# Failure-Atomic Updates of Application Data in a Linux File System

Rajat Verma and Anton Ajay Mendez, *Hewlett-Packard;* Stan Park, *Hewlett-Packard Labs;* Sandya Mannarswamy, *Hewlett-Packard;* Terence Kelly and Charles B. Morrey III, *Hewlett-Packard Labs*

https://www.usenix.org/conference/fast15/technical-sessions/presentation/verma

This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).

February 16–19, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-201

# Failure-Atomic Updates of Application Data in a Linux File System

*Rajat Verma*[1] *Anton Ajay Mendez*[1] *Stan Park*[2]
*Sandya Mannarswamy*[1] *Terence Kelly*[2] *Charles B. Morrey III*[2]

[1]*Hewlett-Packard Storage Division* [2]*Hewlett-Packard Laboratories*

## Abstract

We present the design, implementation, and evaluation of a file system mechanism that protects the integrity of application data from failures such as process crashes, kernel panics, and power outages. A simple interface offers applications a guarantee that the application data in a file always reflects the most recent successful `fsync` or `msync` operation on the file. Our file system furthermore offers a new `syncv` mechanism that failure-atomically commits changes to *multiple* files. Failure-injection tests verify that our file system protects the integrity of application data from crashes and performance measurements confirm that our implementation is efficient. Our file system runs on conventional hardware and unmodified Linux kernels and will be released commercially. We believe that our mechanism is implementable in any file system that supports per-file writable snapshots.

## 1 Introduction

Many applications modify data on durable media, and failures during updates—application process crashes, OS kernel panics, and power outages—jeopardize the integrity of the application data. We therefore require solutions to the fundamental problem of *consistent modification of application durable data* (CMADD), i.e., the problem of evolving durable application data without fear that failure will preclude recovery to a consistent state.

Existing mechanisms provide imperfect support for solving the CMADD problem. Relational databases offer ACID transactions; similarly, many key-value stores allow failure-atomic bundling of updates [2, 13, 14]. Despite the obvious attractions of transactions, both kinds of databases can lead to two difficulties: First, in-memory data structures do not always translate conveniently or efficiently to and from database formats; repeated attempts to smooth over the "impedance mismatch" between data formats have met with limited success [19]. Second, the complexity of modern databases offers fertile ground for implementation bugs that negate the promise of ACID: A recent study has shown that widely used key-value and relational databases exhibit erroneous behavior under power failures; the proprietary commercial databases tested *lose data* [36].

File systems strive to protect their *internal metadata* from corruption, but most offer no corresponding protection for *application* data, providing neither transactions on application data nor any other unified solution to the CMADD problem. Instead, file systems offer primitives for controlling the order in which application data attains durability; applications shoulder the burden of restoring consistency to their data following failures. Added to the inconvenience and expense of implementing correct recovery is the inefficiency of the sequences of primitive operations required for complex updates: Consider, for example, the chore of failure-atomically updating a set of files scattered throughout a POSIX-like file system. Remarkably, the vast majority of file systems do not provide the straightforward operation that CMADD demands: the ability to modify application data in (sets of) files failure-atomically and efficiently.

We present the design, implementation, and evaluation of failure-atomic application data updates in HP's Advanced File System (AdvFS), a modern industrial-strength Linux file system derived from DEC's Tru64 file system [1]. AdvFS provides a simple interface that generalizes failure-atomic variants of `writev` [8] and `msync` [20]: If a file is opened with a new `O_ATOMIC` flag, the state of its application data will always reflect the most recent successful `msync`, `fsync`, or `fdatasync`. AdvFS furthermore includes a new `syncv` operation that combines updates to multiple files into a failure-atomic bundle, comparable to the multi-file transaction support in Windows Vista TxF [17] and TxOS [22] but much simpler than the former and more capable than the latter. The size of transactional updates in AdvFS is limited only by the free space in the file system. AdvFS requires no special hardware and runs on unmodified Linux kernels.

The remainder of this paper is organized as follows: Section 2 situates our contributions in the context of prior work. Section 3 describes AdvFS and the features that made it possible to implement failure-atomic updates of application data. Section 4 presents experimental evaluations of both the correctness and performance of AdvFS, and Section 5 concludes with a discussion.

## 2 Related Work

Most widely deployed mainstream file systems offer only limited and indirect support for consistent modification of application durable data (CMADD).[1] Semantically weak OS interfaces are partly to blame. For example, POSIX permits `write` to succeed *partially*, making it difficult to define atomic semantics for this call [30]. Synchronization calls such as `fsync` and `msync` constrain the order in which application data reaches durable media, and recent research has proposed decoupling ordering from durability [5]. However applications remain responsible for building CMADD solutions (e.g., atomicity mechanisms) atop ordering primitives and for reconstructing a consistent state of application data following a crash. Experience has shown that custom recovery code is difficult to write and prone to bugs. Sometimes applications circumvent the need for recovery by using the one failure-atomicity mechanism provided in conventional file systems: file rename [11]. For example, desktop applications can open a temporary file, write the entire modified contents of a file to it, then use `rename` (or a specialized equivalent [7, 23]) to implement an atomic file update—a reasonable expedient for small files but untenable for large ones.

FusionIO provides an elegant and efficient mechanism for solving the CMADD problem for data on their flash-based storage devices: a failure-atomic `writev` supported by the flash translation layer [8]. The MySQL database exploits this new mechanism to eliminate application-level double writes and thereby improve performance substantially [29]. Still more impressive gains are available to applications architected from scratch around the new mechanism. For example, a key-value store designed to exploit the new feature achieves both performance and flash endurance benefits [15]. The limitations of failure-atomic `writev` are that it requires special hardware, applies only to single-file updates, and does not address modifications to memory-mapped files.

Fully general support for failure-atomic bundles of file modifications is surprisingly rare. Windows Vista TxF supports such a capability, but the feature is deprecated because its formidably complex interface has impaired adoption [17]. TxOS includes a simpler interface to the same capability, but with a limitation: Because TxOS implements atomic file updates via the file system journal, transaction size is limited by the size of the journal [22]. Valor implements in the Linux kernel a transactional file update API with seven new system calls that support inter-process isolation even in the presence of non-transactional accesses by legacy applications [25]. The price that transaction-aware applications pay for this sophisticated support includes a substantial burden of logging: Applications must perform a `Log Append` syscall prior to modifying a page of a file within a transaction, which is awkward at best for the important case of random STOREs to a memory-mapped file.

An attractive approach to the CMADD problem on emerging durable media is a persistent heap supporting atomic updates via a transactional memory (TM) interface. Mnemosyne [31] and Hathi [24] implement such mechanisms for byte-addressable non-volatile memory (NVM) and flash storage, respectively. Persistent heaps obviate the need for separate in-memory and durable data formats: Applications simply manipulate in-memory data structures using LOAD and STORE instructions, which seems especially natural for byte-addressable NVM. One limitation of these systems is that they do not support conventional file operations; another is that they are tailored to specific durable media. Finally, they employ software TM, which carries substantial overheads.

Persistent heaps can be implemented for conventional block storage and need not employ TM. Recent examples include Software Persistent Memory (SoftPM) [9] and Ken [34], whose persistent heaps expose `malloc`-like interfaces and support atomic checkpointing. Such approaches provide ergonomic benefits and are compatible with conventional hardware, but their atomic-update mechanisms entail substantial complexity and overheads. For example, SoftPM automatically copies volatile data into persistent containers as necessary through a novel hybrid of static and dynamic pointer analysis, making development easier and less error-prone. However SoftPM tracks data modification in coarse-grained chunks of 512 KB or larger, which can lead to write amplification at the storage layer. Ken's user-space persistent heap tracks modifications at 4 KB memory-page granularity, which may reduce write amplification, but Ken writes each modified page to storage twice (to a REDO log synchronously and in-place asynchronously).

Failure-atomic `msync` ensures that application data in the backing file always reflects the most recent successful `msync` call [20]. It is easy to layer increasingly sophisticated higher-level abstractions atop this foundation, e.g., persistent heaps which in turn can slide beneath general-purpose libraries of data structures and algorithms such as C++ STL. Although it supports the style of programming natural to non-volatile memory,

[1] Spillane et al. provide an extensive review of research literature on transactional file systems [25].

failure-atomic msync can be implemented on conventional block storage. A kernel-based implementation of failure-atomic msync suffers at least three shortcomings: The need to run a modified kernel impedes adoption, the use of the file system journal limits transaction sizes, and data modifications are written to storage twice (once in the journal and once in-place) [20]. A user-space implementation of a similar mechanism eliminates the first two problems and has been deployed in commercial production systems [3], but it does not support fully general file manipulations and it retains the double write due to logging.

AdvFS solves the CMADD problem directly and combines many of the advantages of prior approaches. The interface is both simple and general: Opening a file with a new O_ATOMIC flag guarantees that the file's application data will reflect the most recent synchronization operation, regardless of whether the file was modified with the write or mmap families of interfaces (or both). Our new syncv operation ensures that updates to a *set* of files are atomic. Because it includes failure-atomic msync as a special case, AdvFS offers the same advantages as a foundation atop which persistent heaps and other abstractions may be layered. AdvFS does not rely on the file system journal to implement atomic updates; it avoids double writes and the size of atomic updates is limited only by the amount of free space in the file system. Adopting AdvFS is relatively easy because it runs on standard Linux kernels and requires no special hardware. Its atomic-update interface admits implementation atop both conventional block storage as well as emerging byte-addressable NVM, and thus it provides a smooth transition path from the former to the latter. Finally, AdvFS for Linux is not a research prototype. It is an extensively modernized production-quality upgrade and Linux port of DEC's Tru64 file system, and it is scheduled for commercial release in March 2015 as part of HP Storage appliances.

As described in Section 3, implementing O_ATOMIC is straightforward in file systems that support per-file writable snapshots [4, 12, 28, 32]. We believe that most could implement O_ATOMIC and syncv without prohibitive cost or complexity, which in turn would make it much easier to write robust applications.

## 3 Implementation

AdvFS is a modern, update-in-place, journaling Linux file system developed internally for commercial storage appliances, designed to be scalable and performant for multiple use cases and workloads. AdvFS for Linux evolved from DEC's Tru64 file system, which was open sourced in 2008 [1, 10] and has been rewritten extensively for modern storage devices, with enterprise scalability and reliability. It supports a number of advanced capabilities such as the ability to add/remove storage devices online, support multiple file systems on the same storage pool, and take *clones* (described below) and snapshots at file, directory, and FS level.

Like other modern file systems that support storage pools [35], AdvFS decouples the logical file hierarchy from the physical storage. The logical file hierarchy layer implements the naming scheme and POSIX-compliant functions such as creating, opening, reading, and writing files. The physical storage layer implements write-ahead logging, caching, file storage allocation, file migration, and physical disk I/O functions. AdvFS is comparable in performance and feature richness to most modern Linux file systems; due to space constraints we omit a detailed description of AdvFS and comparisons with other modern FSes. We designed and implemented O_ATOMIC on AdvFS and exposed it to applications through the conceptually simple and familiar interface of open followed by write/fsync or mmap/msync.

O_ATOMIC leverages a *file clone* feature developed to support use cases such as virtual machine cloning. A file clone is a writable snapshot of the file. AdvFS implements file cloning utilizing a variant of copy-on-write (COW) [21], illustrated in Figure 1. When a file is cloned, a copy of the file's inode is made. The inode includes the file's block map, a data structure that maps logical file offsets to block numbers on the underlying block device. Since the original file and its clone have identical copies of the block map, they initially share the same storage. When a shared block is eventually written to, either in the original file or in its clone, a copy of the block is made and remapped to a different location on the block device. Since COW results in new data blocks being assigned to the original file, it has the downside that it can fragment the original file; AdvFS supports online defragmentation, which can mitigate this difficulty. Efficient clone implementation in AdvFS enabled a simple but effective implementation of O_ATOMIC.

When a file is opened with O_ATOMIC, a clone of the file is made (Figure 1(a)-(b)). This clone is not visible in the user visible namespace but exists in a hidden namespace accessible by AdvFS. When the file is modified the changed blocks are remapped via COW (Figure 1(c)). The clone still points to the blocks of the file at the time the file was opened. On a subsequent call to fsync/msync the existing clone is deleted and a new one is created to track the latest version of the file (Fig-

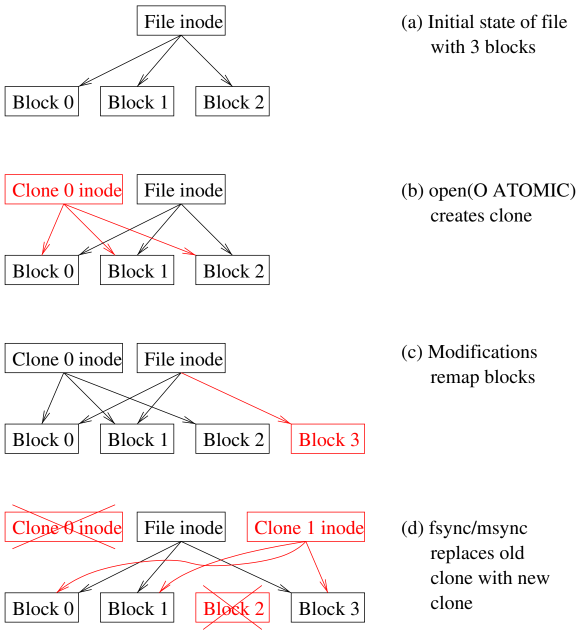

Figure 1: File clones implement atomic updates.

ure 1(d)). On the close of a file opened with `O_ATOMIC`, the original file is replaced with the clone.

If the system crashes, recovery of an `O_ATOMIC` file is delayed until the file is accessed again. The file system's path name lookup function checks if the file has a clone in the hidden name space; this is a very inexpensive check that is performed on *every* file open. If a clone exists, it is renamed to the user visible file and a handle to it is returned (we require *writable* clones rather than read-only snapshots because of this scenario). AdvFS's "lazy," per-file recovery offers several attractions: Consider, for example, a kernel panic that occurs while many processes are atomically updating many files. Upon reboot, the file system will recover quickly because the in-progress updates, interrupted by the crash, trigger no recovery actions when the file system is mounted. The net effect is that applications that do not need recovery from interrupted atomic updates (e.g., applications that are merely *reading* files) do not share the recovery-time penalty incurred by the crash; only those applications that *benefit* from application-consistent recovery pay the penalty. Lazy recovery could in principle lead to the accumulation of hidden clones; it would be straightforward to delete clones in the background.

While our support for atomic file durability for `O_ATOMIC` is built atop the clone feature of AdvFS, alternative implementations are also possible, such as using delayed journal writeback [20]. Using journal writeback to achieve atomic durability has the disadvantage that the size of a single-file atomic update is limited by the size of the journal; another downside is that journaling can lead to double writes of modified data. Our approach does not have these limitations but requires that the file system provide the ability to create per-file clones. In our experience, implementing `O_ATOMIC` atop a per-file cloning capability is relatively straightforward, and we believe that similar implementations on other file systems that support cloning are possible. As of this writing several open source and commercial file systems support per-file clones [4, 12, 28, 32].

Applications that use the `O_ATOMIC` feature of AdvFS must obey a few simple rules. The only new rule is that overlapping or concurrent modifications to a single file via multiple file descriptors void the failure-atomicity guarantee and should be avoided. Due to various subtleties it is not possible to define unambiguous semantics in such cases, so different processes/applications must coordinate their access to files. The remaining rules are not specific to AdvFS or `O_ATOMIC`: As in other file systems, multi-threaded concurrent accesses to a single file must be "orderly," in the sense that data races, atomicity violations, and other concurrency bugs must be avoided. As in other file systems, programmers must ensure that data being committed via `fsync` or `msync` is not being modified by other threads during those calls.

## 3.1 Multi-File Atomic Updates: `syncv`

In many situations it is necessary to failure-atomically update *several* files. For example, the popular SQLite database management system stores separate databases in separate files, and it supports transactions that atomically update multiple databases. To failure-atomically implement the corresponding updates to the underlying files on ordinary file systems, SQLite implements a complex multi-journal mechanism [27]. Such application-level logging can interact pathologically with analogous mechanisms in the underlying file and storage systems [33]. Support for multi-file atomic updates in the file system can simplify and streamline applications that require this capability.

AdvFS supports multi-file atomic durability via its new "`syncv`" mechanism, which is implemented as an `ioctl` for compatibility with the stock Linux kernel. Our `syncv` achieves failure atomicity by leveraging the AdvFS journaling mechanism. AdvFS is a journaling file system that employs write-ahead logging to ensure the integrity of the file system. Modifications to the metadata are completely written to the journal before the actual changes are written to storage. The journal is written to storage at regular intervals. During crash recovery, AdvFS reads the journal to confirm file system transactions. All completed transactions are committed to storage and uncompleted transactions are undone. The number of uncommitted records in the journal, not the amount of data in the file system, determines the speed of recovery.

Our syncv takes as arguments an array of file descriptors opened with O_ATOMIC and the size of the array. Our O_ATOMIC implementation is the building block for implementing syncv. As noted previously O_ATOMIC deletes the existing clone and creates a new one at the time of fsync/msync. In order to make syncv atomic the delete operation on all of the files' clones must be atomic. This is achieved using AdvFS's journaling sub-system. Metadata modifications required to delete the clones are logged to the journal. The journaling sub-system ensures that all of these changes are atomically and durably committed. Apart from this the recovery for syncv is no different from single files opened with O_ATOMIC. Creation of new clones for the files need not be made atomic in syncv because the files and their new clones are mapped to the same storage. Unlike prior work on single-file atomic durability which uses journaling for capturing data changes [20], our multi-file atomic durability mechanism syncv uses the journal not for *application data* but only for the *metadata* changes needed to delete the clones. This metadata change is quite small and hence allows our approach to support a very large number of multiple file updates simultaneously. With the AdvFS default journal size of 128 MB, a single syncv call can atomically update at least 256 files under worst-case conditions. Configuring a larger journal will proportionately increase the number of files that syncv can accommodate.

The 256-file limitation stems from a combination of our current implementation of "clone delete," the worst case size of a single "clone delete," and the size of the AdvFS journal. AdvFS checkpoints the journal at every quadrant, which limits journal transaction size to 25% of the journal size. In a badly fragmented file system, the delete of a single file (or clone) could occupy 128 KB in the journal. So for a 128 MB journal, in the worst case our current implementation of syncv can atomically update $(128 \times 25\% \times 1024 \times 1024)/(128 \times 1024) = 256$ files. In principle we could support far more files per syncv by using Delayed Delete Lists (DDL). A DDL is a list maintained on non-volatile storage that is used to asynchronously delete files. If "clone delete" were to use DDL, then the journal footprint of each would be roughly 100 bytes and syncv would be able to handle hundreds of thousands of files.

It is important to understand that clones are taken of individual files only, *not* the entire file system nor any subtree thereof. Cloning an individual file involves creating a copy of the file's inode and an associated (hidden) dentry. These steps are atomic for the same reason that ordinary file creation is atomic: they are journaled. Atomic update of an individual file involves first flushing changes to the file, then unlinking its clone, then creating a new clone; these operations are journaled separately and sequentially, so partial or full recovery of these three sequential but disjoint operations always leaves the system in a consistent state. Our syncv mechanism, which operates on multiple files, obtains atomicity by leveraging the file system journal to ensure, in REDO-log fashion, that all of the per-file atomic updates in a specified bundle are (eventually) performed.

## 4 Evaluation

We verify that AdvFS O_ATOMIC does indeed protect the integrity of application data across updates in the presence of crashes (Section 4.1), and we compare the performance of our solution to the CMADD problem with existing alternatives (Section 4.2).

### 4.1 Correctness

The O_ATOMIC data integrity guarantees rely upon two realistic assumptions about underlying storage systems. First, it must be possible to commit data to durable media synchronously, which means that volatile write caches in storage hardware must include enough standby power to rescue their contents to durable media if power fails. Second, we assume that writes of 512-byte sectors are atomic. Given these preconditions, the O_ATOMIC feature of AdvFS should protect application data integrity as advertised. We verify that it does so by injecting two types of failure: crash points and power interruptions.

Crash points are manually inserted into the AdvFS source code where the developers believe crashes are most likely to cause trouble, e.g., before, during, and after atomic operations. When a particular crash point is externally activated in a running instance of AdvFS, the result is an immediate storage system shutdown followed by a kernel panic, triggered from the specified crash point in the file system source code. We complement crash-point testing with sudden whole-system power interruptions, because the former test specific developer hypotheses about recovery whereas the latter strive to uncover surprising scenarios. Our power outage tests employ a scriptable device that suddenly cuts power to a computer without warning—the same effect as physically unplugging the machine's power cord.

The goal of both crash-point testing and power interruptions is to cause "impossible" post-crash corruption. Our test suites repeatedly and intensively modify files using `write` or `mmap`/STORE then call `fsync` or `msync`, respectively, or `syncv`. The patterns of data modifications are such that a defective implementation of `O_ATOMIC` would likely introduce obvious evidence of corruption following a crash. Finally, we trigger crashes while our test suites are running. Our crash point-tests ran against an enterprise-class RAID controller and our whole-system power interruptions ran on an enterprise-class SSD, both of which are described in Section 4.2. AdvFS successfully survived over 400 power interruptions and dozens of crash-point tests, with no evidence of application data corruption. The expected kind of application data corruption *does* occur when we inject failures if `O_ATOMIC` is *not* used. For example, crashes during append leave partial data appended and crashes during `seek`/`write` sequences leave partial updates.

## 4.2 Performance

On file system benchmarks such as IOZONE [6], postmark [18] and MDTest [16], AdvFS performance is competitive with other well-known file systems such as ext3, ext4, and XFS; by most performance measures AdvFS is within $\pm 10\%$ of other file systems. Due to space limitations we omit these comparisons and focus on the performance of atomic file updates.

We evaluate the performance of the new `O_ATOMIC` feature via microbenchmarks that mimic a common use case of `fsync` and also via "mesobenchmarks" that compare a simple transactional key-value store built atop failure-atomic `msync` with well-known alternatives. Prior literature has documented the ease with which failure-atomic `writev`/`msync` can be retrofitted onto complex, mature, production software to improve resilience and performance [3, 20, 29].

We ran our tests on two systems: a workstation and an enterprise server. The workstation has two quad-core 2.4 GHz Xeon E5620 processors and 12 GB of 1333 MHz DRAM and ran Linux kernel 2.6.32. We installed AdvFS on the workstation's enterprise-grade 120 GB SATA drive on a 3 Gbps controller. The SSD is powerfail-safe because its write cache is backed by a supercapacitor. Prior to our experiments we "burned in" the SSD by writing to the device at least 180 GB of data (i.e., 1.5$\times$ its rated capacity). Our enterprise server had twelve 1.8 GHz Xeon E5-2450L cores and 92 GB of DRAM; its storage controller had a 1 GB battery-backed cache configured as 90% write cache and a 1 TB 7200 RPM SAS hard drive.

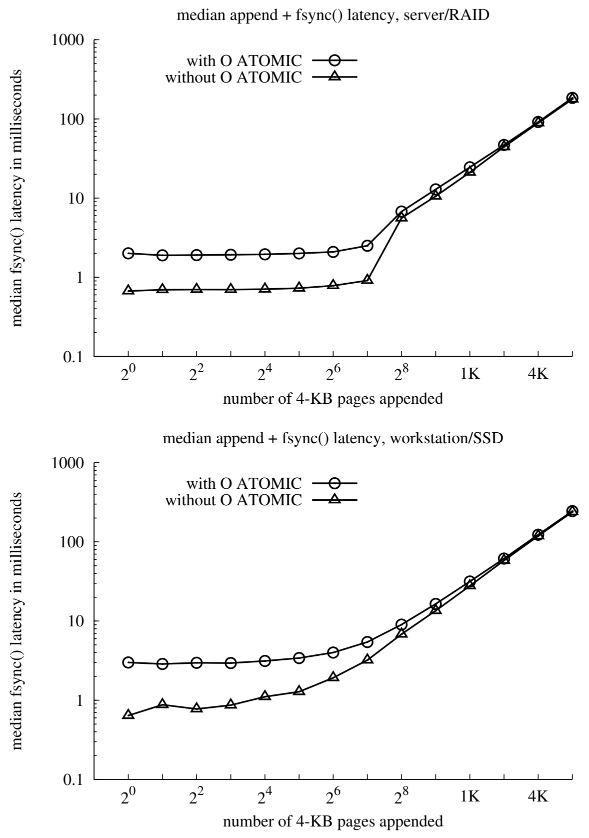

Figure 2: Microbenchmark results.

What overhead does `O_ATOMIC` entail compared to operations on files opened without this flag? Our microbenchmark addresses this question in the context of a common use case: using `write` to append data to a file followed by `fsync` to commit the amendments to durable media. Figure 2 presents median `fsync` latencies for this operation on files opened with and without `O_ATOMIC` on our two test machines. Our results show that `O_ATOMIC` carries a constant overhead on the order of 2 ms, which is clearly visible in Figure 2 for appends of up to $2^7$ pages (512 KB). This overhead occurs because the current implementation of `O_ATOMIC` in AdvFS performs an uncached storage read in connection with creating an inode for the clone when `fsync` is called. This is a simplification in our current implementation and clone creation times could be reduced by copying in-core state rather than reading from storage.

|  | Server/RAID |  |  | Workstation/SSD |  |  |
|---|---|---|---|---|---|---|
|  | insert | replace | delete | insert | replace | delete |
| STL <map>/AdvFS | 1.996 | 2.488 | 2.919 | 1.655 | 2.022 | 2.395 |
| Kyoto Cabinet 1.2.76 | 4.711 | 2.990 | 4.660 | 4.088 | 2.590 | 4.007 |
| SQLite 3.7.14.1 | 2.394 | 2.524 | 2.433 | 2.374 | 2.611 | 2.435 |
| LevelDB 1.6.0 | 0.629 | 0.626 | 0.615 | 0.641 | 0.640 | 0.633 |

Table 1: Mesobenchmarks: Mean per-operation timings (milliseconds).

For large modifications, the roughly constant overhead of `O_ATOMIC` becomes negligible (approximately 2–3.5%) and we approach the rated write bandwidth of the SSD. In other words, the price we pay for failure-atomicity is modest for large updates.

Our "mesobenchmark" repeats the experiment in Section 5.3 of our original failure-atomic `msync` paper [20], which compares four transactional key-value stores: SQLite [26], LevelDB [14], Kyoto Cabinet [13], and a fourth contender implemented as a C++ Standard Template Library (STL) `<map>` container that stores data in a persistent heap backed by a file opened with `O_ATOMIC` and updated with `msync`. Each of these key-value stores performed the following transactional operations on one thousand keys: first, insert all keys paired with random 1 KB values; next, replace the value associated with each key with a different random value; and finally, delete all of the keys, for a total of three thousand transactions. Each of the above three steps visits keys in a different random order, i.e., we randomly permute the universe of keys before each step.

Table 1 presents our results, which are comparable to those in our earlier work. LevelDB wins hands down, with the STL `<map>` atop a memory-mapped file updated with AdvFS failure-atomic `msync` placing second. This is easy to understand: The red-black tree beneath an STL `<map>` makes no attempt to minimize the number of memory pages it modifies, which strongly influences the performance of STL/AdvFS; by contrast, LevelDB implements failure-atomic updates with carefully crafted, compact log file writes. Our simple `<map>`-based key-value store shows that a persistent heap based on atomic file update can very easily slide beneath a rich, full-featured library of in-memory data structures and algorithms—which takes roughly a dozen lines of code in the present case. The net result is to transform software with no failure resilience whatsoever into software that can withstand process crashes, OS kernel panics, and power outages. Our AdvFS-fortified `<map>` furthermore achieves better performance than two far more complex platforms designed to provide failure resilience with good performance. Our experience convinces us that failure-atomic file update enables dramatically simplified application software whose performance rivals all but expertly streamlined code.

## 5 Conclusions

We have shown that a mechanism for consistent modification of application durable data (CMADD) can be implemented straightforwardly atop per-file cloning, a feature already available in AdvFS and in several other modern FSes. Our implementation of `O_ATOMIC` exposes a simple interface to applications, makes file modifications via both `write` and `mmap` failure-atomic, avoids double writes, and supports very large transactional updates of application data. Furthermore, our `syncv` implementation supports failure-atomic updates of application data in *multiple* files. Our empirical results show that the `O_ATOMIC` implementation in AdvFS preserves the integrity of application data across updates in the presence of both surgically inserted crashes and sudden power interruptions. Our performance evaluation shows that `O_ATOMIC` carries tolerable overheads, particularly for large atomic updates.

Implementing a CMADD mechanism in a file system facilitates adoption because it requires neither special hardware nor modified OS kernels. We believe that file systems should implement simple, general, robust CMADD mechanisms, that many applications would exploit such a feature if it were widely available, and that `O_ATOMIC` and `syncv` are convenient interfaces.

## Acknowledgments

We thank Joe Tucek and Haris Volos for helpful advice and feedback on early drafts of this paper. The anonymous FAST reviewers and our shepherd, Raju Rangaswami, provided valuable suggestions and guidance for the final version. We thank HP Storage manager Subbi Mudigere for the support and resources that enabled us to implement the research concept of failure-atomic file update in AdvFS.

## References

[1] Tru64 AdvFS technology. Retrieved 17 September 2014 from `http://advfs.sourceforge.net/`.

[2] Berkeley Database (BDB). `http://www.oracle.com/us/products/database/berkeley-db/overview/index.html`.

[3] A. Blattner, R. Dagan, and T. Kelly. Generic crash-resilient storage for Indigo and beyond. Technical Report HPL-2013-75, Hewlett-Packard Laboratories, Nov. 2013. `http://www.hpl.hp.com/techreports/2013/HPL-2013-75.pdf`.

[4] Btrfs file system, Sept. 2014. `https://btrfs.wiki.kernel.org/index.php/Main_Page`.

[5] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *SOSP*, 2013.

[6] D. Capps. IOzone filesystem benchmark. `http://www.iozone.org/`.

[7] Apple `exchangedata(2)` manual page. Retrieved 22 September 2014 from `https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man2/exchangedata.2.html`.

[8] FusionIO. NVM primitives library, Feb. 2014. See in particular the `nvm_batch_atomic_operations` at `http://opennvm.github.io/nvm-primitives-documents/`.

[9] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *USENIX Annual Technical Conference*, 2012. `https://www.usenix.org/system/files/conference/atc12/atc12-final70.pdf`.

[10] G. Haff. The many lives of AdvFS, June 2008. Retrieved 17 September 2014 from `http://www.cnet.com/news/the-many-lives-of-advfs/`.

[11] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *SOSP*, 2011. `http://doi.acm.org/10.1145/2043556.2043564`.

[12] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference*, 1994. `http://dl.acm.org/citation.cfm?id=1267074.1267093`.

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

[14] Google's LevelDB key-value store. `https://github.com/google/leveldb`.

[15] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan. NVMKV: A scalable and lightweight flash aware key-value store. In *HotStorage*, June 2014. `https://www.usenix.org/system/files/conference/hotstorage14/hotstorage14-paper-marmol.pdf`.

[16] Mdtest: A synthetic benchmark for file systems metadata operations. `sourceforge.net/projects/mdtest/`.

[17] Microsoft Developer Network. Alternatives to using transactional NTFS. Retrieved 17 September 2014 from `http://msdn.microsoft.com/en-us/library/hh802690.aspx`.

[18] Network Appliance. Postmark: A New Filesystem Benchmark, Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html/`.

[19] T. Neward. Object-relational mapping: The Vietnam of computer science, June 2006. `http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx`.

[20] S. Park, T. Kelly, and K. Shen. Failure-atomic `msync()`: A simple and efficient mechanism for preserving the integrity of durable data. In *EuroSys*, 2013. `http://doi.acm.org/10.1145/2465351.2465374`.

[21] Z. Peterson and R. Burns. Ext3Cow: A time-shifting file system for regulatory

compliance. *ACM Transactions on Storage*, 1(2), May 2005. http://doi.acm.org/10.1145/1063786.1063789.

[22] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *SOSP*, 2009. http://www.cs.utexas.edu/users/witchel/pubs/porter09sosp-txos.pdf.

[23] Microsoft Windows ReplaceFile function. Retrieved 22 September 2014 from http://msdn.microsoft.com/en-us/library/aa365512.aspx.

[24] M. Saxena, M. A. Shah, S. Harizopoulos, M. M. Swift, and A. Merchant. Hathi: Durable transactions for memory using flash. In *Non-Volatile Memories Workshop*, Mar. 2012. http://pages.cs.wisc.edu/~swift/papers/nvmw12_hathi.pdf.

[25] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling transactional file access via lightweight kernel extensions. In *FAST*, 2009. https://www.usenix.org/legacy/event/fast09/tech/full_papers/spillane/spillane.pdf.

[26] SQLite database library. http://www.sqlite.org/.

[27] SQLite multi-file atomic commit (Section 5). http://www.sqlite.org/atomiccommit.html.

[28] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *USENIX Annual Technical Conference*, 1996. http://dl.acm.org/citation.cfm?id=1268299.1268300.

[29] N. Talagala. Atomic writes accelerate MySQL performance, Oct. 2011. http://www.fusionio.com/blog/atomic-writes-accelerate-mysql-performance/.

[30] The Open Group. *Portable Operating System Interface (POSIX) Base Specifications, Issue 7, IEEE Standard 1003.1*. IEEE, 2008.

[31] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS*, 2011. http://doi.acm.org/10.1145/1950365.1950379.

[32] Symantec Veritas VxFS file system. Retrieved 23 September 2014 from https://sort.symantec.com/public/documents/sfha/6.0/aix/manualpages/html/man/storage_foundation_for_databases_tools/html/man1m/vxsfadm-filesnap.1m.html.

[33] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman. Dont stack your log on my log. In *USENIX Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*, 2014. https://www.usenix.org/system/files/conference/inflow14/inflow14-yang.pdf.

[34] S. Yoo, C. Killian, T. Kelly, H. K. Cho, and S. Plite. Composable reliability for asynchronous systems. In *USENIX Annual Technical Conference*, 2012. https://www.usenix.org/system/files/conference/atc12/atc12-final206-7-20-12.pdf.

[35] OpenZFS, Sept. 2014. http://www.open-zfs.org/wiki/Main_Page.

[36] 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 *OSDI*, Oct. 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_mai.