Lightweight Application-Level Crash Consistency on Transactional Flash Storage

Changwoo Min, ***Georgia Institute of Technology;*** **Woon-Hak Kang,** ***Sungkyunkwan University;*** **Taesoo Kim,** ***Georgia Institute of Technology;*** **Sang-Won Lee and Young Ik Eom,** ***Sungkyunkwan University***

https://www.usenix.org/conference/atc15/technical-session/presentation/min

This paper is included in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15) is sponsored by USENIX.

Lightweight Application-Level Crash Consistency on Transactional Flash Storage

Changwoo Min[†*] Woon-Hak Kang[‡] Taesoo Kim[†] Sang-Won Lee[‡] Young Ik Eom[‡]
[†]*Georgia Institute of Technology* [‡]*Sungkyunkwan University*

Abstract

Applications implement their own update protocols to ensure consistency of data on the file system. However, since current file systems provide only a preliminary ordering guarantee, notably `fsync()`, these update protocols become complex, slow, and error-prone.

We present a new file system, CFS, that supports a native interface for applications to maintain crash consistency of their data. Using CFS, applications can achieve crash consistency of data by declaring code regions that must operate atomically. By utilizing transactional flash storage (SSD/X-FTL), CFS implement a lightweight mechanism for crash consistency. Without using any heavyweight mechanisms based on redundant writes and ordering, CFS can atomically write multiple data pages and their relevant metadata to storage.

We made three technical contributions to develop a crash consistency interface with SSD/X-FTL in CFS: selective atomic propagation of dirty pages, in-memory metadata logging, and delayed deallocation. Our evaluation of five real-world applications shows that CFS-based applications significantly outperform ordering versions: 2–5× faster by reducing disk writes 1.9–4.1× and disk cache flushing 1.1–17.6×. Importantly, our porting effort is minimal: CFS requires 317 lines of modifications from 3.5 million lines of ported applications.

原子操作一旦开始，就一直运行到结束，中间不会有任何context switch（切换到另一个线程）

1 Introduction

Preserving the consistency of application data is one of the foremost responsibilities of computer systems. Applications, ranging from a simple text editor to more complex relational DBMS, are designed to keep their data crash-consistent. Nevertheless, due to limited file system interfaces, primarily `fsync()`, update protocols of applications to achieve crash consistency are notoriously complex, inefficient, and ad-hoc. As a result, most applications still incur inconsistencies of data upon system crashes or random power failures [54].

Suppose that two database files need to be atomically updated as shown in Figure 1. In current file systems, a typical solution is to use multiple rollback journals as shown in Figure 2. To make a single database update crash-consistent, it first records the original state of the database to a journal, so that it can always restore the database to known state upon a system crash. To make multiple database files crash-consistent, it has to maintain another journal, the so-called master journal, which specifies the database files involved during updates.

```
1  + cfs_begin();
2    write(/db1, "new");
3    write(/db2, "new");
4  + cfs_commit();
```

Figure 1: An example code snippet to implement crash-consistent updates of two database files in SQLite by using CFS. In this pseudo code, `/db1` means a file descriptor of a database `db1` under the directory `/`, and "new" means new data (e.g., database entry) to be updated. Two API calls, `cfs_begin()` and `cfs_commit()`, are included at the beginning and end of two `write()` operations to denote an atomic update.

As a result, popular database systems, such as SQLite [7], end up maintaining three journal files and performing 11 `fsync()` operations for updating just two database files with crash-consistency [31]. Besides complexity, this ad-hoc update protocol imposes a huge performance overhead: under ext4 in ordered journal mode, it generates 48 page disk writes and eight disk cache flush operations to update just two data pages. In spite of the inherent performance overhead, such `fsync()`-based update protocols often can not guarantee crash consistency. This happens because some file systems, device drivers, and virtual machines deliberately ignore such flush requests to optimize runtime performance [12, 13, 50].

A significant amount of research has been done to provide consistency of file system structures (e.g., metadata) [16, 25, 29, 39, 43, 58, 64]. However, upon crashes or power failures, even when file system structures are consistent in a system-wide manner, each application's data can be left inconsistent.

One reason why file systems and applications resort to costly journaling or logging is that current storage devices do not guarantee the atomic write of multiple pages or even a single page. Though recently proposed transactional flash storage supports atomic multi-page writes [20, 33, 51, 53, 56] by extending their log-structured write mechanism, to the best of our knowledge, there is no study on how to use transactional flash storage for

*Some of this work was performed while Changwoo Min was at Sungkyunkwan University.

```
1  // init master journal
2  open(/master.jnl, O_CREATE);
3  write(/master.jnl, "/db1,/db2");
4  fsync(/master.jnl);  fsync(/);
5  // update db1
6  open(/db1.jnl, O_CREATE);
7  write(/db1.jnl, "old");
8  fsync(/db1.jnl);     fsync(/);
9  write(/db1.jnl, "/master.jnl");
10 fsync(/db1.jnl);
11 write(/db1, "new");
12 fsync(/db1);
13 // update db2: do the same as db1
14    ...
15 // clean up master journal
16 unlink(/master.jnl);
17 fsync(/);
18 // clean up db1/2 journal
19 unlink(/db1.jnl);
20 unlink(/db2.jnl);
```

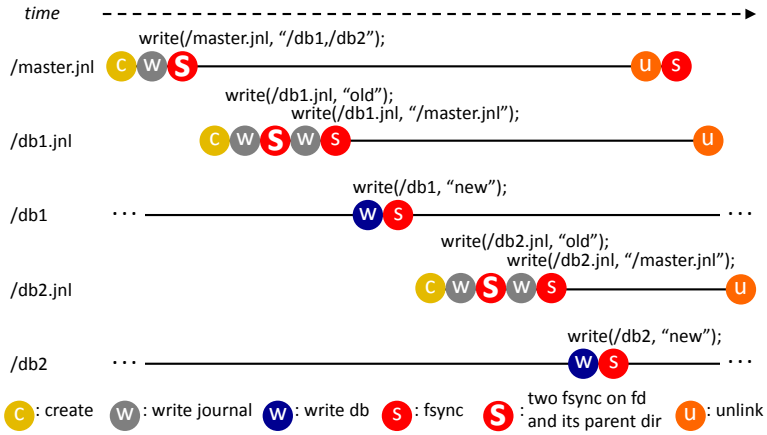


Figure 2: Crash-consistent updates of two database files in SQLite. File systems provide a minimal consistency guarantee upon a crash or a power failure: data and metadata of the latest sync-ed files will be preserved in order. To provide application-level consistency, an application has to carefully coordinate `fsync()` and `unlink()` in an ad-hoc manner. In this example, SQLite maintains a master journal (`master.jnl`) and two journals (i.e., `db1.jnl` and `db2.jnl`) with complex ordering of `fsync()` and `unlink()` calls. Because `fsync()` does not ensure that the entry in the directory containing the file has reached storage, creating a journal file entails two `fsync()` calls: one for the journal itself and another for its parent directory.

direct and efficient support of application-level crash consistency.

In this paper, we present CFS, a file system that natively supports application-level crash consistency on transactional flash storage. We make the following technical contributions:

- **Native Interface for Crash Consistency:** CFS provides a native interface for applications to describe *atomic code regions* which must operate atomically. For each region, CFS creates an *atomic propagation group*, which is a set of data and metadata pages modified in the region. An atomic propagation group is atomically written to storage regardless of system crash (*commit*), or is reverted either explicitly by a user (*abort*) or upon crash. Atomicity is guaranteed by atomic multi-page writes in transactional flash storage without using any journaling or logging.
- **In-Memory Metadata Logging:** The key design challenge is how to propagate metadata pages shared by multiple atomic propagation groups. Suppose that two inodes in different groups happen to be in the same metadata page; committing a group results in unintended propagation of another inode. We call this *false sharing of metadata pages* because irrelevant sub-page-sized metadata structures are in the same page. To resolve this, we introduce an *in-memory metadata logging* mechanism that keeps track of metadata changes for each group and selectively propagates changes of a committing group.
- **Delayed Deallocation:** All operations in an atomic code region must be safely abortable. However, it is tricky to revert deallocation of file system resources such as inodes and blocks. For example, suppose that block `B1`, which was released from atomic propagation group `A1`, was allocated to another group. If `A1` is aborted, it is impossible to revert `A1`'s deallocation of `B1`. To resolve this, we introduce a *delayed deallocation* technique, which defers actual resource deallocation until commit time.
- **Legacy Application Support:** CFS internally manages a system-wide atomic propagation group for legacy applications that do not use CFS system calls. Legacy applications can run with CFS-based ones without any modification.

Unlike transactional file systems [36, 42, 55, 60, 63] , which have been proposed to support DBMS-like ACID transactions, we have designed CFS primarily for crash consistency, and *not* for strong isolation. There are two reasons for this decision. First, modern applications like MariaDB [3, 38] or Kyoto Cabinet [2] already relax their isolation level for performance optimization, without compromising their correctness semantics. If strong isolation is enforced by transactional file systems, atomic updates in these applications may fail to progress efficiently due to frequent conflicts among transactions. We will discuss this in more detail in §8. Second, using rich semantics at the application level, it makes sense to give developers more freedom to choose appropriate synchronization primitives to achieve the required isolation level for their applications. We summarize isolation policies of popular applications in §6.

We have implemented CFS based on ext4 using transactional flash storage (SSD/X-FTL [33]). Our evaluation on real-world applications, including SQLite, MariaDB, and Kyoto Cabinet shows that CFS-based applications significantly outperform their original versions: 2–5× faster by

reducing disk writes 1.9–4.1× and disk cache flushing by 1.1–17.6×. Our porting experience shows that CFS can easily replace a variety of existing update protocols with its native interface.

In the rest of this paper, we will discuss the background (§2) and design principles (§3). Next, we will present the details of CFS design (§4) and implementation (§5). Then, we will show our application case studies (§6) and evaluate performance (§7). Finally, we discuss our limitations (§8) and related work (§9), and conclude (§10).

2 Background

2.1 Problems in Modern File Systems

Modern file systems cannot guarantee application-level crash consistency even with data journal mode for the four following reasons:

No Atomicity on Multiple Files: It is not uncommon that application data, which needs to be atomically updated, spans two or more files, as shown in Figure 2. However, while modern file systems provide three ordering primitives, namely `sync()`, `fsync()` and `msync()` for system-wide or file-wide flushing, they do not provide any primitive to flush multiple files selectively and atomically. As a result, developers have no option other than to implement their own complex update protocols with the given primitives.

Lack of Atomic Page Writes: Although `fsync()` ensures durability and ordering of writes, current storage devices do not guarantee the atomic write of a single page as well as multiple pages. Hence, file systems and applications resort to costly *journaling (or logging)* mechanisms or complicated *copy-on-write (CoW)* mechanisms [19]. However, disk cache flush operations, which are essential to implement such mechanisms, frequently become a performance bottleneck [18, 19, 46].

Shared Metadata Page: Even if an underlying storage device provides atomic page writes, modern file systems cannot directly support application-level crash consistency. Assume that two applications, `A1` and `A2`, are running. Suppose that `A1` finishes atomic updates of its changes and the system crashes before `A2` triggers its atomic updates. To achieve crash consistency of `A1`, all data pages and relevant metadata pages should be written. However, a metadata page can contain information of both `A1` and `A2`, so the *incomplete* metadata changes of `A2` can be accidentally propagated by `A1`. This is because a write unit in storage is a page, not an individual metadata structure. We call this *false sharing of metadata pages*. Depending on the unwanted metadata propagation of `A2`, a directory could have nonexistent files, or a file could have garbage blocks, or the free block counter in a superblock could be incorrect. In other words, `A1` hampers the consistency of `A2` upon a crash.

Steal Policy and Lack of Undo Mechanism: Modern file systems use the *steal policy*: due to page reclamation by the page flusher or `sync()` by applications, any metadata or data page can be written to storage at any time, although its corresponding application is still executing. Upon system recovery after a crash or an application's request to abort its changes, the stolen pages and in-memory data structures, such as metadata and inode cache, should be reverted. Unfortunately no existing file system provides a native undo mechanism. This is one of the main reasons why file systems cannot natively support application-level crash consistency.

2.2 Transactional Flash Storage

Transactional flash storage [20, 33, 51, 53, 56] supports atomic write of multiple pages by extending the log-structured nature of a *flash translation layer* (FTL). They defer the update of the mapping table for new data and achieve the atomicity of multi-page writes by atomically updating the mapping table in response to a commit request from the host. Since atomic writes achieve a high level of data integrity with fewer write commands, the storage industry is working on its standardization [61].

In this paper, we used SSD/X-FTL [33], which is a transactional flash storage providing extended SCSI interfaces such as `write(txid, page)`, `commit(txid)`, and `abort(txid)`. Each write operation is associated with `txid`, and the written pages with the same `txid` become atomically durable upon a `commit(txid)` request. Upon an `abort(txid)` request, they are reverted to their old copies. Though CFS is built on SSD/X-FTL, CFS does not fundamentally require SSD/X-FTL, and it can be built on any transactional storage devices [26, 51, 56, 61] (see §8).

3 Design Principles

For an application to be crash-consistent, a series of file system operations either *all* occur, or *nothing* occurs. From the perspective of file systems, this can be translated into the following technical axiom: "all data pages and their relevant metadata changes should be *atomically* propagated to storage." In this paper, file systems satisfying this technical requirement will be said to provide *application-level crash consistency*. In this section, we discuss four design principles which will lead to our key techniques: selective atomic propagation of dirty pages (§4.1) and in-memory metadata logging (§4.2).

Defining an Atomic Code Region: In CFS, instead of implementing complex update protocols, applications simply specify an *atomic code region*, in which file system operations must be atomically processed. An atomic code region starts with `cfs_begin()` and ends

with `cfs_commit()` (or is canceled with `cfs_abort()`). CFS automatically captures files modified by system calls, but for memory-mapped files, developers should explicitly specify corresponding file descriptors by using `cfs_add(fd)`. Naturally, there are one or more files in an atomic code region. After capturing all the modifications inside the atomic code region (the so-called *atomic propagation group*), `cfs_commit()` will make those changes persistent, and `cfs_abort()` will revert them by undoing all operations performed in the atomic code region.

Atomic Propagation of Data and Metadata Pages: Instead of resorting to costly journaling or complex CoW mechanisms, CFS exploits the atomic multi-page write feature of transactional flash storage. All data and relevant metadata pages modified in an atomic code region are grouped and sent to storage for an atomic write. Since transactional flash storage guarantees atomic durability, there is no need for journaling or CoW mechanisms.

No-Steal and Selective Propagation of Metadata: Even if CFS writes only relevant metadata pages modified in an atomic code region, metadata changes made by other in-progress atomic code regions can be propagated to storage due to the *false sharing of metadata pages*.

To avoid this anomaly, CFS delays writing the in-progress metadata changes to storage until commit time (*no stealing*). Also, to selectively propagate the changes in a metadata page, we propose a technique that logs in-memory metadata changes for each atomic propagation group and replays them at commit time.

Undoing Stolen Data Pages and In-Memory Structures: Unlike metadata pages, we support a *steal policy* for data pages, meaning that data pages do not need to be sent to storage, and rather can be stolen. This provides two benefits. First, effective management of limited page cache becomes possible, because data pages can be reclaimed under memory pressure. Second, the amount of writes at commit can be reduced, hence latency as well, because the page flusher can flush dirty pages during idle time.

To support a steal policy of data pages, CFS should be able to revert every stolen page of the aborted atomic propagation group to its old copy when system recovers from a crash or `cfs_abort()` is invoked. CFS relies on transactional flash storage to revert stolen pages. When a system crashes, transactional flash storage reverts all uncommitted writes to their old copies on system reboot. When `cfs_abort()` is called, CFS asks the transactional flash storage to revert written pages of the aborting group to their old copies. In addition, CFS reverts all in-memory metadata changes for the operating system after the abort operation. This is done by undoing the collected logs of the in-memory metadata structures.

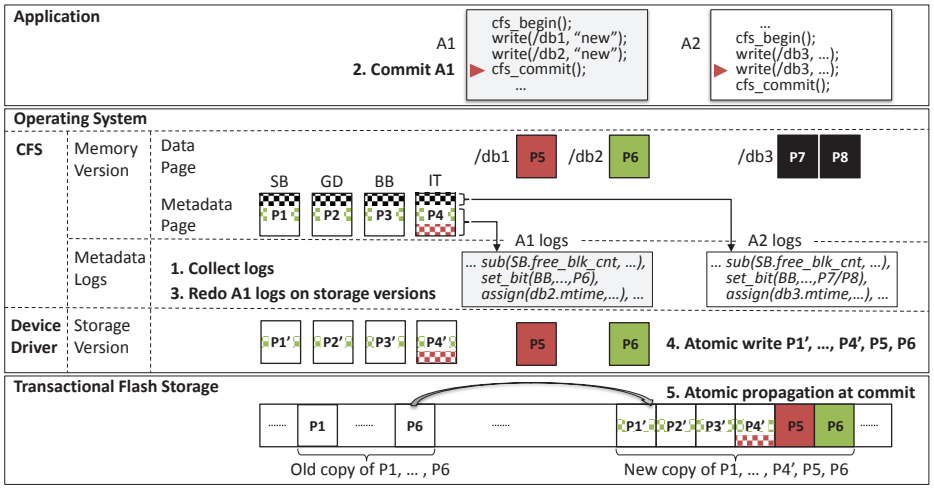
4 CFS Design

In this section, we present the design of CFS, an ext4-based file system that natively supports application-level crash consistency on transactional flash storage. To achieve crash consistency for a series of file system operations, developers define *atomic code regions* in source code and CFS guarantees atomic operations within the regions. For each region, CFS manages an *atomic propagation group* that is a set of data and metadata pages modified in each region. The pages in the group will be written atomically using atomic multi-page write features of transactional flash storage. CFS logs in-memory metadata changes made by each region. At commit time, CFS replays the collected logs of a committing group to selectively propagate changes made in the group to storage. To this end, CFS manages two versions of metadata: *memory version* and *storage version*. The data pages and the storage version of metadata pages are atomically written to storage.

Figure 3 illustrates a running example of CFS. Application `A1`, as in Figure 1, updates two database files, `db1` and `db2`, in its atomic code region. Application `A2` updates two pages of a database file `db3`. Upon `cfs_begin()`, a new atomic propagation group starts and an associated new `txid` is assigned for further interaction with transactional flash storage. CFS logs in-memory metadata changes made in each region (Step 1). Suppose that `A1` starts `cfs_commit()` while `A2` is still in-progress (Step 2). At this moment, the atomic propagation group of `A1` has data page `P5` and `P6`, and metadata pages `P1`–`P4`, which have metadata changes for `db1` and `db2`. The false sharing between `A1` and `A2` occurs in `P1`–`P4`. CFS replays the collected logs of `A1` on the storage versions, `P1'`–`P4'`, for selective propagation of metadata changes (Step 3). Thus, the storage versions `P1'`–`P4'` only contain the changes of `A1` without incorporating the changes of still-in-progress `A2`. All data pages and the storage version of metadata pages in the group are written to storage with the `txid` of `A1` (Step 4). Finally, CFS asks the storage device to make the written pages with the `txid` atomically persistent (Step 5). By using transactional flash storage, CFS can avoid redundant journalings and can significantly improve the performance of file systems. Not only that applications do not need complex and ad-hoc update protocols, but naturally gain better performance by using CFS. For example, SQLite's update protocols, which invokes 11 `fsync()` calls for updating two database files, can be replaced with two native calls to CFS and gain a 16.7× increase in performance compared to the original version, as shown in §7.2.

4.1 Managing Atomic Propagation Groups

For each atomic code region embraced with `cfs_begin()` and `cfs_commit()`, CFS keeps track of modified pages



NOTE. SB: superblock, GD: group descriptor, BB: block bitmap, IT: inode table

Figure 3: Two database applications running in CFS. Checkerboard rectangles denote metadata of files with the same color. Redoing logs of `A1` resolves the false sharing that occurred at `P1`–`P4` and creates storage version `P1'`–`P4'`. Only storage versions of metadata pages with no false sharing are written to storage. Thus, committing `A1` does not interfere with the consistency of `A2`.

as an atomic propagation group. All dirty data pages of the files and modified metadata pages in the same atomic propagation group are atomically and persistently written to storage using atomic write operations in transactional flash storage. An atomic propagation group is inherited by a child task but cannot be nested: a new task spawned inside an atomic code region automatically inherits its parent region (i.e., parent's `txid`) until it starts a new atomic code region. In SSD/X-FTL, a new `txid` is assigned to a task (i.e., `task_struct` in Linux) upon invocation of `cfs_begin()`. CFS then writes all pages with the same `txid` for the task.

4.2 In-Memory Metadata Logging

Memory Version vs. Storage Version: At the core of CFS is the in-memory metadata logging. CFS records changes of in-memory metadata structures, called the *memory version*, for each atomic propagation group. Upon a commit, CFS selectively propagates the changes made in the group to on-disk metadata structures, called the *storage version*, which are updated by *redoing* logs of a committing group, and then writing to storage. Upon an abort, CFS reverts the changes of in-memory metadata structures by *undoing* logs of the aborting group. Creating the storage version is straightforward if a metadata has separate in-memory and on-disk structures (e.g., superblock and inode). Otherwise, in the case that there is no separate structure (e.g., inode bitmap), CFS clones a memory version and uses the cloned structure as a storage version. The storage versions are what will be initially loaded when reading pages from storage.

Operational Logging: CFS uses operational logging, which records executed metadata change operations. Table 1 shows the specification of operations used to capture the metadata changes in CFS. Operations are composed of four primitive operations and one extended operation (`x_op`). Primitive operations directly modify metadata structures and an extended operation runs a registered callback function, noted as argument `f`. All operations have two arguments: `m` for memory version and `s` for storage version of a metadata structure.

Let us suppose that the free inode count in a superblock needs to be decremented when allocating a new inode. To capture this operation, CFS records a `sub` operation with an argument of the free inode count in superblock, so that the metadata change can be part of the logs in the current atomic propagation group. Upon a commit, the free inode count in on-disk superblock (i.e., `s`) will be decremented to reflect the change (*redoing*). Upon an abort, the free inode count of the in-memory superblock (i.e., `m`) will be incremented (*undoing*) to revert the change.

It is worth detailing how to undo `assign` operations (ϕ(`m`) in Table 1). For example, atomic propagation group `A1` creates a new file, thus the timestamp of the parent directory `D` is updated from `t0` to `t1`. After that, another atomic propagation group `A2` creates another file at the same directory so the timestamp is updated from `t1` to `t2`. Now, if `A1` aborts, to what should the timestamp of `D` be reverted? Since `A2` already updated the timestamp from `t1` to `t2`, it should remain `t2`. After that, when `A2` aborts, the timestamp should be reverted to `t0`. After all, CFS always reverts to *its most recent valid value*. For this purpose, CFS maintains a list of `assign` operations for a data entry in order of the operations. Upon an abort, CFS removes the aborting `assign` operation in the list and reverts the value to the head of the list (i.e., its most recent valid value).

Operation	REDO (commit)	UNDO (abort)	Description (example)
add(m,s,v)	s+=v	m-=v	Add v to m (e.g., increments free inode counts)
sub(m,s,v)	s-=v	m+=v	Subtract v from m (e.g., decrements free block counts)
assign(m,s,v)	s=v	m=ϕ(m)	Assign v to m (e.g., changes access mode of an inode)
toggle_bit(m,s,i)	s[i]=¬s[i]	s[i]=¬s[i]	Toggle i-th bit of m (e.g., toggles a bit in block bitmap)
x_op(m,s,f,a)	f(commit,m,s,a)	f(abort,m,s,a)	Run a function f (e.g., allocates a directory entry)

NOTE. m: memory version, s: storage version, m[i]: i-th bit of m, ϕ(m): the most recent valid value of m

Table 1: Specification of operations for in-memory metadata logging.

Extended Operations: To handle complex metadata structures and optimize the use of resources (e.g., caches), CFS introduced a special type of operation, called x_op(). We summarize its usage into three categories:

The first usage is to manipulate complex metadata structures. For example, each directory keeps its directory entries (or dentries) in a list or a hash tree [24]. Inserting or deleting a dentry must follow the semantics of such structures. When CFS allocates a dentry to create a new file, it registers a callback function. Upon a commit, the callback inserts the dentry into the storage version of the directory. Upon an abort, it deletes the dentry from the memory version of the directory.

Second, the extended operation is required to revert file system caches upon an abort. CFS maintains the inodes and dentry caches for efficient accesses as well as the buddy cache [17] for efficient disk block allocation. When allocating a file system resource (i.e., inode, dentry, or block), an associated cache is also updated. CFS needs to revert the changes in the cache if the resource allocation is aborted. To do this, CFS has to register a callback that reverts the cache updates that happened while allocating file system resources.

Lastly, the extended operation is required to correctly deallocate file system resources. For instance, given two atomic propagation groups A1 and A2, let us suppose that a block released from A1 was allocated to A2. After A2 is committed, it is impossible to abort A1 because there is no way to revert the block allocation of already committed A2. In order to prevent this scenario, we propose a technique named *delayed deallocation*. When CFS needs to release file system resources, it registers a callback function. Deallocation is deferred until the actual commit, at which point it finally deallocates the resource by executing the registered callback function.

A Running Example: As in Figure 3, suppose that db1, db2, and db3 are in the same block group [24]. If new data is overwritten in db1, and another new data is appended in db2 and db3, then the size of their database files grows. The last modified time of each database file is updated (P4) and CFS logs three assign operations. Growing the files incurs a series of metadata changes: three block use flags for P6, P7, and P8 in a block bitmap (P3) are turned on and CFS logs three toggle_bit operations; the block maps in the inode table (P4) are changed to refer to the new blocks and the file sizes in the inodes (P4) increase, thus CFS logs five assign operations; each free block count in the superblock (P1) and block group descriptor (P2) decreases, thus CFS logs two sub operations. Since block allocation incurs the changes in the buddy cache, CFS adds one x_op to revert the change in the cache upon abort.

4.3 Commit and Abort Procedures

Upon a commit, CFS first writes all dirty data pages of the files that belong to the committing atomic propagation group. Writing data pages could cause further metadata changes; for example, due to the delayed block allocation scheme [17], the actual block allocation happens when writing data blocks, changing metadata structures such as block bitmap and free block count. Then, CFS applies all of the group's collected logs to the storage version of metadata in the order of their generations and writes the storage version. It writes all pages with the txid issued at cfs_begin() and then asks SSD/X-FTL to make written pages with the txid durable.

Upon an abort, CFS rolls back the atomic propagation group by executing all the collected logs for the group in reverse order of their creation (*undoing*). Then, it also lets the storage revert the stolen data pages to their old copies. In SSD/X-FTL, CFS sends an abort command with the txid of the group. Finally, CFS forcefully drops all the dirty pages of the files in the group so that subsequent access to the page results in reading the reverted valid page from storage. If another application happens to access the aborted files, it could encounter an error depending on its correctness semantics. If this is the case, access to shared files must be coordinated using a synchronization primitive such as locking, or the shared files must be made public only after they are committed. For example, a transactional package manager needs to make new versions of shared libraries public after successful package installation to avoid applications reading the libraries, which are being installed and could be subject to an abort.

4.4 Dealing with Legacy Applications

It is highly desirable to be able to run the legacy applications without any modification while preserving their semantics. To this end, every update from legacy applications is treated as part of an atomic propagation group

in CFS. To be concrete, CFS maintains a *system-wide atomic propagation group*, to which every update from legacy applications belongs. CFS commits the system-wide atomic propagation group either when background flusher threads flush all dirty data and metadata pages or when a `sync()` is invoked. After the commit, CFS creates a new system-wide atomic propagation group for handling subsequent updates from legacy applications. Our current unoptimized `fsync()` simply performs `sync()`. We believe, however, this is not a fundamental limitation of our approach; for example, managing fine-grained (e.g., file-level) atomic propagation group and using group commit can be leveraged to optimize `fsync()` of legacy applications.

4.5 Consistency and Recovery

Despite various system or application failures, CFS guarantees application-level crash consistency as long as an application correctly specifies atomic code regions and a transactional flash storage guarantees atomic multi-page writes. Because CFS enforces durability of *all and only* the data pages and metadata changes of a committing atomic code region, it guarantees version consistency [19], that the metadata version matches the version of the referred data for each commit operation, and does not interfere with the consistency of other commit operations. Also, because updates from legacy applications are treated in the same manner, CFS guarantees file system-level crash consistency.

There are two types of common failures in CFS. First, if an application is terminated abnormally without the entire system failing, the OS kernel aborts all uncommitted atomic propagation groups of the terminating process and thus rolls back the changes of the application. To maintain the semantics of legacy applications, CFS never aborts the system-wide atomic propagation group. Second, if the entire system fails (e.g., a power outage), CFS relies on the recovery mechanism of transactional flash storage. On system reboot, for any incomplete commit at the time of failure, transactional flash storage will invalidate all uncommitted changes and thus roll back the storage to the last successful commit state.

5 Implementation

We implemented CFS in Linux Kernel 3.10.7 based on ext4, modifying about 5,800 lines of code. To capture the operational logs at runtime, we inserted 171 primitive operations and 11 extended operations. We performed experiments on a machine with a quad-core 2.1 GHz Intel Xeon E5606 processor and 4 GB memory. We used the OpenSSD development platform [10] with 8 GB storage capacity and the SATA 2 interface. We implemented two FTL schemes on the OpenSSD device: greedy FTL [35], which is a page-level FTL scheme with a greedy garbage collection policy, and X-FTL [33], which is an extended greedy FTL, to support atomic multi-page writes.

Application	Isolation
SQLite	Strong isolation
MariaDB	Four isolation levels in SQL standards [38]
Kyoto Cabinet	Intentionally no isolation
APT	No isolation
vim	Strong isolation or no isolation

Table 2: Isolation levels in five real-world applications.

In comparison to commercial SSDs, OpenSSD and its FTLs have several limitations: First, its capacity is too small to be considered as typical enterprise setting. Since SSDs use log-structured writing scheme, write performance under high disk utilization would be slower than that in low disk utilization. To avoid such performance anomaly and present fair comparison, we carefully choose the data set size for evaluation. For MariaDB, database size was set to 2.5 GB so there were around 70% free space in the SSD. Next, OpenSSD has a low degree of internal parallelism due to its architectural limitations. Due to this low degree of internal parallelism, performance degradation caused by a disk cache flush is limited in OpenSSD, even though it will be significant in high-end SSDs [32]. Finally, the size of atomic propagation group is limited by the transaction size of X-FTL. However, this limitation could be overcome by adopting other transaction representation schemes (e.g., cyclic representation [56]) to support unlimited (i.e., limited by only disk capacity) transaction size.

6 Application Case Studies

In this section, we show how CFS can simplify the complicated update protocols of existing applications. We choose five real-world applications, which have a variety of isolation levels from strong isolation (e.g., SQLite) to no isolation (e.g., KyotoCabinet), shown in Table 2. The CFS-enabled applications can simply reuse the existing concurrency control code to achieve the same isolation level without any additional overheads. As summarized in Table 3, porting existing applications to CFS is straightforward. For four applications, in which a file is the granularity of atomicity, we simply specified atomic code regions using CFS's native calls. For MariaDB, which uses physiological write-ahead-logging [44] and double-write [1], we replaced the double-write with CFS-protected atomic write of database files. Our experience confirms that CFS can easily replace various existing update protocols: for five real-world applications, we only needed to modify 317 lines of code out of 3.5 million in total.

SQLite: SQLite [6] is a library based DBMS widely used in smart devices. It relies on rollback journaling (RBJ) or write-ahead-logging (WAL) to guarantee crash consistency [7, 8]. In RBJ mode, the original content of a

Application	Lines of code		Mechanisms to guarantee application consistency	
	Original	Modified	Original	Modified
SQLite	217,313	38	Physical RBJ or WAL	Specifying an atomic code region
MariaDB	1,534,980	240	Physiological WAL & double-write	Physiological WAL & atomic database write
Kyoto Cabinet	162,606	26	Physical RBJ on a `mmap`-ed region	Specifying an atomic code region
APT	407,642	4	None	Specifying an atomic code region
vim	1,179,246	9	`rename`-based update	Specifying an atomic code region
Total	**3,501,787**	**317**		

NOTE. RBJ (rollback journaling), WAL (write-ahead-logging)

Table 3: Summary of our porting efforts in applying CFS to five real-world applications. CFS requires only 317 lines of modifications out of 3.5 million lines of ported applications, in order to support the crash consistency.

page is copied to the rollback journal before updating the page. In the WAL mode, the original content is preserved in the database and the modified page is appended to a write-ahead-log file. The change is then later propagated to the database by periodic checkpointing.

In version 3.8.3 of SQLite, the RBJ and WAL mode consist of about 14,500 lines of code. With CFS, we were able to implement the same level of crash consistency by adding just 38 lines of CFS system calls with journal mode off.

MariaDB: MariaDB [3] is a popular open source DBMS, and InnoDB is a popular transactional storage engine used in MariaDB. To preserve crash consistency, InnoDB uses an optimized logging technique known as ARIES-style physiological write-ahead-logging (WAL) [44]. Unlike the *physical WAL* mode in SQLite, in the *physiological WAL* of MariaDB, only changes made to data pages are written to the log device to minimize the amount of log writes. Since logs are directly applied to the data pages in-place, crash recovery is possible only if the data pages are not corrupted. To guarantee atomic update of data pages, InnoDB uses a redundant page write technique known as *double-write* [1]: it first synchronously writes data pages to the dedicated double-write area, then re-writes each page to its original location. In each step, `fsync()` calls are used to enforce ordering and durability.

CFS-based MariaDB directly updates database files in-place after writing the physiological log, and does not require the double-write. CFS can ensure the atomic updates of database files by simply guarding the update code using the CFS system calls.

Kyoto Cabinet: Kyoto Cabinet [2] is a library-based key-value store using a memory mapped region to manage its data. For crash recovery, it writes an unmodified copy to the dedicated *rollback journal* area when a data page becomes dirty. To guarantee that the old copy is flushed to storage ahead of its new copy, it calls `fsync()` upon every write to the rollback journal area.

We were able to achieve the same level of crash consistency by simply turning off the journaling and guarding the atomic update code using the CFS system calls.

APT Package Manager: For a successful software installation or update, numerous files can be created, modified, or deleted, and all these modifications should be carried out atomically. Surprisingly, due to the complexity of guaranteeing atomic package installation, most package managers, including the popular APT [23], do not provide atomic installation, leaving the responsibility to system administrators. We added an atomic installation feature to APT by guarding its package operation code using the CFS system calls.

Vim: When saving an updated file of *document-like data*, many applications, including vim [11], use `rename`-based update schemes: creating a new file and writing the updated document to the new file, then calling `fsync()` on the file to force it to disk, and finally replacing the original with the new one. With CFS, vim is modified to update the file in place and its atomicity is guaranteed by wrapping the update code with `cfs_begin()` and `cfs_commit()`.

7 Evaluation

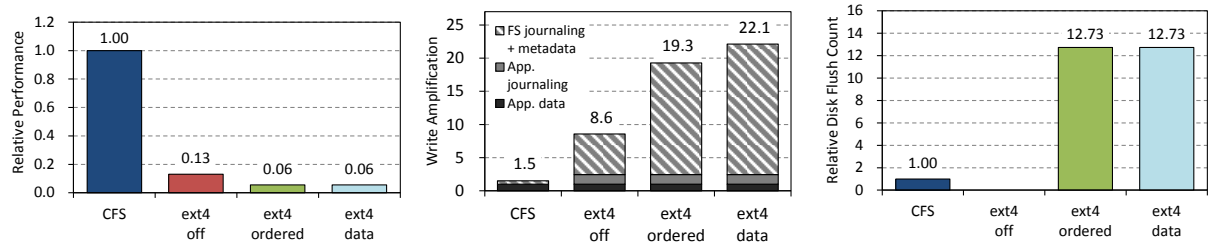
In this section, we present experiments that answer the following questions:

- Does CFS really guarantee application-level crash consistency? (§7.1)
- How do legacy update protocols and CFS-based atomic updates behave differently? (§7.2)
- What are the performance benefits of CFS-based applications? (§7.3)
- What is the performance impact on legacy applications that do not use CFS system calls? (§7.4)

Before running each experiment, we ran the workload independent preconditioning (WIPC) [62], so as to put the SSD in a steady state. The journal size of ext4 is set to 128 MB. We reported the average of three runs.

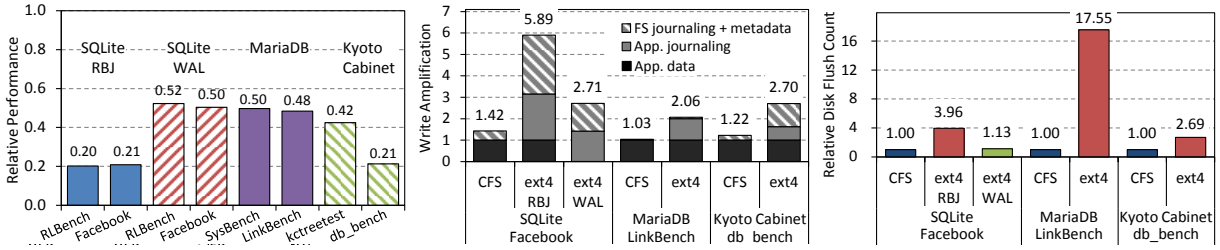
7.1 Consistency over Random Failures

We begin our evaluation of CFS by experimentally verifying whether it preserves application-level crash consistency across sudden power outages. We used MariaDB because it is the most mature and complicated among our test applications and it also provides the tool `mysqlcheck`,



(a) Performance **(b)** Write amplification **(c)** Disk cache flush count

Figure 4: Results of the microbenchmark. Atomic update of two database files in Figure 1 and 2. The CFS-based version is 16.7× faster than the original version in ext4 ordered mode. Because the CFS-based version does not rely on a complex update protocol, disk writes and cache flush operations are reduced by 12.9× and 12.7×, respectively.



(a) Performance **(b)** Write amplification **(c)** Disk cache flush count

Figure 5: Results of real-world applications. The original versions are run on ext4 in ordered journal mode. Compared to the CFS-based versions, the original versions relying on complex update protocols show significant overhead.

which checks for database corruption. While we ran LinkBench [14] on CFS-based MariaDB, we cut the power of the SSD/X-FTL to stress the full system software/hardware stack. After rebooting the test machine, we checked the consistency of CFS and the database using `fsck` and `mysqlcheck`, respectively. If both checks pass, then we conclude that CFS preserves application-level crash consistency. We repeated this test 100 times and passed the consistency check every time. To check the coverage of our test, we further analyzed pages recovered by SSD/X-FTL. In 90% of the tests, SSD/X-FTL recovered 10.3 pages on average. Types of recovered pages were CFS metadata (2.4%), LinkBench data table (95.1%), and InnoDB system table (2.5%). Though it is limited, this experiment is one practical way to validate CFS's correctness. From a theoretical point of view, it is hard to imagine a case where application consistency is vulnerable when data pages and their relevant metadata changes are atomically propagated to the storage.

7.2 Analysis of Atomic Update

To understand the performance characteristics of legacy update protocols and CFS-based atomic updates, we used the atomic update of two database files presented in Figure 1 and 2 as a microbenchmark. The original version was run on ext4 with three different journaling modes: off, ordered, and data journal mode.

In Figure 4, we first present performance comparisons, write amplification, and disk flush count for further analysis. The performance of each original version is normalized to the CFS-based version. Write amplification is the ratio of an application's writing of database files to the file system's writing to storage. It is split into three categories: application data, application journaling (e.g., SQLite RBJ), and file system overhead (i.e., metadata and journaling). We present the normalized disk flush count for CFS. In the case of CFS, we counted commit requests, upon which SSD/X-FTL flushes the disk cache. This data is obtained by instrumenting the microbenchmark and collecting block traces from the host using `blktrace`.

As Figure 4a shows, the CFS-based version significantly outperforms the original version—7.7× to ext4 off mode, and 16.7× to ext4 journaling modes—due to reduced disk writes and disk cache flush operations. The write amplification of the original version is surprisingly high (Figure 4b): 8.6, 19.3, and 22.1 in ext4 off, ordered, and data journal mode, respectively. Application-level journaling incurs significant metadata overhead. When combined with ext4 journaling, the amount of writes is amplified by 2–3× compared to ext4 off mode. As expected, the disk flush count of the original version is very high (Figure 4c). Since ext4 in off mode does not guarantee any consistency, it does not issue any disk cache flush operations. In the other modes, 12.7× more cache flush operations were issued. However, due to the low degree of internal parallelism of OpenSSD [10], performance is largely determined by the write amplification factors rather than disk cache flush count.

7.3 Performance of Real Applications

To see how CFS can improve performance of real-world applications, we evaluated three performance-sensitive applications: SQLite, MariaDB, and Kyoto Cabinet, studied in §6, using six workloads. In Figure 5, we compared performance, write amplification, and disk flush count of each application. As we expected, CFS-based applications significantly improve performance of their original versions, because they prevent the journaling of journal (JoJ) anomaly [31] fundamentally without any consistency compromise. In the rest of this section, we present the performance analysis of each application.

SQLite: We ran two SQL traces [33] collected from running the RL Benchmark [5] and Facebook applications on an Android 4.1.2 Jelly Bean SDK under its typical usage scenario.

The CFS-based version outperforms the original versions with the rollback journal (RBJ) and write-ahead logging (WAL) by approximately five-fold and two-fold, respectively. In RBJ mode, data is always written twice, one for the RBJ file and another for the database file. Moreover, since the RBJ file is created and deleted whenever a new transaction ends, SQLite in RBJ mode has very high file system metadata overhead. As a result, the original version generates 4.1× more writes and 3.9× more disk cache flush operations than the CFS-based version. In WAL mode, the modified data is appended to a WAL file and then the change is propagated to the database file by periodical checkpointing. Since the WAL file is reused by many transactions until the checkpoint occurs, the metadata overhead of SQLite in WAL mode is far lower than that of RBJ mode. As a result, the original version generates about 1.9× more writes and 10% more disk cache flush operations than the CFS-based version.

MariaDB: We used two popular database benchmarks: SysBench [9] and LinkBench [14]. SysBench in an OLTP mode stresses a 2.5 GB database (16 files) with 10 million rows for 10 minutes. LinkBench from Facebook is designed to benchmark performance of database operations with large-scale social graphs. In LinkBench, we ran 80,000 operations for a 2.5 GB database (18 files) after a two minute warm-up. In both experiments, MariaDB was configured to use 100 MB as a buffer pool with eight concurrent threads, and all under `O_DIRECT` I/O mode.

The CFS-based MariaDB performs 2× faster than the original MariaDB. This performance benefit primarily comes from the reduced number of write operations by replacing the double-write with CFS's native interface. While `fsync()` is required for each database file update and every double-write operation in the original MariaDB, the CFS-based MariaDB requires only one disk flush to update all database files. Thus, the CFS-based version invokes 17.6× fewer disk flush operations than the original

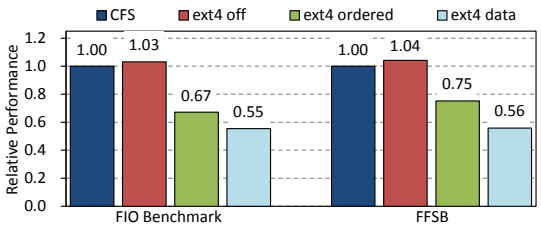


Figure 6: Performance comparison between CFS and ext4 for FIO and FFSB benchmarks

version. However, the performance results in Figure 5a are little affected by such frequent disk flush operations. This is because OpenSSD, as discussed in §5, has low degree of internal parallelism.

To emphasize the criticality of this problem that end-users will face, we ran LinkBench using a commercial SSD [30] with high-level of internal parallelism. Since the SSD does not support transactional interfaces yet, we used the modified MariaDB that flushes disk cache once at updating all database files without double-write. Write amount and cache flush count are the same as those in CFS and so the performance will be similar to that of the CFS-based one. For comparison, we ran the MariaDB only turning off the double-write mode without any other modifications; so the cache flush count is the same as the unmodified MariaDB but the write amount is the same as CFS-based one. The frequent disk cache flush operations in the latter degrades performance by 78%. It shows that frequent cache flush caused by ad-hoc update protocols is a serious performance bottleneck and CFS's native interface can be a solution to overcome this performance degradation problem.

Kyoto Cabinet: We used two workloads for Kyoto Cabinet. First, we ran Kyoto Cabinet's `kctreetest` [2] with eight concurrent test threads. Each test thread writes 10,000 arbitrary key-value pairs and then reads the keys 10,000 times. Second, we ran LevelDB's `db_bench` [28] with a single test thread. We measured the performance of 10,000 arbitrary writes of key-value pairs. We configured Kyoto Cabinet in synchronous transaction mode, guaranteeing consistency.

The CFS-based version significantly outperforms the original version. In `kctreetest`, where the read-to-write ratio is about one, the CFS version is 2.4× faster than the original version. In the write-intensive `db_bench`, the CFS version is 4.8× faster than the original version. The original version issues `sync` system calls three times for a write operation. As a result, it generates 2.2× more writes and 2.7× more disk cache flush operations.

7.4 Performance of Legacy Applications

To understand the performance of legacy applications, which were not designed to use CFS, we compare performances of ext4 and CFS by running two popular bench-

marks: Flexible I/O (FIO) benchmark [15] and Flexible File System Benchmark (FFSB) [59]. We used the FIO benchmark to simulate a data-heavy workload: it is configured to perform random writes to a 4 GB file with an 8 KB write unit while `fsync()` is called every 40 KB. We used the FFSB benchmark to simulate a metadata-heavy workload: it executes a combination of small file creates, writes, reads, and appends.

We ran the benchmarks on CFS and the three journaling modes of ext4. As Figure 6 shows, CFS provides similar performance to ext4 with journaling off while it guarantees the highest level of crash consistency. Results of ext4 in the other journaling modes show significant overhead.

8 Discussion and Future Work

Isolation and Concurrency Control: One may be curious about the difference between the *application-level crash consistency* of CFS and the *transaction* of transactional file systems [36, 42, 55, 60, 63]. In terms of concurrency control, we took an opposite design choice to transactional file systems. Transactional file systems are designed to natively support a DBMS-like ACID transaction, therefore they support strong isolation (i.e., the highest isolation level, *serializable isolation*). Under strong isolation, time-of-check-to-time-of-use (TOCTTOU) races can be easily prevented. In Table 4, we compare two representative transactional file systems, Valor [63] and TxOS [55], with CFS. To support strong isolation, they took two extreme design decisions: Valor uses pessimistic coarse-grained locking and TxOS uses optimistic multi-versioning based on software transactional memory. Suppose that two applications, `A1` and `A2`, create, write, and read files in a directory `D` in each transaction. Since the timestamp of `D` is updated upon every file creation, Valor locks `D` at the expense of concurrent execution of applications. Though TxOS maintains multiple versions of `D` for concurrent execution, due to the conflicting updates of the timestamp, only one application succeeds and the other should be re-executed.

In contrast, CFS does not provide isolation or concurrency control mechanisms. Applications must implement required concurrency control using existing synchronization primitives, such as file lock and mutex. Also, if there is a possibility of TOCTTOU races, applications should prevent the races themselves (for example, by using the `openat()` system call [4]). In fact, of the four ACID properties in DBMS, the isolation property is the most often relaxed. Popular DBMS implementations [38, 40, 48] provide at least four isolation levels for a user to choose. In the above example, if `A1` and `A2` need an isolated view of `D`, they must implement concurrency control themselves. Otherwise, no concurrency control is required resulting no overhead. The rationale behind our design is that isolation can be best implemented at the application level by exploiting correctness semantics of the applications.

	CFS	Valor [63]	TxOS [55]
Atomic update	Trans. flash	FS meta journal + logging	FS full journal
Isolation	None (app.)	Locking	Versioning
Performance	High	Low	Mid
Complexity	Low (5.8K)	Low (4.4K)	High (22.6K)

Table 4: Comparison among CFS and recent transactional file systems. Modified LOCs are in the parentheses on the bottom.

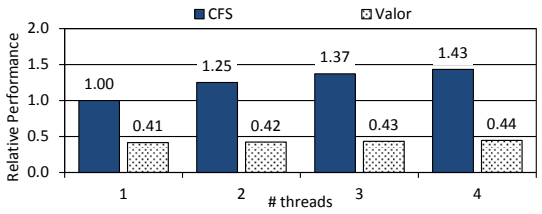


Figure 7: Multithreaded performance of CFS and Valor.

To verify the cost of isolation, we implemented the essential part of Valor in userspace and ran varying numbers of threads of the above example. As expected, performance of CFS increases as thread count increases and is constrained by storage bandwidth (Figure 7). However, due to conservative directory locking, performance of Valor is constant regardless of the number of threads. Also, since Valor uses logging, its single-thread performance is about 2× slower than CFS.

Running CFS on Non-transactional Storage: Another interesting question is whether CFS is applicable to non-transactional storage devices. The performance benefit of CFS is two-fold: simplified update protocols and no redundant writes that rely on transactional flash. Therefore, if atomic multi-page writes can be emulated on non-transactional storage, CFS is applicable and we can expect performance benefits from the simplified update protocols. Such atomic multi-page writes have long been studied (e.g., *atomic recovery unit* in Logical Disk [22]). The obvious design choice is to implement the atomic multi-page write using write-ahead-logging at the device mapper layer [57], which is a higher-level virtual block device on top of physical block devices in the Linux Kernel. To see its potential performance benefit, we ran the code in Figure 1 on ext4 data journal mode replacing `cfs_commit()` with two `fsync` calls. Though it is 2.9× slower than the CFS version, it is still 6.2× faster than the version using ext4 ordered journal (Figure 4). As future work, we will design and implement virtual transactional storage supporting CFS on non-transactional storage devices.

9 Related Work

Crash consistency is critical to operating system design, and many different approaches have been explored.

File System Consistency: To guarantee system-wide consistency of file system data structures, a variety of techniques have been proposed: journaling [39, 64], soft updates [25], copy-on-write [16, 29, 43, 58], and using a DBMS as a file system [27, 34, 45, 47, 49]. However, due to the lack of file system interfaces to support application-level crash consistency, applications had no choice but to implement complex update protocols using `fsync()`. Although several techniques [18, 19, 46] have been proposed to mitigate the performance penalty of `fsync()`, they cannot help to simplify these update protocols. A recent study revealed that widely-deployed applications, such as PostgreSQL, LevelDB, and HDFS, implemented their own ad-hoc update protocols, and thus still remained vulnerable to crashes [54]. We believe CFS is the first principled and practical way to change this landscape.

Transactional File Systems: There have been steady efforts to natively provide transactions with ACID properties to applications via file systems [36, 42, 55, 60, 63]. As we discussed in §8, transactional file systems and applications relying on them support only strong isolation. Considering that relaxation of isolation according to applications' correctness semantics is a key optimization technique, it is the critical limitation in practice. Also, complexity and overheads for strong isolation is not negligible; the most commonly used locking technique limits concurrent execution of multiple transactions [42, 60, 63]; sophisticated multi-versioning still shows non-negligible overhead (14% in TxOS [55]). Moreover, to achieve atomic and durable updates, transactional file systems rely solely on file system journaling [42, 63], or additionally maintain another write-ahead log for transactions [55, 60]. As a result, it was recommended to maintain transactions to small, mostly metadata operations [41].

Transactional Storage Devices: Several interesting approaches [20, 33, 51, 53, 56] have been proposed to support the transactional atomicity inside NAND flash storage devices. They exploit the log-structured mapping in FTL and atomically update the mapping table to achieve transactional atomicity. However, none of them resolve the false sharing of metadata pages. Thus they cannot support atomic update of multiple applications due to this lack of generality. For non-volatile memory storage, MARS [21] supports application transactions. But, it does not mention how MARS can be used to support file system consistency.

Atomic Update of Application Data: Recently, several techniques [37, 52, 65] have been proposed to protect application data from failures without supporting isolation like CFS. None of them handle the false sharing of metadata pages. Thus they cannot support atomic updates for arbitrary file system operations as CFS does. Failure-atomic `msync()` [52] atomically updates the changes of a `mmap`-ed file using REDO journaling. However, it only supports atomic update of a single `mmap`-ed file, and, due to the lack of the UNDO mechanism, the dirty data pages can not be stolen.

10 Conclusion

CFS is the first file system that natively supports application-level crash consistency on transactional flash storage. To guarantee crash consistency, applications can simply specify code regions that need atomic file system operations instead of implementing complex, slow, and error-prone update protocols by themselves. CFS guarantees the atomic propagation of data and metadata pages changed in the code region without relying on journaling through the use of the atomic multi-page write functionality of SSD/X-FTL. Our application case studies confirm that a variety of existing applications can be easily ported to CFS. Our experimental results show that CFS-based applications are 2–5× faster than the original versions.

Acknowledgments

The authors wish to thank our shepherd, Liuba Shrira, and the anonymous reviewers for their helpful comments. We thank to Jin-Soo Kim for motivational discussion, Gihwan Oh for helping us with SSD/X-FTL, and Sangman Kim for his feedback and comments. We also thank the various members of our operations staff who provided proofreading of this paper. This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2010-0020730). This work was supported by ICT R&D program of MSIP/IITP [10041244, SmartTV 2.0 Software Platform]. Changwoo Min and Taesoo Kim are partly supported by ETRI MSIP/IITP [B0101-15-0644] and ONR N00014-15-1-2162.

References

[1] InnoDB Disk I/O in MySQL 5.7 Reference Manual. http://dev.mysql.com/doc/refman/5.7/en/innodb-disk-io.html.

[2] Kyoto Cabinet: a straightforward implementation of DBM. http://fallabs.com/kyotocabinet/.

[3] MariaDB An enhanced, drop-in replacement for MySQL. https://mariadb.org/.

[4] openat(2) - Linux man page. http://linux.die.net/man/2/openat.

[5] RL Benchmark:SQLite. http://redlicense.com/.

[6] SQLite. http://www.sqlite.org/.

[7] SQLite: Atomic Commit In SQLite. http://www.sqlite.org/wal.html.

[8] SQLite: Write-Ahead Logging. http://www.sqlite.org/wal.html.

[9] SysBench: a system performance benchmark. http://sysbench.sourceforge.net/.

[10] The OpenSSD Project. http://www.openssd-project.org/wiki/The_OpenSSD_Project.

[11] Vim the editor. http://www.vim.org/index.php.

[12] Apple. BSD System Calls Manual: FCNTL(2). https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/fcntl.2.html.

[13] Apple. BSD System Calls Manual: FSYNC(2). https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/fsync.2.html.

[14] Armstrong, T. G., Ponnekanti, V., Borthakur, D., and Callaghan, M. LinkBench: a Database Benchmark based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD '13.

[15] Axboe, J. FIO (Flexible IO Tester). http://git.kernel.dk/?p=fio.git;a=summary.

[16] Btrfs. http://btrfs.wiki.kernel.org.

[17] Cao, M., Santos, J. R., and Dilger, A. Ext4 block and inode allocator improvements. In *Proceedings of the Linux Symposium* (2008).

[18] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), SOSP '13.

[19] Chidambaram, V., Sharma, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Consistency Without Ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST '12.

[20] Choi, H. J., Lim, S.-H., and Park, K. H. JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory. *ACM Transactions on Storage 4*, 4 (feb 2009).

[21] Coburn, J., Bunker, T., Schwarz, M., Gupta, R., and Swanson, S. From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (2013), SOSP '13.

[22] de Jonge, W., Kaashoek, M. F., and Hsieh, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (1993), SOSP '93, ACM.

[23] Debian. Apt. https://wiki.debian.org/Apt.

[24] Ext4 Wiki. Ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[25] Frost, C., Mammarella, M., Kohler, E., de los Reyes, A., Hovsepian, S., Matsuoka, A., and Zhang, L. Generalized File System Dependencies. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP '07.

[26] Fusion-io. NVM Primitives API Specification 1.0. http://opennvm.github.io/nvm-primitives-documents/, feb 2014.

[27] Gehani, N. H., Jagadish, H. V., and Roome, W. D. OdeFS: A File System Interface to an Object-Oriented Database. In *Proceedings of the 20th International Conference on Very Large Data Bases* (1994), VLDB '94.

[28] Google. LevelDB Benchmarks. http://leveldb.googlecode.com/svn/trunk/doc/benchmark.html, July 2011.

[29] Hitz, D., Lau, J., and Malcolm, M. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference* (1994), WTEC '94.

[30] Intel. Intel® SSD 330 Series (120GB, SATA 6Gb/s, 25nm, MLC). http://ark.intel.com/products/67287/Intel-SSD-330-Series-120GB-SATA-6Gbs-25nm-MLC.

[31] Jeong, S., Lee, K., Lee, S., Son, S., and Won, Y. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), ATC '13.

[32] Kang, W.-H., Lee, S.-W., Moon, B., Kee, Y.-S., and Oh, M. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014), SIGMOD '14.

[33] Kang, W.-H., Lee, S.-W., Moon, B., Oh, G.-H., and Min, C. X-FTL: Transactional FTL for SQLite Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD '13.

[34] Kashyap, A. File System Extensibility and Reliability Using an in-Kernel Database. Master's thesis, Stony Brook University, December 2004. Technical Report FSL-04-06, http://www.fsl.cs.sunysb.edu/docs/kbdbfs-msthesis/kbdbfs.pdf.

[35] Kawaguchi, A., Nishioka, S., and Motoda, H. A Flash-memory Based File System. In *Proceedings of the 1995 USENIX Technical Conference Proceedings* (1995), ATC '95.

[36] Kim, S., Lee, M. Z., Dunn, A. M., Hofmann, O. S., Wang, X., Witchel, E., and Porter, D. E. Improving Server Applications with System Transactions. In *Proceedings of the 7th ACM European Conference on Computer Systems* (2012), EuroSys '12, ACM.

[37] MariaDB. Fusion-io DirectFS atomic write support. https://mariadb.com/kb/en/mariadb/documentation/getting-started/mariadb-performance-advanced-configurations/fusion-io/fusion-io-directfs-atomic-write-support/.

[38] MariaDB. Isolation level. https://mariadb.com/kb/en/sql-99/37-sql-transaction-concurrency/isolation-level/.

[39] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., and Vivier, L. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux Symposium* (2007).

[40] Microsoft. Isolation Levels in the Database Engine. https://technet.microsoft.com/en-us/library/ms189122%28v=SQL.105%29.aspx.

[41] Microsoft. Performance Considerations for Transactional NTFS. http://msdn.microsoft.com/en-us/library/windows/desktop/ee240893(v=vs.85).aspx.

[42] Microsoft. Transactional NTFS (TxF). http://msdn.microsoft.com/en-us/library/bb968806(v=vs.85).aspx.

[43] Min, C., Kim, K., Cho, H., Lee, S.-W., and Eom, Y. I. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST '12.

[44] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Transaction Database System 17*, 1 (Mar. 1992).

[45] Murphy, N., Tonkelowitz, M., and Vernal, M. The design and implementation of the database file system, 2002.

[46] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), OSDI '06.

[47] Olson, M. A. The Design and Implementation of the Inversion File System. In *Proceedings of the 1993 USENIX Winter Technical Conference* (1993).

[48] Oracle. Database Concepts: 9. Data Concurrency and Consistency. https://docs.oracle.com/cd/E11882_01/server.112/e40540/consist.htm.

[49] Oracle. Oracle Internet File System Setup and Administration Guide. http://docs.oracle.com/cd/A97336_01/cont.102/a81197/toc.htm.

[50] Oracle Corporation. Oracle VM VirtualBox® User Manual. http://www.virtualbox.org/manual/ch12.html#idp59653904.

[51] Ouyang, X., Nellans, D. W., Wipfel, R., Flynn, D., and Panda, D. K. Beyond Block I/O: Rethinking Traditional Storage Primitives. In *17th International Conference on High-Performance Computer Architecture (HPCA)* (2011), pp. 301–311.

[52] Park, S., Kelly, T., and Shen, K. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), EuroSys '13.

[53] Park, S., Yu, J. H., and Ohm, S. Y. Atomic Write FTL for Robust Flash File System. In *Proceedings of the Ninth International Symposium on Consumer Electronics (ISCE 2005)* (june 2005), pp. 155 – 160.

[54] Pillai, T. S., Chidambaram, V., Alagappan, R., Al-Kiswany, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. 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* (2014), OSDI '14.

[55] Porter, D. E., Hofmann, O. S., Rossbach, C. J., Benn, A., and Witchel, E. Operating System Transactions. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (2009), SOSP '09.

[56] Prabhakaran, V., Rodeheffer, T. L., and Zhou, L. Transactional Flash. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), OSDI'08.

[57] Red Hat Software. Device-mapper Resource Page. https://www.sourceware.org/dm/.

[58] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems (TOCS) 10*, 1 (Feb. 1992).

[59] Santos, J., and Rao, S. Flexible File System Benchmark). http://sourceforge.net/projects/ffsb/.

[60] Seltzer, M. I. Transaction Support in a Log-Structured File System. In *Proceedings of the 9th International Conference on Data Engineering* (1993).

[61] SNIA. NVM Programming Model (NPM) Version 1. Tech. rep., December 2013.

[62] SNIA. Solid State Storage (SSS) Performance Test Specification (PTS) Enterprise Version 1.1. Tech. rep., September 2013.

[63] Spillane, R. P., Gaikwad, S., Chinni, M., Zadok, E., and Wright, C. P. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proccedings of the 7th Conference on File and Storage Technologies* (2009), FAST '09.

[64] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS File System. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (1996), ATC '96.

[65] Verma, R., Mendez, A. A., Park, S., Mannarswamy, S., Kelly, T., and III, C. B. M. Failure-Atomic Updates of Application Data in a Linux File System. In *Proceedings of the 13th USENIX conference on File and Storage Technologies* (2015), FAST'15, USENIX Association.