ahead of the writes to f_1; by doing so, the truncate obviates the need for any writes to f_1 at all. Similarly, if the user overwrites f_1 instead of truncating it, only the newer data needs to be written to disk.

We call this effect *write avoidance*: not all user-level writes need to be sent to the disk, but can instead be either forgotten due to future truncates or coalesced due to future overwrites. Re-ordering allows write avoidance across `fsync` calls. Global write ordering, in contrast, implies that if writes to f_2 are being forced to disk, so must writes to f_1. Instead of skipping the writes to f_1, the file system must now both write out its contents (and related metadata), and then, just moments later, free said blocks. If the write to f_1 is large, then this cost can be high.

We call this situation, where `fsync` calls or cache eviction reduce write avoidance in an ordered file system, a *write dependence*. Write dependence is not limited to writes by a single application; any application that forces writes to disk could cause large amounts of other (potentially unneeded) I/O to occur. When write dependence does not improve crash consistency, as when it occurs between independent applications, we term it a *false dependence*, an expected high-cost of global order.

Apart from removing the chance for write avoidance, write dependence also worsens application performance in surprising ways. For example, the `fsync`(f_2) becomes a high-latency operation, as it must wait for all previous writes to commit, not just the writes to f_2. The overheads associated with write dependence can be further exacerbated by various optimizations found in modern file systems. For example, the ext4 file system uses a technique known as *delayed allocation*, wherein it batches together multiple file writes and then subsequently allocates blocks to files. This important optimization is defeated by forced write ordering.

2.4 Order with Good Performance

We believe it is possible to address the overheads associated with maintaining order in practice. To reduce disk-level scheduling overheads, a variety of techniques have been developed that preserve the *appearance* of ordered updates in the event of a crash while forcing few constraints on disk scheduling.

For example, in ext4 data journaling, all file-system updates (metadata and data) are first written to a journal. Once committed there, the writes can be propagated (checkpointed) to their in-place final locations. Note that there are no ordering constraints placed upon the checkpoint writes; they can be re-ordered as necessary by lower layers in the storage stack to realize the benefits of low-level I/O scheduling. Further, by grouping all writes into a single, large transaction, writes are effectively committed in program order: if a write to f_1 occurs before a write to f_2, they will either be committed together (in the same transaction), or the write to f_2 will commit later; never will f_2 commit before f_1. We discuss ext4 journaling in more detail in the next section.

Unfortunately, total write ordering, as provided with data journaling, exacts a high performance cost: each data item must be written twice, thus halving disk bandwidth for some workloads. For this reason, most journaling file systems only journal metadata, maintaining file-system crash consistency but losing ordering among application writes. What would be ideal is the performance of metadata-only journaling combined with the ordering guarantees provided by full data journaling.

However, even if an efficient journaling mechanism is used, it does not avoid overheads due to false dependence. To address this problem, we believe a new abstraction is needed, which enables the file system to separate update orderings across different applications. Within an application, we believe that false dependence is rare and does not typically arise.

Thus, we are left with two open questions. Can a metadata-only journaling approach be adopted that maintains order but with high performance? Second, can a new abstraction eliminate false dependence? We answer these questions in the affirmative with the design and implementation of ccfs.

3 CRASH-CONSISTENT FILE SYSTEM

In this section, we describe ccfs, a file system that embraces application-level crash consistency. Ccfs has two goals corresponding to our two hypotheses in Section 2.1: first, it aims to preserve the program order of updates and provide weak atomicity; second, it aims to achieve performance similar to widely-used re-ordering file systems. To satisfy these goals, we derive ccfs from the ext4 file system. Ext4 is widely used, includes many optimizations that allow it to perform efficiently in real deployments, and includes a journaling mechanism for internal file-system consistency. In ccfs, we extend ext4's journaling to preserve the required order and atomicity in an efficient manner without affecting the optimizations already present in ext4.

The key idea in ccfs is to separate each application into a *stream* and maintain order only within each stream; writes from different streams are re-ordered for performance. This idea has two challenges: metadata structures and the journaling mechanism need to be separated between streams, and order needs to be maintained within each stream efficiently. Ccfs should solve both without affecting existing file-system optimizations. In this section, we first explain ext4's journaling mechanism (Section 3.1), then the streams abstraction (Section 3.2), how streams are separated (Section 3.3) and how order is maintained within a stream (Section 3.4), and our implementation (Section 3.5). We finally discuss how applications can practically start using the streams abstraction (Section 3.6).

3.1 Journaling in Ext4

To maintain file-system metadata consistency, ext4 requires the atomicity of sets of metadata updates (e.g., all metadata updates involved in creating a file) and an order between these sets of updates [6, 25]. Ext4 uses an optimized journaling technique for this purpose. Specifically, the journaling occurs at block granularity, batches multiple sets of atomic metadata updates (*delayed logging* [12]), uses a circular journal, and delays forced checkpointing until necessary. The block-granularity and circular aspects prove to be a challenge for adoption in ccfs, while delayed logging and checkpointing are important optimizations that ccfs needs to retain. We now briefly explain these techniques of ext4 journaling.

Assume the user performs a metadata operation (such as creating a file), causing ext4 to modify metadata structures in the file-system blocks b_1, b_2, b_3. Ext4 associates b_1, b_2, b_3 with an in-memory data structure called the *running transaction*, T_i. Instead of immediately persisting T_i when the metadata operation completes, ext4 waits for the user to perform more operations; when this happens, the resulting set of block modifications are also associated with T_i (i.e., *delayed logging*). Periodically, ext4 *commits* the running transaction, that is, writes the updated contents of all the associated blocks of T_i and some bookkeeping information to an on-disk journal. When T_i starts committing, a new running transaction (T_{i+1}) is created to deal with future metadata operations. Thus, ext4 always has one running transaction, and at most one committing transaction. Once T_i finishes committing, its blocks can be written to their actual locations on disk in any order; this is usually done by Linux's page-flushing daemon in an optimized manner.

If a crash happens, then after rebooting, ext4 scans each transaction written in the journal file sequentially. If a transaction is fully written, then the blocks recorded in that transaction are propagated to their actual locations on disk; if not, then ext4 stops scanning the journal. Thus, the atomicity of all block updates within each transaction is maintained. Maintaining atomicity also implicitly maintains order within a transaction, while the sequential scan of the journal maintains order across transactions.

The on-disk journal file is circular: after the file reaches a maximum size, committed transactions in the tail of the journal are freed (i.e., *checkpointed*) and that space is reused for recording future transactions. Ext4 ensures that before a transaction's space is reused, the blocks contained in it are first propagated to their actual locations (if the page-flushing mechanism had not yet propagated them). Ext4 employs techniques that coalesce such writebacks. For example, consider that a block recorded in T_i is modified again in T_j; instead of writing back the version of the block recorded in T_i and T_j separately, ext4 simply ensures that T_j is committed before T_i's space is reused. Since the more recent version (in T_j) of the block will be recovered on a crash without violating atomicity, the earlier version of the block will not matter. Similar optimizations also handle situations where committed blocks are later unreferenced, such as when a directory gets truncated.

For circular journaling to work correctly, ext4 requires a few invariants. One invariant is of specific interest in ccfs: the number of blocks that can be associated with a transaction is limited by a threshold. To enforce the limit, before modifying each atomic set of metadata structures, ext4 first verifies that the current running transaction (say, T_i) has sufficient capacity left; if not, ext4 starts committing T_i and uses T_{i+1} for the modifications.

3.2 Streams

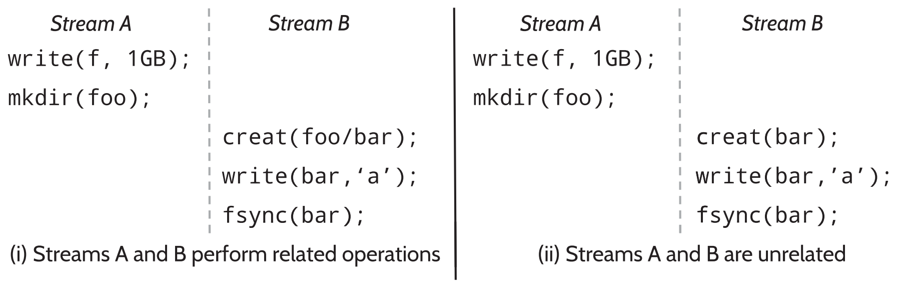
Ccfs introduces a new abstraction called the *stream*. We now first explain the stream abstraction, and then the API for the abstraction.

Stream Abstraction. In ccfs, each application usually corresponds to a single stream. Writes from different streams are re-ordered for performance, while order is preserved within streams for crash consistency. We define the stream abstraction such that it can be easily used in common workflows; as an example, consider a text file $f1$ that is modified by a text editor while a binary file $f2$ is downloaded from the network, and they are both later added to a VCS repository. Initially, the text editor and the downloader must be able to operate on their own streams (say, A and B, respectively), associating $f1$ with A and $f2$ with B. Note that there can be no constraints on the location of $f1$ and $f2$: the user might place them in the same directory. Moreover, the VCS should then be able to operate on another stream C, using C for modifying both $f1$ and $f2$. In such a scenario, the stream abstraction should guarantee the order required for crash consistency, while allowing enough re-ordering for the best performance possible.

Hence, in ccfs, streams are transient and are not uniquely associated with specific files or directories: a file that is modified in one stream might be later modified in another stream. However, because of such flexibility, while each stream can be committed independently without being affected by other streams, it is convenient if the stream abstraction takes special care when two streams perform operations that affect logically related data. For example, consider a directory that is created by stream A, and a file that is created within the directory by stream B; allowing the file creation to be re-ordered after the directory creation (and recovering the file in a *lost+found* directory on a crash) might not make logical sense from an application's perspective. Hence, when multiple streams perform logically related operations, the file system takes sufficient care so the temporal order between those operations is maintained on a crash.

We loosely define the term *related* such that related operations do not commonly occur in separate streams within a short period of time; if they do, the file system might perform inefficiently. For example, separate directory entries in a directory are not considered related (since it is usual for two applications to create files in the same directory), but the creation of a file is considered related to the creation of its parent. Figure 2(i) illustrates two logically related streams, where stream A first creates a directory (*foo*), after which stream B creates a file (*bar*) inside *foo*. When stream B calls an `fsync` on *bar*, the `creat` of *bar* has to be flushed, which requires flushing the `mkdir` of its parent directory, ultimately requiring the 1GB write to *f* in A also to be flushed. In Figure 2(ii), streams A and B are unrelated; thus, neither the `mkdir` of *foo* nor the 1GB write need to be flushed when B calls an `fsync` on *bar*. We leave the exact definition of logically related operations to file systems that implement streams, instead of the stream abstraction itself; in Section 3.5, we discuss the set of logically related operations in ccfs.

Although each application (perhaps containing multiple threads and processes) usually corresponds to a single stream, the abstraction allows for more flexibility; Figure 3 shows different ways



```
      Stream A      |     Stream B          ||      Stream A      |    Stream B
write(f, 1GB);      |                       || write(f, 1GB);     |
mkdir(foo);         |                       || mkdir(foo);        |
                    | creat(foo/bar);       ||                    | creat(bar);
                    | write(bar,'a');       ||                    | write(bar,'a');
                    | fsync(bar);           ||                    | fsync(bar);
```

(i) Streams A and B perform related operations
(ii) Streams A and B are unrelated

Fig. 2. Related and non-related operations across streams. *Panel (i) shows a scenario where streams A and B are related: B creates a file* `bar` *inside the directory* `foo` *created in stream A. In panel (ii), the streams perform similar operations but are unrelated and can commit independently.*

in which streams can be used and shared by applications. In addition to the usual case (Figure 3(a)), each thread in the application can use a different stream (Figure 3(b)) if necessary, a single thread can switch between different streams (Figure 3(c)), and multiple applications can share a stream (Figure 3(d)). We next explain the interface that allows such flexibility.

Stream API. Table 2 shows the stream API. The `setstream(s)` call creates a new stream (if not already existing) and associates the current thread with the stream `s`. All future updates in that thread will be assigned to stream `s`; when forking (a process or thread), a child will adopt the stream of its parent.

With the default `setstream(s)` call, all `fsync` calls issued in the stream will be obeyed. Thus, in this mode, applications will observe similar performance to the case where they run on a re-ordering file system. However, the in-order update semantics of ccfs results in better consistency guarantees.

To improve performance, all *unnecessary* `fsync` calls can be ignored. An `fsync` call is unnecessary if it only intends to enforce ordering but not durability. Since ccfs inherently provides the required ordering, such `fsync` calls can be eliminated from the application. To ease this elimination, the `setstream()` call also takes a flag called `IGNORE_FSYNC` specified as part of the optional *flags* parameter. If the `IGNORE_FSYNC` flag is set, then all `fsync` calls in the stream will be ignored, improving performance while maintaining correctness.

However, some `fsync` calls may be required to enforce durability not just ordering. For example, while SQLite may issue several `fsync` calls to order updates, it also issues `fsync` calls at a few places to enforce durability (e.g., transaction commit). To enforce durability where required, the `streamsync()` call can be utilized; this call flushes all updates in the current stream. Identifying places where `streamsync()` calls are required is straightforward. Developers and users can add these calls at points where durability is desired (e.g., transaction commit points in SQLite). In addition, tools such as Alice can help identify where the `streamsync()` calls are required.

Figure 4 shows how the write-ahead logging protocol (shown earlier in Figure 1) can be made crash consistent atop ccfs. Part (i) shows a variation of the original protocol that works on ext3 and ext4 but might not work atop file systems such as ext2 or btrfs [38]. Part (ii) shows the changes required to make this protocol crash consistent atop ccfs while providing the same performance as the one shown in part (i). To improve performance atop ccfs, one can use the `IGNORE_FSYNC` flag and issue a `streamsync()` call for durability at the end of the protocol as shown in part (iii). Note that issuing the `streamsync()` call at any point after the journal write ensures durability.

The stream API also provides a `getstream()` call that is used, for example, to find if the current process is operating on the *init* stream (explained in Section 3.6) or a more specific stream. The API usage is further explained in Section 3.6.

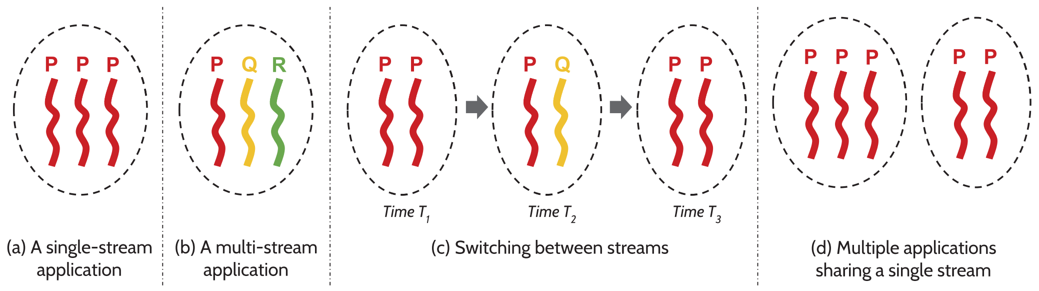

P P P
(a) A single-stream application
P Q R
(b) A multi-stream application
P P
Time T_1
P Q
Time T_2
P P
Time T_3
(c) Switching between streams
P P P
P P
(d) Multiple applications sharing a single stream

Fig. 3. Using Streams. *The figure shows different ways in which streams can be used and shared. A dotted oval represents an application consisting of multiple threads (or processes), and each wavy line within the oval represents a thread. Panel (a) shows the case where all threads of an application share a single stream, P. Panel (b) shows the case where each thread of an application uses a different stream. Panel (c) shows a scenario where a thread switches between two streams P and Q. Panel (d) shows the case where multiple applications share the same stream, P.*

Table 2. Stream API

Call	Description
`setstream(id, flags)`	Creates the stream `id` and associates the calling thread with it. The `flags` parameter is optional. If the `IGNORE_FSYNC` flag is set, then all `fsync` calls issued in the stream will be ignored.
`getstream()`	Gets the stream identifier of the calling thread. This call is used to check whether a stream is already set.
`streamsync()`	Flushes all updates in the stream.

3.3 Separating Multiple Streams

In ccfs, the basic idea used to separately preserve the order of each stream is simple: ccfs extends the journaling technique to maintain multiple in-memory running transactions, one corresponding to each stream. Whenever a synchronization system call (such as `fsync`) is issued, only the corresponding stream's running transaction is committed. All modifications in a particular stream are associated with that stream's running transaction, thus maintaining order within the stream (optimizations regarding this are discussed in the next section).

Using multiple running transactions poses a challenge: committing one transaction without committing others (i.e., re-ordering between streams) inherently re-orders the metadata modified across streams. However, internal file-system consistency relies on maintaining a global order between metadata operations; indeed, this is the original purpose of ext4's journaling mechanism. It is hence important that metadata modifications in different streams be logically independent and be separately associated with their running transactions. We now describe the various techniques that ccfs uses to address this challenge while retaining the existing optimizations in ext4.

3.3.1 Hybrid-Granularity Journaling. The journaling mechanism described previously (Section 3.1) works at block-granularity: entire blocks are associated with running transactions, and committing a transaction records the modified contents of entire blocks. Ccfs uses *hybrid-granularity journaling*, where byte-ranges (instead of entire blocks) are associated with the running transaction, but transactional commits and checkpointing still happen at the granularity of blocks.

Ccfs requires byte-granularity journaling, because separate metadata structures modified by different streams might exist in the same file-system block. For example, a single block can contain

```
creat(jrnl);
write(jrnl, "<offset>,
        <size>,<data>");
fsync(jrnl);
write(dbfile, offset,
        data);
fsync(dbfile);
unlink(jrnl);
```

(i) Works on ext3 and ext4 but not ext2 or btrfs

```
setstream("wal")
creat(jrnl);
write(jrnl, "<offset>,
        <size>,<data>");
fsync(jrnl);
write(dbfile, offset,
                data);
fsync(dbfile);
unlink(jrnl);
```

(ii) Same performance atop ccfs

```
setstream("wal",IGNORE_FSYNC)
creat(jrnl);
write(jrnl, "<offset>,
        <size>,<data>");
fsync(jrnl); // ignored
write(dbfile, offset, data);
fsync(dbfile); // ignored
unlink(jrnl);
streamsync();
```

Fig. 4. Crash-consistent write-ahead logging protocol atop ccfs. *Panel (i) shows a slight modification of Figure 1 that works correctly on some versions of ext4 and ext3, but might not on other file systems (such as ext2 or btrfs). Panel (ii) shows the change (a single* `setstream()` *call) required to the previous figure for guaranteeing correctness using streams while achieving similar performance. Panel (iii) shows the changes required to improve performance using streams.*

the inode structure for two files used by different applications; in block-granularity journaling, it is not possible to associate the inodes with the separate running transactions of two different streams.

Block-granularity journaling allows many optimizations that are not easily retained in byte-granularity. A major optimization affected in ext4 is data coalescing during checkpoints: even if multiple versions of a block are committed, only the final version is sent to its in-place location. Since the Linux buffer cache and storage devices manage data at block granularity, such coalescing becomes complicated with a byte-granularity journal.

To understand hybrid-granularity journaling, consider the example illustrated in Figure 5. In this example, block X initially contains the bytes $\langle a_0 b_0 \rangle$. Before allowing any writes, ccfs makes an in-memory copy (say, X^0) of the initial version of the block. Let the first byte of X be modified by stream A into a_1; ccfs will associate the byte range X_{0-0} with the running transaction T_A of stream A (X_{i-j} denotes the ith to jth bytes of block X), thus following byte-granularity. Let stream B then modify the second byte into b_1, associating X_{1-1} with T_B; the final in-memory state of X will be $\langle a_1 b_1 \rangle$. Now, assume the user calls `fsync` in stream B, causing T_B to commit (T_A is still running). Ccfs converts T_B into block-granularity for the commit, by super-imposing the contents of T_B (i.e., X_{1-1} with the content b_1) on the initial versions of their blocks (i.e, X^0 with content $\langle a_0 b_0 \rangle$), and committing the result (i.e., $\langle a_0 b_1 \rangle$). When T_B starts committing, it updates X^0 with the value of X that it is committing. If the user then calls `fsync` in A, then X_{0-0} is super-imposed on X^0 ($\langle a_0 b_1 \rangle$), committing $\langle a_1 b_1 \rangle$.

Thus, hybrid-granularity journaling performs in-memory logging at byte-granularity, allowing streams to be separated; the delayed-logging optimization of ext4 is unaffected. Commits and checkpoints are block-granular, thus preserving delayed checkpointing.

3.3.2 Delta Journaling. In addition to simply associating byte ranges with running transactions, ccfs allows associating the exact changes performed on a specific byte range (i.e., the *deltas*). This technique, which we call *delta journaling*, is required when metadata structures are actually shared between different streams (as opposed to independent structures sharing the same block). For

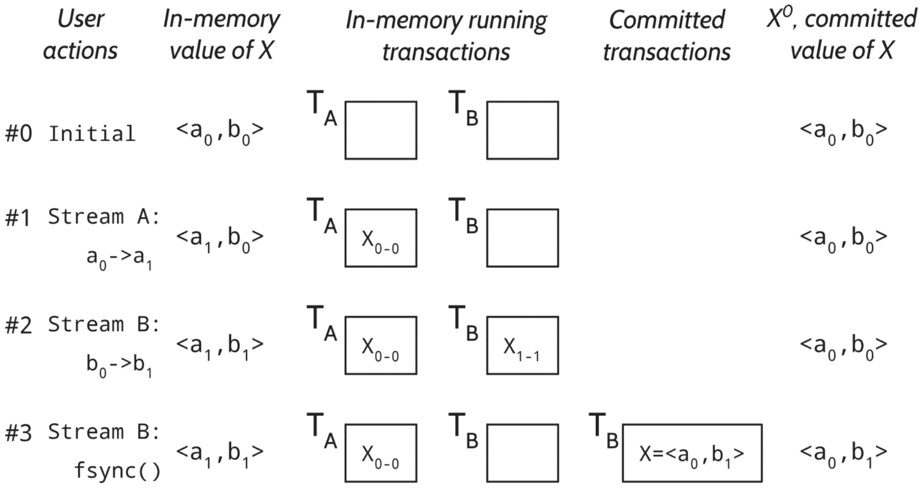


Fig. 5. Hybrid-granularity Journaling. *Timeline showing hybrid-granularity journaling in ccfs. Block X initially contains the value $\langle a_0, b_0\rangle$, T_A and T_B are the running transactions of streams A and B; when B commits, X is recorded at the block level on disk.*

example, consider a metadata structure tracking the total free space in the file system: all streams need to update this metadata structure.

Delta journaling in ccfs works as follows. Assume that the byte range X_{1-2} is a shared metadata field storing an integer, and that stream A adds i to the field and stream B subtracts j from the field. Ccfs associates the delta $\langle X_{1-2} : +i\rangle$ to the running transaction T_A and the delta $\langle X_{1-2} : -j\rangle$ to the running T_B. When a transaction commits, the deltas in the committing transaction are imposed on the initial values of their corresponding byte ranges, and then the results are used for performing the commit. In our example, if X_{1-2} initially had the value k, and stream B committed, the value $(k - j)$ will be recorded for the byte range during the commit; note that hybrid-granularity journaling is still employed, that is, the commit will happen at block-granularity.

In ext4, shared metadata structures requiring delta journaling are the *free inode count* and the *free block count*, which concern the global state across the file system. Delta journaling is also needed for the *nlink* and the modification time fields of directory inodes, since multiple streams can modify the same directory.

3.3.3 Pointer-less Data Structures. Metadata in file systems often use data structures such as linked lists and trees that contain pointers, and these cause metadata operations in one stream to update pointers in structures already associated with another stream. For example, deleting an entry in a linked list will require updating the *next* pointer of the previous entry, which might be associated with another stream. Ccfs eliminates the need to update pointers across streams by adopting alternative data structures for such metadata.

Ext4 has two metadata structures that are of concern: directories and the *orphan list*. Directories in ext4 have a structure similar to linked lists, where each entry contains the relative byte-offset for the next entry. Usually, the relative offset recorded in a directory entry is simply the size of the entry. However, to delete a directory entry d_i, ext4 adds the size of d_i to the offset in the previous entry (d_{i-1}), thus making the previous entry point to the next entry (d_{i+1}) in the list. To make directories pointer-less, ccfs replaces the offset in each entry with a *deleted* bit: deleting an entry sets the bit. The insert and scan procedures are modified appropriately; for example, the insert procedure recognizes previously deleted entries in the directory and uses them for new entries if possible.

The orphan list in ext4 is a standard linked list containing recently freed inodes and is used for garbage collecting free blocks. The order of entries in the list does not matter for its purposes in ext4. We convert the orphan list into a pointer-less structure by substituting it with an orphan directory, thus reusing the same data structure.

3.3.4 Order-Less Space Reuse. Ccfs carefully manages the allocation of space in the file system such that re-ordering deallocations between streams does not affect file-system consistency. For example, assume stream A deletes a file and frees its inode, and stream B tries to create a file. The allocation routines in ext4 might allocate to B the inode that was just freed by A. However, if B commits before A, and then a crash occurs, the recovered state of the file system will contain two unrelated files assigned the same inode.

Ext4 already handles the situation for block allocation (for reasons of security) by reusing blocks only after the transaction that frees those blocks has fully committed. In ccfs, we extend this solution to both inode and directory-entry reuse. Thus, in our example, B will reuse A's freed inode only if A has already been committed.

3.4 Maintaining Order Within Streams

We saw in the previous section how to separate dependencies across independent streams; we now focus on ordering the updates within the same stream. Ext4 uses metadata-only journaling: ext4 can re-order file appends and overwrites. Data journaling, that is, journaling all updates, preserves application order for both metadata and file data, but significantly reduces performance, because it often writes data twice. A hybrid approach, selective data journaling (SDJ) [6], preserves order of both data and metadata by journaling only overwritten file data; it only journals the block pointers for file appends. Since modern workloads are mostly composed of appends, SDJ is significantly more efficient than journaling all updates.

We adopt the hybrid SDJ approach in ccfs. However, the approach still incurs noticeable overhead compared to ext4's default journaling under practical workloads, because it disables a significant optimization, *delayed allocation*. In our experiments with ext4 (Linux 3.13) on a Toshiba MK1665GSX 160GB HDD, the createfiles benchmark results in 8,795ops/s with delayed allocation, and 7,730ops/s without (12% overhead).

Without delayed allocation, whenever an application appends to files, data blocks are allocated and block pointers are assigned to the files immediately, as shown in the second column of Figure 6. With delayed allocation (third column), the file system does not immediately allocate blocks; instead, allocations for multiple appends are delayed and done together. For order to be maintained within a stream, block pointers need to be assigned immediately (for example, with SDJ, only the order of allocations is preserved across system crashes): naive delayed allocation inherently violates order.

Ccfs uses a technique that we call *order-preserving delayed allocation* to maintain program order while allowing delayed allocations. Whenever a transaction T_i is about to commit, all allocations (in the current stream) that have been delayed so far are performed and added to T_i before the commit; further allocations from future appends by the application are assigned to T_{i+1}. Thus, allocations are delayed until the next transaction commit, but not across commits. Since order is maintained within T_i via the atomicity of all operations in T_i, the exact sequence in which updates are added to T_i does not matter, and thus the program order of allocations is preserved.

However, the running transaction's size threshold poses a challenge: at commit time, what if we cannot add all batched allocations to T_i? Ccfs solves this challenge by reserving the space required

System calls	No delayed allocation	Order-violating delayed alloc	Order-preserving delayed alloc
`write(f1, 1);`	`alloc(f1, 1);`		
`write(f2, 1);`	`alloc(f2, 1);`		
`write(f1, 1);`	`alloc(f1, 1);`		
`write(f2, 1);`	`alloc(f2, 1);`		
`fsync(f2);`		`alloc(f2, 2);`	`atomic{alloc(f1, 2) alloc(f2, 2)};`

Fig. 6. Order-preserving Delayed Allocation. *Timeline of allocations performed, corresponding to a system-call sequence.*

for allocations when the application issues the appends. Order-preserving delayed allocation thus helps ccfs achieve ext4's performance while maintaining order. For the createfiles benchmark, the technique achieves 8,717ops/s in ccfs, and thus performs similar to the default configuration of ext4 (8,795ops/s).

3.5 Implementation

Ccfs changes 4,500 lines of source code (original ext4 total: 50,000 lines). Of these, preserving order (Section 3.4) required only 500 lines while implementing multiple streams (Section 3.3) was more complicated and involved the rest of the changes. Overall, most of the changes (3,000 lines) are related to the journaling code within ext4. We now describe our implementation.

Related Operations Across Streams. The current version of ccfs considers the following operations as logically related: modifying the same regular file, explicitly modifying the same inode attributes (such as the owner attribute), updating (creating, deleting, or modifying) directory entries of the same name within a directory, and creating a directory and any directory entries within that directory. To understand how ccfs maintains temporal ordering between related operations from different streams, consider that stream A first performs operation O_A at time t_1 and stream B then performs a related operation O_B at t_2. If stream A gets committed between t_1 and t_2 (either due to an `fsync` or a periodic background flush),then the required temporal order is already maintained, since O_A is already on disk before O_B is performed. If not, then ccfs temporarily merges the streams together and treats them as one, until the merged streams get committed to disk; the streams are then separated and allowed to proceed independently.

Maintaining Order Within Streams. An implementation challenge for order-preserving delayed allocation is that the allocations need to be performed when a transaction is about to commit, but before the actual committing starts. We satisfy these requirements without much complexity by performing the allocations in the `T_LOCKED` state of the transaction, a transient state in the beginning of every commit when all file-system updates are blocked. A more efficient implementation could perform these allocations before the `T_LOCKED` state.

To correctly maintain the order of file updates, SDJ requires careful handling when data is both appended and overwritten on the same block. For example, consider an append when T_i was running and an overwrite when T_i is committing (when T_{i+1} is running); to maintain order, two versions of the block must be created in memory: the old version (that does not contain the overwrite) must be used as part of T_i's commit, and the new version must be journaled in T_{i+1}. Ccfs handles these cases correctly.

3.6 Discussion

We now discuss how we expect applications to use streams. Overall, the abstraction is flexible: while we expect most applications to use a single stream, if needed, applications can also use separate streams for individual tasks, or multiple applications can share a single stream. In the current version of ccfs, the *init* process is assigned an *init* stream; hence, all applications inherit this stream by default. We expect most applications whose write performance is user visible to issue a single `setstream()` call in the beginning of the application (but to not make any other code changes). Thus, applications by default will have improved crash consistency, and applications issuing `setstream()` will have both improved consistency and high performance. If so desired, then applications can also significantly improve their performance (while maintaining consistency) by first setting the `IGNORE_FSYNC` flag and removing any `O_SYNC` flags, and issuing `streamsync()` calls only when durability is actually desired.

4 EVALUATION

In our evaluation, we first confirm our ordering hypothesis by answering the following question: does the ordering guarantee provided by ccfs improve application crash consistency?

Next, we confirm our efficiency hypothesis by answering the following questions:

- Does ccfs effectively use streams to eliminate the overhead of write dependencies?
- How does ccfs perform in standard file system benchmarks run in a single stream?
- What is the performance effect of maintaining order on real applications?

Finally, we analyze how much developer effort is needed to make an application crash consistent atop ccfs.

We performed a set of experiments to answer these questions. For the experiments, we use an Intel Core 2 Quad Processor Q9300 with 4GB of memory running Linux 3.13, with either an SSD (Samsung 840 EVO 500GB) or a HDD (Toshiba MK1665GSX 160GB).

4.1 Reliability

We first examine whether the in-order semantics provided by ccfs improves application crash consistency compared to the widely used ext4 file system (which re-orders writes). We follow a model-based testing strategy to check application consistency on both file systems using the Alice tool [38]. The tool records the system-call trace for a given application workload, and then uses a file-system model to reproduce the possible set of file-system states if a system crash occurs. We configured Alice with the models of ext4 (model provided with the tool) and ccfs (system calls are weakly atomic and in-order). We tested five applications previously reported [38] to exhibit crash inconsistencies on ext4: LevelDB, SQLite, Git, Mercurial, and ZooKeeper. We use workloads similar to the previous study, but newer versions of the applications; we do not check durability in Git and Mercurial, since they never call `fsync`.

The results of our testing are shown in Table 3(a). Ext4 results in multiple inconsistencies: LevelDB fails to maintain the order in which key-value pairs are inserted, Git and Mercurial can result in repository corruption, and ZooKeeper may become unavailable. With ccfs, the only inconsistencies were with Mercurial. These inconsistencies are exposed on a process crash with any file system, and therefore also occur during system crashes in ccfs; they result only in *dirstate corruption*, which can be manually recovered from and is considered to be of minor consequence [29]. Thus, our model-based testing reveals that applications are significantly more crash consistent on ccfs than ext4.

Table 3. Consistency Testing

Application	ext4	ccfs
LevelDB	1	0
SQLite-Roll	0	0
Git	2	0
Mercurial	5	2
ZooKeeper	1	0

(a) Vulnerabilities found

Application		ext4	ccfs
LevelDB	Images	158 / 465	427 / 427
	Time (s)	24.31 / 30	30 / 30
Git	Images	84 / 112	96 / 96
	Time (s)	9.95 / 40	40 / 40

(b) Consistent post-reboot disk states produced by BoB

The first table shows the results of model-based testing using Alice, and the second shows experimental testing with BoB. Each vulnerability reported in the first table is a location in the application source code that has to be fixed. The Images*rows of the second table show the number of disk images reproduced by the BoB tool that the application correctly recovers from; the* Time*rows show the time window during which the application can recover correctly from a crash (x / y: x is the time window, y is the total workload runtime, both in seconds). For Git, we consider the default configuration instead of a safer configuration with bad performance (Section 4.4).*

We used the BoB tool [38] to test whether our implementation of ccfs maintains weak atomicity and ordering, that is, whether the implementation reflects the model used in the previous testing. BoB records the block-level trace for an application workload running on a file system, and reproduces a subset of disk images possible if a crash occurs. BoB generates disk images by persisting blocks in and out of order; each image corresponds to a time window during the runtime where a crash will result in the image. These windows are used to measure how much time the application remains consistent.

We used Git and LevelDB to test our implementation and compare it with ext4; both have crash vulnerabilities exposed easily on a re-ordering file system. Table 3(b) shows our results. With ext4, both applications can easily result in inconsistency. LevelDB on ext4 is consistent only on 158 of the 465 images reproduced; a system crash can result in being unable to open the datastore after reboot, or violate the order in which users inserted key-value pairs. Git will not recover properly on ext4 if a crash happens during 30.05s of the 40s runtime of the workload. With ccfs, we were unable to reproduce any disk state in which LevelDB or Git are inconsistent. We conclude that our implementation provides the desired properties for application consistency.

Thus, our results show that the in-order semantics provided by ccfs noticeably improves the state of application crash consistency, confirming our ordering hypothesis. We next validate our efficiency hypothesis by evaluating whether ccfs provides this improved consistency with good performance.

4.2 Multi-Stream Benefits

Maintaining order causes write dependence during `fsync` calls and imposes additional overheads, since each `fsync` call must flush all previous dirty data. In the simplest case, this results in additional `fsync` latency; it can also prevent writes from being coalesced across `fsync` calls when data is overwritten, and prevent writes from being entirely avoided when the previously written data is deleted. We now evaluate if using separate streams in ccfs prevents these overheads.

We devised three microbenchmarks to study the performance effects of preserving order. The *append* microbenchmark appends a large amount of data to file *A*, then writes 1 byte to file *B* and calls `fsync` on *B*; this stresses the `fsync` call's latency. The *truncate* benchmark truncates file *A* after calling `fsync` while *overwrite* overwrites *A* after the `fsync`; these benchmarks stress whether or not writes are avoided or coalesced.

Table 4. Single-fsync Experiments

Micro Benchmark	**File system**	fsync **latency (s)**	fsync **written (MB)**	**Total written (MB)**
Append	ext4	0.08	0.03	100.19
	ccfs-1	1.28	100.04	100.18
	ccfs-2	0.08	0.03	100.20
Truncate	ext4	0.07	0.03	0.18
	ccfs-1	1.28	100.04	100.21
	ccfs-2	0.05	0.03	0.20
Overwrite	ext4	0.08	0.03	100.19
	ccfs-1	1.27	100.04	300.72
	ccfs-2	0.07	0.03	100.20

fsync*latencies in the first column correspond to the data written by the* fsync*shown in the second column on HDD, while the total data shown in the third column affects the available device bandwidth and hence performance in more realistic workloads. In the ccfs-1 configuration, updates to both files A and B are performed in a single stream. In the ccfs-2 configuration, Modifications to A and B are performed in two separate streams. The numbers reported are the average over 10 runs.*

We use two versions of each benchmark. In the simpler version, we write 100MB of data in file *A* and measure the latency of the fsync call and the total data sent to the device. In another version, a foreground thread repeatedly writes *B* and calls fsync every 5s; a background thread continuously writes to *A* at 20MB/s, and may truncate *A* or overwrite *A* every 100MB, depending on the benchmark. The purpose of the multi-fsync version is to understand the distribution of fsync latencies observed in such a workload.

We ran the benchmarks on three file-system configurations: ext4, which re-orders writes and does not incur additional overheads, ccfs using a single stream (ccfs-1), and ccfs with modifications of *A* and *B* in separate streams (ccfs-2). Table 4 and Figure 7 show our results.

For the append benchmark, in ext4, the fsync completes quickly in 0.08s, since it flushes only *B*'s data to the device. In ccfs-1, the fsync sends 100MB and takes 1.28s, but ccfs-2 behaves like ext4, since *A* and *B* are modified in different streams. Repeated fsync follows the same trend: most fsync calls are fast in ext4 and ccfs-2 but often take more than 1s in ccfs-1. A few fsync calls in ext4 and ccfs-2 are slow due to interference from background activity by the page-flushing daemon and the periodic journal commit.

With truncates, ext4 and ccfs-2 never send file *A*'s data to disk, but ccfs-1 sends the 100MB during fsync, resulting in higher latency and more disk writes. Most repeated fsync calls in ext4 and ccfs-2 are fast, as expected; they are slow in ccfs-1, but still quicker than the append benchmark, because the background thread would have just truncated *A* before some of the fsync.

With overwrites, in both ext4 and ccfs-2, only the final version of *A*'s data reaches the disk: in ccfs-2, SDJ (selective data journaling) considers the second modification of *A* an append, because the first version of *A* is not yet on disk (this still maintains order). In ccfs-1, the first version is written during the fsync, and then the second version (overwrite) is both written to the journal and propagated to its actual location, resulting in 300MB of total disk writes. Repeated fsync calls are slow in ccfs-1 but quicker than previous benchmarks because of fewer disk seeks: with this version of the benchmark, since *A* is constantly overwritten, data is only sent to the journal in ccfs-1 and is never propagated to its actual location.

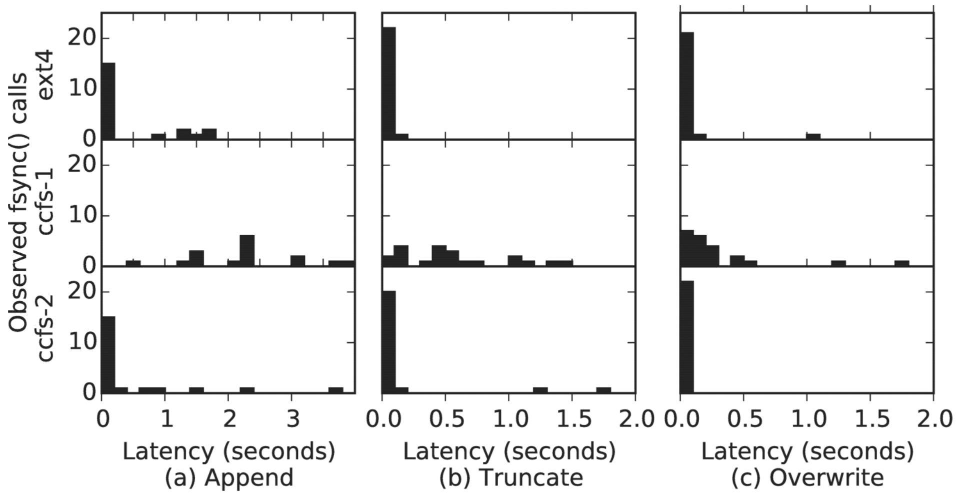


Fig. 7. Repeated fsync Experiments. *Histogram of user-observed foreground latencies in our multi- fsync experiments. Each experiment is run for two minutes on an HDD. We bucketize the latencies in 200ms buckets in panel (a) and 100ms buckets in panels (b) and (c).*

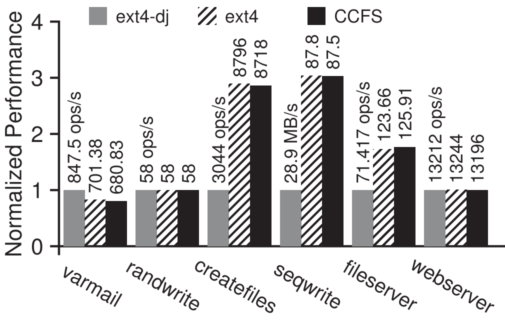
These results show that ccfs is effective at avoiding write dependence overheads when multiple streams are used (in comparison to a file system providing global order). The results also show that, within a stream, write dependence can cause noticeable overhead. For certain applications, therefore, it is possible that dividing the application into multiple streams is necessary for performance. The subsequent sections show that the majority of the applications do not require such division.

4.3 Single-Stream Overheads

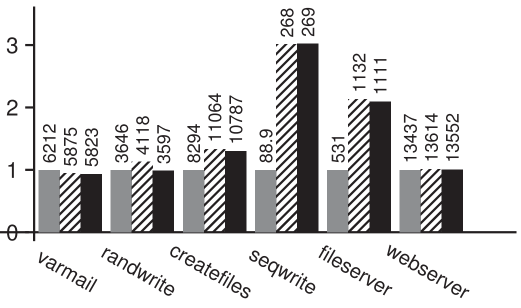
The previous experiments show how ccfs avoids the performance overheads across streams; we now focus on performance within a stream. The performance effects of maintaining order within a stream are affected by false dependencies between updates within the stream, and hence depend significantly on the pattern of writes. We perform our evaluation using the Filebench [13, 55] suite that reflects real-world workload patterns and microbenchmarks, and compare performance between ext4 (false dependencies are not exposed) and ccfs (false dependencies are exposed because of ordering within streams). Another source of overhead within streams is the disk-level mechanism used to maintain order, that is, the SDJ technique used in ccfs. Hence, we compare performance between ext4 (no order), ccfs (order-preserving delayed allocation and SDJ), and ext4 in the *data=journal* mode (*ext4-dj*, full data journaling). We compare performance both with an HDD (disk-level overheads dominated by seeks) and an SSD (seeks less pronounced).

The overall results are shown in Figure 8; performance is most impacted by overwrites and fsync calls. We now explain the results obtained on each benchmark.

The *varmail* benchmark emulates a multithreaded mail server, performing file creates, appends, deletes, reads, and fsync calls in a single directory. Since each append is immediately followed by an fsync, there is no additional write dependence due to ordering. Performance is dominated by seek latency induced by the frequent fsync calls, resulting in similar performance across ext4 and ccfs. Ext4-dj issues more writes but incurs fewer seeks (since data is written to the journal rather



(a) HDD Performance



(b) SSD Performance

	Writes (*KB*/*op*)			CPU (*μs*/*op*)	
	ext4-dj	**ext4**	**ccfs**	**ext4**	**ccfs**
varmail	3.42	2.91	2.98	59.9	67.2
randwrite	8.10	4.10	8.10	16.8	24.4
createfiles	12.22	5.53	5.49	89.1	94.4
seqwrite	2.00	1.00	1.00	0.9	2.4
fileserver	1093.00	321.00	327.00	1040.0	2937.0
webserver	0.49	0.24	0.15	74.4	75.5

(c) HDD Writes and CPU usage

Fig. 8. Imposing Order at Disk-Level: Performance, Data Written, and CPU usage. *Panels (a) and (b) show throughput under standard benchmarks for ccfs, ext4, and ext4 under the data=journal mode (ext4-dj), all normalized to ext4-dj. Panel (c) shows the total writes and CPU usage with an HDD. Varmail emulates a multithreaded mail server, performing file creates, appends, deletes, reads, and* fsync *in a single directory. Randwrite does 200,000 random writes (each of 4KB in size) over a 10GB file with an* fsync *every 100 writes. Webserver emulates a multithreaded web server performing open-read-close on multiple files and a log file append. Createfiles uses 64 threads to create 1M files, each of size 16KB. Seqwrite writes 32GB to a new file (1KB is considered an operation in panel (c)). Fileserver emulates a file server, using 50 threads to perform creates, deletes, appends, and reads, on 80K files. The fileserver, varmail, and webserver workloads were run for 300s. The numbers reported are the average over 10 runs.*

than the in-place location during each `fsync`), and performs 20% better in the HDD and 5% better in the SSD.

Randwrite overwrites random locations in an existing file and calls `fsync` every 100 writes. Since the `fsync` calls always flush the entire file, there is no additional write dependence due to ordering. However, the overwrites cause both ccfs (SDJ) and ext4-dj (full journaling) to write twice as much data as ext4. On the HDD, all file systems perform similarly, since seeks dominate performance; in the SSD, additional writes cause a 12% performance decrease for ccfs and ext4-dj.

Createfiles and *seqwrite* keep appending to files, while *fileserver* issues appends and deletes to multiple files; they do not perform any overwrites or issue any `fsync` calls. Since only appends are involved, ccfs writes the same amount of data as ext4. On the HDD, similar performance is observed in ccfs and in ext4. Under SSDs, createfiles is 4% slower atop ccfs because of delayed allocation in the `T_LOCKED` state, which takes a noticeable amount of time (an average of 132 ms during each commit); this is an implementation artifact and can be optimized. For all these benchmarks, ext4-dj writes data twice and hence is significantly slower. *Webserver* involves mostly reads and a few appends; performance is dominated by reads, so all file systems perform similarly.

Figure 8(c) compares the CPU usage of ccfs and ext4. For most workloads, our current implementation of ccfs has moderately higher CPU usage; the significant usage for *fileserver* and *seqwrite* is because the workloads are dominated by block allocations and de-allocations, which are especially CPU-intensive for our implementation. This can be improved by adopting more optimized structures and lookup tables (Section 3.5). Thus, while it does not noticeably impact performance in our experiments, reducing CPU usage is an important future goal for ccfs.

Overall, our results show that maintaining order does not incur any inherent performance overhead for standard workloads when the workload is run in one stream. False dependencies are rare and have little impact for common workloads, and the technique used to maintain order within streams in ccfs is efficient.

4.4 Case Studies

Our evaluation in the previous section shows the performance effects of maintaining order for standard benchmarks. We now consider three real-world applications: Git, LevelDB, and SQLite with rollback journaling; we focus on the effort required to maintain crash consistency with good performance for these applications in ccfs and the default mode (*data=ordered*) of ext4. For ext4, we ensure that the applications remain consistent by either modifying the application to introduce additional `fsync` calls or using safe application configuration options. All three applications are naturally consistent on ccfs when run on a single stream.

Single-Application Performance. We first ran each application in its own stream in the absence of other applications, to examine if running the application in one stream is sufficient for good performance (as opposed to dividing a single application into multiple streams). Specifically, we try to understand if the applications have false dependencies. We also consider their performance when `fsync` calls are omitted without affecting consistency (including user-visible durability) on ccfs.

The results are shown in Table 5. For Git, we use a workload that adds and commits the Linux source code to an empty repository. While Git is naturally consistent atop ccfs, it requires a special option (*fsyncobjectfiles*) on ext4; this option causes Git to issue many `fsync` calls. Irrespective of this option, Git always issues 242MB of appends and no overwrites. In ccfs, the 242MB is sent directly to the device and the workload completes in 28.9s. In ext4, the `fsync` calls needed for correctness prevent updates to metadata blocks from being coalesced; for example, a block bitmap that is repeatedly updated by the workload needs to be written to the journal on every `fsync`. Moreover, each `fsync` call forces a separate journal transaction, writing a separate descriptor block and commit block to the disk and causing two disk cache flushes. Thus, in ext4, the workload results in 1.4GB of journal commits and takes 2,294s to complete (80× slower).

For SQLite, we insert 2,000 rows of 120 bytes each into an empty table. SQLite issues `fsync` calls frequently, and there are no false dependencies in ccfs. However, SQLite issues file overwrites (31.83MB in this workload), which causes data to be sent to the journal in ccfs. Sending the overwritten data to the journal improves the performance of ccfs in comparison to ext4 (1.28×). Because SQLite frequently issues an `fsync` after overwriting a small amount (4KB) of data, ext4 incurs a seek during each `fsync` call, which ccfs avoids by writing the data to the journal. SQLite can also be heavily optimized when running atop ccfs by omitting unnecessary `fsync` calls; with our workload, this results in a 685× improvement.

For LevelDB, we use the fillrandom benchmark from the db_bench tool to insert 250K key-value pairs of 1,000 bytes each to an empty database. Atop ext4, we needed to add additional `fsync` calls to improve the crash consistency of LevelDB. LevelDB on ccfs and the fixed version on ext4 have similar write avoidance, as can be seen from Table 5. Since LevelDB also does few file overwrites,

Table 5. Case Study: Single-Application Performance

		Throughput		User-level Metrics			Disk-level Metrics		
								Data (MB)	
				fsync()	*Append (MB)*	*Overwrite(kB)*	*Flushes*	*Journal*	*Total*
Git	ext4	17	*files/s*	38599	242.00	0.00	77198	1423.000	1887
	ccfs	1351		0	242.00	0.00	10	18.000	243
	ccfs+	1351		0	242.00	0.00	10	18.000	243
SQLite	ext4	5.23	*ops/s*	6000	31.56	31.83	12000	70.000	170
	ccfs	6.71		6000	31.56	31.83	12000	117.000	176
	ccfs+	4598.00		0	0.32	0.00	0	0.000	0
LevelDB	ext4	5.25	*MB/s*	598	1087.00	0.01	1196	16.300	1131
	ccfs	5.10		523	1087.00	0.00	1046	16.200	1062
	ccfs+	25.50		0	199.00	0.00	2	0.074	157

The table shows the performance and observed metrics of Git, LevelDB, and SQLite-rollback run separately under different file-system configurations on HDD. ccfs+ denotes running ccfs with unnecessary `fsync`*calls omitted; in both ccfs configurations, the application tuns in a single stream. The user-level metrics characterize each workload; "appends" and "overwrites" show how much appended and overwritten data needs to be flushed by* `fsync`*calls (and also how much remain buffered when the workload ends). Overhead imposed by maintaining order will be observed by* `fsync`*calls in the ccfs configuration needing to flush more data. The disk-level metrics relate the characteristics to actual data written to the device.*

it performs similarly on ccfs and ext4. With ccfs, existing `fsync` calls in LevelDB can be omitted, since ccfs already provides ordering, increasing performance 5×.

Thus, the experiments suggest that false-dependency overheads are minimal within an application. In two of the applications, the ordering provided by ccfs can be used to omit `fsync` calls to improve performance.

Multiple-Application Performance. We next test whether ccfs is effective in separating streams: Figure 9 shows the performance when running Git and SQLite simultaneously. The situation in current real-world deployments is exemplified by the *ext4-bad* configuration in Figure 9: both applications are run on ext4, but Git runs without the *fsyncobjectfiles* option (i.e., consistency is sacrificed). The *ccfs-2* configuration is the intended use case for ccfs: Git and SQLite are in separate streams on ccfs, achieving consistency while performing similarly to *ext4-bad*. (SQLite performs better under *ccfs-2*, because ccfs sends some data to the journal and reduces seeks, as explained previously.) Thus, ccfs achieves real-world performance while improving correctness.

The *ccfs-1* configuration demonstrates the overhead of global order by running Git and SQLite in the same stream on ccfs; this is *not* the intended use case of ccfs. This configuration heavily impacts SQLite's performance because of (false) dependencies introduced from writes issued by Git. Running applications in separate streams can thus be necessary for acceptable performance.

The *ext4* configuration reiterates previous findings: it maintains correctness using Git's *fsyncobjectfiles* on ext4, but Git is unacceptably slow due to `fsync` calls. The *ccfs+* configuration represents a secondary use case for ccfs: it runs the applications in separate streams on ccfs with unneeded `fsync` calls omitted, resulting in better SQLite performance (Git is moderately slower, since SQLite uses more disk bandwidth). Thus, running each application in its stream achieves correctness with good performance, while global order achieves correctness but reduces performance.

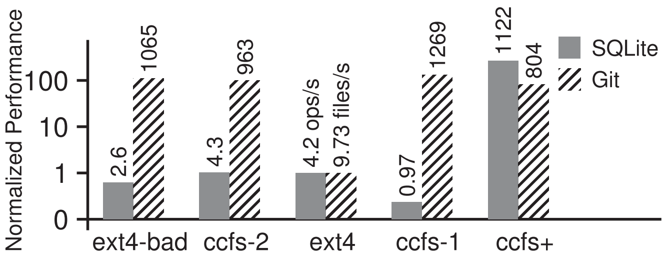


Fig. 9. Case Study: Multiple-Application Performance. *The figure shows the performance of Git and SQLite-rollback run simultaneously under different configurations on HDD, normalized to performance under the* `ext4` *configuration (higher is better). The* `ext4-bad` *configuration runs the applications on ext4, with consistency sacrificed in Git.* `ccfs-2` *uses separate streams for each application on ccfs.* `Ext4` *uses ext4 with consistent Git.* `ccfs-1` *runs both applications in the same stream on ccfs.* `ccfs+` *runs applications in separate streams without unnecessary* `fsync` *calls. Workload: Git adds and commits a repository 25 times the size of Linux; SQLite repeatedly inserts 120-byte rows until Git completes.*

Overall, all our performance experiments show that ccfs provides in-order update semantics while achieving performance similar to a re-ordering file system, confirming our efficiency hypothesis.

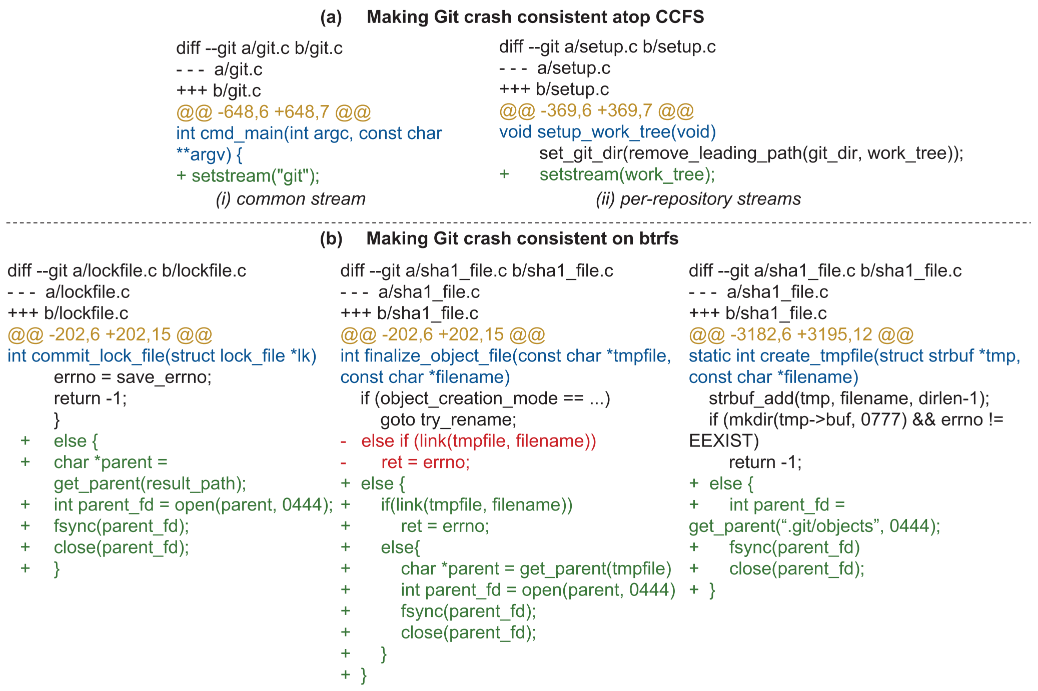
4.5 Developer Effort

We now discuss the developer effort required to make an application crash consistent atop ccfs and other re-ordering file systems. Figure 10(a(i)) shows the code changes needed to make Git crash consistent atop ccfs. Achieving correctness atop ccfs (while maintaining performance) required negligible developer overhead: we added a single line to the main function that sets the stream using the `setstream()` call. Similarly, for all other applications, we added one `setstream()` call to the beginning of each application, without examining the applications any further. To omit unnecessary `fsync` calls in ccfs and improve performance (i.e., for the ccfs+ configuration), we used the `IGNORE_FSYNC` flag on the `setstream()` calls, and added `streamsync()` calls where durability is desired. Identifying such locations for `streamsync()` calls was straightforward: we added them to places in the code where the user is guaranteed durability (i.e., transaction commit points). Specifically, we added the `streamsync()` calls in one and two places in LevelDB and SQLite, respectively.

Note that the change shown in Figure 10(a(i)) makes all Git processes share the same stream, since the same string ("git") is used to initialize the stream. Assigning Git processes that work on different repositories to different streams is also straightforward: one can simply use the repository path as the stream identifier, as shown in Figure 10(a(ii)). We note here that we do not discuss in detail the security issues associated with using simple stream identifiers. For example, an application could affect another application's performance by deliberately using the same identifier. However, one could tackle this problem by using a file-system pathname as the identifier and its associated permissions to prevent other applications from using the same identifier.

Achieving crash consistency atop re-ordering file systems required considerably more developer effort. For example, correctness with ext4 required two additional `fsync` calls on LevelDB and the *fsyncobjectfiles* option on Git. The changes in ext4 both reduced performance and were complicated; we carefully used results from the Alice study to determine the additional `fsync` calls necessary for correctness.

Note that, while we happened to find that Git's *fsyncobjectfiles* makes it correct on ext4, other changes are needed for other file systems such as btrfs and ext2. Figure 10(b) shows the code



(a) Making Git crash consistent atop CCFS

```
diff --git a/git.c b/git.c
- - -  a/git.c
+++ b/git.c
@@ -648,6 +648,7 @@
int cmd_main(int argc, const char
**argv) {
+ setstream("git");
```

(i) common stream

```
diff --git a/setup.c b/setup.c
- - -  a/setup.c
+++ b/setup.c
@@ -369,6 +369,7 @@
void setup_work_tree(void)
      set_git_dir(remove_leading_path(git_dir, work_tree));
+     setstream(work_tree);
```

(ii) per-repository streams

(b) Making Git crash consistent on btrfs

```
diff --git a/lockfile.c b/lockfile.c
- - -  a/lockfile.c
+++ b/lockfile.c
@@ -202,6 +202,15 @@
int commit_lock_file(struct lock_file *lk)
       errno = save_errno;
       return -1;
       }
  +    else {
  +    char *parent =
       get_parent(result_path);
  +    int parent_fd = open(parent, 0444);
  +    fsync(parent_fd);
  +    close(parent_fd);
  +    }
```

```
diff --git a/sha1_file.c b/sha1_file.c
- - -  a/sha1_file.c
+++ b/sha1_file.c
@@ -202,6 +202,15 @@
int finalize_object_file(const char *tmpfile,
const char *filename)
   if (object_creation_mode == ...)
      goto try_rename;
-  else if (link(tmpfile, filename))
-     ret = errno;
+  else {
+     if(link(tmpfile, filename))
+        ret = errno;
+     else{
+        char *parent = get_parent(tmpfile)
+        int parent_fd = open(parent, 0444)
+        fsync(parent_fd);
+        close(parent_fd);
+     }
+  }
```

```
diff --git a/sha1_file.c b/sha1_file.c
- - -  a/sha1_file.c
+++ b/sha1_file.c
@@ -3182,6 +3195,12 @@
static int create_tmpfile(struct strbuf *tmp,
const char *filename)
   strbuf_add(tmp, filename, dirlen-1);
   if (mkdir(tmp->buf, 0777) && errno !=
EEXIST)
      return -1;
+  else {
+     int parent_fd =
get_parent(".git/objects", 0444);
+     fsync(parent_fd)
+     close(parent_fd);
+  }
```

Fig. 10. Achieving crash consistency in Git. *Panel (a) shows the code changes required to make Git crash consistent atop ccfs. Panel (a(i)) shows the code changes to assign all Git processes to a shared stream, while panel (a(ii)) shows the changes to assign Git processes that work on different repositories to different streams. In both choices, only a single line of code is added atop ccfs to realize correctness. Panel (b) shows the code changes required to achieve crash consistency atop the btrfs file system. Code differences shown in the figure are produced by the git diff tool.*

changes required to make Git crash consistent atop a file system that reorders directory operations (e.g., btrfs).

However, note that the changes shown in Figure 10(b) are not complete for two main reasons. First, the changes only address vulnerabilities Alice explicitly finds with the `git-add` and `git-commit` workloads. To ensure that all Git commands are crash consistent, more ordering points and barriers might be required similar to those shown in Figure 10(b). On the other hand, ccfs preserves ordering in all code paths in Git as the stream is initialized at the start of the application.

Second, the code changes address vulnerabilities only specific to btrfs. Additional changes are required to make Git crash consistent on other file systems that have more freedom to reorder operations. For example, a few file systems such as ext2 do not implement even popular ordering heuristics such as *safe rename* (all appends to a file are persisted before the file is renamed) and *safe file flush* (the directory entry of a newly created file is persisted when an `fsync` is called on the file) [38]. Realizing crash consistency atop such file systems requires additional `fsync` calls on directories at several other places than those shown in Figure 10(b).

Thus, the developer effort required to achieve correctness atop re-ordering file systems is significant; furthermore, such changes adversely affect common-case performance. In contrast, the effort required to achieve correctness atop ccfs while maintaining performance is negligible; additional effort can improve performance significantly.

5 DISCUSSION

The stream abstraction and ccfs have been designed to significantly improve crash consistency and performance, with minimal developer effort, in a broad range of applications. Consequently, there are specific applications for which the abstraction and the current implementation of ccfs are not ideal.

In this section, we discuss three separate aspects that determine whether a given application would find streams useful in practice. We first outline how the stream abstraction can be a fundamental mismatch for certain applications because of false dependencies. We then discuss how the abstraction imposes a trade-off on *all* file systems that implement streams, and how this affects applications. We finally explain specific limitations of ccfs that can possibly degrade application performance.

5.1 False Dependencies

False dependencies are a fundamental aspect in the stream abstraction that can cause performance overhead. In theory, for any application, one can always construct streams with no false dependencies by determining sets of writes that really require sequential order among them. In the extreme, each write can be associated with its own stream (note that `fsync`-like calls already in the application are still enforced, and consistency is not affected).

In practice, in many application workloads, most false dependencies can be avoided with little developer effort by associating entire threads to each stream (as shown in the previous section). But this strategy does not work in some applications, especially in applications where multiple layers of code hide the logical relationships between writes. As an example, consider a type-2 hypervisor that stripes multiple guest-level drives across many host-level files written by a single thread. It might be impractical for the hypervisor thread to use streams while avoiding false dependencies (note that streams can still be used within the guest OS): the natural relationships between individual writes inside the guest operating systems are forgotten by the time those writes reach host-level files. Streams are not ideal in such applications because of the developer effort involved in separating out related sets of writes into different streams.

5.2 File System Trade-Offs

Even in the absence of false dependencies, the stream abstraction imposes overhead due to the guarantee of sequential ordering within a stream. For example, consider the following code sequence: `write`(f_1, "*hello*"); `write`(f_1, "*world*"); `fsync`(f_1);. Write avoidance is not affected by streams, since in any file system, we expect *hello* and *world* to be persistent on the disk when the `fsync` returns. There are hence no false dependencies according to our previous definition (Section 2.3). But, with streams, we expect *hello* to be persistent before *world*, an additional requirement over ordinary file systems that can cost performance.

File systems that implement streams, therefore, have to use techniques to hide the performance overhead of maintaining order. We know of two basic techniques that allow the appearance of sequential order to be enforced efficiently (without explicit hardware support): journaling and copy-on-write (or log structuring). While both approaches still pose overheads (writing twice and affecting data locality, respectively), file systems can adopt different optimizations and make trade-offs for maintaining order. For example, ccfs avoids any overhead for appended data and chooses journaling over copy-on-write for overwritten data; a more complex file system could introduce additional optimizations, such as learning from workload characteristics and then dynamically choosing between journaling and copy-on-write for overwritten data. Thus, the ordering technique should not affect application performance in practice; we believe file systems can be constructed to make the overhead insignificant for most workloads, as evidenced in the previous section.

However, there remains a practical concern for some application workloads even with efficient file-system design: the need to adopt significant techniques for maintaining sequential order, along with trade-offs and optimizations, changes disk-level behavior from the perspective of an application. For example, assuming the read and write performance behavior of a hard-drive, a database might use an optimized B-tree structure for efficiency; a file system implementing streams will change that behavior with its ordering technique. Furthermore, different file systems can make different trade-offs to maintain order, and can adopt different optimizations. Streams might hence not be ideal for applications that assume particular disk-level effects from writes, such as heavily optimized traditional databases.

5.3 CCFS-Specific Effects

The current implementation of ccfs has limitations. First, it journals overwritten data, and applications that heavily overwrite data might find performance degraded. As previously explained, this could be improved by more optimizations, such as using copy-on-write when multiple contiguous blocks are overwritten (journaling can actually improve performance with random writes [41]).

Second, ccfs uses linear directories in the pursuit of pointer-less data structures (ext4 uses a hash-tree structure as its default). Although we did not observe any performance effects due to this in our evaluation, directory-heavy workloads could be affected. If necessary, however, then ccfs directories could be further optimized; for example, the linear directory currently implemented could be periodically compacted into a hash-tree in the background, in a manner reminiscent of LSM-trees.

5.4 Summary

While streams are useful for most applications, it is important to recognize that they are not compatible with all applications. Application incompatibility can arise from the stream abstraction itself, or due to the implementation of a particular file system. We identified two example applications in this section that are not ideal considering the abstraction itself: type-2 hypervisors (because of false dependencies) and traditional relational databases (because streams make disk-level effects less transparent). Note that both are significantly complex applications that might detect and be optimized for very specific device characteristics; it is reasonable to expect that they spent significant effort to verify their crash consistency. We believe that, unlike such complex applications, for applications that actually desire streams to improve consistency or performance, the abstraction will not impose overhead.

We also identified in this section that the ordering technique is an important design choice for file systems that implement streams, and explained how a combination of journaling and copy-on-write is a good choice. We considered two specific limitations of ccfs (journaling overwritten data and linear directories) in this section, and suggested alternatives for them. In general, while file systems that implement streams need to be more complex, sufficient optimizations can be introduced to make any overhead insignificant.

6 RELATED WORK

We briefly describe how ccfs differs from previous work:

Atomicity interfaces. Transactional file-system interfaces have a long history [48] and allow applications to delegate most crash-consistency requirements to the file system. Recent work in this space includes file systems providing ACID semantics such as Amino [59], Valor [52], and

Windows TxF [30], atomicity-only file systems as proposed by Vermat et al. [58], Park et al. [35], and CFS [31], and OS-level transaction support as advocated by TxOS [40]. Such interfaces allow adding crash consistency easily to applications that do not already implement them, and help heavily optimized applications that trade portability for performance [28].

For applications with existing consistency implementations, proponents of atomicity interfaces and transactional file systems advocate replacing the existing implementation with the interface provided by the file system. This is not trivial to achieve (though perhaps much easier than writing a new consistency implementation). For instance, consider the SQLite database, and assume that we replace its consistency implementation using a straightforward *begin_atomic()–end_atomic()* interface provided by the file system. This does not work for two reasons. First, it does not offer SQLite's ROLLBACK command [54] (i.e., abort transaction) and the SAVEPOINT command (which allows an aborted transaction to continue from a previous point in the transaction). Second, unless the file system provides isolation (which recent research argues against [31]), it requires reimplementing isolation and concurrency control, since SQLite's isolation mechanism is inherently tied to its consistency mechanism [53]. With applications such as LevelDB, where the consistency mechanism is tightly coupled to query-efficient on-disk data structures [26, 34], adopting alternative consistency mechanisms will also cause unnecessary performance changes.

To summarize, adopting atomicity interfaces to overcome vulnerabilities is nonoptimal in applications with existing consistency implementations. One challenge is simply the changes required: CFS [31], with arguably the most user-friendly atomic interface, requires changing 38 lines in SQLite and 240 lines in MariaDB. Another challenge is portability: until the interfaces are widely available, the developer must maintain both the existing consistency protocol and a protocol using the atomic interface; this has deterred such interfaces in Linux [10]. Finally, the complexity of data structures and concurrency mechanisms in modern applications (e.g., LSM trees) are not directly compatible with a generic transactional interface; Windows TxF, a transactional interface to NTFS, is being considered for deprecation due to this [30]. In contrast, streams focus on masking vulnerabilities in existing application-level consistency implementations. Ccfs advocates a single change to the beginning of applications and running them without more modification on both stream-enabled and stream-absent file systems.

Ordering interfaces. Fine-grained ordering interfaces [5, 6, 14] supplement the existing `fsync` call, making it less costly for applications to easily achieve crash consistency. They allow better performance, but require developers to specify the exact ordering required, and as such are not optimal for fixing existing protocol implementations. Ext4's data-journaled mode and LinLogFS [11] provide a globally ordered interface, but incur unacceptable disk-level ordering and false-dependence overhead. Xsyncfs [33] provides global order and improves performance by buffering user-visible outputs; this approach is complementary to our approach of reducing false dependencies. Other proposed ordering interfaces [9, 36] focus only on NVMs.

Implementation. Ccfs builds upon seminal work in database systems [18, 32] and file-system crash consistency [8, 12, 15, 16, 19, 44, 46, 49], but is unique in assembling different techniques needed for efficient implement of the stream API. Specifically, ccfs uses journaling [8, 19] for order within a stream but applies techniques similar to soft updates [15, 16, 49] for separating streams. Such a design is necessary: using soft updates directly for a long chain of dependent writes ordered one after the other (as ccfs promises within a stream) will result in excessive disk seeks. Block-level guarantees of atomicity and isolation, such as Isotope [50] and TxFlash [42], can simplify separation of streams in ccfs; however, techniques in Section 3 are still necessary. IceFS [27] extends ext3 to support multiple virtual journals, but requires data journaling within each journal to support ordered data writes, and hence cannot be directly used to improve application consistency without reducing performance. IceFS also does not use techniques similar to soft updates to separate the virtual journals, associating only a static and coarse-grained partition of the file-system namespace to each journal (compared to the dynamic and fine-grained stream abstraction).

In principle, one should be able to easily construct a stream-ordered file system atop a fine-grained ordering interface. However, the direct implementation of ordering in Featherstitch [14] uses the soft-updates approach, which is incompatible as described. The interface provided by OptFS [6] is insufficient for implementing streams. Ccfs uses the selective data journaling (SDJ) technique from OptFS but optimizes it; the original relies on specialized hardware (durability notifications) and decreased guarantees (no durability) for efficiency.

7 CONCLUSION

In this article, we present the stream abstraction as a practical solution for application-level crash consistency. We describe the stream API and the ccfs file system, an efficient implementation of the API. We confirm the ordering hypothesis by showing that crash consistency in real applications can be significantly improved by ccfs's in-order update and weak atomicity semantics. We confirm our efficiency hypothesis by comparing ccfs's performance with ext4, finding that ccfs maintains (and sometimes significantly improves) performance while improving correctness. Our results suggest that developer effort for using the streams API is negligible and practical.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and Ashvin Goel (our shepherd for the FAST version of the article) for their insightful comments and suggestions. We thank the members of the Advanced Systems Laboratory (ADSL) for their valuable discussions.

REFERENCES

[1] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Correlated crash vulnerabilities. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI'16)*.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2014. *Operating Systems: Three Easy Pieces* (0.9 ed.). Arpaci-Dusseau Books.

[3] Andrew D. Birrell. 1989. *An Introduction to Programming with Threads*. Technical Report SRC-RR-35.

[4] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*.

[5] Nathan C. Burnett. 2006. *Information and Control in File System Buffer Management*. Ph.D. Dissertation. University of Wisconsin-Madison.

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

[7] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST'12)*. 101–116.

[8] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. 1992. The episode file system. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter'92)*. 43–60.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP'09)*.

[10] Jonathan Corbet. 2009. Better than POSIX? March 2009. Retrieved April 2016 from https://lwn.net/Articles/323752/.

[11] Christian Czezatke and M. Anton Ertl. 2000. LinLogFS: A log-structured filesystem for linux. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track'00)*.

[12] Linux Documentation. 2016. XFS Delayed Logging Design. Retrieved April 2016 from https://www.kernel.org/doc/Documentation/filesystems/xfs-delayed-logging-design.txt.

[13] Filebench. 2016. Filebench. Retrieved March 2016 from https://github.com/filebench/filebench/wiki.

[14] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. 2007. Generalized file system dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*. 307–320.

[15] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. 2000. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst. (TOCS)* 18, 2 (May 2000), 127–153.

[16] Gregory R. Ganger and Yale N. Patt. 1994. Metadata update performance in file systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI'94)*. 49–60.

[17] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1992. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*. Seattle, WA.

[18] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann.

[19] Robert Hagmann. 1987. Reimplementing the cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP'87)*.

[20] Timothy L. Harris. 2001. A Pragmatic Implementation of Non-blocking Linked-lists. *DISC.*

[21] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2011. A file is not a file: Understanding the I/O behavior of apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*.

[22] Maurice Herlihy. 1991. Wait-free synchronization. *Transactions on Programming Languages* 11, 1 (January 1991).

[23] D. M. Jacobson and J. Wilkes. 1991. *Disk Scheduling Algorithms Based on Rotational Position.* Technical Report HPL-CSP-91-7. Hewlett Packard Laboratories.

[24] Jaeho Kim, Jongmoo Choi, Yongseok Oh, Donghee Lee, Eunsam Kim, and Sam H. Noh. 2009. Disk schedulers for solid state drives. In *EMSOFT.* Grenoble, France.

[25] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. 2013. A study of linux file system evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST'13)*. San Jose, CA.

[26] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*.

[27] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. Physical disentanglement in a container-based file system. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*.

[28] MariaDB. 2016. Fusion-io NVMFS Atomic Write Support. Retrieved April 2016 from https://mariadb.com/kb/en/mariadb/fusion-io-nvmfs-atomic-write-support/.

[29] Mercurial. 2016. Dealing with Repository and Dirstate Corruption. Retrieved April 2016 from https://www.mercurial-scm.org/wiki/RepositoryCorruption.

[30] Microsoft. 2016. Alternatives to using Transactional NTFS. Retrieved April 2016 from https://msdn.microsoft.com/en-us/library/hh802690.aspx.

[31] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. 2015. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*.

[32] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Data. Syst.* 17, 1 (March 1992), 94–162.

[33] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*.

[34] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informat.* 33, 4 (1996), 351–385.

[35] Stan Park, Terence Kelly, and Kai Shen. 2013. Failure-atomic msync (): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the EuroSys Conference (EuroSys'13)*.

[36] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA'14)*.

[37] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application crash consistency and performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*.

[38] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. 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)*.

[39] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. Crash consistency. *Commun. ACM* 58, 10 (October 2015).

[40] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. 2008. Operating systems transactions. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI'08)*.

[41] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. Analysis and evolution of journaling file systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX'05)*. 105–120.

[42] Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. 2008. Transactional flash. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI'08)*.

[43] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A Lock-free Approach to Parallelizing Stochastic Gradient Descent. *Advances in Neural Information Processing* (2011), 693–701.

[44] Mendel Rosenblum and John Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (February 1992), 26–52.

[45] Chris Ruemmler and John Wilkes. 1994. An introduction to disk drive modeling. *IEEE Comput.* 27, 3 (March 1994), 17–28.

[46] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. 1993. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter'93)*. 307–326.

[47] Margo Seltzer, Peter Chen, and John Ousterhout. 1990. Disk scheduling revisited. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter'90)*. 313–324.

[48] Margo I. Seltzer. 1993. *File System Performance and Transaction Support*. Ph.D. Dissertation. EECS Department, University of California, Berkeley.

[49] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. 2000. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX'00)*. 71–84.

[50] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. 2016. Isotope: Transactional isolation for block storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*.

[51] Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2011. A Primer on Memory Consistency and Cache Coherence. *Synthesis Lectures on Computer Architecture*, 1st Morgan & Claypool (November 2011).

[52] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. 2009. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST'09)*.

[53] SQLite. 2016. Isolation In SQLite. Retrieved Dec 2016 from https://www.sqlite.org/isolation.html.

[54] SQLite. 2016. SQL As Understood By SQLite. Retrieved Dec 2016 from https://www.sqlite.org/lang.html.

[55] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *;login: USENIX Mag.* 41, 1 (June 2016).

[56] Linus Torvalds. 2009. Linux 2.6.29. Retrieved April 2016 from https://lkml.org/lkml/2009/3/25/632.

[57] Theodore Ts'o. 2012. ext4: remove calls to ext4_jbd2_file_inode() from delalloc write path. Retrieved April 2016 from http://lists.openwall.net/linux-ext4/2012/11/16/9.

[58] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Srivilliputtur Mannarswamy, Terence P. Kelly, and Charles B. Morrey III. 2015. 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)*.

[59] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. 2007. Extending ACID semantics to the file system via ptrace. *ACM Trans. Storage (TOS)* 3, 2 (June 2007), 1–42.

[60] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*.

[61] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. 2014. Torturing databases for fun and profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*.

Received June 2017; revised June 2017; accepted June 2017