Making Application-level Crash Consistency Practical on Flash Storage

Dong Hyun Kang, Changwoo Min, Sang-Won Lee, and Young Ik Eom

Abstract—We present the design, implementation, and evaluation of a new file system, called *ACCFS*, supporting application-level crash consistency as its first-class citizen functionality. With *ACCFS*, application data can be correctly recovered in the event of system crashes without any complex update protocol at the application level. With the help of the *SHARE* interface supporting atomic address remapping at the flash storage layer, *ACCFS* can easily and efficiently achieve crash consistency as well as single-write journaling. We prototyped *ACCFS* by slightly modifying the full data journal mode in ext4, implemented the *SHARE* interface as firmware in a commercial SSD available in the market, and carried out various experiments by running *ACCFS* on top of the SSD. Our preliminary experimental results are very promising. For instance, the performance of an OLTP benchmark using MySQL/InnoDB engine can be boosted by more than 2–6x by offloading the responsibility of guaranteeing the atomic write of MySQL data pages from the InnoDB engine's own journaling mechanism to *ACCFS*. This impressive performance gain is in part due to the single-write journaling in *ACCFS* and in part comes from the fact that the frequent `fsync()` calls caused by the complex update protocol at the application level can be avoided. *ACCFS* is a practical solution for the crash consistency problem in that (1) the *SHARE* interface can be, like the TRIM command, easily supported by commercial SSDs, (2) it can be embodied with a minor modification on the existing ext4 file system, and (3) the existing applications can be made crash consistent simply by opening files in `O_ATOMIC` mode while the legacy applications can be run without any change.

Index Terms—File system-level consistency, application-level consistency, flash storage device, address remapping.

1 Introduction

To guarantee the consistency of file metadata, data blocks, and versions, modern file systems have heavily resorted to various techniques such as journaling and copy-on-write [1]. Unfortunately, they suffer from heavy read/write amplification incurred by the mechanisms inherent in each scheme, including redundant write [2], [3], segment cleaning [4], [5], and tree wandering [6]. Furthermore, because they do not provide the higher application-level crash consistency[1] (hereafter, for short, crash consistency) [7], [8], [9], [10], [11], many consistency-critical applications (*e.g.*, MySQL [12], SQLite [13], git, and VMware) should implement their own idiosyncratic mechanisms for ensuring the secure recovery of their data from unexpected crashes, which are, in some cases, still crash-vulnerable [10], [11], [14].

Considering that flash memories are being used as main storage, especially for performance critical applications, it is an urgent and practical problem for file system communities to develop flash-tailored solutions for higher consistency level (*e.g.*, crash consistency) and for higher performance (*e.g.*, no redundant write), by leveraging the new interface such as *SHARE*. To this end, we address two problems in file systems, IO amplification for data consistency and lack of crash consistency, especially focusing on ext4 journaling mechanism, with the *SHARE* flash storage interface [15]. The *SHARE* interface allows host programs to explicitly remap one or more pairs of LBAs atomically at the flash storage FTL layer. Though simple, it is very effective in eliminating the overhead of *redundant writes for guaranteeing atomic write, excessive read/writes in compaction, and the tree-wandering problem* in database applications such as MySQL DWB (double-write buffer) [12] and Couchbase storage engine [15]. One coincident and intriguing observation is that these database overheads have essentially same characteristics with those consistency overheads in modern file systems.

Based on this observation, in this paper, we propose *ACCFS* (*Application-Crash-Consistent File System*), which extends the existing ext4 journaling file system naturally and minimally so as to utilize the *SHARE* interface, thus achieving both higher performance (*i.e.*, no redundant writes) and higher-level consistency (*i.e.*, crash consistency). *ACCFS* makes two main contributions: single-write journaling and application-level crash consistency. For single-write journaling, *ACCFS* provides *SHARE*-aware data journaling (SDJ) mode, which can achieve the highest data consistency (*i.e.*, version consistency [1]) at the same performance of ordered journal (OJ) mode. We slightly modified the data journal (DJ) mode in ext4 file system so that, after (metadata and data) blocks have been successfully written in journal area, a *SHARE* call for the multiple blocks is made, instead of making `pdflush daemon` to write them redundantly in their original locations. Thus, *ACCFS* can achieve high data consistency with single write.

Next and more importantly, *ACCFS* guarantees the atomic write of multiple scattered pages in either single or multiple files opened with `O_ATOMIC` flag. We adopted the semantics

- *Dong Hyun Kang is with the Department of Computer Engineering, Dongguk University-Gyeongju, Gyeongju, 38066, Korea. E-mail: dhkang@dongguk.ac.kr*
- *Changwoo Min is with the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24060, USA. E-mail: changwoo@vt.edu*
- *Sang-Won Lee and Young Ik Eom are with the College of Computing, Sungkyunkwan University, Suwon, 16419, Korea. E-mail: {swlee | yieom}@skku.edu*

Manuscript received April 19, 2005; revised August 26, 2015.

1. A file system allows upper-layer applications to explicitly guarantee their own transactions without any interferences.

of `fsync()` and `syncv()` calls slightly, added two new system calls, `abort()` and `abortv()`, and modified the commit/checkpoint/recovery operations in DJ mode (minimal changes and very compatible to DJ mode). Without *SHARE*, as will be detailed later, this light-weight implementation of solid application-level crash consistency would not be possible even with double-write journaling. While designing *ACCFS*, we identified an interesting recovery property, called *A-property*.

The focus of *ACCFS* is on crash consistency and thus it leaves concurrency control management (*i.e.*, isolation) to the applications, as other file systems do [7], [8], [9], [10], [11]. We prototyped *ACCFS* by modifying ext4 journal (kernel version: 4.6.7) on top of a commercial SSD available in the market, inside which we implemented the *SHARE* interface as a firmware. Our preliminary evaluations confirm that the effect of *ACCFS* is very promising. As an example, when we ran an OLTP benchmark using MySQL/InnoDB DBMS on top of *ACCFS*, we observed 6x TPS improvement over the default configuration where the benchmark was run with InnoDB engine's own journaling mode (*i.e.*, DWB: double-write buffer[2]) enabled. This surprising performance improvement can be explained as follows: by offloading the responsibility of guaranteeing the atomic full page write from InnoDB engine itself to *ACCFS*, it can halve the amount of data being written to the flash storage, and can, more importantly, reduce the number of `fsync()` system calls by 16.4 times. Also in SQLite database, we observed that, compared to the case when it was run in either RBJ (roll-back journal) or WAL (write-ahead logging) mode with ext4 ordered-mode journaling, SQLite on *ACCFS* can achieve better performance (by up to 3.3x) as well as the same crash consistency even when its journaling mode is turned off. From these results, we confirm that *ACCFS* can make many consistency-critical applications high-performant and also free from the burden of devising their own idiosyncratic mechanisms for crash consistency. The benefits of *ACCFS* can be summarized as follows:

1. TRIM-like: As will be shown in this paper, like the well known TRIM command, which has successfully been incorporated into major OS/file system kernel, 1) the *SHARE* interface could be easily supported by commercial SSDs, 2) *ACCFS* can be built with minimal extension on the existing file systems (*e.g.*, 300 lines added to DJ mode), and 3) it will have high performance impact on a variety of applications.

2. Portability [10]: The existing legacy applications can run under *ACCFS* without any modification, and they are provided with higher data consistency. And, *ACCFS* allows applications to be made crash consistent simply by adding `O_ATOMIC` flag to `fopen()`. Both types of applications can run concurrently under *ACCFS*.

In the rest of this paper, we will discuss background (Section 2) and related work (Section 3). Next, we will present the details of *ACCFS* design (Section 4) and implementation (Section 5). Then, we will show our evaluation results (Section 6). Next, to study the performance impact on different file systems, we present the evaluation results of ACCFS on a log-structured file system (Section 7). Finally, we conclude the paper (Section 8).

2. It appends a new copy to the double-write-buffer and then overwrites the old copy in its original location.

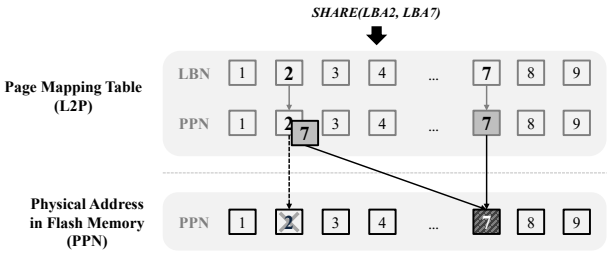


Fig. 1: *SHARE* Interface.

2 Flash Memory, FTL, and *SHARE*

Because flash memory does not allow to update pages in place, an out-of-place update strategy is commonly taken by every flash storage devices. Thus, to maintain the ever-changing mapping between logical addresses and physical flash memory addresses, every flash storage device are equipped with a firmware module called *FTL* (flash translation layer), and the fine-grained page-mapping approach is popular mainly for performance reasons.

In order to leverage this indirection of page-level address mapping in flash storage, recently in database community, Oh *et al.* [15] proposed the *SHARE* interface. It exposes an abstraction that allows host applications to explicitly ask FTL to change the *internal* address mapping maintained by FTL. To be concrete, as illustrated in Figure 1, upon receiving a `share` command from the host with a pair of two logical block addresses, *LBA2* and *LBA7*, as its parameter, FTL changes the PPN (physical page number) of *LBA2* in its page-mapping table to that of *LBA7*, thus the latter physical page being shared by the former logical address. A `share` command can have an optional third argument, `length`, when the length of data to be shared is longer than the FTL mapping granularity (*i.e.*, 4KB). Though the description so far assumes that a `share` command is associated with a single pair of LBAs, it can have multiple LBA pairs in a batch. In this case, FTL should be able to support the *atomic address remapping* for the given set of LBA pairs upon a system crash or power-off failure.

The *SHARE* interface, though simple, has proven to be very effective in reducing the read/write amplification in various database applications [15]: with the help of *SHARE*, 1) the atomic full page write, which is critical to MySQL database, can be achieved without the double-write mechanism in InnoDB engine, 2) the read/write amplification of the compaction operation in NoSQL databases can be replaced by zero-copy compaction, and 3) the write amplification problem in CoW B-tree, so called tree-wandering, can be avoided in Couchase NoSQL storage engine. One strong motivation of our work on *ACCFS* is that journaling-based file systems can also benefit from *SHARE* by applying the single-write journaling of *SHARE*-aware InnoDB engine to the ext4 data journal mode. The idea of applying *SHARE* to file systems is not limited to journaling file system, and would be also very helpful in optimizing the segment cleansing overhead in log-structured file systems [4], [5] and the tree-wandering problem in CoW B-tree file systems [6].

3 Related Work

The major challenge of both file systems and consistency-critical applications is how to update their data objects persistently even when a power loss or system crash occurs. Many researchers in academia and industry focused on studying the crash consistency mechanism that guarantees the data objects on storage devices are left in a consistent state even when system crash or power failure occurs after write operations. In this section, we briefly review and compare exiting researches that are closely related to our work: 1) file system consistency and 2) application-level crash consistent file systems.

3.1 File System Consistency

By using a variety of techniques such as journaling and copy-on-write, modern file systems provide various consistency levels including metadata, data, and version consistency [1]. However, the historical consistency techniques often raise the issue of the performance drop because they have been built on redundant I/O operations and synchronization operations (*e.g.*, `fsync()`, `fdatasync()`, and `msync()`). Therefore, many prior works tried to efficiently solve the performance challenges of consistency techniques in the file system's point of view [1], [16], [17], [18], [19]. For example, RFLUSH [17] allows a fine-grained `flush` command to reduce it's processing time. Park *et al.* [19] introduced OFTL, where it includes an ordering mechanism, to simplify the journaling procedure. Besides, the system-wide consistency currently provided by file systems is a broken abstraction for application-level crash consistency [10]. Therefore, many applications such as SQLite [13] and Vim [20] should craft their own complex update protocols that ensure its data to be correctly recovered upon unexpected system crash or power failure, mainly by calling `fsync()` system calls for ordering and durability. Unfortunately, they suffer from poor performance due to frequent `fsync()` calls [1], [16], and some of them are still even vulnerable to crashes [11].

Meanwhile, our *SHARE*-aware *ACCFS* is not the first work on exploiting the address remapping for file system optimization with the hardware support. To our knowledge, JFTL [21] is the first approach to suggest the atomic address remapping functionality in FTL so as to avoid the redundant write overhead in journaling file system. In this sense, it is the closest approach to *ACCFS*. But, unlike *ACCFS*, the JFTL did not consider the application-level crash consistency at all, and it uses a proprietary interface between the host and flash storage for remapping the journaled data. Finally, we argue that *ACCFS* is a principled and practical way to change this landscape, which supports crash consistency as its first-class citizen functionality inside the file system. In addition, *ACCFS* can, with the help of *SHARE*, provide all consistency levels at no cost of redundant writes.

3.2 Application-Level Crash Consistent File System

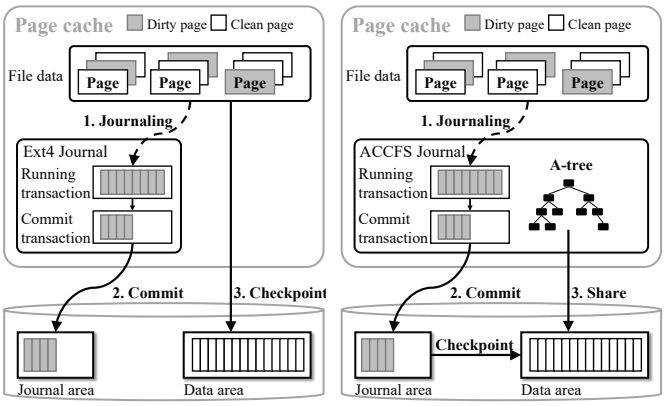
As far as we know, many crash consistent file systems have been proposed: TxFlash [22], MARS [23], Failure-atomic Msync [7], [8], CFS [9], and T2FS [24]. CFS is similar in spirit to our *ACCFS* in that it achieves application-level crash consistency by utilizing a transactional storage, called X-FTL [25], which can atomically update multiple (scattered) pages in place, and by extending an existing file system. However, our *ACCFS* is more practical than prior works in that *SHARE* is simple enough to be easily added to the existing SSDs while X-FTL requires SSD to support complex concepts including transaction identifier, commit, and abort. Moreover, while CFS introduced new APIs such as `cfs_begin` and `cfs_end` to define the transactional scope, *ACCFS* utilizes existing APIs such as `fsync()` and `syncv()` for that purpose. For this reason, existing applications can be made more crash consistent with *ACCFS* rather than CFS. Next, TxFlash [22] and MARS [23] have been built on transactional storage systems to ensure the atomicity property. Park *et al.* [7] proposed failure-atomic msync(), which can atomically update the changes of an mmap-ed file using REDO journaling. Verma *et al.* [8] extended the work mainly in two directions; firstly, in order to avoid the redundant writes, data blocks are managed in a CoW style, and secondly, in order to support the failure-atomicity for multiple files, they suggested the `syncv()` call. In particular, the second work [8] is unique in two folds: 1) it demonstrates that the crash consistency can be achieved without help of special hardware, and 2) it proposes small but elegant set of APIs for developing crash consistent applications. However, its CoW style data management will newly introduce the space fragmentation and thus may require costly garbage collection overhead; CoW style amplifies write operations by 8x compared to ext4 in case of file overwrite [26]. In contrast, our *ACCFS* is built with minimal changes in the existing ext4 file system with the help of *SHARE* interface, and thus we believe this would be more practical approach for crash consistent file system.

4 Design of *ACCFS*

4.1 Overview

Guaranteeing crash consistency is one of the most important factors in designing a file system. But, there is a trade-off between consistency level and performance. For this reason, the ext4 file system takes a relaxed consistency, i.e., the ordered journaling mode (OJ), as its default mode. This mode of OJ provides *data consistency* [1], which only guarantees that metadata are entirely consistent to the data and that the same data read by a file legitimately belongs to that file. Therefore, under the OJ mode, a file can point to the older version of its data, which is the source of the well-known *torn page* problem in database systems. In contrast, the full data journal mode (DJ) supports *version consistency* [1], where the metadata version is guaranteed to match to the version of the referred data. But, this higher consistency in the DJ mode comes at the expense of considerable performance degradation due to double-write journaling of data as well as metadata.

Basically, *ACCFS* is based on ext4 journaling file system. But, unlike ext4 file system, one of its main design goal is to provide higher consistency at no compromise of performance. To satisfy this goal, *ACCFS* takes advantage of the atomic address remapping provided by *SHARE* interface at the flash storage layer, thus offloading the burden of guaranteeing system-wide version consistency from file system to flash storage and, at the same time, achieving higher performance almost for free without resorting to costly journaling scheme.



(a) ext4 **(b)** *ACCFS*

Fig. 2: The overview of journaling process in ext4 and *ACCFS* file systems.

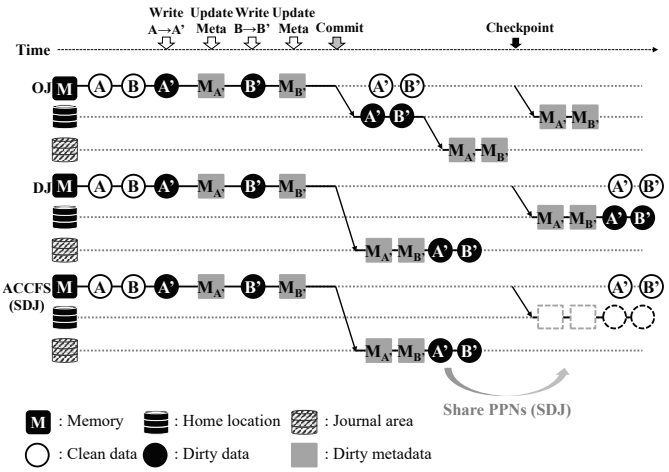


Fig. 3: Comparison of journaling modes in ext4 and *ACCFS*. In the scenario of the figure, two different files are updated with small changes. Ext4 file system provides two journal modes: ordered journaling mode (OJ) and full data journaling mode (DJ). Checkerboard rectangles denote journaled blocks into the journal area. OJ mode in ext4 guarantees a minimal crash consistency: data and metadata of the file system will be preserved in ordered manner and flushed to the home location at checkpoint time (total 6 block writes). DJ mode in ext4 provides *version consistency*: data and metadata of the file system are synchronously logged into the journal area and then flushed to the home location by periodic checkpoint operation (total 8 block writes). SDJ in *ACCFS* follows the default rules of DJ mode except for checkpoint operation: data and metadata of the file system are synchronously logged and then reflected to the home location through *SHARE* interface (total 4 block writes with 1 `share` command).

In addition, by slightly modifying the existing data journaling mode of ext4 and also causing no extra run-time overhead, *ACCFS* can support higher application-level crash consistency as its first-class citizen functionality, thus freeing the application developers from the burden of devising a complex and costly update protocol for application crash consistency.

Figure 2 illustrates the overview of journaling process in ext4 and *ACCFS*. As depicted in Figure 2, while the journaling and commit processes in *ACCFS* are almost same to those in ext4, the checkpoint process in *ACCFS* is in stark contrast with that in ext4. While the second write of each journaled block to its home location happens in ext4, the second write of the same block is replaced by *SHARE* command in *ACCFS*. *ACCFS* uses an auxiliary red-black tree, called A-tree (atomic-tree), which helps to batch multiple journal writes into a single `share` command by keeping a set of LBA pairs of home location and journaled location on DRAM; a home location is used as a key for the given LBA pair to handle consecutive updates within one transaction. *ACCFS* uses A-tree to make the checkpoint operation faster in SDJ and to support isolation among multiple transactions in SADJ, which we will discuss in more detail in the later sections. When commit operation is triggered in *ACCFS*, each write operation for journaling is first recorded in the journal area and then the relevant LBA pair is inserted into the A-tree of the *SHARE* interface. At each checkpoint, *ACCFS* generates a `share` command by searching LBA pairs on the A-tree belonging to the checkpoint transaction, and issues the `share` command to the underlying storage. At this time, the home locations in the storage are atomically shared with the journaled locations at hardware level. Finally, for the next checkpoint, the previous LBA pairs in the A-tree are discarded. In this way, *ACCFS* can avoid unnecessary overhead caused by redundant journaling writes, which we call single-write journaling.

4.2 *SHARE*-aware Data Journaling (SDJ)

For efficient single-write journaling, *ACCFS* provides *SHARE*-aware data journaling (SDJ) mode that can achieve system-wide version consistency at the same performance of ordered journal (OJ) mode. In this section, we show in detail how *SHARE* interface can be leveraged to guarantee the highest data consistency (*i.e.*, version consistency). Figure 3 illustrates the SDJ procedure in comparison with ordered journaling (OJ) mode and data journaling (DJ) mode in ext4. When a commit operation is triggered by time (*e.g.*, 5 second) or a synchronization operation (*e.g.*, `fsync()`, `fdatasync()`, and `msync()`), OJ mode first writes dirty pages (A′ and B′) to their home locations[3] and then synchronously writes a journal descriptor (JD) block and metadata pages ($M_{A'}$ and $M_{B'}$) to the journal area. The JD block has the home locations of journaled metadata blocks [2] for recovery. After that, OJ mode writes a journal commit (JC) block together with the force unit access (FUA) command to the journal area, to mark the end of a journal commit transaction. Note that dirty metadata pages have not yet been written to the home location. These metadata pages asynchronously flush to their home location ($M_{A'}$ and $M_{B'}$) by either checkpoint or flush daemon. On the other hand, upon a commit operation, DJ mode in ext4 synchronously writes data (A′ and B′) and metadata blocks ($M_{A'}$ and $M_{B'}$) with a JD block to the journal area and then synchronously writes the JC block together with FUA command to the journal area. Since the same version of data and metadata blocks are in the journal area, DJ mode completely guarantees the version consistency.

3. The dirty page writes are asynchronously issued but their completion should be enforced prior to writing a journal commit (JC) block using an FUA command.

Later upon a checkpoint operation, DJ mode asynchronously writes dirty pages (A′, B′, $M_{A'}$, and $M_{B'}$) to their home location of journaled blocks.

SDJ in *ACCFS* follows the similar steps to the DJ mode in ext4 until the commit operation is completed. At commit time, unlike DJ mode, SDJ fetches the LBA address for the home location (which keeps the persistent original data), and the LBA address for the journaled location (which keeps the up-to-date data) from the `journal_head` structure, after each journal write would succeed. In other words, SDJ stores the home location and journaled location as an LBA pair at the commit time. A given LBA pair is inserted into the A-tree with an integer key that is used to find the pair at checkpoint time. Once a checkpoint operation is triggered (*e.g.*, due to no free space in the journal area, periodical time interval, or an application's calling a `sync()` system call), SDJ scans all the nodes in A-tree to build a `share` command. Note that SDJ can quickly build `share` commands by scanning the A-tree. After finishing the checkpoint operation, SDJ discards not only a set of LBA pairs, which were reflected to the storage via the last `share` command, but also a set of invalid LBA pairs that were invalidated during the last commit operation. In this way, *ACCFS* builds more robust and reliable system-wide version consistency while improving the overall performance. Therefore, *ACCFS* uses SDJ as its default consistency mechanism.

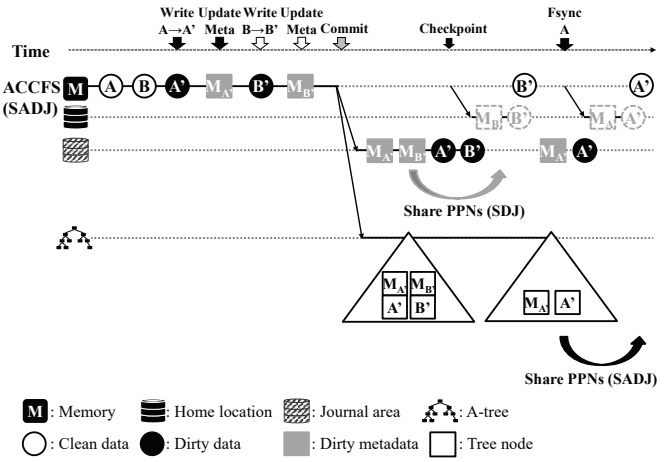


Fig. 4: Two applications running on *ACCFS*. One process opens file A with `O_ATOMIC` flag and the other process opens file B without the flag. Upon commit time, *ACCFS* synchronously writes data and metadata blocks into the journal area and inserts pairs of LBAs into the A-tree. Upon checkpoint, the blocks associated with file B (B′ and $M_{B'}$) are reflected to the home location using a `share` command. However, at checkpointing file B, the blocks associated with file A (A′ and $M_{A'}$) will not be reflected to the home location. They will be reflected to the home location by a `fsync()` operation of file A.

4.3 *SHARE*-aware Application-level Data Journaling (SADJ)

For some applications such as databases and key-value stores, even the system-wide version consistency by ext4 DJ mode, despite its double-write journaling, fails to meet their stringent requirements for transactional atomicity. For this reason, each application should devise its own application-level crash consistency mechanism. However, such application-level crash consistency mechanisms bring about two problems. First, they usually suffer from poor performance and the reduced lifespan of the underlying flash storage mainly because of write amplification and frequent `fsync()` calls from the application layer. Second, the application-level update protocols are complex and error-prone so that, as shown in recent studies [10], [11], [14], there still exist some subtle bugs even in widely-deployed applications. For example, SQLite has an update transaction protocol that issues lots of `fsync()` operations to consistently update its own database files. Unfortunately, this complex update protocol is error-prone because the `fsync()` operations are ignored willfully by the underlying kernel stacks, such as file systems and device drivers, for performance optimization. Therefore, it is imperative for file systems to support application-level crash consistency that has the same spirit of update transaction protocol of applications.

In this section, we describe *SHARE*-aware application-level data journaling (SADJ), which can provide application-level crash consistency by slightly extending SDJ mode. It was designed with the following two goals in mind:

- The interface for using SADJ should be simple and intuitive so that application developers can easily adopt it.
- Legacy applications should be able to run together with SADJ-based one with small changes

For an application to run in SADJ mode in *ACCFS*, it can use the failure-atomic update APIs (*i.e.*, `O_ATOMIC`, `syncv()`, and `msync()`) [7], [8]. Some applications (*e.g.*, SQLite [13]) require `abort()` protocols to roll back the changes to the most recent successful committed state. To this end, *ACCFS* newly introduced `abort()` and `abortv()` systems calls. Multiple files can be committed or roll-backed at once using `syncv()` or `abortv()`, respectively. These APIs can be easily incorporated to conventional applications with only a few lines of code changes.

If an application opens a file with `O_ATOMIC` option, its dirty pages are isolated from the normal journal pages that are handled by SDJ mode. Therefore, when a checkpoint is triggered by the pre-defined time (*e.g.*, 5 minutes), the dirty pages belonging to the file with `O_ATOMIC` not be reflected in their home location. After that, if an application calls one of the synchronization operations (*e.g.*, `fsync()`, `fdatasync()`, `msync()`, or `syncv()`), all dirty pages whose the file referred by the synchronization operation are permanently recorded to their home location in the atomic manner of "all or nothing". In order to realize such isolated management, there should exist a mechanism which allows us to easily determine whether each dirty page (*i.e.* journaled block) belongs to a file opened with `O_ATOMIC` or not. Therefore, we add a new flag, called `JBD2_A_FLAG`, to both `journal_head` structure and A-tree. Figure 4 briefly illustrates the sequence of SADJ mode.

Now let us explain in detail how *ACCFS* can guarantee both the application-level crash consistency and system-wide version consistency with *SHARE* interface as shown Figure 5.

1. Journaling: Once a page on the page cache is written by an application, its status is changed from clean to dirty

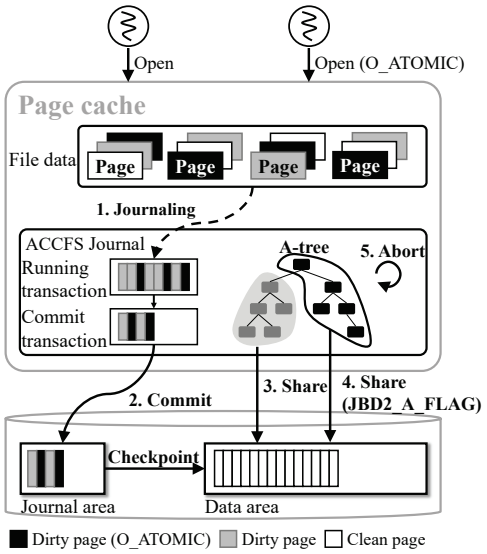


Fig. 5: Application-level data journaling in *ACCFS*.

(we call it a dirty page). Like ext4, SADJ in *ACCFS* inserts information for each dirty page, which keeps file data, to the running transaction.

2. Commit: At commit time, the running transaction is updated to the commit transaction. SADJ synchronously write all dirty pages of data and metadata belonging to the committing transaction to the journal area in the storage; each dirty page is mapped to one journal block in terms of ext4. To correctly recover such journaled blocks after system crashes or power failures, ext4 additionally records `journal_heads` of each journal block to the journal area. While writing each journal block to the journal area, SADJ checks whether the dirty page belongs to the file that was opened with `O_ATOMIC`, and in that case SADJ sets its `JBD2_A_FLAG` inside `journal_head` structure and inserts a LBA pair with `JBD2_A_FLAG` into A-tree.

3. Share: When checkpoint is triggered, SADJ first searches for LBA pairs belonging to the checkpoint transaction in A-tree and then checks their `JBD2_A_FLAG`. If a `JBD2_A_FLAG` is set, SADJ skips the LBA pair and move on to the next for building `share` command. Finally, like SDJ, SADJ also discards a set of invalid LBA pairs in A-tree. Note that the skipped LBA pairs can be updated more than once until the application issues a synchronization command. In other words, LBAs that belong to an application-level transaction is isolated from LBAs belonging to other concurrent filesystem-level transactions.

4. Share (`JBD2_A_FLAG`): Meanwhile, a set of LBA pairs whose `JBD2_A_FLAG` was set to 1 at commit time is sent to the storage via `share` command when an application calls `fsync()` to make the updates data of the file (which was opened with `O_ATOMIC`) persistent. After issuing the `share` command, SADJ discards those LBA pairs in A-tree for the next synchronization operation.

5. Abort: When an application calls `abort()` against a specific file, SADJ first searches for all dirty pages (which belong to the file) on the page cache and then fetches each journaled LBA address from the `journal_head` of each page. Finally, SADJ disrupts a set of LBA pairs by scanning A-tree for them using the searched LBA addresses. Note that SADJ does nothing for rollback during storage operations because journal blocks that contain data to be ignored were only placed in the journal area. Those journal blocks will never be reflected to the home locations because SADJ reuses them based on a round-robin order.

4.4 Technical issues of *ACCFS*

In comparison to ext4, *ACCFS* raises two technical issues about performance and correctness: optimal size of journal area and identification of valid journal block. Let us discuss each issue in turn.

Size of journal area: In *ACCFS*, the size of journal area has a huge effect on the overall performance because *ACCFS* triggers a checkpoint operation to reclaim journal blocks in the storage when it runs out of journal space. And, since *ACCFS* keeps data blocks as well as metadata blocks in the journal area, this performance issue is exacerbated when it uses the small-sized journal area (*e.g.*, 128MB). In addition, in terms of application-level crash consistency, perhaps some applications would likely need large journal area to guarantee their application-level consistency. To resolve this issue caused by small-size journal area, we decided to allocate rather large-sized journal area (*e.g.*, 1GB) which can be expected to preserve all data requiring application-level crash consistency. Of course, one possible solution to completely address the issue is to extend the fixed journal area in a dynamic way. We will leave it for our future work.

Valid journal block identification: SADJ in *ACCFS* should maintain some journal blocks (which we call valid journal block) in the journal area for guaranteeing the application-level crash consistency. However, those journal blocks can be unintentionally over-written because a new journal block is assigned in a round-robin manner. To prevent such data corruption, SADJ allows skipping those journal blocks and allocates a new journal block in the journal area; this allocation is similar to the slack space recycle (SSR) of F2FS [27]. To achieve this, SADJ looks for a new LBA address in A-tree, which holds LBA addresses for all journaled blocks, before assigning a new journal block. If the new LBA address exist in A-tree, SADJ increments the LBA address until it finds an LBA address that does not exist in A-tree (*i.e.*, invalid journal block). Fortunately, this situation rarely happens because journal area of *ACCFS* (*e.g.*, 1GB) is large enough to guarantee application-level crash consistency without any data corruption.

4.5 Recovery

In the event of system crash or application failure, *ACCFS* can completely guarantees both system-wide version consistency and application-level crash consistency. Basically, it takes the roll-back recovery in the ext4 journaling scheme. But the recovery process in *ACCFS* is unique in two aspects: (1) *SHARE*-aware zero-copy for all committed blocks with `JBD2_A_FLAG` disabled and (2) *A-property*, which states that all `JBD2_A_FLAG`-ed blocks can be safely ignored during the recovery process.

***SHARE*-aware zero-copy recovery:** During the recovery, for each journal commit block (JC) encountered while scanning the journal area, *ACCFS* finds its corresponding journal

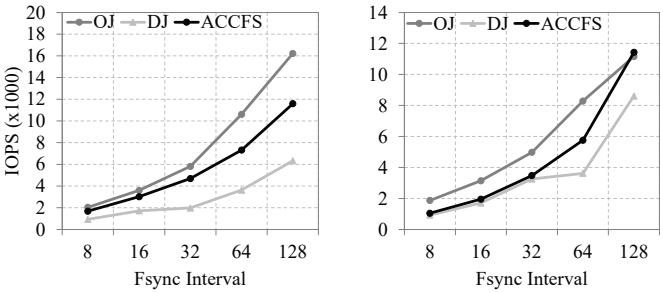
descriptor (JD) block [2]. Then, for all non-JBD2_A_FLAG-ed blocks between JD and JC, it generates and issues a share command, which contains pairs of LBAs, so as to keep data and metadata up-to-date. This step is repeated until the last commit block is encountered in the journal area. As in ext4, all the remaining blocks after the last commit block can be safely ignored from the recovery perspective. Considering that the redundant write of every journaled block to its home location in ext4 is replaced by a zero-copy share command in *ACCFS*, it is quite obvious that *ACCFS* can recover much faster than ext4. Note that this *SHARE*-based recovery process is idempotent and thus the recovery process can be simply repeated when another crash is encounter during the recovery.

A-property: Now let us explain the *A-property* which is used in the recovery process of SADJ mode. It states that every JBD2_A_FLAG-ed block in the journal area can be safely ignored during the recovery. As stated above, at the moment when an application calls fsync() against a specific file, every data or metadata block with JBD2_A_FLAG enabled belonging to the file will be propagated to its original location in an atomic manner by a share command, which will be executed only after the JC block for the fsync() is synchronously saved in the journal area. Therefore, it is guaranteed that all the blocks of a successfully fsync()-ed file is propagated to their original location. In case when the system crashed before the application is acknowledged for the fsync() call, each block belonging to the file should not be propagated to its original location. Consequently, all the JBD2_A_FLAG-ed blocks could be simply skipped during the recovery. Also, *A-property* makes it easy to implement abort() and abortv() APIs because these APIs do not need to remove the journaled blocks in the journal area at the time of abort(). In summary, this property is essential in making our *ACCFS* guarantee application-level crash consistency in SADJ mode.

One interesting question with regard to *A-property* is whether it can also be embodied in the existing data journaling mode only if the concepts of A-tree and JBD2_A_FLAG are introduced, and the answer is no. In fact, *A-property* is a combined effect of taking all three techniques, A-tree, JBD2_A_FLAG, and *SHARE* interface. Let us assume that a file opened with O_ATOMIC is being updated by an application and for any reason one or more JBD2_A_FLAG-ed blocks from the file are already journaled before the application invokes the fsync() call to make its recent update durable. In this case, upon recovery, the existing data journal mode can not decide whether those blocks should be copy-backed to their original locations because it has no information regarding whether the fsync() call for those blocks succeeded.

5 Implementation

We implemented *ACCFS* in Linux kernel 4.6.7 by modifying about 400 lines of code (LoC) of ext4 and JBD2. We note that other file systems supporting application-level crash consistency [8], [9] need significant changes (*e.g.*, 5,800 LoC for CFS [9]), which inhibit wide and rapid adoption of the new storage interface in practice. In *ACCFS*, A-tree is implemented using a red-black tree maintaining mappings between destination LBA and source LBA. To enable the



(a) 1 job / 1000 files **(b)** 10 jobs / 100 files

Fig. 6: Performance comparison between *ACCFS* and ext4 for FIO microbenchmark (128MB journal size). For each graph, the x-axis shows fsync() interval that means the number of write operations between two consecutive fsync().

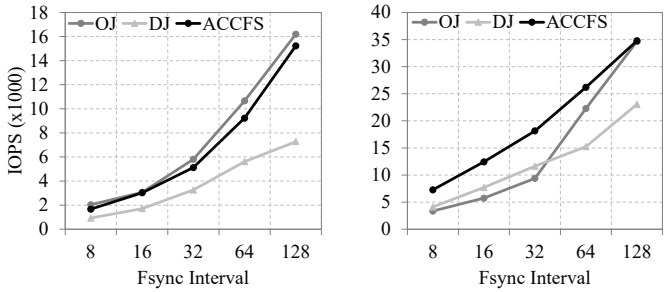
O_ATOMIC flag, we partially ported the clone feature of CoW to the A-tree of *ACCFS* since Verma *et al.* implemented the same flag based on CoW-based file system [8], [28]. We also added two new system calls, syncv() and abortv(), by using ioctl facility of the Linux. The new system calls are exposed to applications for transferring file descriptors whose files were opened with O_ATOMIC flag. Implementation details of the O_ATOMIC flag and syncv() system call can be found in [28] and [8], respectively. We also realized the idea of SADJ mode by modifying MySQL and SQLite. In general, MySQL and SQLite maintain a special area (*i.e.*, *double-write-buffer*, *write-ahead logging* or *roll-back journal*) to prevent their database corruption; they write up-to-date data its original location after recording the data into the special area according to pre-defined conditions (*e.g.*, an eviction operation or time threshold). Therefore, we modified MySQL and SQLite to make use of the share command in processing the write operation at its original location. In our implementation, each LBA of the database file is the key associated with a share command, and thus the part of the modified code keeps track of the mapping between LBA address of the original location and LBA address of the special area where it has up-to-date data. Some lines of new codes were added to issue the share command instead of the redundant write of data in the original location. Finally, we also inserted the new codes related to database files to enable O_ATOMIC flag.

We implemented a *SHARE*-enabled SSD by modifying an FTL firmware of a commercial high-end PCIe M.2 SSD supporting 360K and 280K IOPS for random read and write operations, respectively. And, since no matching command exists in current storage interface such as SATA and NVMe, the share command has been added as a *vendor unique command* (VUC) in the NVMe SSD. Implementation details of the share command can be found in [15].

6 Evaluation

For performance comparision, we ran all experiments on a system with a quad-core processor (Intel i7-6700) and 8GB memory. In this section, we present experiments that answers following questions:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2959305, IEEE Transactions on Parallel and Distributed Systems



(a) 1 job / 1000 files **(b)** 10 jobs / 100 files

Fig. 7: Performance comparison between *ACCFS* and ext4 for FIO microbenchmark (1GB journal size). For each graph, the x-axis shows `fsync()` interval that means the number of write operations between two consecutive `fsync()`.

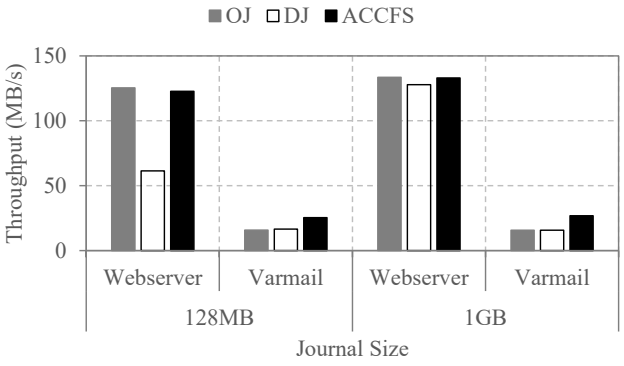


Fig. 8: Performance comparison between *ACCFS* and ext4

- Does SDJ in *ACCFS* provide high performance while guaranteeing the *version consistency*? (Section 6.1)
- How much can *version consistency* of SDJ help to improve real application performance? (Section 6.2)
- How well does SADJ in *ACCFS* guarantee application-level consistency? (Section 6.3)

6.1 Effect of *ACCFS* on Microbenchmarks

Normally, microbenchmark is widely used to evaluate the performance impact of file systems. So, we used two microbenchmarks, Flexible I/O (FIO) [29] and Filebench [30] benchmark, to compare *ACCFS* with the conventional ext4 file system. We compare three journal mode: ordered mode (OJ) journal, data mode (DJ) journal, and SDJ in *ACCFS*.

FIO microbenchmark: We first evaluated *ACCFS* using the FIO microbenchmark, which was configured to simulate data-heavy workload. We performed random writes 10GB of data with 8KB write granularity, and varied the number of threads and files to better investigate the performance of *ACCFS*. Since `fsync()` directly affects the amount of journal data and the overall performance, we ran the same pair of FIO with varying `fsync()` interval, which indicates the number of write operations between two consecutive `fsync()` calls.

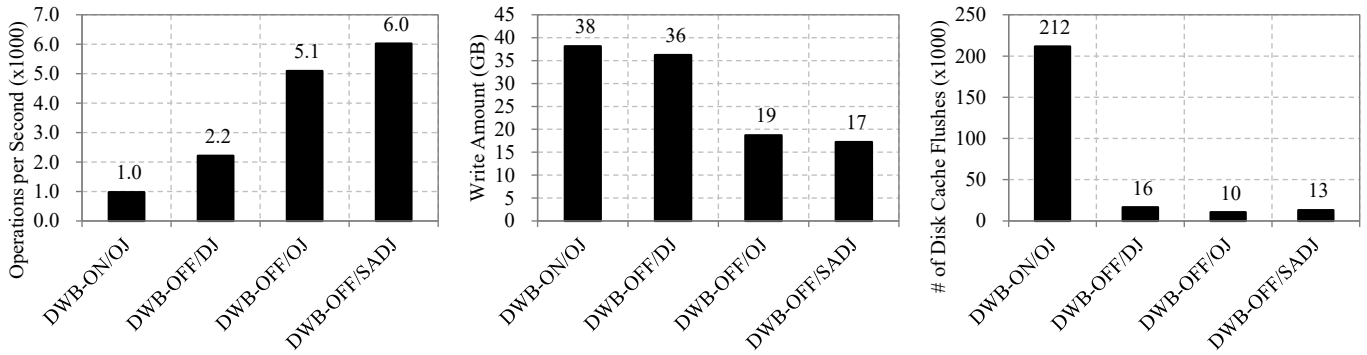
Figure 6 presents the throughput in IOPS for all the experiments when the journal size is 128MB, which is default for the ext4 file system. As we expected, this figure shows the notable performance gap between OJ and *ACCFS*. The major reason of the performance gap is that *ACCFS* frequently triggers checkpoint operations to reclaim journal blocks in the storage as mentioned in Section 4.4, because it preserves both data and metadata in the journal area unlike OJ mode. To confirm this consideration at runtime, we monitored the block traces by using Blktrace while running the benchmark and figured out that *ACCFS* significantly increases the number of `fsync()` operations for a short time. For fair comparison, we extended the journal size to 1GB, as in previous work [16] and again evaluated the same pair of FIO (Figure 7). As expected, Figure 7a clearly shows that *ACCFS* provides similar performance to OJ mode in all cases while it guarantees the system-wide *version consistency*. In addition, Figure 7b presents that the performance of *ACCFS* outperforms that of OJ mode by up to 2.16x. Meanwhile, one interesting finding from the results is that OJ mode reveals performance drop compared with DJ when it runs 10 FIO threads with short `fsync()` intervals, such as 8, 16, and 32 interval. This is because OJ mode suffers from the scalability issue of file systems [31] and random pattern writes. Another interesting finding is that 1GB journal size for *ACCFS* is large enough to hide the overhead (*e.g.*, frequent checkpoints) caused by the journal size while guaranteeing the system-wide version consistency.

Filebench microbenchmark: To emulate real-world I/O workload, we used the write-intensive Varmail and the read-intensive Webserver workload. The Varmail consists of 16 concurrent threads to simulate a mail server and each thread performs a set of create-append-sync and read-append-sync operations. The Webserver workload is also composed of 100 threads, each of which sequentially reads a whole file and then writes a small chuck of data. Figure 8 demonstrates the throughput of Filebench. From this figure, we can confirm that *ACCFS* works well in real-world workload. In addition, Even when the journal size is 128MB, the performance of *ACCFS* outperforms other modes in most cases. These results do not match to those of Figure 6. Therefore, we analyzed read and write performance, respectively. We found two reasons behind such improvements. First, *ACCFS* can improve the read performance by sequentially writing data in the journal area. In other words, *ACCFS* can maximize the effect of read-ahead within the flash storage (*i.e.*, internal parallelism). Second, *ACCFS* reduces the number of write operations without any redundant write as shown in Figure 3. This observation is indeed interesting because we had not expected any improvement of read operations with *SHARE* interface.

6.2 Effect of ACCFS on MySQL/InnoDB

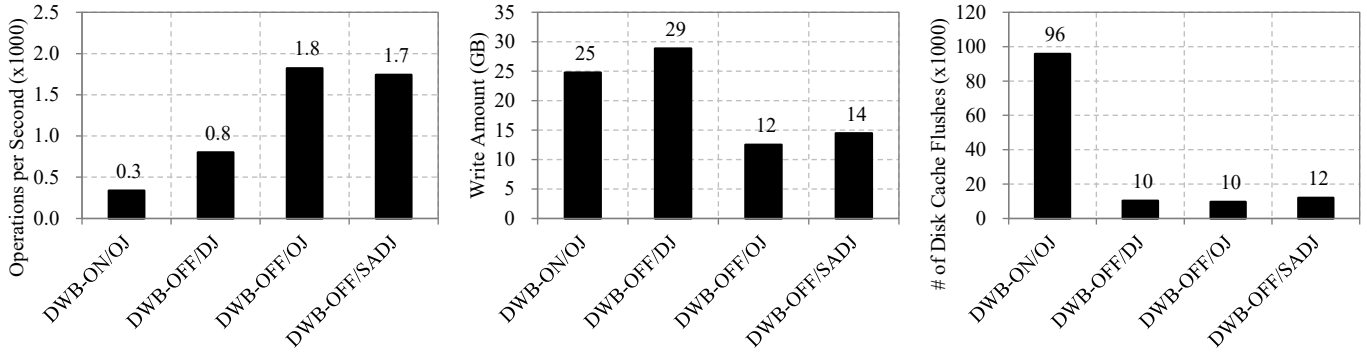
The atomicity of each page write is a uncompromisable assumption in database storage engines because a *torn* page can not be restored even with the Aries-style recovery scheme. However, since modern file systems and storage devices do not generally guarantee page write atomicity, every database engine has its own update protocol to prevent the *torn* page problem. For example, the MySQL/InnoDB storage engine takes a variant of journaling, called *double-write-buffer* (for short, DWB) [12]: when a dirty page is replaced from the buffer cache, its new copy is first appended to a separate

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2959305, IEEE Transactions on Parallel and Distributed Systems



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

Fig. 9: OLTP benchmark results of LinkBench using MySQL. The original MySQL versions were tested in three different configurations: (1) `DWB-ON/OJ`(default), (2) `DWB-OFF/DJ`, and (3) `DWB-OFF/OJ`, while MySQL on *ACCFS* were in `DWB-OFF/SADJ`.



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

Fig. 10: OLTP benchmark results of Sysbench using MySQL. The original MySQL versions were tested in three different configurations: (1) `DWB-ON/OJ`(default), (2) `DWB-OFF/DJ`, and (3) `DWB-OFF/OJ`, while MySQL on *ACCFS* were in `DWB-OFF/SADJ`.

journal area, *double-write-buffer*, and then the old copy in its original location is overwritten. In each step, an `fsync()` call is made to enforce ordering and durability.

Because SADJ mode in *ACCFS* can guarantee the *version consistency*, MySQL/InnoDB on *ACCFS* is safe from the torn page problem even when the DWB mode is turned off. And due to the system-wide *version consistency* of *ACCFS*, the amount of writes to the storage is halved, and hence the performance could be doubled. Hence, to evaluate the effect of *ACCFS* on MySQL/InnoDB database, we ran two popular OLTP benchmarks, SysBench [32] and LinkBench [33] under four different modes: (1) `DWB-ON/OJ`(default), (2) `DWB-OFF/DJ`, (3) `DWB-OFF/OJ`, and (4) `DWB-OFF/SADJ`(*ACCFS*-based version). Note that the third mode `DWB-OFF/OJ` does not prevent the torn page problem while the other three modes do.

Figure 9 presents the results obtained from running OLTP benchmark of Linkbench using MySQL; we ran 4,800,000 operations for a 50 GB database (24 files) after a two minute warm-up. In addition, Figure 10 shows the results from running the Sysbench in OLTP mode; 10 GB database (20 files) with 40 million rows for 1,000,000 operations. In both experiments, MySQL/InnoDB engine was configured to use 5 GB as a buffer pool with sixteen concurrent threads, and all under buffered I/O mode. Finally, we deliberately added the *crash-inconsistent* `DWB-OFF/OJ` mode to Figure 9 and Figure 10 so as to stress that the *ACCFS*-based version can outperform the crash-inconsistent mode in terms of performance. As Figure 9a and Figure 10a show, the *ACCFS*-based MySQL outperforms the default mode `DWB-ON/OJ` by 6.16x and the second option `DWB-OFF/DJ` by 2.73x. This performance gain is, as is clearly shown in Figure 9b and Figure 10b, in part due to the reduction of write amount by replacing the redundant write at either `DWB` or `DJ` mode with *ACCFS*'s single-write journaling. However, the wider performance gap between `DWB-ON/OJ` and `DWB-OFF/SADJ` modes can not be explained solely with the write reduction. The other main reason for the gap is the difference in the number of disk flush operations invoked in two modes. While the default `DWB-ON/OJ` mode, as explained before, calls `fsync()` in every step of database file writes and double-write-buffer write, the *ACCFS*-based version calls one disk flush after writing all database files together. Thus, as Figure 9c and Figure 10c show, the *ACCFS*-based version invokes 16.4x less disk flush operations than the original version. In summary, MySQL/InnoDB can, by offloading the responsibility for preventing the torn page problem to *ACCFS*, benefit significantly in terms of performance at no compromise of data consistency.

6.3 Effect of ACCFS on SQLite

SQLite is a light-weight library based DBMS widely used in mobile devices. Hence, unlike enterprise-class DBMS engines

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2959305, IEEE Transactions on Parallel and Distributed Systems

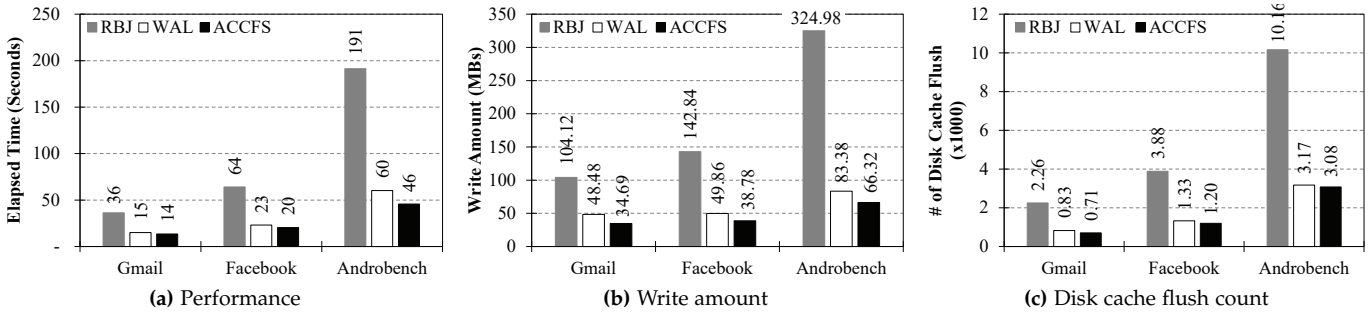


Fig. 11: SQLite Performance: RBJ vs. WAL vs. *ACCFS*. The original SQLite database (version 3.8.13) were tested under three different modes : 1) `RBJ/OJ`(default), 2) `WAL/OJ`, and 3) `WRITEBACK/SADJ`. A set of three mobile workloads was used in the experiment: Facebook, Gmail, and AndroBench.

such as MySQL, it takes a less complicated page-oriented scheme for its transactional atomicity support: the force policy for commit and the steal policy for buffer replacement [25]. For this reason, when a transaction commits, all the updated pages (from single or multiple files) by the transaction should be atomically propagated to the storage. Please note that this requirement is more stringent than the data consistency provided in either ext4 DJ mode or MySQL/InnoDB's DWB scheme. Therefore, in order to meet this application-level crash consistency, SQLite takes costlier journaling modes of rollback journaling (RBJ) [34] and write-ahead-logging (WAL) [35].

RBJ mode takes a *undo-based journaling* in that the original content of a page is copied to the rollback journal before updating the page. In contrast, the WAL mode takes a *redo-based journaling* in that 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 checkpoint operation. Unfortunately, the WAL mode can not, although faster than the RBJ mode, guarantee the transactional atomicity when updates made by a transaction are spanning over multiple database files [35], and in this respect, it is an *incomplete* solution to the crash consistency.

In order to evaluate the effect of *ACCFS* on SQLite database, we ran a set of representative mobile traces in three different SQLite modes, `RBJ/OJ`, `WAL/OJ`, and `WRITEBACK/SADJ`. As noted earlier, our SDJ mode in *ACCFS* can not meet the crash consistency requirement in SQLite and thus the SAJD mode should be used instead. To guarantee the crash consistency in `WRITEBACK/SADJ` mode, where any journaling mode of SQLite is turned off, the only change made in SQLite source code is to add `O_ATOMIC` flag to an `fopen()` call which opens SQLite database files. We used three SQLite traces, and one trace is a synthetic AndroBench [36], and the other two real traces are collected from running Facebook and Gmail applications on an Andriod 4.1.2 Jelly Bean SDK. And the experimental results are presented in Figure 11.

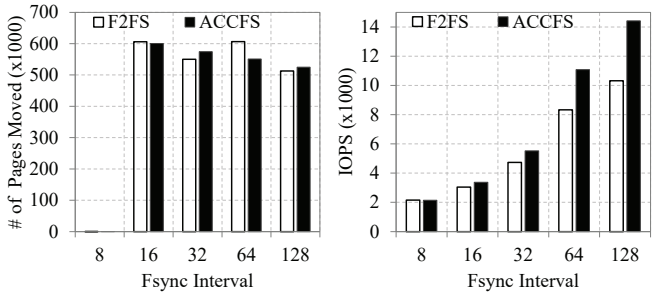
As shown in Figure 11, the *ACCFS*-based version outperforms the two SQLite journaling modes consistently over all three workloads by approximately 3.3x and 1.2x, respectively. It is well known that the RBJ mode suffers from its double-write journaling and excessive `fsync()` calls [25], [37]: the frequent `fsync()` calls are in part due to journal file creation/deletion per every transaction, and are in part necessary to guarantee the durability of database and journal files, and also to ensure the strict write ordering between those two files. As a result, it is not surprising to see from Figure 11b and Figure 11c that the original RBJ mode generates about 3.8x more writes and 3.2x more disk cache flush operations than the *ACCFS*-based version.

In the WAL mode, the updated pages are appended to a WAL file and then they are later checkpointed to the database file. For this reason, when the database size is relatively small and the workload has clear locality in write patterns, the write amplification in the WAL mode could be significantly smaller than that in the RBJ mode. And because the WAL file is reused once created, the `fsync()` calls are not made so frequently as in the RBJ mode. As a result, as shown in Figure 11b and Figure 11c, the WAL mode generates less writes and disk cache flush operations than the RBJ mode consistently over all three traces. This is why the performance gap between *ACCFS*-based version and WAL mode is rather marginal. However, it should be recalled that the original WAL mode in SQLite does not guarantees the crash consistency against multiple files, while *ACCFS*-based SQLite and the RBJ mode do guarantee.

Before closing this subsection, we would like to stress that with *ACCFS*, the existing application can be easily made crash-consistent. Please recall that the SQLite `WRITEBACK` mode becomes crash-consistent by adding one flag to the `fopen()` call in SQLite. In contrast, in version 3.8.13 of SQLite, the RBJ and WAL mode consist of about 14,500 lines of code, and CFS also requires to add 38 lines of its system calls in SQLite source code [9].

7 Discussions

ACCFS built on top of log-structured file system (LFS) [4]. In this section, we discuss the impact of *SHARE* command on LFS. A log-structured writing is widely adopted on flash storage devices, but it still suffers from inevitable segment cleaning overhead to secure large chunks of free space. Various techniques, such as data grouping [38], slack space recycling [39], and in-place-update (IPU) mode in F2FS [27], have been proposed, but none of them are free from the burden of relocating valid blocks in victim segment. The overhead, in contrast, can be almost completely eliminated by



(a) Number of moved pages **(b)** Performance

Fig. 12: Comparison between F2FS and *ACCFS*. While the number of copy-backed pages is similar, *ACCFS* shows significantly better performance by completely removing copy-back during segment cleaning. *ACCFS* performs *SHARE* operations to logically move pages without physical copying. For each graph, the x-axis shows `fsync()` interval that means the number of write operations between two consecutive `fsync()`.

incorporating the *SHARE* interface into the segment cleansing procedure in F2FS: the segment cleansing can complete by simply calling the *SHARE* interface with old and new LPNs of valid blocks as its parameters, instead of copying valid blocks from a victim segment to a new segment. We implemented *ACCFS* (*i.e.*, *SHARE*-aware segment cleaning) by modifying about 100 LoC of F2FS. For evaluation, we first filled up the file system to make its utilization to 50% of total space. Then, we performed experiments with FIO benchmark, which was configured to perform random writes to 40% of the total storage capacity, by varying `fsync()` interval from 8 to 128. Figure 12 shows the total number of moved pages during the segment cleaning and the performance results of each segment cleaning. Figure 12a shows how many valid pages are moved during the segment cleaning. Interestingly, when `fsync()` interval is 8, *ACCFS* and F2FS do nothing. This is because current F2FS was modified to allow an in-place update when the `fsync()` interval is smaller than 16. If the `fsync()` interval grows beyond 16, the segment cleaning is triggered to move valid pages. As a result, *ACCFS* outperforms F2FS by 10%–30% as shown in Figure 12b. The reason behind the speed up is that *ACCFS* completely removes the copy-back overhead of data blocks and only updates metadata blocks, such as segment information table and segment summary area.

8 Conclusion

We have presented *ACCFS*, which natively supports both *system-wide version consistency* and the *application-level crash consistency* on flash storage with an atomic address remapping interface, called *SHARE*. Our *ACCFS* can relieve the applications of the burden of guaranteeing their crash consistency as well as data consistency, and boost the application performance because of its single-write journaling and less frequent `fsync()` calls from applications. Therefore, with *ACCFS*, consistency-critical applications do not need to devise complex, tardy, error-prone update protocols by themselves. The existing applications can be run without any changes under *ACCFS*, while enjoying the higher data consistency, and they can easily be made application-level crash-consistent simply by opening their data files in `O_ATOMIC` mode. In addition, the single-write journaling in *ACCFS* will double the life span of flash storage devices. We have prototyped *ACCFS* by modifying *ext4* file system with only minimal changes, and also have implemented the *SHARE* interface inside a commercial SSD as firmware. Using the *ACCFS* prototype and the *SHARE*-enabled M.2 SSD, we have carried out a set of synthetic and realistic benchmark tests. Our experimental results show that *ACCFS*-based applications are 2–6x faster than their original versions.

Finally, in this paper, we presented the strength and potential of *ACCFS*. Unfortunately, we acknowledge that the applicability of *ACCFS* is quite limited as of now, especially taking into account that it requires a holistic modification across several layers. Though, the *ACCFS*-style approach deserves to be encouraged in that its performance improvement could be significant and its abstraction is rather easily applicable to a wide set of file systems as well as applications.

Acknowledgment

This work was partly supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (IITP-2015-0-00284, (SW Starlab) Development of UX Platform Software for Supporting Concurrent Multi-users on Large Displays) and (No.2015-0-00314,NVRam Based High Performance Open Source DBMS Development). Young Ik Eom is the corresponding author of this paper.

References

[1] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency Without Ordering," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 1–16.
[2] Ext4 Disk Layout/Journal (jbd2). [Online]. Available: https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout#Journal_.28jbd2.29
[3] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," in *Proc. USENIX'96 Annu. Tech. Conf.*, 1996, pp. 1–15.
[4] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-structured File System," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992.
[5] C. Lee, D. Sim, J.-Y. Hwang, and S. Cho, "F2FS: A New File System for Flash Storage," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 273–286.
[6] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-Tree Filesystem," *ACM Trans. Storage*, vol. 9, no. 3, pp. 9:1–9:32, Aug. 2013.
[7] S. Park, T. Kelly, and K. Shen, "Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data," in *Proc. 8th ACM Eur. Conf. Computer Syst.*, 2013, pp. 225–238.
[8] R. Verma, A. A. Mendez, S. Park, S. Mannarswamy, T. Kelly, and C. B. Morrey, "Failure-atomic Updates of Application Data in a Linux File Syst." in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 203–211.
[9] C. Min, W.-H. Kang, T. Kim, S.-W. Lee, and Y. I. Eom, "Lightweight Application-Level Crash Consistency on Transactional Flash Storage," in *Proc. USENIX'15 Annu. Tech. Conf.*, 2015, pp. 221–234.
[10] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Crash Consistency," *Communi. of the ACM*, vol. 58, no. 10, pp. 46–51, Sep. 2015.
[11] ——, "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 433–448.
[12] MySQL 5.7 Reference Manual. [Online]. Available: http://dev.mysql.com/doc/refman/5.7/en/
[13] SQLite. [Online]. Available: http://www.sqlite.org/

[14] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh, "Torturing databases for fun and profit," in *Proc. 11th USENIX Symp. Operating Syst. Des. Implementation*, 2014, pp. 449–464.
[15] G. Oh, C. Seo, R. Mayuram, Y.-S. Kee, and S.-W. Lee, "SHARE Interface in Flash Storage for Relational and NoSQL Databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2016, pp. 343–354.
[16] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic Crash Consistency," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 228–243.
[17] J. Yeon, M. Jeong, S. Lee, and E. Lee, "RFLUSH: Rethink the Flush," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 201–209.
[18] Y. Won, J. Jung, G. Choi, J. Oh, S. Son, J. Hwang, and S. Cho, "Barrier-Enabled IO Stack for Flash Storage," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 211–226.
[19] D. Park, D. H. Kang, and Y. I. Eom, "OFTL: Ordering-aware FTL for Maximizing Performance of the Journaling File System," in *Proc. 55th Annu. Des. Automat. Conf.*, 2018, pp. 1–6.
[20] "Vim the editor." [Online]. Available: http://www.vim.org/index.php
[21] H. J. Choi, S.-H. Lim, and K. H. Park, "JFTL: A Flash Translation Layer Based on a Journal Remapping for Flash Memory," *ACM Trans. Storage*, vol. 4, no. 4, pp. 14:1–14:22, Feb. 2009.
[22] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, "Transactional Flash," in *Proc. 8th USENIX Symp. Operating Syst. Des. Implementation*, 2008, pp. 147–160.
[23] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 197–212.
[24] Y. Hu, Y. Kwon, V. Chidambaram, and E. Witchel, "From Crash Consistency to Transactions," in *Proc. USENIX Hot Operating Syst.*, 2017, pp. 1–6.
[25] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, and C. Min, "X-FTL: Transactional FTL for SQLite Databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, ser. SIGMOD '13. ACM, 2013, pp. 97–108.
[26] J. Mohan, R. Kadekodi, and V. Chidambaram, "Analyzing IO Amplification in Linux File Systems," in *Proc. 8th Asia-Pacific Workshop on Syst.*, 2017, pp. 1–2.
[27] J. Kim. f2fs: introduce flash-friendly file system. [Online]. Available: https://lwn.net/Articles/518718/
[28] Tru64 AdvFS Technology. [Online]. Available: http://advfs.sourceforge.net/
[29] J. Axboe. FIO (Flexible IO Tester). [Online]. Available: http://git.kernel.dk/?p=fio.git;a=summary
[30] Filebench. [Online]. Available: http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench
[31] C. Min, S. Kashyap, S. Mass, W. Kang, and T. Kim, "Understanding Manycore Scalability of File Systems," in *Proc. USENIX'16 Annu. Tech. Conf.*, 2016, pp. 71–85.
[32] "SysBench (Branch 1.0)." [Online]. Available: https://github.com/akopytov/sysbench
[33] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: a Database Benchmark based on the Facebook Social Graph," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1185–1196.
[34] SQLite: Atomic Commit In SQLite. [Online]. Available: http://www.sqlite.org/wal.html
[35] SQLite: Write-Ahead Logging. [Online]. Available: http://www.sqlite.org/wal.html
[36] AndroBench (SQLite Benchmark). [Online]. Available: http://www.androbench.org/wiki/AndroBench
[37] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O Stack Optimization for Smartphones," in *Proc. USENIX'13 Annu. Tech. Conf.*, 2013, pp. 309–320.
[38] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "SFS: Random Write Considered Harmful in Solid State Drives," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 1–16.
[39] Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Optimizations of LFS with slack space recycling and lazy indirect block update," in *Proc. 3rd Annual Haifa Experimental Syst. Conf.*, 2010, pp. 1–9.

Dong Hyun Kang is an Assistant Professor with the Department of Computer Engineering at Dongguk University-Gyeongju in South Korea. Before joining Dongguk University, he was a software engineer of Samsung Electronics in South Korea (2018-2019). He received his Ph.D. degree in College of Information and Communication Engineering from Sungkyunkwan University in 2018. His research interests include file and storage systems, operating systems, and emerging storage technologies.

Changwoo Min is an Assistant Professor of the Electrical and Computer Engineering Department at Virginia Tech, where his research focuses on many-core scalability and concurrency of in-memory and non-volatile memory systems. His prior research includes operating systems, storage systems, database systems, and system security. Before joining Virginia Tech in 2017, he was a research scientist in Computer Science at Georgia Institute of Technology. He received his Ph.D. degree from Sungkyunkwan University in 2014. Before starting his Ph.D., he developed various software products including Linux-based mobile platform (Tizen), Java virtual machine (J9), and desktop operating system (OS/2) in Samsung Electronics and IBM Korea.

Sang-Won Lee has been a Professor with the College of Computing at Sungkyunkwan University, Suwon, South Korea, since 2002. Before that, he was a research professor at Ewha Womans University and a technical staff at Oracle, South Korea. He received a Ph.D. degree from the Computer Science Department of Seoul National University in 1999. His research interest is in flash-based database technology.

Young Ik Eom received the B.S., M.S., and Ph.D. degrees in Computer Science and Statistics from Seoul National University, South Korea, in 1983, 1985, and 1991, respectively. Since 1993, he has been a Professor with Sungkyunkwan University, South Korea. From 2000 to 2001, he was a visiting scholar with the Department of Information and Computer Science, University of California at Irvine. He also was a president of Korean Institute of Information Scientists and Engineers in 2018. His research interests include virtualization, operating systems, file and storage systems, cloud systems, and UI/UX system.