Reproducing Crash Consistency Experiments with ALICE

Milan Bhandari and Mridu Nanda

1 Introduction

File systems use a variety of well-studied techniques to provide crash consistency for file system metadata. Some of these techniques include logging, copy-on-write, and soft updates. Many modern applications are built atop these file systems, and therefore get file-system level crash consistency guarantees for free. However, an application must initiate its own application-level crash consistency protocol to ensure that user-level data structures are consistent post-crash. This sequence, known as an update protocol, invokes system calls that update the underlying files and directories in a recoverable way [4]. For example, the default setting of SQLite uses an update protocol called rollback journaling to maintain transactional atomicity.

Unfortunately, applications often implement update protocols incorrectly. Incorrect update protocols are primarily a result of a mismatch of expectations: the application developer assumes incorrect invariants about the underlying file system, causing the update protocol to behave in an unexpected manner. For example, an update protocol might assume the writes are persisted in program order; however, most modern file systems re-order writes to avoid high latency seek times. The multitude of possible application states and the non-deterministic nature of application state post-crash makes it even more difficult to write a complete and correct update protocol.

The authors of "Application Crash Consistency and Performance with CCFS" hypothesize that existing update protocols would work (mostly) correctly on an ordered and weakly atomic file system. To this end, they designed the crash consistent file system (CCFS), which improves application crash consistency. Experiments verified that applications running atop CCFS were significantly more crash consistent than ext4 [3]. In this paper, we attempt to reproduce the crash consistency experiment from the original paper.

The rest of the paper is structured as follows. In section 2 we give a recap of CCFS. Next, we outline the design for our reproduction experiment by describing the experimental tools (Sections 3.1 and 3.2) and shortcomings (Section 3.3) of the original experiment. Section 4 describes the implementation of our reproduction experiment and section 5 presents our results. We conclude with a discussion of CCFS in light of our reproduction experiment.

2 CCFS

CCFS improves application crash consistency by providing two guarantees: *ordering* and *weak atomicity*. The ordering property states that the effects of system calls should be persisted on disk in program order. This property disallows the standard re-ordered write optimization, which is ubiquitous in modern file systems. More generally, the order property ensures that a system crash cannot produce a state where the system calls appear re-ordered. Weak atomicity guarantees that system calls are atomic across a crash. For example, a directory operation like `rename` should appear as an atomic operation across a system crash. Weak atomicity also requires: (1) writing to a file in sector-sized increments be atomic, and (2) writing to a file while increasing the file's size to be atomic.

CCFS implements ordering and weak atomicity with the stream abstraction. A stream represents a sequence of operations that are committed in program order. An application typically corresponds to a single stream; however, applications with multiple streams are also possible. CCFS re-orders operations from different streams to reduce the overheads associated with ordering per-stream. Since streams are not uniquely associated with a specific file or directory, two streams may perform operations on logically related data. Therefore, the stream implementation takes special care to handle these dependencies.

3 Experiment Design

Our goal was to reproduce Table 1, as copied from the original CCFS paper. This experiment compared the crash consistency of applications running atop CCFS with that of applications running atop ext4. Five different programs, which included version control systems (e.g., Git) and data systems programs (e.g., SQLite), were tested.

Application	**ext4**	**ccfs**
Level DB	1	0
SQLite-Roll	0	0
Git	2	0
Mercurial	5	2
Zookeeper	1	0

Table 1. Vulnerabilities found using ALICE in the original CCFS experiment. Each reported vulnerability is location in the application source code that has to be fixed.

Rather than running workloads from these applications directly on hardware, the Application-Level Intelligent Crash Explorer (ALICE) tool was used to simulate the guarantees provided by each filesystem. This led the experiment to be split in roughly two parts: (1) determining the relevant persistence properties for each file system, and (2) simulating these properties using ALICE. We describe both these steps and give our critique of the overall experiment in the following sections.

3.1 BoB

The first step in the experiment was to determine the persistence properties for each filesystem to be tested. Persistence properties are broadly split into two categories: ordering guarantees and atomicity guarantees. Ordering guarantees specify which operations a filesystem will persist to disk before others. Atomicity guarantees specify which disk operations are atomic in the context of a system crash.

The authors of the CCFS paper used previous results from the Block Order Breaker (BoB) tool to determine the atomicity and ordering guarantees provided by ext4. These results are reproduced in Table 2.

Persistence Property	**File System**				
	ext4-writeback	ext4-ordered	ext4-nodelalloc	ext4-datajournal	ccfs
Atomicity					
Single sector overwrite					
Single sector append	×				
Single block overwrite	×	×	×		×
Single block append	×				×
Multi-block append/writes	×	×	×	×	×
Multi-block prefix append	×				
Directory op					
Ordering					
Overwrite → Any op	×	×	×		
[Append, rename] → Any op	×				
O_TRUNC Append → Any op	×				
Append → Append (same file)	×				
Append → Any op	×				
Dir op → Any op	×	×			

Table 2. The table shows atomicity and ordering persistence properties for ext4 and CCFS. The CCFS authors empirically determined the properties of ext4 using the BoB tool. We included CCFS's properties, as described by the paper, for completeness. $X \rightarrow Y$ specifies that X must be persisted to disk before Y. $[X, Y] \rightarrow Z$ indicates that Y follows X in program order, and that they both must be persisted to disk before Z. A $\times$ indicates that the original authors found a reproducible test case where the property fails in that file system

At a high level, BoB collects block-level traces underneath a file system and re-orders them to explore possible on-disk crash states that may arise [4]. Unfortunately, BoB is not available as an open-source project, so we did not verify these results. However, recent work [2] [1] independently verifies the results in Table 2 from BoB.

Luckily, the CCFS paper describes the relevant ordering and atomic guarantees for CCFS. Even though these properties were not tested by BoB, we include these properties in the table below for sake of completeness.

A running challenge in our reproduction effort involved decoding the exact meaning of the persistence properties listed in Table 2. For example, it was unclear if the property 'Overwrite → Any op' should be interpreted as:

1. An overwrite to a file should be persisted before any other operation *on the same file*
2. OR An overwrite to a file should be persisted before any other operation *that affects the file system*

We clarified some of the properties while corresponding with the author of the CCFS paper. We used our own interpretation for the remaining properties, taking extra care to keep these assumptions consistent. For example, we interpreted ordering constraints of the form 'X → Any Op' to mean "X should be persisted before *any file system operations following X in program order*". We describe our interpretation for each of the persistence properties below.

3.1.1 Atomicity.

- *Single sector overwrite* is a file write within one disk sector that does not modify the file's size.
- *Single sector append* is a file write within one disk sector that also modifies the file's size.
- *Single block overwrite* is a file write within one blocksize region that does not modify the file's size.
- *Single block append* is a file write within one blocksize region that also modifies the file's size.
- *Multi-block append/writes* are file writes that cross blocksize boundaries. Suppose the user makes a write call with data *AB* where each letter represents a block. If the filesystem guarantees multi-block append/write atomicity, then possible states are either empty or *AB*. In contrast, if the file system promises no append atomicity, the file could have multiple states - empty, *AB*, 0*B*, *A*0, *A*%, %*B*, etc. where % represents a block of random data and 0 represents a block of 0 data.
- *Multi-block prefix append* are file writes that cross blocksize boundaries that are persisted to disk in atomic blocksize units, in sequential order. Suppose the user makes a write call with data *AB* where each letter represents a block. If the filesystem guarantees multi-block prefix append atomicity, then possible states are either empty, *A* or *AB*.

3.1.2 Ordering.

- Overwrite → Any op: A write that does not modify a file's size should be persisted before any other file system operation that follows in program order.

- [Append, rename] → Any op: An append operation followed by a rename on the same file should be persisted before any other file operation that follows in program order.
- O_TRUNC Append → Any op: A truncate operation that extends the length of a file should be persisted before any other file system operation that follows in program order.
- Append → Append (same file): A write that modifies a file's size should be persisted before a following write that modifies the same file's size.
- Append → Any op: A write that modifies a file's size should be persisted before any other file system operation that follows in program order.
- Dir op → Any op: A directory operation (like `mkdir`) should be persisted before any other file system operation that follows in program order.

3.2 ALICE

The ALICE tool models the behavior of an application under different filesystem guarantees. The user must implement two components for each program she wishes to test: a workload script and a checker script. The user must also implement an abstract persistent model (APM) for each filesystem she wishes to test.

3.2.1 Micro Operations & Disk Operations. ALICE requires a trace of the program's system calls, and a byte dump to check the consistency of a workload with an APM. First, ALICE parses system-calls into micro-operations, which abstract away many of the details of individual system calls. For example, the `write()`, `pwrite()`, `writev()`, and `mmap()` system calls are all translated into either `overwrite` or `append` micro-operations. System calls that do not affect disk state are discarded. From here, the micro-operations are transformed into disk-operations. Disk operations are the smallest atomic modification that can be performed on a disk region. There are six disk operations: `sync`, `stdout`, `write`, `truncate`, `create dir entry`, and `delete dir entry`.

3.2.2 Abstract Persistence Model. The APM for a filesystem codifies the filesystem's constraints as defined by Table 2 into Python code. These constraints enable ALICE to generate a set of *crash states*: states in which a working directory could appear to a program after a system crash. ALICE explores these crash states to find vulnerabilities in the application's update protocol.

3.2.3 Simulation Modes. It is not feasible for the APM to simulate all possible crash states. Consider N 1-byte `write()` calls that overwrite different parts of file. If the APM does not include any ordering constraints, then the `write()` calls will be persisted in an arbitrary order. As a result, the APM would have to run 2^N simulations to capture all possible crash states.

A user can define a simulation mode to limit the number of crash states the APM must explore. The simulation mode consists of a split mode (either `count` or `aligned`) and a number of splits. By setting the APM's mode, the user controls the granularity of the mapping between micro-operations and disk operations. Consider a `write` micro-operation that is writing from offset 4,000 to 13,000 in a file. If the user simulates the filesystem in `(aligned, 4096)` mode, then ALICE will produce the following disk operations:

1. `Write` from offset 4000 to 4095 (inclusive)
2. `Write` from offset 4096 to offset 8191
3. `Write` from offset 8192 to 12,287
4. `Write` from offset 12,288 to 13,000

In contrast, if the user runs the simulation in `(count, 1)` mode, then ALICE will only produce a single `write` diskop, which writes from offset 4000 to 13,000. However, if the user runs the simulation in `(count, 3)` mode, then ALICE will produce the following disk operations:

1. `write` from offset 4000 to 7000
2. `write` from offset 7000 to 10,000
3. `write` from offset 10,000 to 13,000

More generally, in the `count` mode, a single micro-operation will be broken down into s disk operations of equal size, where s represents the split number. In the `aligned` mode, a micro-operation will be partitioned into disk operations of size s.

Naturally, simulation modes can help model an APM's atomicity constraints. For example, if the file system guarantees atomic sector writes, the user can run the corresponding APM in `(aligned, 512)` mode. By splitting a large write into multiple sector-sized writes, ALICE creates the appropriate intermediates crash states for the APM. However, the simulation mode itself is not sufficient for implementing all atomicity constraints. For example, ext4-ordered requires atomic directory operations, which cannot be modeled by only specifying a simulation mode. Rather, we created cyclic dependencies between disk operations to model a set of operations that should be perceived as an atomic unit.

Since the user determines which subset of crash states ALICE will explore, the user must carefully specify the appropriate modes to detect all relevant crash states for a given APM. Consider a filesystem does not guarantee atomicity for certain disk operations. In this case, the user must specify a simulation mode that includes *crash states* where those operations are partially persisted. The user can specify multiple simulation modes to run a single ALICE experiment. For example, a single ALICE experiment can be run in `(aligned, 512)` and `(aligned, 4096)` modes concurrently. In any case, the aggregate results from all simulation modes should contain all the relevant crash states for a given filesystem.

3.2.4 Application Scripts. As mentioned above, the ALICE user must write workload and checker scripts for each tested application. The workload script generates a system call trace and byte-dump using a customized version of strace. The checker script verifies each crash state against the user's expectations about the durability and consistency of the application. As such, the checker application takes two arguments, the path to the crash state and the path to the file that contains the stdout until the simulated crash.

3.2.5 Dynamic and Static Vulnerabilities. ALICE reports discovered vulnerabilities as either dynamic or static:

- *A dynamic vulnerability* is a system call that results in an inconsistency. ALICE finds dynamic vulnerabilities by incrementally simulating each system call, and running the outputted state through the user-defined application checker.
- *A static vulnerability* is a source code line that results in an inconsistency. Multiple dynamic vulnerabilities can correspond to a single static vulnerability. Suppose an inconsistency-causing write system call is called in a for loop ten times. Using the strace log, ALICE would detect ten distinct dynamic vulnerabilities. However, ALICE would only detect one static vulnerability, since each system call originated from the same source line within the for loop.

Table 1 reports static vulnerabilities.

3.3 Critique

In this section we give our critiques of the original experiment. We include two broad categories of critiques: experimental setup and result quality.

3.3.1 Experimental Setup. The experimental setup for ALICE included installing Python 2.7 and a few other dependencies. Overall, the tool was straightforward to set up.

We found that variable definitions in the ALICE code conflicted with the description of the tool in the original ALICE paper [4]. For example, the micro-operations in the ALICE code refer to the logical operations from the ALICE paper. Also, the disk-operations in the code refer to micro-operations from the paper. We use the definitions from the code as described in Section 3.2 rather than the ALICE paper.

The open-source version of ALICE came with one default APM implementation. Unfortunately, APM models for CCFS and ext-4 were not available. The ALICE tool itself is not as modular as we would have expected. Writing new APMs required us to copy multiple files and required careful modification to program logic.

3.3.2 Result Quality. Table 1 compares results from simulating ext4 and CCFS using ALICE. However, it fails to specify which mode of ext4 was used during these simulations. After communicating with one of the authors, we learned that the experiment was run with the ordered mode. Interestingly, Table 2 shows that the nodelalloc, and datajournal modes of ext4 provide stronger ordering and atomicity guarantees than those provided by the ordered mode of ext4. Therefore, we hypothesized that ext4-nodelalloc and ext4-datajournal might see fewer vulnerabilities than the ext4-ordered mode. To provide a fairer comparison, our experiment compared workloads on CCFS with each of the ext4 modes.

Table 1 documents the crash consistency of five different programs, which include version control systems (e.g., Git) and data systems programs (e.g., SQLite). We assume that these programs are sensitive to the underlying assumptions of the filesystem on which they run, both in terms of reordering and atomicity. The results may have been even more convincing if the authors had tested (1) several workloads for each program, and (2) a greater number of programs.

4 Experiment Implementation

Our primary implementation effort was to create APMs for CCFS and different modes of ext4. We also created workload and checker scripts for Mercurial and SQLite3, basing them closely on the provided Git and LevelDB scripts, respectively.

4.1 CCFS APM

The CCFS ordering model requires that all operations within a single stream are persisted in program order. We assumed that each application uses exactly one stream. The following Python snippet from our CCFS APM simulates the dependencies required by stream abstraction:

```
for i in range(1, len(ops)):
    ops[i].hidden_dependencies.add(i-1)
```

This for loop iterates through disk operations in program order. For each disk operation, a dependency is added to the previous disk operation. As a result disk operation i cannot be persisted before disk operation $i - 1$.

CCFS's weak atomicity property makes the following guarantees:

- *Sector-sized overwrite are atomic.* To model this guarantee, we ran CCFS in `('aligned', 512)` mode. As mentioned above, this breaks write micro-operations into multiple disk-operations that each write up to 512 bytes. Thus, the crash states are limited to those on sector size boundaries.
- *Directory operations are atomic.* Directory operations include the creation, deletion, and renaming of files and hard links. Implementing an atomic `Rename` micro-operation was particularly interesting since it decomposes into multiple disk operations. We achieved atomicity by adding a cyclic dependency between each of the disk operations.
- *Atomic appends.* If a system call appends data to the end of a file, then both increasing the file size and the writing of data to the newly appended portion of the file should be atomic together [3]. To model this

semantic, we commented code from the default APM, which simulated non-atomic sector-sized appends.

We have marked each of these changes in the aliceccfs.py and aliceccfsexplorer.py files.

4.2 ext4 APMs

As seen in Table 2, the various ext4 modes have differing atomicity and ordering constraints. Some of the guarantees overlapped with CCFS's guarantees. For example, all modes of ext4 guarantee atomic directory operations. Therefore, we reused some of the CCFS APM implementation in our various ext4 APM implementations.

Implementing ext4-ordered and ext4-nodelalloc modes provided an interesting challenge. Both of these modes guarantee atomic single block appends, but do not guarantee atomic single block overwrites. To simulate the crash states relating to block-sized operations, we ran Alice with the `(aligned, 4096)` mode. We also added conditional logic to ensure that micro-operations decomposed to block-sizes disk operations during overwrites, but not during appends.

5 Results

We tested four applications using the ext4 and CCFS APMs. We used the static vulnerability count for our measurements, as was done in the CCFS paper. Table 3 summarizes our results:

Application	**File System**				
	ext4-writeback	ext4-ordered	ext4-nodelalloc	ext4-datajournal	ccfs
Level DB	NA	3*	3*	0	0
SQLite-Roll	0	0	0	0	0
Git	1	0	0	0	0
Mercurial	0	0	0	0	0

Table 3. Vulnerabilities found using ALICE in our reproduction experiment. Each reported vulnerability is location in the application source code that has to be fixed. NA indicates the experiment could not be completed due to memory constraints. ∗ indicates that the ALICE tool may be over-reporting the vulnerabilities.

The LevelDB experiments took the longest to run. A combination of (1) a lack of constraints, and (2) numerous append micro-operations generated by the LevelDB workload, caused the number of crash states to explode in ext4-writeback mode. As a result, our machines did not have enough memory to complete the LevelDB experiment for this mode (indicated by an 'NA' in Table 3). We found three vulnerabilities while simulating ext4-ordered mode and ext4-writeback mode. However, we suspect that ALICE may be overreporting the number of vulnerabilities is this instance: all three of the static vulnerabilities involved a similar stack trace pattern. Thus, we hypothesize the same update sequence is used multiple times across LevelDB.

Like the original paper, we found zero vulnerabilities while simulating CCFS. We also found zero vulnerabilities in the ext4-datajournal simulation, which is intuitive since ext4-datajournal provides strictly more guarantees than CCFS according to Table 2.

Our Mercurial experiment yielded different results from the original CCFS paper. The original experiment discovered five vulnerabilities using ext4 and two vulnerabilities using CCFS. In contrast, our reproduction effort found Mercurial to have zero vulnerabilities under all of our APMs. This discrepancy could be attributed to us using a newer version of Mercurial that fixed any previous vulnerabilities. It is also possible that our Mercurial workload differed from the original experiment's Mercurial workload. Unfortunately, the original paper does not provide much detail or any code for any application workloads.

We found one vulnerability while running the Git workload on the ext4-writeback APM. We saw this vulnerability disappear when testing the remaining APMs, which provide stricter ordering guarantees. On further investigation, we found that this was an ordering vulnerability. Specifically, the vulnerability was triggered by a reordered append operation and rename operation. This observation is consistent with the persistence properties listed in Table 2: ext4-writeback is the only filesystem that does not provide the Append $\rightarrow$ Any op ordering guarantee.

We found zero vulnerabilities in SQLite across all of our APMs. This result was consistent with Table 1. Upon further investigation, we found that SQLite implements an application-level journal as its update protocol. As a result, SQLite achieves consistency and durability without having to rely on the underlying file system. However, application-level journaling has its costs. In particular, we observed many redundant writes in the strace logs produced by the SQLite workloads. Unfortunately, we did not have enough time to run performance experiments.

6 Discussion

Overall, the results of our reproduction experiment were consistent with those reported in the original CCFS paper. We found that CCFS had zero vulnerabilities for all of the applications we checked. Therefore, we conclude that *ordering* and *weak atomicity* are sufficient conditions to improve crash consistency.

Both ext4-datajournal mode and ext4-nodelalloc mode provide stronger guarantees about ordering and atomicity compared to ext4-ordered. Consequently, we hypothesized that ext4-nodelalloc would result in fewer vulnerabilities, downplaying the results reported in Table X. While we found that ext4-datajournal yielded the same results as CCFS, we predict that ext4-datajournal must pay a higher performance cost for crash consistency. Ext4-datajournal achieves crash consistency by enforcing a total write order. Total write order imposes a high performance cost because each data item must be written twice (rather than only writing the metadata twice).

Ext4-nodelalloc also provides stronger guarantees about ordering and atomicity compared to ext4-ordered. However, our reproduction effort showed that ext4-nodelalloc and ext4-ordered mode found the same vulnerabilities. Therefore, we can conclude the original experiment did not overstate the benefit of CCFS. Ideally, we would have liked to test more applications to differentiate between ext4-ordered mode and ext4-nodelalloc mode.

While we were able to test the consistency and durability guarantees of CCFS, we were unable to test its actual implementation and performance. The actual CCFS source code is not available. Our experiments were also limited to workloads that we could test on our laptops. Simulating the crash states generated by LevelDB seemed to be very computationally expensive, even on a relatively simple workload. Given more time, we'd liked to have conducted additional tests including workloads with multiple concurrent users and distributed data systems.

References

[1] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and Checking File System Crash-Consistency Models. *SIGARCH Comput. Archit. News* 44, 2 (March 2016), 83–98. https://doi.org/10.1145/2980024.2872406

[2] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 33–50.

[3] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application Crash Consistency and Performance with CCFS. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 181–196. https://www.usenix.org/conference/fast17/technical-sessions/presentation/pillai

[4] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 433–448.