Application Crash Consistency

By

Thanumalayan Sankaranarayana Pillai

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 12/13/2016

The dissertation is approved by the following members of the Final Oral Committee:

Andrea C. Arpaci-Dusseau, Professor, Computer Sciences
Remzi H. Arpaci-Dusseau, Professor, Computer Sciences
Ben Liblit, Associate Professor, Computer Sciences
Michael Swift, Associate Professor, Computer Sciences
Jing Li, Assistant Professor, Electrical and Computer Engineering

To my parents.

Abstract

APPLICATION CRASH CONSISTENCY
Thanumalayan Sankaranarayana Pillai

Crash consistency, i.e., making sure that data stays in a consistent state after an unexpected system crash or power loss, is important for storage reliability. Much previous research has focused on the crash consistency of file systems and of traditional relational database management systems. However, to ensure data safety, it is necessary to consider the crash consistency of all layers of the storage stack. These include the single-node application layer, the distributed systems layer if data is stored in a distributed fashion, and the internal consistency of storage devices when using certain modern storage devices such as intelligent SSDs.

In this thesis, we focus on the application layer, which (along with the file-system layer) forms a mandatory component of most modern storage stacks. Modern applications offer users a rich set of features and manage a huge amount of user data, in many cases requiring their data to be crash consistent. For example, a version control system requires version commit information to match the stored data even if a crash happens while committing a new version, and a photo-viewing application requires that thumbnails match photos.

In the first part of this thesis, we describe ALICE, a tool that examines real-world applications to determine whether they maintain crash consistency correctly, and if they do not, reports the locations of errors in their

source code. ALICE uses state-space exploration to traverse targeted states that will occur if a crash happens during the given application's execution. ALICE is practical and easy to use: the state space is determined automatically from a trace of the application's execution, and each explored state is checked for invariants directly by running the application's recovery code.

In the second part of this thesis, we use ALICE to study 11 real-world applications (including databases, key-value stores, version control systems, distributed systems, and virtualization software) and investigate whether they maintain crash consistency. We find a total of 60 vulnerabilities (i.e., error locations in source code that affect the application's crash consistency), many of which lead to severe consequences such as data corruption and data loss. The study reveals that applications use complex update protocols to persist state, and that the correctness of these protocols is highly dependent on subtle behaviors of the underlying file system. We examine which vulnerabilities are exposed in different existing file systems and how the observed vulnerabilities relate to file-system behavior. The results can be used to determine if a file system is compatible with an application and to design future file systems.

In the final part of this thesis, we present c^{2}fs, a file system that improves the correctness of application-level crash consistency protocols while maintaining high performance. A key idea in c^{2}fs is the abstraction of a *stream*. Within a stream, updates are committed in program order, thus helping correctness; across streams, there are no ordering restrictions, thus enabling scheduling flexibility and high performance. We empirically demonstrate that applications running atop c^{2}fs achieve high levels of crash consistency. Further, we show that c^{2}fs performance under standard file-system benchmarks is excellent, in the worst case on par with the highest performing modes of Linux ext4, and in some cases notably better. Overall, we demonstrate that both application correctness and high performance can be realized in a modern file system.

Acknowledgements

First and foremost, I would like to thank my advisors, Andrea and Remzi Arpaci-Dusseau. They were excellent guides for my research work and for the pursuit of my PhD degree, and excellent role models I learnt much from. Their style of managing multiple students and handling challenging tasks will keep influencing me.

The other members of my thesis committee – Mike Swift, Ben Liblit, and Jing Li – helped shape this dissertation, and my discussions with them gave rise to interesting ideas. I was fortunate to have taken classes under both Mike Swift and Ben Liblit, and my interest in robustness, reliability, and testing, came from these classes.

My PhD would not have been possible without the help and company of other students in the computer science department, especially from the ADSL group. During my initial years at grad school, they helped me understand the difficulty and importance of getting a PhD. During the later years, they kept me sane with arbitrary coffee breaks, random exciting discussions in the hallways, useful feedback on research, and shared feelings of existential crisis.

I survived grad school because of constant reminders that there were more important things than rejected papers – from my family and friends in India, and my roommates at Madison. The nature of grad school often tempted (and sometimes succeeded) me to slip into my own cocoon and remain detached from friends and family; I cannot thank them enough for always bringing me back. I dedicate this thesis to my parents.

Contents

List of Tables

List of Figures

1 Introduction

Application complexity is increasing over the years: compared with simple, modular UNIX applications [45], modern applications are huge monoliths, offering users a rich set of features and managing a huge amount of user data [32]. In many cases, storing extra data allows the application to provide new features (e.g., Firefox stores browsing history to provide autocomplete). Many applications require that certain invariants on the data hold across system crashes. For example, Firefox requires that entries in the browsing history correctly correlate URLs with possible search terms, while a photo-viewing application requires that thumbnails match photos [110]. An application is deemed *consistent* when its invariants hold.

Crash consistency is a fundamental problem in systems research [8, 31, 59, 73], particularly in file systems, database management systems, and key-value stores. Crash consistency is hard to get right: consider the ten-year gap between the release of commercial database products (e.g., System R [8] and DB2 [59]) and the development of a working crash consistency algorithm (ARIES [58]). Even after ARIES was invented, another five years passed before the algorithm was proven correct [42, 52].

Crash consistency for file systems has been heavily studied and improved. The file-systems community has developed a standard set of techniques to provide metadata consistency in the face of crashes: logging [4, 12, 31, 72, 93, 103], copy-on-write [34, 53, 73, 92], soft updates [27], and other similar approaches [14, 25]. While bugs exist in the file systems

that implement these methods [49], the core techniques are highly tested and well understood.

However, few researchers have focused on application-level crash consistency, despite its importance. For example, consider a typical modern photo management application (e.g., iPhoto) that not only stores the photos a user takes, but also information relevant to a photo library, including labels, events, and other photo metadata. No user wants a system that loses photos or other relevant information simply because a crash occurred while the photo application was trying to update its internal database.

Because of the absence of research, unlike file-system crash consistency, crash consistency for applications is still problematic. Most data-management applications (databases like SQLite [91], key-value stores like LevelDB [29]) implement complex *update protocols* to remain crash-consistent while still achieving high performance. Similar to early file system and database schemes, it is difficult to ensure that update protocols maintain consistency after a crash [83]. Maintaining application consistency would be relatively simple (though not trivial) if all state were mutated synchronously; however, such an approach is prohibitively slow. Moreover, to achieve robustness in practice, since applications are varied and many by nature (compared to file systems), significant effort is needed to test each of them. The update protocols must hence handle a wide range of corner cases that are executed rarely, are relatively untested, and are (perhaps unsurprisingly) error-prone. Thus, safety measures for correctness are desired and necessary.

Moreover, since the majority of modern applications are built atop file systems, application-level crash consistency depends on the behavior and semantics (after a crash) of the underlying file system. For instance, many applications depend (for crash consistency) on how file systems behave when a file is replaced using the `rename()` system call [101]. Specifically, crash invariants are held only if the entire file (including both the contents

of the file and the file name) is replaced atomically with respect to a system crash. This dependency is sometimes a bug, caused by a misunderstanding of the behavior guaranteed by file systems after a crash. It is also sometimes intentional, to provide reasonable application performance over different file systems: the complex application techniques required for correctly maintaining crash invariants in file systems with weak behavior (such as non-atomic file replacement) might degrade performance in those already providing strong behavior (such as atomic replacement). For the safe execution of applications, many modern file systems explicitly ensure atomic file replacement (during certain sequences of system calls), even though this is not a part of the POSIX standard.

Beyond atomic file replacement, which is perhaps a widely known problem, application-level consistency is also affected by other undocumented (and unexplored) crash-related behavior that differs between file systems [97]. This severely constrains the portability of applications. There is no consensus on what file-system behavior affects application-level consistency, and what behaviors file systems should hence guarantee. In addition, the lack of well-defined file-system behavior prevents careful applications from optimizing their update protocols, thus reducing efficiency. Previous techniques for file-system internal consistency [5, 13, 15, 27, 34, 73, 103, 104] have not investigated their (unintended) consequences on application-level consistency.

This thesis studies and improves crash consistency at the application layer. The thesis focuses on single-node applications and libraries that are built directly atop file systems, but touches upon distributed applications as examples. Focusing on single-node applications is necessary as a first step, since distributed applications are often built as a higher layer upon single-node data-management components (e.g., BigTable upon LevelDB [9]; this thesis studies LevelDB). Single-node applications are also sufficiently complex, as the subsequent chapters will show.

In the following sections, we outline and elaborate the main parts of the thesis. We then provide a summary of its contributions and an outline for the rest of this thesis.

1.1 ALICE

To study application-level crash consistency, we need to understand how consistency is implemented in practice in modern applications, and whether it is implemented correctly. This is no easy task: since building a high-performance application-level crash-consistency protocol is not straightforward, many applications implement complex update protocols that are spread across multiple source files. The protocols are also inherently tied to other aspects of the application, such as concurrency isolation.

To analyze applications, we develop ALICE, a novel framework that enables us to systematically study application-level crash consistency. ALICE takes advantage of the fact that, no matter how complex the application source code, the update protocol boils down to a sequence of file-system related system calls. Given the system call trace of an application workload, ALICE can find *crash vulnerabilities* (i.e., incorrectness in update protocols) that are exposed for a specified set of file-system crash behavior. ALICE achieves this by constructing a state space from possible permutations of the trace, and then exploring the space in a targeted, heuristical manner. The heuristics allow ALICE to discover most vulnerabilities in the application within a practical amount of time.

Note that the correctness of update protocols (i.e., vulnerabilities that are exposed) depends on file-system behavior. With ALICE, the user supplies an *abstract persistence model* (APM) of the file system to be considered. The abstract persistence model is a novel representation that expresses the crash-behavior of a file system. While being simple to specify, an APM allows calculating (and generating) all possible states of the file system

that become visible after a system crash, given a certain application workload. Thus, APMs can also be used to specify and understand file-system behavior in general, in addition to their usecase in ALICE.

APMs are more advantageous than providing a file-system implementation because they can also be used for future file systems that are still being designed. They also ensure that vulnerabilities are not omitted simply because the provided file-system version has trivial differences in crash behavior (which nevertheless influence application consistency) from other versions. Thus, APMs allow testing whether applications work portably across different versions of a file system. ALICE can also be used to determine all file-system behaviors that the given application depends on for correctness: to find vulnerabilities that can be exposed on any file system, one simply uses an abstract persistence model that is generic and unconstrained.

ALICE presents the discovered vulnerabilities to the user using the unique notion of *static crash vulnerabilities* (defined in the next chapter) that allows incorrectness in update protocols to be expressed and understood in an intuitive manner. Static crash vulnerabilities directly correlate incorrectness to the source code lines responsible for the incorrectness. Furthermore, ALICE produces as its output an annotated representation of update protocols (called protocol diagrams) that abstract away low-level details to clearly present the underlying logic, thus allowing the user to easily understand the application's consistency mechanism.

1.2 Vulnerabilities Study

To study how consistency is implemented in practice in modern applications, it is necessary to consider a variety of popular applications that differ on the format of data stored, access patterns optimized for, and crash-consistency requirements. To achieve this goal, we first analyze two

applications (SQLite and LevelDB) without using ALICE, examining past bug reports and mailing list interactions with developers, and manually analyzing their source code and system-call trace. This allows us to understand the context within which to interpret ALICE's results when applied to applications. We then apply ALICE to eleven important applications, to study and analyze their update protocols.

The applications studied include relational databases (SQLite [91], PostgreSQL [99], HSQLDB [35]), key-value stores (GDBM [28], LMDB [94], LevelDB [29]), version control systems (Git [47], Mercurial [54]), distributed systems (HDFS [81], ZooKeeper [2]), and virtualization software (VMWare Player [107]). We find a total of 60 vulnerabilities across applications: several discovered vulnerabilities have severe consequences such as data corruption or application unavailability. The study includes a detailed understanding of update protocols constructed directly from ALICE output, finding which vulnerabilities are exposed on various existing file systems, determining whether a new file-system design will harm current applications, and, a high-level discussion of why vulnerabilities occur.

More interestingly, we found interesting patterns that provide insights for file-system developers. For example, we found that many applications use a variant of write-ahead-logging; while applications are very careful about updating data in-place, they are not as careful about logged data, resulting in a number of vulnerabilities. Since writes to logged data are file appends, it happens that simple file system design decisions can prevent these vulnerabilities without affecting performance.

To summarize, the study indicates that a high number of vulnerabilities result because the current file-system interface is not intuitive for developers. More specifically, vulnerabilities occur because of two reasons. First, file systems persist user-issued updates to disk in an order different from the order in which they are issued. For example, writing data to a file and then renaming it can result in the rename operation sent to the disk before

the data, and can send the written data to disk in an order different from the issued write. Second, file systems might persist user-issued updates partially in different ways. For example, renaming a file can either happen atomically with respect to a system crash, result in both the source and the destination links present after the crash, or neither present.

1.3 C^{2}FS

Thus, a file system that persists user-issued updates to the disk in their issued order helps applications achieve crash consistency. However, many file system developers have determined that such ordering is performance prohibitive; as a result, most modern file systems reduce internal ordering constraints. For example, many file systems (including ext4, xfs, btrfs, and the 4.4BSD fast file system) re-order application writes, and some file systems commit directory operations out of order (e.g., btrfs [65]). Similarly, lower levels of the storage stack re-order aggressively, to reduce seeks and obtain grouping benefits [36, 39, 74, 76].

However, we hypothesize that a carefully designed and implemented file system can achieve *both* ordering and high performance. We explore this hypothesis in the context of the Crash-Consistent File System (c^{2}fs), a new file system that enables crash-consistent applications by providing intuitive behavior (guaranteeing both the required ordering and the atomicity) while delivering excellent performance.

The key new abstraction provided by c^{2}fs, which enables the goals of high performance and correctness to be simultaneously met, is the *stream*. Each application's file-system updates are logically grouped into a stream; updates within a stream, including file data writes, are guaranteed to commit to disk in order. Streams thus enable an application to ensure that commits are ordered (making recovery simple); separating updates between streams prevents *false write dependencies* and enables the file system

to re-order sufficiently for performance.

Underneath this abstraction, c^{2}fs contains numerous mechanisms for high performance. Critically, while ordering updates would seem to overly restrict file-system implementations, this thesis shows that the journaling machinery found in many modern systems can be adopted to yield high performance while maintaining order. More specifically, c^{2}fs uses a novel hybrid-granularity journaling approach that separately preserves the order of each stream; hybrid-granularity further enables other needed optimizations, including delta journaling and pointer-less metadata structures. C^{2}fs takes enough care to retain optimizations in modern file systems (like ext4) that appear at first to be incompatible with strict ordering, with new techniques such as order-preserving delayed allocation.

We show that most applications and standard benchmarks perform excellently on c^{2}fs with only a single stream per application. Thus, c^{2}fs makes it simple and straightforward to achieve crash consistency efficiently in practice without much developer overhead.

1.4 Summary of Contributions

The following are the contributions of this thesis.

ALICE. The ALICE tool allows a user to study the update protocol of any application and find crash vulnerabilities in the application.

Abstract Persistence Models. The thesis provides a way to model a file system that describes all possible states that a partition (of the file system) might be left in, in the event of a sudden crash.

Static Crash Vulnerabilities. The number of failures observed during crash-recovery cannot be directly used to understand the causes of the failures or quantitatively evaluate the correctness of application code. This thesis provides a quantitative and intuitive way to describe the incor-

rectness seen in an application's update protocol.

Vulnerabilities Found. This thesis studies 11 widely-used applications and finds 60 vulnerabilities among them, and in the process, demonstrates ALICE's utility with real applications.

Vulnerability Patterns Found. This thesis identifies file-system behaviors that cause most of the found vulnerabilities, along with the common impacts of the vulnerabilities when they are exposed.

Stream API. The Stream API is an extension to the file-system interface that drastically reduces the number of vulnerabilities exposed, with little developer effort, while allowing file systems to maintain high performance.

C^{2}fs. The c^{2}fs file system is an efficient implementation of the Stream API that preserves the desirable qualities of the widely used ext4 file system, while increasing the correctness of application update protocols.

1.5 Outline

The rest of this thesis is organized as follows.

Background. The behavior of a file system after recovering from a crash (and the semantics it provides in such an event) differs between file systems; the behavior is often related to the crash-consistency technique used internally within the file system. Chapter 2 explains the relationship between application-level crash consistency and the crash-recovery behavior of file systems, highlighting the behavioral differences between popular file systems. It also briefly explains the crash-consistency technique used by the ext4 file system, since we later build upon this technique to improve application-level crash consistency.

ALICE. Chapter 3 describes the ALICE tool, abstract persistence models, and static crash vulnerabilities. The chapter also includes preliminary

evaluation about the effort required to use ALICE, the running time of ALICE, and the effectiveness of the heuristics used by ALICE to target a subset of the state space of crashes.

Vulnerabilities Study. Chapter 4 describes the manual examination of LevelDB and SQLite, and the manual analysis of 11 applications. This chapter also serves to demonstrate the practical usability of ALICE.

C^{2}FS. Chapter 5 explains the Stream API, c^{2}fs, and their evaluation.

Related Work. Chapter 6 describes other work focusing on application-level crash consistency and how they relate to this thesis.

Conclusion and Future Work. Chapter 7 summarizes the thesis, describes possible future work, and the lessons learnt.

2
Background and Motivation

In this chapter, we explain how application-level crash consistency is achieved atop the file-system interface (§2.1), how the correctness of application-level consistency varies with file-system behavior (§2.2), and the internal crash-consistency mechanism of the ext4 file system (§2.3). We also define a few terms (§2.4) and conclude with a summary (§2.5).

2.1 An Example

To ensure crash consistency, the application developer must craft an update protocol that orchestrates modifications of the persistent state of the file system. Since applications communicate with file systems through the POSIX system-call interface [98], an update protocol is essentially a carefully-constructed sequence of *system calls* (such as file writes, renames, and other file-system calls) that updates underlying files and directories in a recoverable way.

Let us look at an example demonstrating the complexity of implementing crash consistency: a simple database management system (DBMS) that stores its data in a single file. Whenever the DBMS updates the file to perform a transaction, it wants the effects of the transaction to be atomic across a system crash. To maintain transactional atomicity across a system crash, the DBMS can use an update protocol called *undo logging*: before

```
creat(log);
# Making a backup in the log file
write(log, "<offset>,<size>,<data>");
write(dbfile, offset, data); # Actual update
unlink(log); # Deleting the log file
```

Figure 2.1: **Incorrect undo logging pseudocode.** *This pseudocode directly works only on few file-system configurations, such as the* data=journal *mode of* ext3/4. `offset` *and* `size` *correspond to the portion of the dbfile that should be modified. Whenever the DBMS is started, the DBMS rolls back the transaction if the log file exists and is fully written (determined using the* `size` *field).*

updating the file, the DBMS simply records those portions of the file that are about to be updated in a separate *log* file. Figure 2.1 shows the pseudocode, using POSIX system calls. After completing the transaction, the DBMS will delete the log file. Whenever the DBMS is started, it checks if the log file exists – if so, a system crash happened during a previous transaction, and the DBMS appropriately rolls-back the transaction using the log file.

In an ideal world, one would expect the pseudocode in Figure 2.1 to work on all file systems implementing the POSIX interface. Unfortunately, the pseudocode does not work on *any* widely-used file-system configuration; in fact, even within a single file system, a different set of measures are required to make it work on each configuration of the file system.

2.2 File-system Behavior

The correctness of the application's crash-consistency protocol inherently depends on the semantics of system calls with respect to a system crash, i.e., the crash behavior of the file system. However, while the POSIX standard

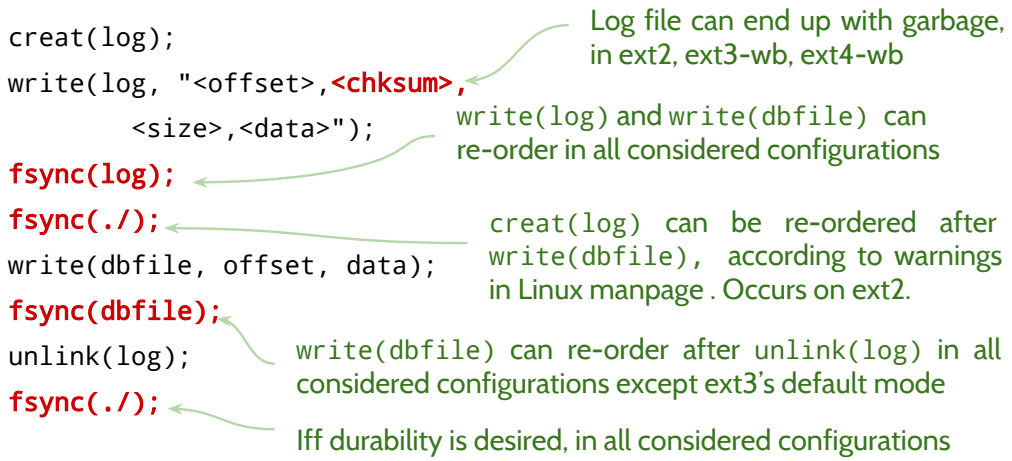


Figure 2.2: **Correct undo logging pseudocode.** *This pseudocode works in Linux file-system configurations. "./" refers to the current directory. The red parts are each additional measures needed for correctness. Comments explain which measures are required by different file systems: we considered the default configurations of ext2, ext3, ext4, xfs, and btrfs, and the* data=writeback *configuration of ext3/4 (denoted as ext3-wb and ext4-wb).*

specifies the effect of a system call in memory, it does not fully define how disk state is mutated in the event of a crash. File systems take advantage of this lack of standardization to improve performance, without much regard about the viewpoint of an application developer. Each file system (and each configuration within a single file system) persists application data differently, resulting in different kinds of intermediate disk states, and leaving programmers guessing.

Because file systems buffer writes in memory and send them to disk later, from the perspective of an application, most file systems can *re-order* the effects of system calls before persisting them on disk. In the previous example, with some file systems (*ext2*, *ext4*, *xfs*, and *btrfs* in their default configurations, but not *ext3*), the deletion of the log file can be re-ordered before the write to the database file. On a system crash in these file systems, the log file might be found already deleted from the

disk, while the database has been updated partially. Other file systems can persist a system call partially in seemingly nonsensical ways: in *ext2* and non-default configurations (the *data=writeback* mode) of *ext3* and *ext4*, while writing (appending) to the log file, a crash might leave garbage data in the newly-appended portions of the file. In such file systems, during recovery, one cannot differentiate whether the log file contains garbage or undo information.

Figure 2.2 shows the measures needed for undo logging to work on different Linux file-system configurations. Almost all measures simply resort to using the `fsync()` system call, which flushes a given file (or directory) from the buffer cache to the disk, and is used to prevent the file system from re-ordering updates. The `fsync()` calls can be arbitrarily costly, depending on how the file system implements them; an efficient application will hence try to avoid `fsync()` calls when possible. With only a subset of the `fsync()` calls, however, an implementation will be consistent only on some file-system configurations.

Note that it is not practical to use a verified implementation of a single update protocol across all applications; the update protocols found in real applications vary widely and can be more complex than Figure 2.2. The choice can depend on performance characteristics; some applications might aim for sequential disk I/O, and prefer an update protocol that does not involve seeking to different portions of a file. The choice can also depend on usability characteristics. For example, the presence of a separate log file (as described above) unduly complicates common workflows, shifting the burden of recovery to include user involvement. The choice of update protocol is also inherently tied to the application's concurrency mechanism and the format used for its data structures.

What *can* applications rely on? File-system developers seem to agree on two rules that govern what information is preserved across system crashes. The first is subtle: information already on disk (file data, directory

entries, file attributes etc.) is preserved across a system crash, unless one explicitly issues an operation affecting it. In other words, information should be preserved when the user does not issue any modifications.

The second rule deals with `fsync()` and similar constructs (`msync()`, `O_SYNC`, etc.) in UNIX-like operating systems. An `fsync()` on a file guarantees that the file's data and attributes are on the storage device when the call returns, but with some subtleties. A major subtlety with `fsync()` is the definition of "storage device": after information is sent to the disk by `fsync()`, it can reside in an on-disk cache, and hence can be lost during a system crash (except in some special disks). Operating systems provide ad-hoc solutions to flush the disk cache *to the best of their ability*; since you might be running atop a fake hard drive [16], nothing is promised. Another subtlety relates broadly to directories – directory entries of a file and the file itself are separate entities, and can each be separately sent to the disk; an `fsync()` on one does not imply persistence of the others.

2.3 Journaling in Ext4

The difference in crash behavior between file systems is not deliberate, and is often a side effect of the internal crash-consistency mechanism of each file system. For instance, many file systems use a technique known as *journaling* (also known *write-ahead logging*) to simplify and ensure internal crash consistency. Journaling is similar to the undo logging protocol discussed previously (though employed at the file-system layer). Specifically, before making updates, the file system records the updates in a journal; if a crash happens, the file system retrieves from the journal those updates that were in progress and replays them.

Since updates are added sequentially to the journal, this simplistic form of journaling (often known as *full data journaling*) makes the file system exhibit a fully-ordered crash behavior. However, an optimized variation of

journaling only records updates to certain file-system metadata structures (since only these concern file-system consistency), thus changing the crash behavior exhibited to applications. File systems can also adopt other crash-consistency techniques, such as soft updates and copy-on-write, along with numerous optimizations and implementation variations that might change with each version of the file system.

We now examine the internal crash-consistency technique used by the ext4 file system, an optimized form of journaling, highlighting a few optimizations and design details. The case study provides a background for the c^{2}fs file system that forms the final part of this thesis and is built upon ext4; it also illustrates the complexity of ext4's technique and how crash behavior can be subtly dependent on implementation details.

We first describe a simplified version of ext4's technique; the description also contains an important characteristic of ext4 journaling: journaling occurs at *block granularity*. We then explain optimizations over the simplified version explained so far, focusing on three more important characteristics: ext4 journaling batches multiple sets of atomic metadata updates (*delayed logging* [23]), uses a *circular journal*, and *delays forced checkpointing* until necessary. The block-granularity and circular aspects are a challenge for a file system that implements the Stream API, while delayed logging and checkpointing are important optimizations that any derivative file system should retain.

Basics

In simple Linux file systems, modifications made by the user are first made to an in-memory copy (in the Linux buffer cache) of the corresponding on-disk block. They are then propagated to their locations on disk. To maintain internal file-system metadata consistency, ext4 requires the atomicity (in the event of a crash) of sets of metadata modifications (e.g., all metadata modifications involved in creating a file). It also requires an

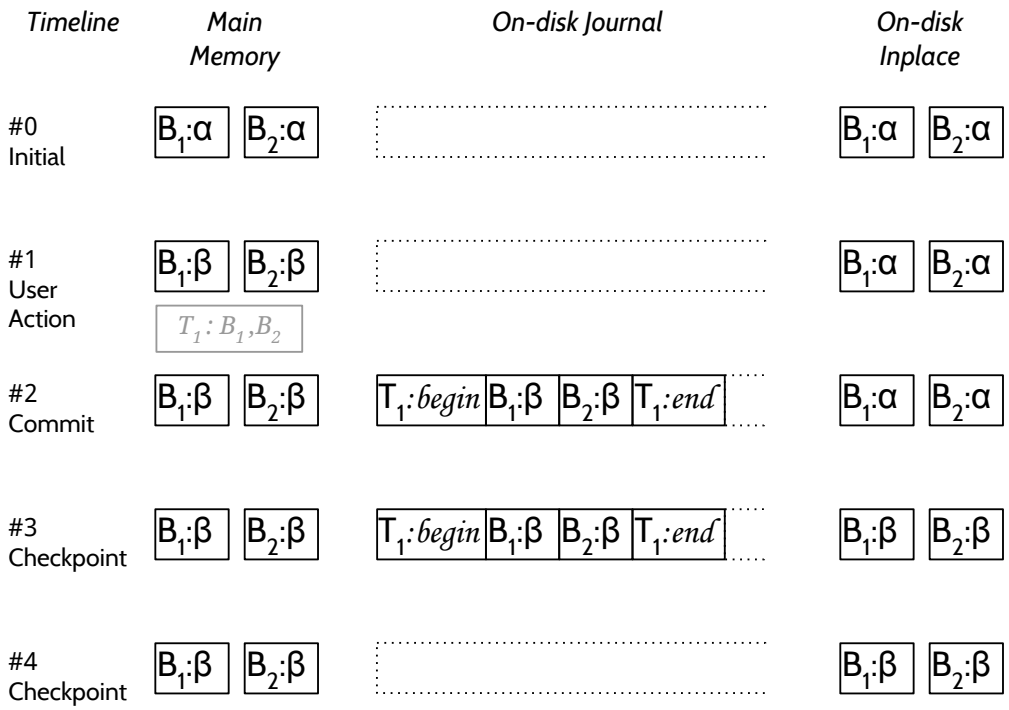


Figure 2.3: **Simplified Ext4 Journaling.** *The figure shows the sequence of events that happen when the user causes ext4 to modify metadata in the file-system blocks* B_1 *and* B_2 *from* α *to* β*. The structures residing in main memory, the state of the on-disk journal, and the on-disk (inplace) locations of blocks are shown.* $\boxed{\mathbf{B_1 : \alpha}}$ *illustrates the block* B_1 *containing the value* α*. The structure* $\boxed{T_1 : B_1, B_2}$ *shows the running transaction in main memory, containing a reference to the in-memory blocks* B_1 *and* B_2*.* $\boxed{\mathbf{T_1 : begin}}$ *and* $\boxed{\mathbf{T_1 : end}}$ *in the on-disk journal are bookkeeping information denoting the beginning and ending of the committed transaction; this is a simplification, and more complex bookkeeping is added in ext4. The figure also shows only metadata blocks: data blocks are not written on the journal, and are not shown.*

order to be maintained between these sets of modifications. To achieve this, ext4 creates a journal: a sequential region on disk, allocated when the file-system is created.

Figure 2.3 illustrates a simplified version of ext4 journaling. The in-memory copies of blocks, the state of the on-disk journal, and the on-disk locations of blocks are shown. We refer to the actual on-disk locations of

blocks as *inplace* locations, to distinguish them from any copies that are temporarily stored in the journal.

Figure 2.3 illustrates the sequence of events that happen when the user performs a metadata operation (such as creating a file), and causes ext4 to modify metadata structures in the file-system blocks B_1 and B_2. The figure shows the initial state of the in-memory copies of the file-system blocks in #0, containing the data α, and their final version after the modifications in #1 (containing the data β). Since both block modifications relate to a single logical metadata change (i.e., file creation), ext4 requires them to be atomic across a system crash. Hence, at this point, the modifications to the in-memory copies are not directly propagated to their on-disk copies.

Instead, ext4 associates B_1 and B_2 with an in-memory data structure called the *running transaction*, T_i (shown in Figure 2.3 #1). The running transaction is then *committed* as shown in Figure 2.3 #2, i.e., the updated contents of all the associated blocks of T_i and some bookkeeping information are written to an on-disk journal. Now, the transaction is *checkpointed*: first, the updated contents of the in-memory copies are propagated to their inplace locations (Figure 2.3 #3), and then the transaction is deleted from the on-disk journal (Figure 2.3 #4). If a crash happens in between, and the bookkeeping information of any transaction in the on-disk journal indicates that the complete transaction has been recorded, then steps #3 and #4 are repeated after the crash.

Delayed logging

In our previous description, we explained that after metadata is modified in main memory, the running transaction is committed to the on-disk journal. However, ext4 does not immediately commit the running transaction, since that would be inefficient if a single metadata block is modified multiple times. Instead, ext4 waits for the user to perform more operations; the resulting set of block modifications are also associated with T_i. Ext4

commits the running transaction only periodically, or when the user desires durability (by issuing a system call such as `fsync()`). When T_i starts committing, a new running transaction (T_{i+1}) is created to deal with future metadata operations. Thus, ext4 always has one running transaction, and at most one committing transaction (although each transaction in ext4 can include multiple user-level operations).

Delayed checkpointing

Once T_i finishes committing, the updated blocks are not immediately propagated to their inplace locations on disk. Instead, they are written in the background by Linux's page-flushing daemon in an optimized manner (for example, taking care of disk scheduling). Furthermore, ext4 employs techniques that coalesce such writebacks. For example, consider that a block recorded in T_i is modified again in T_j; instead of writing back the version of the block recorded in T_i and T_j separately, ext4 simply ensures that T_j is committed before T_i's space is reused. Since the more recent version (in T_j) of the block will be recovered on a crash without violating atomicity, the earlier version of the block will not matter. Similar optimizations also apply when committed blocks are later unreferenced, such as when a directory gets truncated.

Thus, there can be multiple transactions on the journal that have not yet been written back to their inplace locations on disk, i.e., that have been committed but not yet checkpointed. If a crash happens, after rebooting, ext4 scans each transaction written in the journal sequentially. If a transaction is fully written, the blocks recorded in that transaction are propagated to their actual locations on disk; if not, ext4 stops scanning the journal. Thus, the atomicity of all block updates within each transaction are maintained. Maintaining atomicity implicitly also maintains order within a transaction, while the sequential scan of the journal maintains order across transactions.

Circular journal

The previous subsection explains that delaying checkpointing as much as possible might coalesce and reduce the amount of total I/O; however, it does not deal with the size of the journal being finite. The on-disk journal file is circular: when space runs out, committed transactions in the tail of the journal are freed and that space is reused for recording future transactions. Ext4 ensures that before a transaction's space is reused, the blocks contained in it are first propagated to their inplace locations (if the page-flushing mechanism had not yet propagated them).

For circular journaling to work correctly, ext4 requires a few invariants. One invariant is of specific interest in c^{2}fs: the number of blocks that can be associated with a transaction is limited by a threshold. To enforce the limit, before modifying each atomic set of metadata structures, ext4 first verifies that the current running transaction (say, T_i) has sufficient capacity left; if not, ext4 starts committing T_i and uses T_{i+1} for the modifications.

2.4 Definitions

The following are the detailed definitions of a few terms used in this thesis.
System crash. A sudden power loss of the entire computer system, or a software bug (such as a kernel panic) that immediately prevents further communication with the disk subsystem and requires the system to be restarted for further operation. Note that this thesis mostly assumes a single-node machine such as a workstation or a laptop.
Application crash guarantees. Any invariants the user expects from an application, after a system crash happens, and when the user reboots the machine and restarts the application. For some applications, guarantees are well specified; for example, some database management systems promise ACID transactions. Guarantees for some applications are not defined, but are desired (and sometimes assumed) by convention; a photo-

editing application is expected to not corrupt a photo on a power loss. The thesis specifies the guarantees assumed when they are not well defined.

Crash consistency protocol. The steps that the application takes to ensure its crash guarantees. We use the term *crash-consistency mechanism* to describe the general logic of the protocol instead of the exact steps taken. We also use the term *crash-consistency implementation* when the protocol might vary slightly with different versions of the application, and we are referring specifically to a particular version's protocol.

Update protocol. One part of the crash-consistency protocol. It refers specifically to the steps that the application takes while modifying its data; for example, the application might make a backup of the original data before modifying it.

Recovery protocol. Other part of the crash-consistency protocol. Refers to the steps taken when the application is restarted after a system crash; for example, the application might restore data from a backup taken during the update protocol.

File-system crash behavior. When an application issues system calls that modify files and directories, and a system crash happens, after rebooting the machine (and allowing the file system to perform its recovery), there can be a variety of states in which the files and directories might be found. For example, if the application issues a `write()` system call to modify two bytes in a file and a system crash happens, the file might have both bytes modified, only one modified, or both unmodified. The variety of states that might occur depends on the file system; only certain file systems might allow the state where only one byte has been modified. The term *crash behavior* broadly defines the states that the file system allows to occur when the application modifies files and directories in a specific way.

Crash vulnerability. A violation of application crash guarantees possible under certain file-system behavior and certain timings of system crash. In the context of the terms *errors, faults, and failures* [33], the term *vulnerability*

encompasses errors and faults; we specifically use the term *static vulnerabilities* and *dynamic vulnerabilities* (defined later) similar to errors and faults, respectively. However, the term *faults* is usually used for misbehaviors that result in failures *unless* they are handled; vulnerabilities result in failures *only if* a system crash happens under certain conditions. Thus, our usage of *vulnerabilities* is similar to security literature, if a system crash is considered equivalent to a malignant adversary.

Vulnerability exposure. The event of the vulnerability actually occurring during a system crash (i.e., the system-crash timing is satisfied and the file system behaves in the required way). A *failure*.

Window of vulnerability. The time interval during which, if a system crash happens, a particular vulnerability will be exposed. For the window of vulnerability to exist, the underlying file system must behave such that the vulnerability can be exposed.

Ordering. The term *ordering* refers to a specific constraint on file-system crash behavior. Consider an application that issues two modifications (to files and directories) and a system crash happens; for example, the application creates file A and then file B. The file system might only allow states where, if the second modification is visible to the user, the first is also visible; in the example, if the application finds file B to exist after the crash, it can certainly also find file A. We say such a file system has *ordered* the modifications, *maintained the ordering* of the modifications, or persisted the modifications *in-order*. If the application finds B but not A, we say the file system has *re-ordered* the modifications, *violated the ordering*, or persisted the modifications *out-of-order*.

Full ordering and global ordering. When the file system maintains ordering between all types of modifications, we say the file system is fully ordered or maintains full ordering (however, we omit the word *fully* when the meaning is apparent from the context). If a file system does not maintain ordering between all modifications, we term it a re-ordering file system

or out-of-order file system. The term *global ordering* refers to ordering being maintained across all processes and threads. For example, a file system that maintains the ordering between all modifications within the same thread but not across threads, maintains full ordering only within each thread, and does not maintain global ordering.

2.5 Summary

To summarize this chapter, efficient implementations of application-level crash consistency are inherently complex. The varying and non-intuitive crash behavior of different file systems make it further difficult to implement application consistency correctly. The exact crash behavior of file systems are an unintentional consequence of their internal crash-consistency implementations, which are often intricate and complex.

3
ALICE

In this chapter, we describe a tool, the application-level intelligent crash explorer (ALICE), for studying the update protocols of existing I/O applications. ALICE examines whether the update protocols are correct, or if they assume any behavior about the underlying file systems. ALICE achieves its goal by constructing different on-disk file states that may result due to a crash and then verifying application correctness on each created state.

Unlike other approaches [109, 111] that simply test an application atop a given storage stack, ALICE finds the generic set of file-system behaviors required for application correctness without being restricted to only a specified file system. ALICE targets specific states that are prone to reveal crash vulnerabilities in different source lines and associates discovered vulnerabilities with the exact source lines involved. ALICE achieves this by constructing file states *directly* from the system-call trace of an application workload. The states to be explored and verified can be described purely in terms of system calls: the actual storage stack is not involved. ALICE can also be used to abstractly test the safety of new file systems.

This chapter first provides an overview of ALICE, and then how it calculates states possible during a system crash using an Abstract Persistence Model (APM). Next, the chapter describes how these states are selectively explored so as to discover application requirements and how discovered vulnerabilities are reported associated with source code lines. The chapter also describes our prototype implementation of ALICE and its limitations.

3.1 Overview

The basic idea employed by ALICE for checking a particular application is simple. The user first supplies ALICE with an initial snapshot of the files used by the application (typically an entire directory), and a workload script that exercises the application (such as performing a transaction). The user also supplies ALICE with a checker script corresponding to the workload that verifies whether invariants of the workload are maintained (such as atomicity of the transaction). ALICE runs the checker atop different *crash states*, i.e., the state of files after rebooting from a system crash that can occur during the workload. ALICE then produces a logical representation of the update protocol (that was executed during the workload), vulnerabilities in the protocol, and the source code lines associated with each vulnerability. ALICE also lists the file-system crash behaviors that need to be guaranteed for the update protocol to be correct.

Listing 3.1 shows example workload and checker scripts for a key-value store written in Python. The workload inserts a few key-value pairs, while the checker tries opening the database and retrieving any inserted keys. Note that the checker does not deal with the internal implementation of the application, but is rather simply another application workload that is run on simulated crash states. If the checker finds that the database cannot be opened, or the retrieved key-value pairs do not match expectations, it exits with an error and additionally prints an error message to *stderr* identifying the violated invariant (Python's *assert* statement does both). Notice how the checker also checks for the durability expected from the key-value store; ALICE supplies the checker with any externally visible information that has been output when the crash happened, and the checker in Listing 3.1 simply checks if all keys can be retrieved if the message *'Done'* had been printed at the time of the crash.

The exact crash states possible for a workload varies with the file system. For example, depending on the file system, appending to a file can result

```
# Workload

# Opening database
db = simplekv.open('mydb')

# Inserting key-value pairs
db['x'] = 'foo1'
db['y'] = 'foo2'

# Flushing them to disk
db.sync()
print 'Done'
```

```
# Checker

db = simplekv.open('mydb')
assert(db is not None)

c = len(db.keys())
if alice.printed('Done'):
    # Should be durable
    assert c == 2
else:
    # Only inserted pairs
    # can exist
    assert c in [0,1,2]

# The inserts were atomic
if c == 1:
    assert db['x'] == 'foo1'
elif c == 2:
    assert db['x'] == 'foo1'
    assert db['y'] == 'foo2'
```

Listing 3.1: **Workload and Checker.** *Simplified Python workload and checker scripts for a key-value store. The workload inserts two key-value pairs into an empty database (assumed to already exist) and flushes them to disk. The checker checks whether the database can be opened, and whether the retrieved key-value pairs are sensible, atomic, ordered, and durable after the* db.sync() *call.*

in the file containing either a prefix of the data persisted, with random data intermixed with file data, or various combinations thereof. ALICE uses file-system *Abstract Persistence Models (APMs)* to define the exact crash states possible in a given file system. By default, ALICE uses an APM with few restrictions on the possible crash states, so as to find the generic file-system crash guarantees required for application correctness. However, ALICE can be restricted to find vulnerabilities occurring only on a specific file system, by supplying the APM of that file system. We discuss how

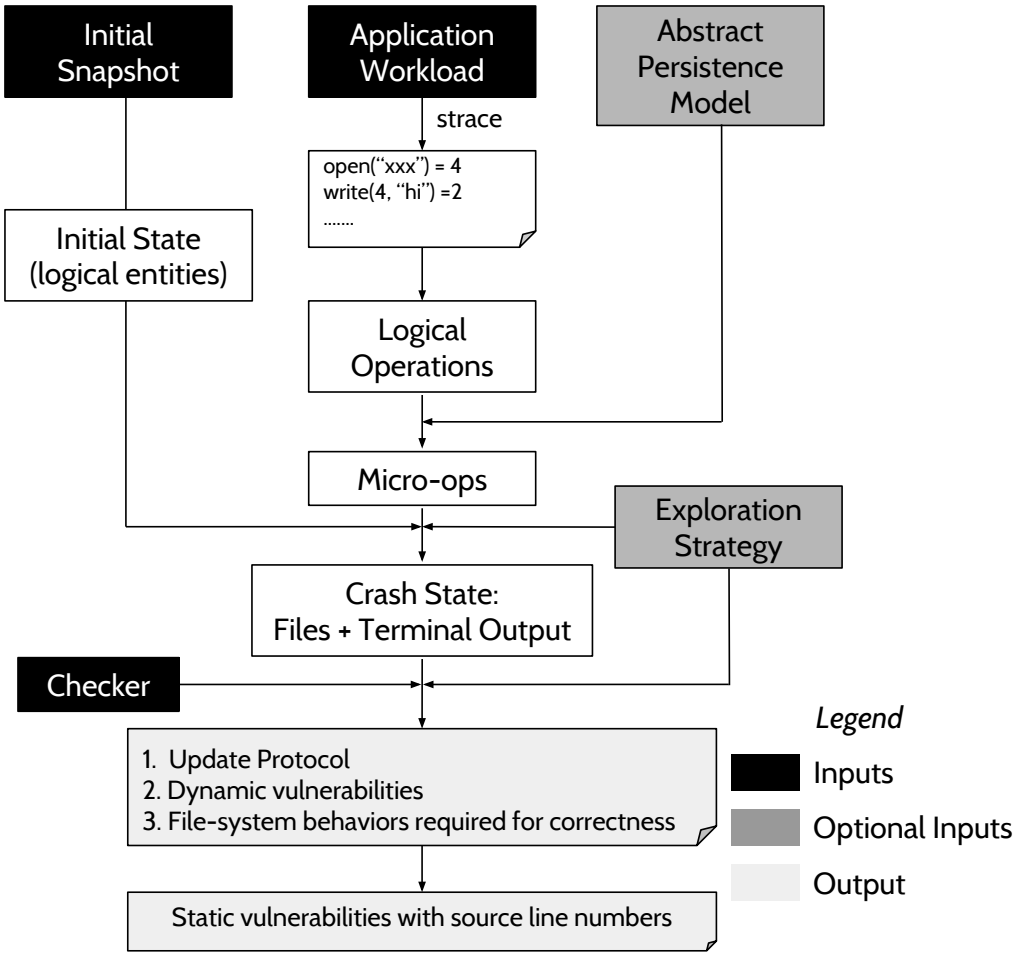


Figure 3.1: **ALICE: Steps To Find Vulnerabilities.** *The figure shows how* ALICE *converts user inputs into crash states and finally into crash vulnerabilities. Black boxes are user inputs and grey boxes are optional inputs.*

APMs are specified and used by ALICE, in the next section.

3.2 Constructing Possible Crash States

Figure 3.1 shows the various steps followed by ALICE to find crash vulnerabilities. ALICE first records the initial state of the files and directories (on

which the user-supplied workload is about to be run) as a set of *logical entities*. It then runs the application workload and obtains a system-call trace, which is converted into a sequence of *logical operations*; the trace represents an execution of the application's update protocol. Finally, ALICE uses the sequence of logical operations along with the *Abstract Persistence Model* of a file system to calculate the different crash states that are possible from the initial state. We now explain these steps in detail.

Logical File-System State and Logical Operations

ALICE represents the initial state of the file system (as well as other states, described subsequently) as a set of logical entities. There are two types of logical entities: *file inode* entities store the size and data contents of a regular file, while *directory inode* entities are an associative map of file names to either file or directory inodes. Note that neither of these entities correspond to the internal data structures of a particular file system. Rather, the entities represent the state of files and directories as observed by the application, in a format independent of the file-system implementation.

After running the application workload, ALICE converts the trace of system calls and memory accesses obtained to a sequence of *logical operations*. Logical operations abstract away details such as current read and write offsets, and file descriptors, and transform a large set of system calls and other I/O producing behavior into a small set of file-system operations. For example, `write()`, `pwrite()`, `writev()`, `pwritev()`, and `mmap()`-writes are all translated into `overwrite` or `append` logical operations.

The list of all logical operations is detailed in Table 3.1, and the system calls they result from are shown in Table 3.2. Logical operations capture three aspects of the obtained trace: modifications performed by the application to the in-memory version of the initial state, the times at which data is explicitly flushed by the application to the disk (`sync()` logical operation), and any outputs of the application that relate to durability expectations

Logical operation	Meaning
overwrite(inode, offset, count, data)	Write to regular file, with no change to its size.
append(inode, old_size, new_size, data)	Write just beyond the current size of a regular file and increase the size of the file.
truncate(inode, old_size, new_size)	Change the size of a file without writing data to the file; the size can be either increased or decreased.
link(dir_inode, link_name, link_inode, link_count)	Create a link to a new or existing regular file or a directory. The *link_count* parameter records the total number of links that point to the *link_inode* across the entire file system after this operation.
unlink(dir_inode, link_name, link_count)	Remove a file or directory link.
rename(source_dinode, source_name, dest_dinode, dest_name, inode)	The `rename()` system call.
sync(inode)	Synchronize the in-memory and on-disk state of an inode.
output(data)	Application prints to the screen; can be configured for other relevant outputs, such as network messages.

Table 3.1: **Logical Operations.** *List of logical operations employed by* ALICE.

after a crash. Note that all these aspects are file-system agnostic.

Producing logical operations requires more than examining only system calls and memory accesses that directly induce I/O. For example, the `clone()` and `close()` system calls influence file descriptors, and hence must be examined as well; other such ancillary system calls are shown in the last row of Table 3.2. ALICE produces logical operations by simulating a replay of the system-call trace and associating logical entities (i.e., file or directory inodes) with the resulting operations.

Note that, while many different logical operations can be performed on

Logical operation	**Results from**
overwrite	pwrite, pwritev, write, writev, mwrite*
append	pwrite, pwritev, write, writev
truncate	fallocate, ftruncate, truncate, pwrite, pwritev, write, writev, openat, open
link	link, mkdir, openat, open, creat
unlink	rmdir, unlink
rename	rename
sync	fdatasync, fsync, msync, sync, pwrite, pwritev, write, writev
output	write
ancillary	chdir, clone, close, dup, dup2, dup3, execve, fchdir, fchmod, fchown, fcntl, fcntl64, lseek, mmap, mmap2, mremap, munmap, shmat, shmctl, shmdt, shmget, vfork, read, readv

Table 3.2: **System Calls and Logical Operations.** *The table lists which system calls each logical operation might result from. Some system calls can result in multiple logical operations. For instance, a* `write()` *system call can result in both a* overwrite *and an* append *logical operation, if the write straddles the current size of the file; a* `write()` *can also result in a* truncate *followed by an* append *if the user did a* `lseek()` *beyond the end of the file. The "ancillary" row simply lists system calls that never directly result in a logical operation, but influence the generation of other logical operations. Note that the current version of* Alice *does not handle all system calls (e.g.,* `mknod()`*), and might need to be extended in a straightforward manner for applications that use them. * mwrite is not an actual system call, but denotes memory accesses to memory-mapped files.*

logical entities, there is no direct operation that creates or deletes an entity; Alice instead models an infinite number of logical entities that are never allocated or deallocated, but only rather simply changed. Specifically, the `link()` operation, if resulting from the creation of a file or a directory, associates with itself an entity from the infinite pool that has never been previously associated.

Micro operation	Meaning
write(inode, offset, count, data)	A data write to a file. Two special values for *data* are *zeros* and *random*; *random* writes garbage data to the file, imitating a file system that exposes uninitialized data regions. This micro operation does not change the file size; writes beyond the end of a file cause data to be stored without changing file size.
change_file_size(inode, size)	Changes the size of a file inode. Increasing the file size beyond the point where data has been explicitly written to the file exposes uninitialized data region similar to a *write(random)*
create_dir_entry()	Creates a directory entry in a directory, and associates a file inode or directory with it.
delete_dir_entry()	Removes a directory entry.
output()	Application output

Table 3.3: **Micro Operations.** *List of micro operations employed by* ALICE.

Abstract Persistence Models

Logical operations are sufficient to determine all possible in-memory file-system states that are visible to the application while it is running. By nature, the in-memory states are independent of the particular file system being used underneath. ALICE, however, is interested in the states that become visible following a system crash and reboot; the set of such possible crash states depends on the file system used.

ALICE determines the possible crash states of the workload by considering a particular file system's Abstract Persistence Model (APM). An APM is a model of the file system that, when applied to a sequence of logical operations and initial state, enumerates all possible crash states for the file system. More specifically, an APM converts a sequence of logical operations into a list of *micro-operations* and ordering constraints between the operations in the list, which can then be applied on the initial state to

Logical Operation	Micro Operations
overwrite	count× write(data)
append	count× { change_file_size write(zeros) write(data)
truncate	count× { change_file_size write(zeros)
link	create_dir_entry
unlink	delete_dir_entry + *truncate if last link*
rename	*if dest exists: unlink(dest)* create_dir_entry(dest) delete_dir_entry(source)
sync	*No micro operation*
output	output

(a) Converting Logical Operations to Micro Operations.

Description	Constraint
Overwriting same offset	$\text{write}_i(\text{inode}_A, \text{offset}_B, \text{count=1}, \text{data}_C) \rightarrow \text{write}_j(\text{inode}_A, \text{offset}_B, \text{count=1}, \text{data}_D)\ \forall\ i < j$
`sync(A)` flushes all operations on A	$\textit{any-op}_i(A) \rightarrow \textit{any-op}_k\ \forall\ \text{sync}_j(A)$ s.t. $i < j < k$
Output	$\text{output}_i() \rightarrow \textit{any-op}_j\ \forall\ i < j$

(b) Ordering Constraints.

Table 3.4: **Default APM Informal Description.** *(a) translates logical operations into micro operations;* count× *indicates micro operations generated multiple times for a single logical operation (corresponds to the size of the overwrite, append, or truncate). (b) shows ordering constraints;* X_i *is the* i^{th} *micro operation,* $X_i \rightarrow X_j$ *means that* X_i *reaches the disk (is ordered before)* X_j*, and any-op(A) is any operation on the logical entity A.*

determine the possible crash states. Each micro-operation represents an atomic modification that is internally performed by the file system on the logical entities; possible micro operations are shown in Table 3.3. Note that, while the conversion of logical operations is dependent on the file

system's APM, the resultant micro operations and ordering constraints can be interpreted in a manner independent of the file system.

To understand APMs further, consider a single `rename()` logical operation where the destination doesn't previously exist. The logical operation can be achieved through two micro operations: a `create_dir_entry(destination)` and a `delete_dir_entry(source)`. In the absence of any ordering constraints, such conversion can result in four different crash states: both the destination and the source links existing, neither existing, only the source existing, and only the destination existing. Some file systems do not allow a crash state where neither the destination or the source exist; the APM of such file systems specify an ordering constraint that the `create_dir_entry` needs to be performed before the `delete_dir_entry`. If the file system performs renames atomically, its APM can specify an additional ordering constraint, that `delete_dir_entry` needs to be performed before `create_dir_entry`; this results in a circular ordering dependency which imposes the desired atomicity.

Note the `output` micro operation in Table 3.3. In ALICE, each crash state, in addition to specifying the state of the file system at the time of the crash, also includes information about all externally visible outputs at the time of the crash. This leads to two mandatory rules in all APMs (irrespective of the file system). The first rule is that all `output` logical operations are converted directly into a `output` micro operation. The second rule is an ordering constraint: all `output` micro operations are performed before all micro operations resulting from future logical operations (i.e., if the application prints a message on screen and then writes a file, the write cannot reach the disk before the message is printed).

ALICE can be configured with the APM of any file system, but uses a hypothetical file system by default, with the aim of finding and reporting many vulnerabilities that can be exposed in multiple real-world file systems (ensuring that the application can be crash consistent atop multi-

ple file systems). We now first explain the default APM and then briefly describe how other APMs can be used with ALICE.

Default APM

ALICE is always required to be configured with an APM, and will hence always examine crash states possible with a particular file system (corresponding to the APM). However, ideally, by default, we wanted ALICE to find all vulnerabilities in the given application without assuming a file system. To reconcile these contradictory requirements, ALICE uses the APM of a hypothetical file system that can result in many crash states as its default APM, intending to capture the worst-case behavior of a multitude of file systems. The default APM achieves this goal by generating micro operations from logical operations at byte-granularity and imposing only a few ordering constraints on them. In the next subsection, we highlight how this strategy can capture the worst-case behavior of many real-world file systems.

Table 3.4 describes the default APM, while Listing 3.2 shows an example translating system calls into micro operations and ordering constraints using the APM. Table 3.4(a) shows how each logical operation is broken down into micro operations. File data operations (overwrites, appends, and truncates) are broken down at the byte level, thus encompassing our expected worst-case behavior of future NVM-specific file systems. The `count`$\times$ notation in Table 3.4(a) for the file operations thus corresponds to the number of bytes involved in the logical operation. Both the append and truncate operation increment the file size and write information for each individual byte; appending to a file introduces garbage data (the state when only `change_file_size` has happened), then zeros (`write(zeros) has happened`), and finally the actual data. Note that directory operations are broken into individual directory-entry operations, again encompassing our expected worst-case behavior of file systems.

The ordering constraints imposed by the default weak APM are shown in Table 3.4(b). First, if data in a particular position in the file is written more than once, the earlier versions should reach the disk before the later versions; this follows the working of unified buffer caches found in modern operating systems [82]. Second, all micro operations followed by a `sync` on a file A are ordered after writes to A that precede the `sync`; we thus assume a file system that takes sufficient care to force file data (regular or directory) to disk on a `fsync()`, `fdatasync()`, or `msync()` system call. The final constraint concerns the `output` logical operation described previously.

Other APMs

ALICE can also model the behavior of real file systems when configured with other APMs. Table 3.5 summarizes the APMs of five Linux file-system configurations: ext3 under the *data=journal* mode, under the default *data=ordered* mode, and under the *data=writeback* mode, and the default mode of ext4 and btrfs. We based the APMs in Table 3.5 on previous work that seeks to understand their behavior [65], and validated them based on our understanding of ext3 and ext4 and by communicating with btrfs developers [64].

Note that, by their nature, the APMs in Table 3.5 will not correspond exactly to the file systems: behavior of these file systems has so far been unspecified, can change between versions [66, 67], and even their developers are confused about the behavior [64]. Since file systems are not *required* to maintain good crash behavior, and our APMs are not officially guaranteed by file systems, file-system developers can always change the provided behavior in future versions. Hence, we have based the APMs on their current (as of May 2014) high-level design and optimizations, intending to cover past, current, and future versions of their file systems. In the future, APMs can allow file-system developers themselves to accurately specify the crash behavior that is to be expected of their file system, and

	Ordering	SAG	CA	DO
ext3-j	All operations are ordered.	4K	✓	✓
ext3-o	Directory operations, appends, and truncates ordered among themselves. Overwrites ordered before non-overwrites. All operations ordered before every sync.	4K	✓	✓
ext3-w	Directory operations and file-size operations ordered among themselves, and before sync operations. All operations on inode X ordered before sync on X.	4K	×	✓
ext4-o	Safe rename, safe file flush are satisfied. Directory operations ordered among themselves. All operations on inode X ordered before sync on X.	4K	✓	✓
btrfs	Safe rename, safe file flush are satisfied. All operations on inode X ordered before sync on X.	4K	✓	✓
default APM	All operations on inode X ordered before sync on X.	1 byte	×	×

Table 3.5: **APMs Used.** *The table summarizes the APMs of common Linux file systems. Legend: SAG: Size-Atomicity Granularity. CA: Content-Atomicity. DO: Directory-Operations atomicity. The terms content atomicity, size-atomicity granularity, directory-operations atomicity, safe file flush, and safe rename, are explained below. Ordering between file overwrites at the same offset and ordering related to output operations (further explained in Table 3.4) are satisfied in all the APMs of this table but are not explicitly mentioned.*

to implement their file system accordingly.

Table 3.5 summarizes the ordering constraints of each APM and how logical operations are split. All APMs obey the ordering constraints specified in Table 3.4(b), and these constraints are not explicitly listed in Table 3.5; the additional ordering constraints for each APM is described. The constraints for *ext4-o* and *btrfs* include *safe rename* and *safe file flush*. The

safe rename constraint orders all writes to a file before a future rename of the file. The safe file flush constraint orders all directory operations on directory entries in a file's path (up to the root of the file system) before a `sync` on the file. These ordering constraints are further explained in Chapter 4 (§4.5).

We describe how logical operations are split in Table 3.5 using the columns *size-atomicity granularity*, *content atomicity*, and *directory-operations atomicity*. Logical operations are split for all APMs as in Table 3.4(a), except for the changes described in these columns. The size-atomicity granularity column describes how `overwrite`, `append`, and `truncate` logical operations are split; if the granularity of size atomicity is 4096, they are split at 4096-byte boundaries of file offset.

The content atomicity column actually describes circular ordering constraints that impose atomicity for micro operations generated from the `append` and `truncate` logical operations. If there is a ✓ on the column, circular ordering constraints are added for each set of `change_file_size`, `write(zeros)`, and `write(data)` micro operations generated from an `append` logical operation. Similarly, with `truncate`, circular ordering constraints are added for each pair of `change_file_size` and `write(zeros)`. Thus, if there is a ✓ on the column, on a crash, an `append` or `truncate` cannot result in garbage being visible at the end of the file after rebooting. The directory-operations atomicity column similarly describes circular ordering constraints that impose the atomicity of all micro operations resulting from each `link`, `unlink`, and `rename` logical operation. The terms *size atomicity*, *content atomicity*, and *directory-operations atomicity* are further used in Chapter 4 (§4.5).

As an example, let us consider the APM in the *ext3-j* row in Table 3.5; it differs from the default APM (final row) as follows. First, logical operations are broken down into micro operations at the block (4096 byte) level for file data and without zero writes during appends. Second, or-

dering constraints specify that each micro operation requires all previous micro operations to be persisted. Finally, the APM imposes the atomicity of rename, truncate, and the unlink logical operations, and block-level atomicity of the append operation (i.e., no garbage will be observed by the user) by using circular ordering constraints.

State Re-construction

As explained, using the APM, ALICE can translate the system-call trace into micro operations and calculate ordering dependencies amongst them. We now explain how to generate crash states from the micro operations and ordering dependencies.

Consider a simple workload that generates two micro operations (say, A and B) without any ordering constraints between them, and that a crash happens during the workload or immediately after. By the time of the crash, both operations might have been applied to the initial state on disk, or neither operation might have, or only one of the operations might have. Thus, there are four possible crash states (i.e., ⟨A B⟩, ⟨_ _⟩, ⟨A _⟩, ⟨_ B⟩).

Note that the permutation among the micro operations does not matter in the construction of the crash state: ⟨A B⟩ and ⟨B A⟩ represent identical crash states. This is because of a property of micro operations and logical entities that is inherent in their definition: applying a *set* of micro operations to an initial state results in the same final state, irrespective of the order in which they are applied. This property is apparent for a simple workload, such as writing two bytes of file data. To stress this further, assume a slightly complex workload with the following two system calls:

```
mkdir(/X); mkdir(/X/Y);
```

They generate the following micro operations, assuming the inode entity of the root directory is 2, the inode entity assigned to `X` is 3, and to `Y` is 4:

```
create_dir_entry(dir=2, entry='X', inode=3) — (A)
create_dir_entry(dir=3, entry='Y', inode=4) — (B)
```

Applying the micro operation A first and then B (⟨A B⟩) represents creating the directory entry X on inode 2 (pointing to inode 3) and then the entry Y on inode 3 (pointing to inode 4). Applying B first and then A (⟨B A⟩) first creates entry Y on inode 3 (inode 3 is not yet reachable from the root directory and is thus invisible to the application) and then creates entry X on inode 2 (making inode 3 visible). Also note that, in this example, the states ⟨_ _⟩ and ⟨_ B⟩ appear equivalent to the application, since inode 3 is not visible without micro operation A, and micro operation B operates on inode 3. We do not discuss such application-visible state equivalence for now, and consider each set of micro operations to form a unique crash state; optimizations using such equivalence are discussed in Section 3.5.

Thus, each set of the micro operations can result in a unique crash state. If we consider a workload that generates N micro operations without any ordering constraints, there are 2^N possible unique crash states. Ordering constraints reduce the number of valid crash states. For example, if the constraints state that B should be ordered after A in the initial example, the crash state ⟨_ B⟩ is invalid.

Hence, to reconstruct crash states, ALICE selects different sets of the translated micro operations that obey the ordering constraints. It then applies each operation in the selected set sequentially to the logical entities in the initial file-system state, producing the state of the logical entities after the crash. For each such reconstructed logical crash state, ALICE then converts the logical entities back into actual files and supplies them to the checker, which then verifies the crash state.

3.3 State Exploration

The space of all possible crash states for a given sequence of micro operations and ordering constraints (i.e., for a given system-call trace) can be huge. By default, ALICE targets specific crash states that concern the

ordering and atomicity of each individual system call, as explained in the following subsections. The intuition behind this strategy is to explore a few representative crash states for each error that a developer might make while implementing the application's update protocol. If desired, the user can configure ALICE with a different exploration technique.

Atomicity *across* System Calls

The programmer might have accidentally assumed that multiple system calls are persisted together atomically, and ALICE first constructs a set of crash states designed to check for such a requirement. The requirement is easy to check: if the protocol has N system calls, ALICE constructs one crash state for each prefix (i.e., the first X system calls, $\forall\ 1 < X < N$) applied. In the sequence of crash states generated in this manner, the first crash state to have an application invariant violated (i.e., where the checker fails) indicates the start of an atomic group. The invariant will hold once again in crash states where all the system calls in the atomic group are applied. If ALICE determines that a system call X is part of an atomic group, it does not proceed with constructing further crash states that involve the ordering and atomicity of X as discussed in the next subsections.

System-Call Atomicity

The programmer might have assumed that a single system call will always be persisted atomically. ALICE tests this for each system call by applying all previous system calls to the initial state, and then generating crash states corresponding to different intermediate states of the system call. Note that there can be many possible intermediate states for a file write or truncate system call. For example, a 10-byte `write()` system call in the default APM can result in 2^{10} states, since it generates 10 micro operations. Similarly, 10-block writes would result in 2^{10} states in file systems such as

btrfs and ext3, since these file systems break `write()` system calls at block granularity.

Hence, for file overwrites, file appends, and file truncates, we explore only a subset of the possible intermediate states. Assuming that the application writes some user-level data structure in the system call, the goal is to produce intermediate states that result in a medley of cases where different fields in the data structure are persisted partially, and when some fields are persisted without the others. Examining crash states where the data structure is persisted partially will likely reveal any crash vulnerabilities.

We now explain the intermediate states explained for file overwrites. We then briefly describe the states explored for truncates and appends, and for directory operations.

File Overwrites

With file overwrites, we explore three intermediate states for each micro operation constituting the system call; to understand the intermediate states explored, consider an example where the system call has been split into three micro operations, A, B, and C. First, the state where only the part is persisted among all parts is explored (i.e., only A, only B, only C). Second, the state where only the part is not persisted is explored (i.e., B C for part A, A C for part B, A B for part C). Third, we explore the state where all parts up to the considered part are persisted (i.e., only A for part A, AB for part B, ABC for part C).

Thus, for each overwrite, we explore $3 \times M$ states, where M is the number of micro operations the system call has generated. For most non-default APMs modeling real file systems, since micro operations are generated from the system call at the block (or sector) granularity of the file system, this strategy results in a practical number of states. However, since the default APM has byte-granularity micro-operations, the strategy is impractical for bigger writes.

Hence, with the default APM, instead of exploring three intermediate states for *each* micro operation, the micro operations are grouped into chunks, and the three intermediate states are explored per chunk. Specifically, the micro operations are first grouped into 4 KB chunks (a common block size), and the three states are explored per chunk; then, they are grouped into 512 B chunks (a common sector size) and the states explored; then, all the micro operations forming the system call are grouped into three equally sized chunks and the states explored.

File Appends and Truncates

We follow a similar strategy for appends and truncates. With truncates, for non-default APMs, if micro operations can result in the file containing garbage, the strategy automatically investigates states containing garbage. For the default APM, chunks are grouped together such that the garbage-filled state corresponding to each chunk is also investigated. For appends, the strategy also investigates zero-filled states.

Directory Operations

For directory operations, enumerating all possible intermediate states when testing atomicity is practical, and is followed by ALICE. Note that a system call performing a directory operation (such as `unlink()`) might also include a file truncate; in such cases, only selective intermediate states are considered for the truncate (per the strategy described previously), while all possible states are considered for the rest of the system call.

Ordering Dependency among System Calls

The programmer might assume various system calls are persisted in the relative order in which they were issued. Consider a workload that produces N system calls without any ordering constraints imposed by the

underlying APM, such as N file creations in the default APM. There are $2^N - N - 1$ different crash states where a crash has happened with some system calls persisted out-of-order (but all persisted system calls were persisted atomically). Thus, testing all states where system calls have been persisted out-of-order is impractical, similar to the situation explained in the previous section.

Hence, the default exploration strategy tests a subset of the crash states caused by re-ordering. Specifically, for every possible pair of system calls in the workload, we test the crash state produced by re-ordering those two system calls, with all the other system calls having persisted atomically and in-order. To achieve this for the pair of two system calls A and B, where A occurs before B during the application's execution, ALICE applies the micro operations corresponding to every system call from the beginning of the protocol until B, except for A.

Summary

To summarize, ALICE's default exploration strategy is not exhaustive (and hence not complete). The default strategy does not test crash states where more than one system call had persisted partially (i.e., not atomically), more than one pair of system calls were persisted out-of-order at the time of the crash, or a combination of both these scenarios. It also only tests a subset of the states concerning parial persistence of each system call. However, we found that the exploration strategy discovers most application vulnerabilities in a practical amount of time; this is quantitatively discussed in Section 3.6.

3.4 Static Vulnerabilities

The previous sections elaborate how ALICE re-constructs a targeted set of crash states possible under the given workload, and how the user-supplied

checker is run on each re-constructed state. The checker might determine that the expected application-level invariants are satisfied in some of the re-constructed states, and are not satisfied on others. Each of the crash states where invariants are violated represents a possible failure scenario for the application. We now address how to present to the user the list of crash states where invariants are violated. We first introduce the reader to a simple but useful method, dynamic vulnerabilities, and then to a better method, static vulnerabilities.

Dynamic Vulnerabilities

A naive strategy is to simply list the crash states where invariants are violated, with each state described in terms of the micro operations that constitute the state. However, this strategy does not provide information on how each state was constructed, and hence does not provide any directly usable information regarding the cause behind the observed inconsistencies.

With the default state-exploration technique followed by ALICE, a better strategy can be used; the observed inconsistencies can be described in terms of whether they can be fixed by atomically persisting multiple system calls together, by atomically persisting a particular system call, or by maintaining the order between a pair of system calls. We term the vulnerabilities described in such a way that provides directly usable information (in terms of system calls in the trace) about the inconsistency as **dynamic vulnerabilities**. Note that the precise definition of dynamic vulnerabilities depends on the state-exploration technique. Hence, if a user decides to customize the exploration technique (as described in the previous section), the user also has to re-define the method used to identify unique dynamic vulnerabilities from a list of inconsistent crash states.

Even though dynamic vulnerabilities provide better information than a simple list of crash states, they still represent *failures* and not *errors*. For

example, consider an application workload issuing ten writes in a loop; the system-call trace would then contain ten `write()` system calls. If each write is required to be atomic for application correctness, ten dynamic vulnerabilities are detected, since dynamic vulnerabilities are described in terms of the system calls in the trace; however, they are all caused by a single source line. Thus, simply providing ALICE with a longer running workload will output more dynamic vulnerabilities, although they might be caused by the same number of errors as reported in a shorter workload.

Static Vulnerabilities

ALICE solves this problem using stack-trace information to correlate all 10 system calls (i.e., the 10 dynamic vulnerabilities, or the 10 failures) to a single error, or a single **static vulnerability**. One way to report static vulnerabilities is to simply report the set of unique stack traces among the different dynamic vulnerabilities detected. In the previous example, all 10 dynamic vulnerabilities have the same stack trace, and will hence correspond to a single static vulnerability. While this method of reporting static vulnerabilities might be sufficient in practice for developers, it is not sufficient for a detailed study of the number of logical errors and their implications in applications, as in Chapter 4. This is because a single error might correspond to two different stack traces; for example, an application workload might trigger the same application action (that happens to cause a dynamic vulnerability) from different places.

Hence, ALICE chooses the innermost stack frame from the stack trace of each dynamic vulnerability, and reports the number of such unique stack frames across all dynamic vulnerabilities. This is more meaningful than using the entire stack trace: even if the same application action (that causes a dynamic vulnerability) is called from different places in the application workload, since the resulting dynamic vulnerabilities will have the same innermost stack frame, only one static vulnerability will be reported. In

the rest of this thesis, any report of the number of vulnerabilities in a particular application corresponds to the number of such static vulnerabilities discovered, i.e., the number of errors discovered in the application instead of the number of failures discovered. To identify errors, we use the number of unique innermost stack frames rather than the number of unique stack traces.

Note that, in some applications, the innermost stack frame is actually a wrapper to the C-library, and hence the number reported might be less than the actual number of vulnerabilities. As an example, consider two separate modules of an application, each requiring a `write()` to be atomic. If both modules issue the `write()` via the same wrapper function, since the wrapper function forms the innermost stack frame in the dynamic vulnerability, both will be reported as a single static vulnerability. To combat this, ALICE can be configured to choose an alternate set of representative stack frames (instead of only the innermost) while determining static vulnerabilities. In Chapter 4, we configure ALICE to choose a non-default stack frame for applications that we know use such wrapper libraries (LevelDB, Git, and SQLite).

3.5 Implementation

ALICE consists of around 4000 lines of Python code, and also traces memory-mapped writes in addition to system calls. It employs a number of optimizations. First, ALICE caches crash states, and constructs a new crash state by incrementally applying micro-operations onto a cached crash state. Second, we found that the time required to check a crash state was often much higher than the time required to incrementally construct a crash state. Hence, ALICE constructs crash states sequentially, but invokes checkers concurrently in multiple threads. Third, different micro-op sequences can lead to the same crash state, as explained in Section 3.2. For example,

different micro-op sequences may write to different parts of a file, but if the file is unlinked at the end of sequence, the resulting disk state is the same. Therefore, ALICE hashes crash states and only checks the crash state if it is new. Finally, we found that many applications write to debug logs and other files unrelated to application correctness. ALICE allows filtering out system calls related to these files with configuration options.

While logical entities, APMs, and micro operations are necessary abstractions for the basic methodology used by ALICE to discover crash vulnerabilities, logical operations only simplify the methodology and are not strictly necessary. In some cases, however, we found that using logical operations as a hard abstraction makes the methodology more complex. For example, consider the fact that APMs describe the conversion of logical operations into micro operations. It is possible that, for some file systems, the micro operations for a `overwrite` logical operation depends on whether the overwrite resuled from a `pwrite()` or a `write()` system call. Therefore, defining APMs to be entirely based on logical operations requires each logical operation to also encode such information. Instead, we treat logical operations as a soft abstraction in ALICE: while they are often used as such, in unusual cases, an APM or a customized state-exploration technique can instead refer directly to the system-call trace.

3.6 Evaluation and Discussion

In this section, we discuss the following questions:

- How difficult is it to write workloads and checkers?
- How much time does running ALICE take?
- Does ALICE's default state-space exploration strategy offer sufficient space coverage?

We answer these questions based on our experience in using ALICE to find crash-consistency vulnerabilities in eleven important applications, further described in Chapter 4. Specifically, in this section, we examine and discuss in detail a specific case that is relevant to each question, and perform additional experiments when necessary.

Workloads and Checkers

ALICE can be considered similar to a unit-testing framework: it stresses the given testcase to find vulnerabilities, and hence only discovers vulnerabilities associated with the testcase. In ALICE, a testcase corresponds to the combination of a workload and a checker. We now discuss the workload and checker used to study the LevelDB application in the next chapter; we find LevelDB to be a representative example for the effort required to write workloads and checkers. We first provide a brief overview of LevelDB, and then discuss the workload and the checker.

LevelDB is a key-value store that mainly supports *Put(key, value)* and *Get(key)* operations, along with other operations such as deletes and atomic batches of multiple *Put*s. LevelDB guarantees that *Put* operations are atomic and ordered with respect to a system crash. *Put*s are also guaranteed to be immediately durable if a *sync* flag is set.

On *Put* operations, LevelDB stores the inserted key-value pairs in a *log* file. When the file reaches a threshold size, it is *compacted* into a query-optimized *table* file, and any future *Put*s are stored in a new log file. When certain thresholds are reached, old table files are further compacted into newer table files (optimizing them further for queries). Note that this description of LevelDB's write protocol is simplified in many aspects, including concurrency and crash consistency; LevelDB auguments the protocol carefully with checksums and `fsync()` for crash consistency. More details of LevelDB are discussed in the next chapter.

Workload

Our workload for LevelDB consists of 40 lines of C++ code, including 15 lines of helper code that generates random strings. Although small, the workload was designed carefully based on a conceptual understanding of the application's write protocol.

The workload performs 10 *Put* calls (50 KB each) that trigger two compactions (including a compaction of old table files into new table files), with the *sync* flag set in the last *Put*. The workload also, initially, adjusts the threshold sizes triggering compaction in LevelDB to a smaller value (64 KB) than the default. In addition, the workload closes and opens the database again, to trigger a part of LevelDB's write protocol that is run only when opening a pre-existing database.

Thus, we used a conceptual understanding of the write protocol (explained previously) to design the workload. The conceptual understanding was required to trigger each unique part of the protocol, such as normal insertions into the log, durable (synchronous) insertions, and compactions. We also used the understanding to examine all the parts of the write protocol while reducing the overall size of the workload: we did this by reducing the compaction threshold. Smaller workloads result in smaller system-call traces, reducing the number of states explored by ALICE and its running time as described in the next section.

Such a conceptual understanding of the write protocol will be easily available to the authors of the protocol, i.e., the developers of the application. In our case, we initially studied the protocol from LevelDB's documentation. We then wrote a workload that simply inserted 10,000 key-value pairs (20 lines of C++ code) and supplied it to ALICE, and examined ALICE's representation of the exact workload protocol (which ALICE outputs in a user-friendly format using logical operations). This examination revealed that the exact write protocol remained the same for different compactions of log files, and for different compactions of old table files. Thus,

we gained confidence that one log compaction and one table compaction in our final workload were sufficient to examine the compaction-related portion of the write protocol. We also repeated this process to test whether the exact protocol differed for different key-value sizes, or if separate parts of the write protocol were invoked while opening and closing the database. We used our observations to craft the 40-line final workload described previously.

Other applications described in the next chapter required similar effort to design the workload. With four applications (Git, Mercurial, HDFS, and Zookeeper), since the applications allowed many different actions to be performed (instead of just *Put* and *Get*), our workloads examined only a few of the actions.

Checker

Our LevelDB checker contains 90 lines of C++ code. It tries to open the database, and if successful, checks for corrupted data, atomicity, ordering, and durability if the crash happened after the synchronous *Put*.

Unlike the workload, which required a understanding of the application's protocol for effectiveness, the checker only required knowing the crash-consistency guarantees of the application. However, we had to examine the documentation manually to determine the guarantees and the correct options to open the database correctly; many applications include configuration options that either disable proper recovery after a crash or trade-off crash consistency for performance while running. Our LevelDB checker enables the options *paranoid_checks*, and tries the function *RepairDB()* on a detected error before actually failing.

Other applications required more effort for writing the checker. For instance, consider Git; Git specifies many interactive, manual steps that are meant to be performed by an user if failures are observed after a system crash. Our checker hence requires much text-processing code.

Summary

To summarize, writing an effective workload is easy if a conceptual understanding of the application's write protocol is known. Similarly, writing a checker is easy once the correct usage of the application, and the exact guarantees expected of it, are determined.

Running Time

We measured the time taken by ALICE to find vulnerabilities in LevelDB-1.15 on a machine with an Intel Core 2 Quad Processor Q9300 and 4 GB of memory running Linux 3.13, with a HDD (Toshiba MK1665GSX 160 GB). ALICE uses the default APM and the default state exploration strategy, and uses 4 threads to invoke checkers, with the state reconstruction performed in a separate thread.

The LevelDB workload described in the previous section results in 166 logical operations corresponding to 369 micro operations with the default APM. ALICE explores a total of 5121 crash states in 1321 seconds, of which 802 seconds is spent in invoking checkers. Note that the 802 seconds represents running checkers in parallel: each individual checker execution takes 0.626 seconds on average.

With other applications, invoking checkers contributed to an even higher percentage of the total running time. For example, with Git, the workload results in 53 logical operations (99 micro operations), and ALICE explores a total of 1692 crash states in 1434 seconds. If the Git checker were to be substituted with the */bin/echo* tool, however, ALICE takes 124 seconds to test Git.

Effectiveness of State Space Exploration

The default state-space exploration strategy in ALICE is intuitively designed to explore programmer errors. However, it is possible that other crash

states also result in vulnerabilities. To measure the effectiveness of ALICE's default exploration strategy, we compare it to a customized exploration strategy described here.

The customized strategy explores all unique crash states (i.e., all possible sets of micro operations), except for those associated with system calls already found to cause vulnerabilities using the default strategy. Specifically, the customized strategy chooses each possible set of micro operations, validates it with the ordering constraints of the default APM, and verifies whether all previously-known culprit system calls are present in their entirety in the set. If the currently chosen set satisfies these conditions, the corresponding state is explored.

On checking LevelDB-1.15 with the customized strategy, after running for 72 hours, 514126 states were explored and no additional vulnerabilities were found. As will be described in the next chapter, a detailed manual examination of LevelDB also did not find more vulnerabilities. Thus, these results provide more confidence that the default exploration strategy sufficiently covers the state space to find most crash vulnerabilities.

Summary

In summary, ALICE proves to be a practical tool to find vulnerabilities in applications. Effective workloads and checkers can be designed with a simple conceptual understanding of the write protocol. ALICE takes less than an hour to examine complex applications, with the majority of the time consumed by the user-specified checker. The state-space exploration strategy followed by ALICE proves effective in finding vulnerabilities while reducing the total time taken.

3.7 Limitations

ALICE is not complete, in that there may be vulnerabilities that are not detected by ALICE: ALICE's default exploration technique does not explore all possible crash states. It also requires the user to write application workloads and checkers; we believe workload automation is orthogonal to the goal of ALICE, and various model-checking techniques can be used to augment ALICE. For workloads that use multiple threads to interact with the file system, ALICE serializes system calls in the order they were issued; in most cases, this does not affect vulnerabilities as the application uses some form of locking to synchronize between threads. ALICE currently does not handle file attributes or features such as file holes; it would be straight-forward to extend ALICE to do so.

```
open(path="/x2VC") = 10
```

Logical operation: None
Micro operations: None
Ordered after: None

```
write(fd=10, size=2, "ab")
```

Logical operation: overwrite(inode=8, offset=0, count=2, data="ab")
Micro operations: #1 write(inode=8, offset=0, size=1, data="a")
#2 write(inode=8, offset=1, size=1, data="b")
Ordered after: None

```
fsync(10)
```

Logical operation: sync(inode=8)
Micro operations: None
Ordered after: None

```
write(fd=10, size=2, "cd")
```

Logical operation: overwrite(inode=8, offset=2, count=2, data="cd")
Micro operations: #1 write(inode=8, offset=2, size=1, data="c")
#2 write(inode=8, offset=3, size=1, data="d")
Ordered after: #1, #2

```
link(oldpath="/x2VC", newpath="/file")
```

Logical op: link(dir_inode=2, link_name='file', link_inode=8, link_count=2)
Micro operations: #5 create_dir_entry(dir=2, entry='file', inode=8)
Ordered after: #1, #2

```
write(fd=1, data="Writes recorded", size=15)
```

Logical operation: output("Writes recorded")
Micro operations: #6 output("Writes recorded")
Ordered after: #1, #2

Listing 3.2: **Example: System Calls Converted To Micro Operations.** *Micro-operations resulting from each system call in a trace are shown along with their ordering dependencies when using the default APM. Logical entities are shown as inode numbers; the inode number assigned to* `x2VC` *is 8, and for the root directory is 2. Some details of the shown system calls have been omitted.*

4
Vulnerabilities Study

In this chapter, we study the update protocols of real-world applications. We first perform an extensive manual study of two applications (LevelDB and SQLite), examining their update protocol, previously reported crash vulnerabilities, and the relationship between the application's configuration, performance, and crash consistency. The manual study provides us insight into how an automatic study can be designed for real-world applications. We then use ALICE to examine 11 widely-used applications (focusing on single-node applications), to find whether their update protocols have been implemented correctly, whether file-system behavior significantly affects application users, and which file-system behaviors are thus important. Our study thus requires finding vulnerabilities under an abstract file system and understanding them in terms of relevant file-system behavior, a unique advantage of ALICE.

The applications used in the study each represent different domains, and range in maturity from a few years-old to decades-old. We study three key-value stores (LevelDB [29], GDBM [28], LMDB [94]), three relational databases (SQLite [91], PostgreSQL [99], HSQLDB [35]), two version control systems (Git [47], Mercurial [54]), two distributed systems (HDFS [81], ZooKeeper [2]), and a virtualization software (VMWare Player [107]). We study two versions of LevelDB (1.10, 1.15), since they vary considerably in their update-protocol implementation. Note that the consequences of the vulnerabilities we find are not easily quantified; they vary based on user

expectations in the deployed environment. Moreover, we also consider file-system behaviors that may not be common now (such an non-atomic sector-level data writes) but may become prevalent in the future by using the default APM in ALICE. Hence, the results are not suited to relatively quantify and compare the correctness between different applications.

4.1 Manual Case Studies

We performed a manual study of crash consistency in two applications, SQLite and LevelDB-1.10. The purpose of our manual study is to understand how crash consistency has been previously dealt with by application developers and users in real applications, so as to provide the correct context within which we can perform a broader automated study that is described in the next section.

To achieve this goal, we first understood the crash invariants guaranteed by both applications, and their conceptual update protocol as described in the design documents. We also examined developer attitudes towards file-system behavior and past vulnerabilities already fixed in each application. We then manually examined system-call traces to identify potential vulnerabilities and verified them by introducing environmental changes to increase the window of vulnerability and a system crash during the window.

4.1.1 LevelDB

LevelDB [29], already introduced in the previous chapter, is a persistent key-value store originally developed in Google. LevelDB is widely deployed; the Chromium web browser uses it as an embedded storage library. To our knowledge, LevelDB does not formally document the guarantees it provides on a system crash. However, the documentation and examples [22] suggested that inserting a key-value pair is atomic (with respect

to a system crash), ordered after all previous inserts, and depending on a configuration option (*sync*), durable. LevelDB also provides a few configuration options on how to deal with I/O errors and recovery that concerned crash consistency: we use the safer (but not default) configuration options of *verify_checksums* and *paranoid_checks*.

Update Protocol

LevelDB's write protocol was briefly described in Chapter 3. To review, LevelDB adds inserted key-value pairs to a *log* file until it reaches a threshold, and then switches to a new log file. During the switch, a background thread starts compacting the old log file to a query-friendly table (*ldb* or *sst*) file. During compaction, LevelDB first writes data to the new table file, updates pointers to point to the new file (by appending to a *manifest*), and then deletes the old log file. LevelDB also periodically compacts multiple old table files into a new table file.

Vulnerabilities

There was one bug related to crash consistency in LevelDB that was reported previously and subsequently fixed (Bug #68, detailed later); this bug happens only in select file systems. With manual examination, we discovered four more places where LevelDB was vulnerable and reported them to the LevelDB bug database [46].

For the manual examination, we first gained a conceptual understanding of LevelDB's write protocol, and then wrote workloads that exercised various parts of the protocol; these steps are similar to writing a workload for ALICE as described previously. We then ran the workloads, collected system-call traces, and examined the traces. With our knowledge of LevelDB's recovery protocol and the crash behavior of various file systems, we looked for suspicious patterns in the system-call trace that did not correspond to our understanding of LevelDB's recovery protocol. Unlike

writing a workload for ALICE, we had to examine system calls in detail; the process took about 15 days. We then tested each suspicious pattern in the system-call trace for a crash vulnerability; this was done by modifying the source code to mainly introduce timing changes, re-running the workload, and then manually hard-rebooting the machine.

We now describe both the previously reported vulnerability in LevelDB and those we found using manual examination. We also describe the file-system behavior required to expose the vulnerability, how we reproduced the vulnerability manually, and the ticket number in the bug database corresponding to the vulnerability.

Bug #68: This bug had been previously reported and fixed. When opening a database, LevelDB updates a file by first creating a temporary file, writing new contents to it, then renaming it over the original file. However, LevelDB does not flush the contents of the file to disk before issuing the `rename()`. If the file system does not make sure that the new data within the file is persisted before the `rename()`, a system crash might result in the (renamed) file not containing the desired contents; LevelDB reports corruption in this case.

Bug #183: During the initial creation of a database, LevelDB writes initialization data to a newly-created table file, then writes a pointer to the data in a newly-created manifest file (without any `fsync()` in-between). If the system crashes during the creation, the pointer might get written to the disk first, before the data in the table file; after reboot, LevelDB would report an error if the user tries to recreate (or open) the database.

We reproduced the vulnerability by modifying LevelDB to call an `fsync()` on the manifest file and sleep for a few seconds, immediately after writing the pointer. We ran the workload on ext4 and rebooted the machine during the sleep, after which we observed the pointer on the manifest file without the table file containing initialization data, and LevelDB reporting an error.

Bug #187: LevelDB switches log files when they reach a certain size. LevelDB does not ensure that the old log is completely on disk before switching to the new log file; if the new log is persisted but the old one is not, LevelDB recovers the entries in the new log file without being aware of the loss of older entries, violating the order of the inserted key-value pairs.

We reproduced the vulnerability similar to the previous vulnerability, by introducing an `fsync()` on the new log file and a sleep, immediately after writing to the new log file. We manually crashed the machine as soon as the sleep started, thus avoiding the old log file's data from being flushed (from the buffer cache) to the disk by the background flushing daemon. On reporting this vulnerability, we found there was some confusion among the developers and users of LevelDB as to whether the ordering of inserts is guaranteed, but it was then resolved.

Bug #189: When compacting log files into table files, the table file is first created with a temporary name, all data is written to it (and flushed), then atomically renamed; finally, the log files are unlinked. However, the rename is not explicitly persisted before unlinking the log files. On a crash, the old log files might be permanently deleted, while the new file still exists under the temporary name (and is hence not recognized by LevelDB). After recovery, either corruption is reported, or some (previously existing) key-value pairs disappear without any indication of an error.

This vulnerability cannot be reproduced on the current version of ext4 using only timing changes; it only happens on other file systems (such as btrfs). Since we were not experienced with timing effects in other file systems, to reproduce the vulnerability, we removed the rename calls, and crashed the machine after the unlinks. We observed the previously mentioned failures when we tried to read the database after rebooting.

Bug #190: LevelDB assumed that performing an `fsync()` or `fdatasync()` on a file also flushes the directory entry of the file (as well as directory entries in its path). If the file system does not guarantee this property,

there were multiple chances of corruption within LevelDB.

This vulnerability can also not be reproduced in ext4 with only timing changes. To reproduce it, we took a file that we suspected required its directory entry to be flushed for correctness. We modified LevelDB to unlink the file after an `fsync()` was called on it (but not on its parent directory), and immediately crashed the machine after the unlink (by sleeping after the unlink and manually rebooting the machine). We observed corruption when we tried to read the database after rebooting the machine. This vulnerability has been partially fixed by the developers since we reported it (LevelDB now explicitly also flushes the directory entry, but only for *manifest* files).

Summary

LevelDB does not formally document the invariants maintained across a crash, but a reasonable set of invariants can be determined from the usage examples provided in the documentation. LevelDB's update protocol is complex, and some parts of the update protocol are triggered only for specific input workloads (e.g., periodic compaction of old table files requires long-running workloads). A vulnerability was previously reported and resolved; careful manual examination allows us to identify four more.

4.1.2 SQLite

SQLite [91] is a relational database that provides ACID guarantees, and is widely used as an embedded library in desktop and mobile applications. SQLite features an inbuilt crash-recovery test suite [89] that rigorously tests SQLite's crash invariants during power failures. This, combined with the long history of bug fixing in SQLite, seemingly leaves it portable across file systems: on examining it manually, we found no evidence of vulnerabilities. A few past bugs are interesting, and are described further

subsequently in this section. We also describe a mismatch between the crash invariants documented and implemented by SQLite.

Because of the extensive testing, the protocol used by SQLite to ensure crash invariants is *pessimistic*: performance is sacrificed. The developers recognize this, and present a solution in the form of a set of configurable options, each of which slightly changes the protocol (thus improving performance). The developers suspect that, given different underlying storage and file systems, a subset of these configuration options can be switched on without sacrificing correctness. Thus, each option depends on a specific file-system behavior; however, the developers do not understand which of these behaviors are satisfied by existing file systems [86]. In this section, we also describe the configuration options (related to file-system behaviors) that SQLite provides and their performance impact.

Update Protocol

SQLite allows a choice of two different protocols, *rollback journaling* and *write-ahead logging*. *Rollback journaling* is older and the default; *write-ahead logging* is newer, faster, and seems more performance-resistant to different file-system behaviors, but does not support some of SQLite's features such as transactions across multiple databases. We focus on *rollback journaling* for our manual study.

In *rollback journaling*, when the user modifies a database, SQLite first creates a temporary journal file, and appends a copy of some information in the database file to the journal file. The database file is then actually updated, and after that, the journal is deleted. If the update was interrupted due to a crash, the journal file will be left on-disk; if SQLite finds a valid journal while opening the database, it recovers the contents appropriately.

Configurable File-System Behavior

We present here five relevant configuration options in SQLite that optimize the protocol according to the behavior of the underlying file system. We also evaluated SQLite's performance when each of these options are (separately) toggled, using two micro-benchmarks. The insert benchmark creates six tables of two columns each, and inserts a number of rows in each of them; each insert is a separate transaction. The update benchmark first inserts 300 rows into each table, then measures the performance of the SQL UPDATE statement on all rows in each table; each update (containing all rows in a table) is a transaction.

The observed performance is shown in Table 4.1. Note that the evaluated configurations do not necessarily represent correct behavior: depending on the file system, SQLite might be vulnerable during a crash. We used a single core machine running Ubuntu 12.04, a 80 GB hard drive with ext4-current, and SQLite version 3.7.17.

Safe append: During recovery after a system crash, SQLite must find out if the journal file is valid, or if a crash happened while the journal file was being written to. Validity is checked using a header in the journal file; the header is marked valid only after the rest of the journal file is updated and flushed using `fsync()`. All writes to the journal file are appends; the extra `fsync()` is hence necessary only if the file system allows an user to observe, after a crash, the file-size increase of an append without actual data in the appended region (i.e., allows observing garbage in the appended region). If a file system does not allow such behavior, SQLite can detect the validity of the journal file without the extra `fsync()`.

Power-safe overwrite: This option assumes that no data in a given file, other than those explicitly over-written, are ever affected by a system crash. SQLite decides the *granularity* of information copied to the journal file based on this parameter. This property is probably true in most modern file systems (and storage stacks), and is switched on by default in the

current version of SQLite.

Atomic writes: Given a granularity of atomicity, this option assumes that all `write()` calls lesser than that granularity are atomic (with respect to system crash). Thus, if a transaction is smaller than this granularity, SQLite can directly modify the database without using a journal file.

Sequential writes: This option assumes that, if a sequence of `write()` calls are issued, all calls only get persisted in-order. Thus, all `fsync()` operations required while creating the journal file and writing information to it, can be omitted.

Synchronous directory operations: When the temporary journal file is created, and also during a couple of other directory operations, SQLite normally flushes the directory after the operation. This configuration option omits the directory flushes.

Table 4.1 shows that each option significantly affects throughput. Especially important are the performance effects of *safe append*: this is an option that can be safely switched on with most modern file systems. Also, we modified SQLite to implement a new option corresponding to only those directory flushes that can be omitted on ext4; it performs similar to *synchronous directory operations*. We believe that *power-safe overwrite* improves performance because writes now happen at file-system-block granularity.

Past Bugs and Vulnerabilities

We examined all bugs from October 2009 to May 2013 in SQLite's bug database; we also studied a few older bugs that seemed interesting. Among the studied bugs, three specifically affect system crashes [87, 88, 90].

An early bug reported in SQLite [87] (before the introduction of the crash-test suite) concerns the requirement of separately flushing the directory entry of a file after flushing the file; SQLite was modified to `fsync()` the parent directory after creating files. The developers were concerned about performance while fixing this issue.

File-System Behavior Configuration	Throughput (X/s)	
	Insert	*Update*
Default	10.25	9.23
Atomic writes	33.39 (+226%)	32.62 (+253%)
Safe append	14.47 (+41%)	12.62 (+37%)
Sequential write	33.88 (+231%)	32.52 (+252%)
Power-safe overwrite	10.43 (+2%)	10.33 (+12%)
Synchronous directory	11.24 (+10%)	9.98 (+8%)

Table 4.1: **SQLite performance under rollback journaling.** *The table represents throughput obtained when different configuration options are toggled. Each row reports throughput when separately toggling the relevant configuration option with all other options are set to their default values. "Default" represents running SQLite without changing any of the defaults (this switches on power-safe overwrite, and switches off the others). The "Power-safe overwrite" row represents the performance when the option is switched off. We believe that power-safe overwrite improves performance because writes now happen at file-system-block granularity. "Atomic writes" represent a configuration in which all writes are atomic. The configurations do not necessarily represent correct behavior, and SQLite might be vulnerable during a crash.*

A later bug [90] deals with recovery. During recovery, SQLite wrote the contents of the journal file to the database file, and then deleted the journal file; the database file was never flushed in-between. Thus, if a system crash happened again during recovery, the database file might be permanently left in a partially-recovered (corrupt) state.

The third bug [88] deals with entries being appended to a log file, in SQLite's other update protocol (*write-ahead logging*). When the log of entries exceeds a certain threshold, SQLite wraps around, and starts writing entries again from the beginning of the log. However, without any flushes, such a wrap-around of the log could cause an older, invalid transaction to get replayed during recovery, causing corruption.

Summary

SQLite has reasonably well-defined crash invariants, two sets of update protocols, and a testing framework for its update protocols. SQLite offers many configuration options that tweak its update protocols to take advantage of specific file-system behaviors, complicating reasoning about correctness. However, because of its crash-testing framework, three vulnerabilities have been previously reported and fixed in the protocols. We performed a detailed manual examination similar to that described for LevelDB, involving around 15 days of manual effort, but were unable to find any suspicious patterns that could be vulnerable in the trace.

4.2 Workloads and Checkers

As explained in the previous section, many applications have configuration options that change the update protocol and the guarantees offered (i.e., invariants maintained) on a system crash. Furthermore, the guarantees might not be well defined; indeed, for some applications (Git, Mercurial), we could not find any documented guarantees. Hence, when using an automated tool to understand application update protocol and discover vulnerabilities, it is important to consider the configurations and the workload tested, and the checker used. We now discuss the workloads and checkers for each application class. Where applicable, we also present the guarantees we believe each application makes to users, information garnered from documentation, mailing-list discussions, interaction with developers, and other relevant sources.

Key-value Stores and Relational Databases. For LevelDB, LMDB, GDBM, HSQLDB, SQLite-Rollback, SQLite-WAL, and PostgreSQL, the workload tests different parts of the protocol, typically opening a database and inserting enough data to trigger checkpoints (or compactions with LevelDB). The checkers check for atomicity, ordering, and durability of

transactions. We note here that GDBM does not provide any crash guarantees, though we believe lay users will be affected by any loss of integrity. Similarly, SQLite does not provide durability under the default update protocol (i.e., default journal mode; we became aware of this only after interacting with developers), but its documentation seems misleading. We enable checksums on LevelDB.

Version Control Systems. Git's crash guarantees are fuzzy; mailing-list discussions suggest that Git expects a fully-ordered file system [48]. Mercurial does not provide *any* guarantees, but does provide a plethora of manual recovery techniques. Our workloads add two files to the repository and then commit them. The checker uses commands like `git-log`, `git-fsck`, and `git-commit` to verify repository state, checking the integrity of the repository and the durability of the workload commands. The checkers remove any leftover lock files, and perform recovery techniques that do not discard committed data or require previous backups.

Virtualization and Distributed Systems. The VMWare Player workload issues writes and flushes from within the guest; the checker repairs the virtual disk and verifies that flushed writes are durable. HDFS is configured with replicated metadata and restore enabled. HDFS and ZooKeeper workloads create a new directory hierarchy; the checker tests that files created before the crash exist. In ZooKeeper, the checker also verifies that quota and ACL modifications are consistent.

If ALICE finds a vulnerability related to a system call, it does not search for other vulnerabilities related to the same call. If the system call is involved in multiple, logically separate vulnerabilities, this has the effect of hiding some vulnerabilities. Most tested applications, however, have distinct, independent sets of failures (e.g., *dirstate* and *repository* corruption in Mercurial, consistency and durability violation in other applications). To combat this, we use different checkers for each type of failure, and report vulnerabilities for each checker separately. Specifically, we use a

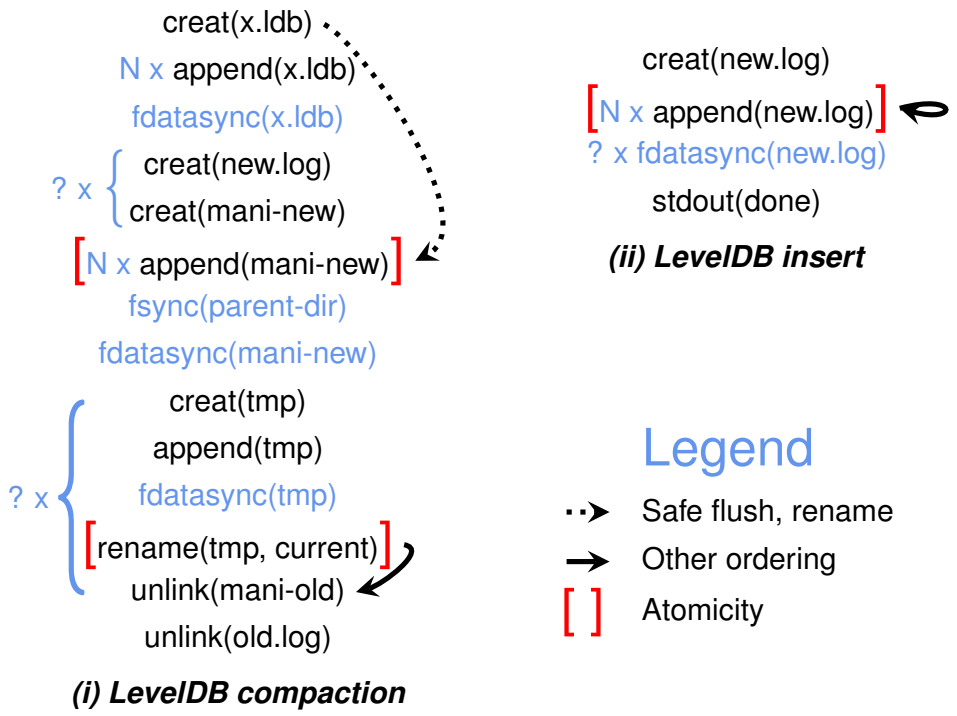


Figure 4.1: **LevelDB Protocol Diagram.** *The diagram shows the modularized update protocol for LevelDB-1.15. Uninteresting parts of the protocol and a few vulnerabilities (similar to those already shown) are omitted. Repeated operations in the protocol are shown as 'N ×' next to the operation, and portions of the protocol executed conditionally are shown as '? ×'. Blue-colored text simply highlights such annotations and sync calls. Ordering and durability dependencies are indicated with arrows; durability dependency arrows end in an* stdout *micro-op. Dotted arrows correspond to* safe file flush *vulnerabilities. Operations inside brackets must be persisted together atomically. Vulnerabilities shown are based on the default APM of* ALICE.

separate checker for *dirstate* and *repository* corruption in Mercurial, and separate checkers for durability and the other consistency requirements (atomicity and ordering) for LevelDB, HSQLDB, and GDBM.

Summary. Across the 11 applications, our workloads test a total of 34

configuration options that tweak the update protocol and change application guarantees. Our checkers are conceptually simple: they do read operations to verify workload invariants for that particular configuration, and then try writes to the datastore. However, some applications have complex invariants, and recovery procedures that they expect users to carry out (such as removing a leftover lock file). Our checkers are hence complex (e.g., about 500 LOC for Git), invoking all recovery procedures we are aware of that are expected of normal users. If application invariants for the tested configuration are explicitly and conspicuously documented, we consider violating them as failure; otherwise, our checkers consider violating a lay user's expectations as failure.

4.3 Per-Application Summary

We now discuss the logical update protocols of the applications examined. Figures 4.1–4.6 illustrate the update protocols for different applications, showing the logical operations in the protocol (organized as modules) and discovered vulnerabilities. The vulnerabilities shown are for the default APM in ALICE, and thus make the weakest assumptions about the crash behavior of the underlying file system.

Databases and Key-Value Stores

LevelDB's protocol, explained earlier, is designed to efficiently work with LSM trees [61] and allows multiple levels of compaction. The protocol is shown in Figure 4.1; ALICE finds more vulnerabilities in LevelDB compared to our manual study. For example, a crash can result in the appended portion of a log file containing garbage. LevelDB's recovery code does not properly handle this situation, and the user gets an error if trying to access the inserted key-value pair. The proper behavior for LevelDB, in this situation, is to simply discard they key-value pair and report to the

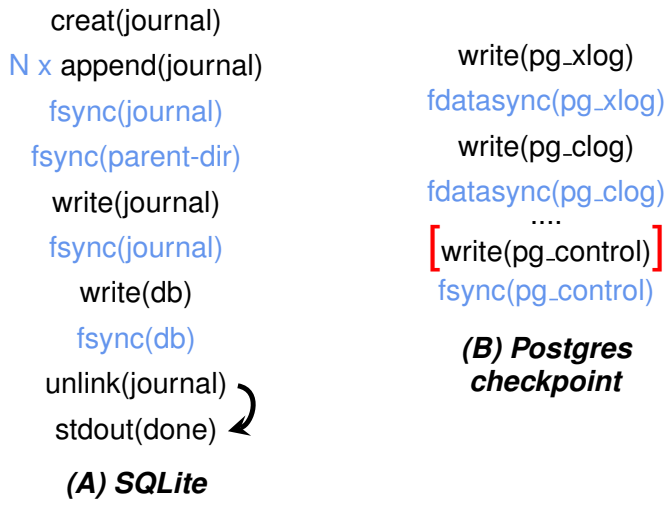


Figure 4.2: **SQLite and Postgres Protocol Diagrams.***The diagram shows the update protocol for SQLite under the Rollback journaling mode and for Postgres. Labelings have the same meaning as in the LevelDB protocol diagram. Vulnerabilities shown are based on the default APM of* ALICE.

user that the key was not found (since a crash occurred before the entire key-value pair was made durable).

Figure 4.2(A) shows the rollback configuration of SQLite which uses a simple version of *undo logging*. SQLite stores its entire database in a single file, shown as `db` in Figure 4.2(A). For undo logging, the existing contents of the database that are about to be modified are first copied onto a `journal` file, the database is then modified, and finally, the journal file is deleted. Another configuration of SQLite (SQLite-WAL) uses write-ahead logging: the new data is written to a `log` file first, and the database is then modified. Postgres uses a more complex version of write-ahead logging, briefly shown in Figure 4.2(B). The log file to which new data is written is called `pg_xlog`, while `pg_clog` and `pg_control` maintain the metadata required for logging (such as transaction identifiers) with yet another log;

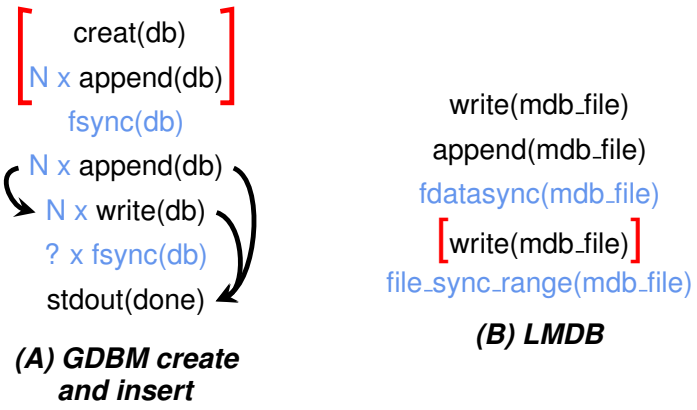


Figure 4.3: **GDBM and LMDB Protocol Diagrams.** *The diagram shows the update protocol for GDBM and LMDB. Labelings have the same meaning as in the LevelDB protocol diagram. Vulnerabilities shown are based on the default APM of* ALICE.

more details can be found in the Postgres documentation [69].

LMDB (a key-value store) stores all database information in a single file (in a tree structure) and uses shadow-paging (copy-on-write), as shown in Figure 4.3(B). Any *Put()* to the database is first written to unused portions of the database file (sometimes appended), and then a pointer in the beginning of the file is changed to indicate the newly written data; space use information is recorded along with the data write to the unused portion. ALICE found that LMDB requires the final pointer update (106 bytes) in the copy-on-write tree to be atomic. HSQLDB (Figure 4.4) uses a combination of write-ahead logging and update-via-rename, on the same files, to maintain consistency. The update-via-rename is performed by first separately unlinking the destination file, and then renaming; out-of-order persistence of `rename()`, `unlink()`, or log creation causes problems.

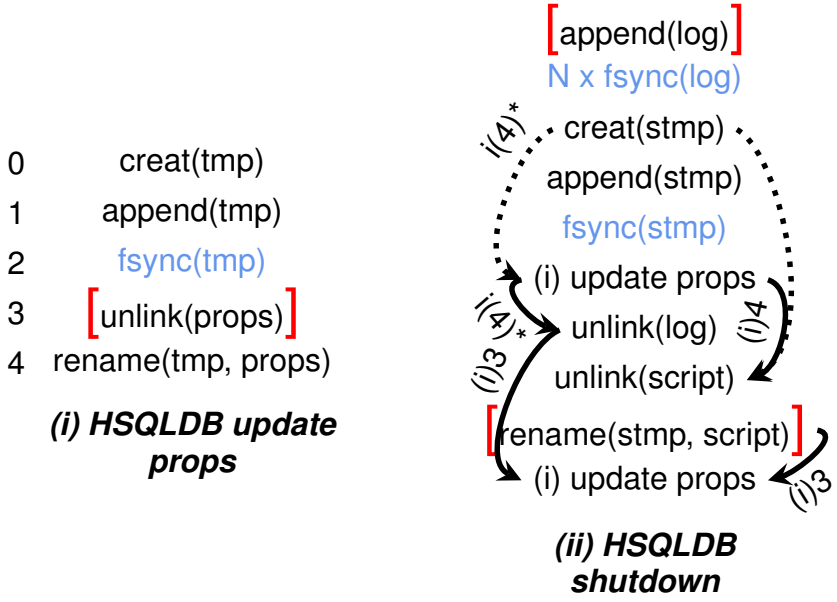


Figure 4.4: **HSQLDB Protocol Diagram.** *The diagram shows the modularized update protocol for HSQLDB. Dependencies between modules are indicated by the numbers on the arrows, corresponding to line numbers in modules. The two dependencies marked with * are also durability dependencies. Dotted arrows correspond to* safe rename *or* safe file flush *vulnerabilities. Other labelings have the same meaning as in the LevelDB protocol diagram. Vulnerabilities shown are based on the default APM of* ALICE.

Version Control Systems

Git and Mercurial maintain meta-information about their repository in the form of logs. The Git protocol is illustrated in Figure 4.5(A). Git stores information in the form of object files, which are never modified; they are created as temporary files, and then linked to their permanent file names. Git also maintains pointers in separate files, which point to both the meta-information log and the object files, and are updated using update-via-rename. Mercurial, in contrast, uses a journal to maintain consistency,

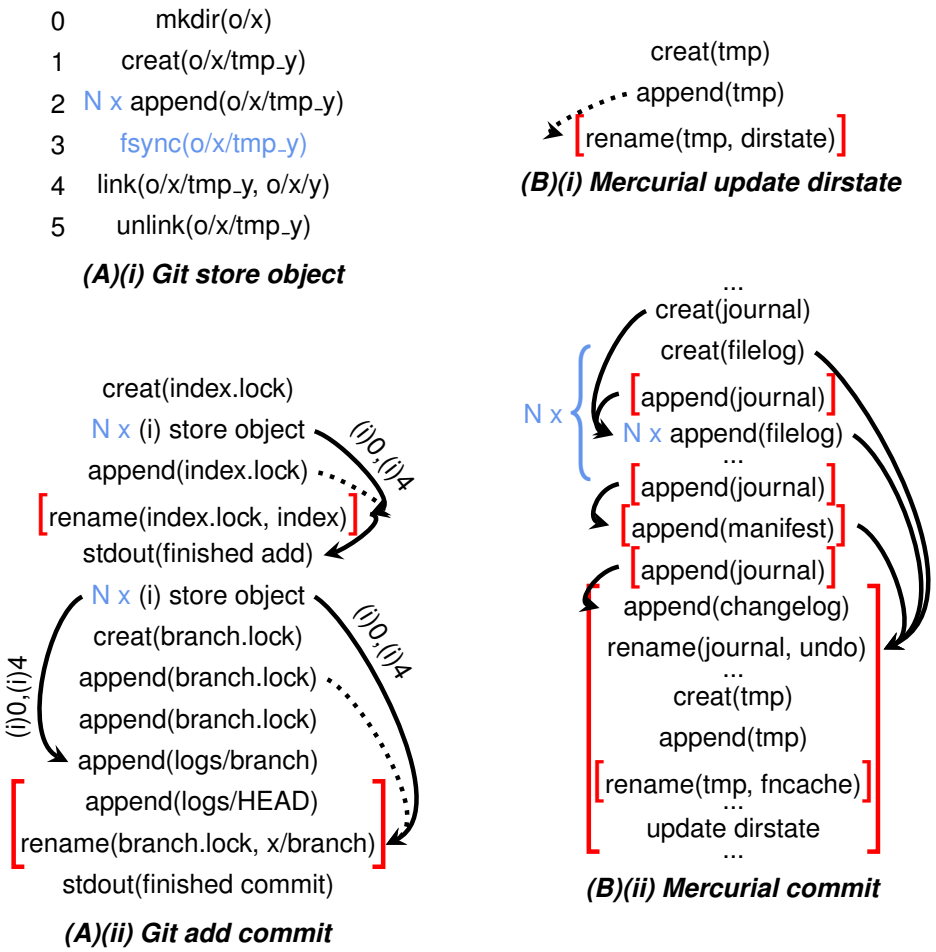


Figure 4.5: **Protocol Diagrams for Version Control Systems.** *The diagram shows the modularized update protocol of Git and Mercurial. Dependencies between modules are indicated by the numbers on the arrows, corresponding to line numbers in modules. Dotted arrows correspond to* safe rename *vulnerabilities. Other labelings have the same meaning as in the LevelDB protocol diagram. Vulnerabilities shown are based on the default APM of* ALICE.

using update-via-rename only for some unimportant information.

We find many ordering dependencies in the Git protocol, as shown in

Figure 4.5(A). This result is not surprising, since mailing-list discussions suggest Git developers expect total ordering from the file system. We also find a Git vulnerability involving atomicity across multiple system calls; a pointer file being updated (via an append) has to be persisted atomically with another file getting updated (via an update-via-rename). In Mercurial, we find many ordering vulnerabilities for the same reason, not being designed to tolerate out-of-order persistence.

Virtualization and Distributed Systems

Figure 4.6 shows the update protocol for VMWare Player, HDFS, and ZooKeeper. VMWare Player's protocol is simple. VMWare maintains a static, constant mapping between blocks in the virtual disk, and in the `VMDK` file (even for dynamically allocated `VMDK` files); directly overwriting the `VMDK` file maintains consistency (though VMWare does use update-via-rename for some small files). Both HDFS and ZooKeeper use write-ahead logging. We find that ZooKeeper does not explicitly persist directory entries of log files, which can lead to lost data. ZooKeeper also requires some log writes to be atomic.

4.4 Summary of Vulnerabilities Found

ALICE finds 60 static vulnerabilities in total across all applications; most vulnerabilities are shown in the protocol diagrams in the previous section, and all vulnerabilities are shown in Tables 4.2 and 4.3. The 60 static vulnerabilities correspond to 156 dynamic vulnerabilities; as explained in the previous chapter, ALICE uses stack trace information to coalesce the higher number of dynamic vulnerabilities (i.e., failures) to the fewer number of static vulnerabilities (i.e., errors). Dynamic vulnerabilities themselves are already coalesced from multiple failed crash states: applications failed in more than 4000 crash states. However, multiple failed crash states often

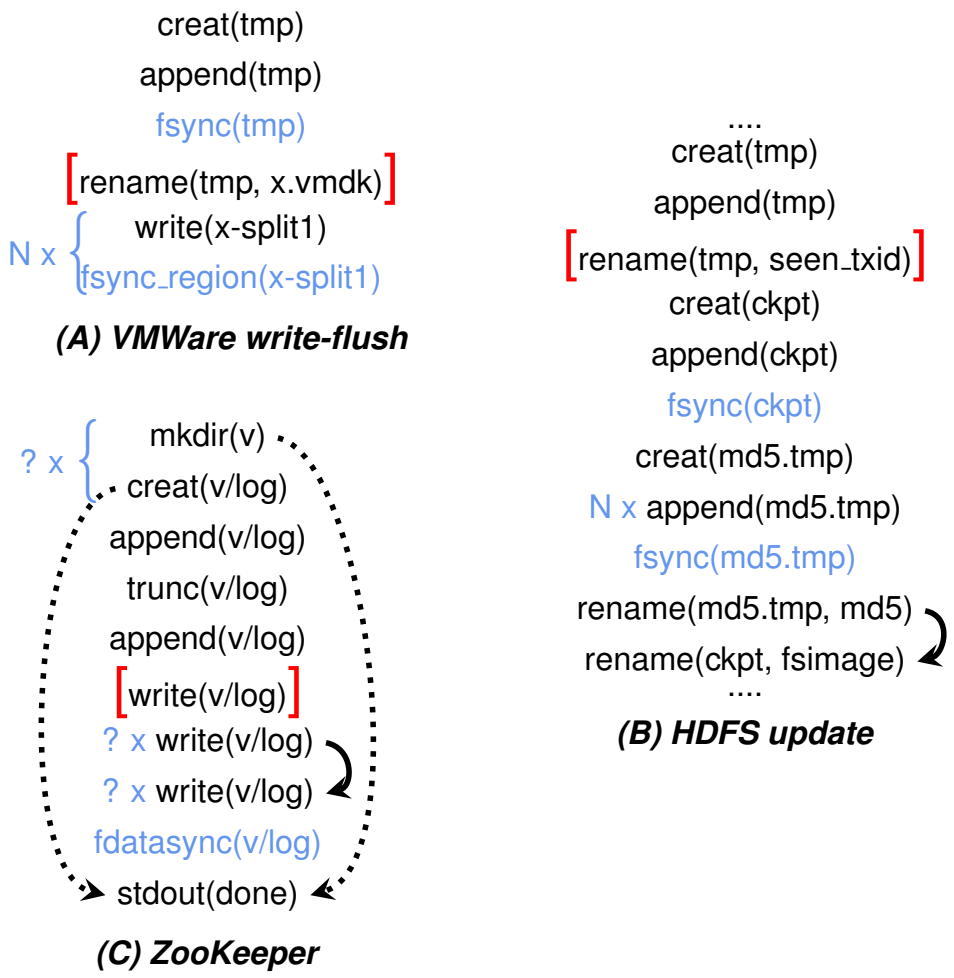


Figure 4.6: **Protocol Diagrams for Virtual Machines and Distributed Systems.** *The diagram shows the protocols for VMWare, HDFS, and ZooKeeper. Labelings have the same meaning as in the LevelDB protocol diagram. Vulnerabilities shown are based on the default APM of* ALICE.

correspond to the same system call, especially when testing for system-call atomicity in the default exploration strategy. Specifically, breaking a single system call in multiple ways might cause many of the corresponding crash

states to fail; however, they are considered a single dynamic vulnerability.

Application	Types									Unique static vulnerabilities
	Across-syscalls atomicity	Atomicity			Ordering			Durability		
		Appends and truncates	Single-block overwrites	Renames and unlinks	Safe file flush	Safe renames	Other	Safe file flush	Other	
Leveldb1.10	1‡	1		1	2	1	3	1		**10**
Leveldb1.15	1	1		1	1		2			**6**
LMDB			1							**1**
GDBM	1			1			1		2	**5**
HSQLDB		1		2	1		3	2	1	**10**
Sqlite-Roll									1	**1**
Sqlite-WAL										**0**
PostgreSQL			1							**1**
Git	1			1	2	1	3		1	**9**
Mercurial	2	1		1		1	4		2	**10**
VMWare				1						**1**
HDFS				1			1			**2**
ZooKeeper			1				1	2		**4**
Total	**6**	**4**	**3**	**9**	**6**	**3**	**18**	**5**	**7**	**60**

Table 4.2: **Vulnerabilities: File-System Behavior (default ALICE APM).** *The table shows the discovered static vulnerabilities categorized by the type of file-system behavior they are related to, using* ALICE*'s default APM. The number of unique vulnerabilities for an application can be different from the sum of the categorized vulnerabilities, since the same source code lines can exhibit different behavior.* ‡ *The atomicity vulnerability in Leveldb1.10 corresponds to multiple mmap() writes.*

Table 4.2 shows the vulnerabilities classified by the affected file-system

behavior, and 4.3 shows the vulnerabilities classified by failure consequence. Table 4.3 also separates out those vulnerabilities related only to user expectations and not to documented guarantees, with an asterik (*); many of these correspond to applications for which we could not find any documentation of guarantees.

Two different configurations considered in SQLite use different protocols, and the different versions of LevelDB differ on whether their protocols are designed around the `mmap()` interface. Tables 4.2 and 4.3 hence show these configurations of SQLite and LevelDB separately. All other configurations (in all applications) do not change the basic protocol, but vary on the application invariants; among different configurations of the same update protocol, all vulnerabilities are revealed in the *safest* configuration. Tables 4.2 and 4.3, and the rest of the paper only show vulnerabilities we find in the safest configuration, i.e., we do not count separately the same vulnerabilities from different configurations of the same protocol.

We find many vulnerabilities have severe consequences such as silent errors or data loss. Seven applications are affected by data loss, while two (both LevelDB versions and HSQLDB) are affected by silent errors. The *cannot open* failures include failure to start the server in HDFS and ZooKeeper, while the *failed reads and writes* include basic commands (e.g., `git-log`, `git-commit`) failing in Git and Mercurial. A few *cannot open* failures and *failed reads and writes* might be solvable by application experts, but we believe lay users would have difficulty recovering from such failures (our checkers invoke standard recovery techniques). We also checked if any discovered vulnerabilities are previously known, or considered inconsequential. The single PostgreSQL vulnerability is documented; it can be solved with non-standard (although simple) recovery techniques. The single LMDB vulnerability is discussed in a mailing list, though there is no available workaround. All these previously known vulnerabilities are separated out in Table 4.3 (†). The five *dirstate fail* vulnerabilities in

Mercurial are shown separately, since they are less harmful than other vulnerabilities (though frustrating to the lay user). Git's *fsck-only* and *reflog-only* errors are potentially dangerous, but do not affect normal usage.

We interacted with the developers of eight applications, reporting a subset of the vulnerabilities we found. Our interactions convince us that the vulnerabilities will affect users if they are exposed. The other applications (GDBM, Git, and Mercurial) were not designed to provide crash guarantees, although we believe their users will be affected by the vulnerabilities found should an untimely crash occur. Since the vulnerabilities will not surprise a developer of these applications, we did not report them. We also did not report documented vulnerabilities and those concerning partial renames (usually dismissed as they are not commonly exposed).

To our knowledge, application developers have acted on six of the vulnerabilities we find; one (LevelDB-1.10) is now fixed, another (LevelDB-1.15) was fixed parallel to our discovery, and three (HDFS, and two in ZooKeeper) are under consideration as of November 2016. The SQLite developers suggested that the discovered durability vulnerability (under `rollback` journaling) is not guaranteed by SQLite, but we opined that the documentation is misleading; the latest version of SQLite has an explicit configuration option that controls durability (although still switched off by default).

We have found that developers often dismiss other vulnerabilities which do not (or are widely believed to not) get exposed in current file systems, especially relating to out-of-order persistence of directory operations. The fact that only certain operating systems allow an `fsync()` on a directory is frequently referred to; both HDFS and ZooKeeper respondents lament that such an `fsync()` is not easily achievable with Java. An interesting fact is that the developers did consider a particular set of `fsync()` calls on directories important: of the six acted-on vulnerabilities, three relate to not explicitly issuing an `fsync()` on the parent directory

Application	Silent errors	Data loss	Cannot open	Failed reads and writes	Other
Leveldb1.10	1	1	5	4	
Leveldb1.15	2		2	2	
LMDB					read-only open†
GDBM		2*	3*		
HSQLDB	2	3	5		
Sqlite-Roll		1*			
Sqlite-WAL					
PostgreSQL			1†		
Git		1*	3*	5*	3$^{\#*}$
Mercurial		2*	1*	6*	5 dirstate fail*
VMWare			1*		
HDFS			2*		
ZooKeeper		2*	2*		
Total	**5**	**12**	**25**	**17**	**9**

Table 4.3: **Vulnerabilities: Failure Consequences.** *The table shows the number of static vulnerabilities resulting in each type of failure.* † *Previously known failures, documented or discussed in mailing lists.* * *Vulnerabilities relating to unclear documentation or typical user expectations beyond application guarantees.* $^{\#}$ *There are 2 fsck-only and 1 reflog-only errors in Git.*

after creating and calling `fsync()` on a file. However, not issuing such an `fsync()` is perhaps more safe in modern file systems than out-of-order persistence among different directory operations. We believe the developers' interest in issuing `fsync()` calls on parent directories arises from the

Linux documentation explicitly recommending this action.

Summary. ALICE detects 60 vulnerabilities in total, with 5 resulting in silent failures, 12 in loss of durability, 25 leading to inaccessible applications, and 17 returning errors while accessing certain data. ALICE is also able to detect previously known vulnerabilities.

4.5 Common Patterns

We now examine vulnerabilities related to different file-system behaviors, describing common patterns amongst them. Since durability vulnerabilities show a separate pattern, we consider them separately.

Atomicity across System Calls

Four applications (including both versions of LevelDB) require atomicity across system calls. The required atomicity are all shown in previous protocol diagrams: appends to `mani-new` and `new.log` in LevelDB (Figure 4.1), the creation and appends to `db` in GDBM (Figure 4.3(A)), an append and rename to different files in Git (Figure 4.5(A)) and a long sequence of different system calls in Mercurial (Figure 4.5(B)).

For three applications, the consequences seem minor: inaccessibility during database creation (of an empty database) in GDBM, dirstate corruption in Mercurial, and an erratic `reflog` in Git. LevelDB's vulnerability has a non-minor consequence, but was fixed immediately with LevelDB-1.15 (when LevelDB started using `read()-write()` instead of `mmap()`).

In general, we observe that this class of vulnerabilities seems to affect applications less than other classes. This result may arise because these vulnerabilities are easily tested: they are exposed independent of the file system (i.e, via process crashes), and are easier to reproduce. Alternatively, if such vulnerabilities have serious consequences, there is a high chance

that they affect many users (due to process crashes) and will hence be noticed and fixed by the developers.

Atomicity within System Calls

Append atomicity. Surprisingly, three applications require appends to be *content-atomic*, i.e., the appended portion should contain actual data. In the absence of content atomicity, if a crash happens after appending to a file (but before an `fsync()` or `fdatasync()` is called), the application might observe that the size of the file has increased after rebooting, but with the extended portion of the file containing garbage data. The failure consequences are severe, such as corrupted reads (HSQLDB), failed reads (LevelDB-1.15), and repository corruption (Mercurial). Filling the appended portion with zeros instead of garbage still causes failure; in ext4, only the current implementation of delayed allocation (where file size does not increase until actual content is persisted) works.

Applications might also be dependent on the *size-atomicity* of appends, i.e., when performing an append of size X, is the file size atomically extended by X? Size-atomicity can be thought of in different granularities; we investigate whether applications require the file size to be extended at a granularity of 4K blocks. However, most appends seemingly do not need to be size-atomic at even a 4K-block granularity: only Mercurial is affected, and the affected append also requires content-atomicity.

Overwrite atomicity. LMDB, PostgreSQL, and ZooKeeper require small writes (< 200 bytes) to be atomic. Both the LMDB and PostgreSQL vulnerabilities are previously known.

We do not find any multi-block overwrite vulnerabilities, and even single-block overwrite requirements are typically documented. This finding is in stark contrast with append atomicity; some of the difference can be attributed to the default APM (overwrites are content-atomic), and to some workloads simply not using overwrites. However, the major cause

seems to be the basic mechanism behind application update protocols: modifications are first logged, in some form, via appends; logged data is then used to overwrite the actual data. Applications have careful mechanisms to detect and repair failures in the actual data, but overlook the presence of garbage content in the log.

Directory operation atomicity. Since most file systems provide atomic directory operations, one would expect that most applications would be vulnerable to such operations not being atomic. However, we do not find this to be the case for certain classes of applications. Databases and key-value stores do not employ atomic renames extensively; consequently, we observe non-atomic renames affecting only three of these applications (GDBM, HSQLDB, LevelDB). Non-atomic unlinks seemingly affect only HSQLDB (which uses unlinks for logically performing renames), and we did not find any application affected by non-atomic truncates.

Ordering between System Calls

Applications are extremely vulnerable to system calls being persisted out of order; we find 27 vulnerabilities.

Safe renames. On file systems with delayed allocation, a common heuristic to prevent data loss is to persist all data (including appends and truncates) of a file before subsequent renames of the file [50]. We find that this heuristic only matches (and thus fixes) three discovered vulnerabilities, one each in Git, Mercurial, and LevelDB-1.10. A related heuristic, where if an existing file is opened with the `O_TRUNC` flag and modified, then the file is flushed to the disk immediately on a corresponding `close()` system call, does not affect any of the vulnerabilities we discovered. Also, the effect of the heuristics varies with minor details: if the safe-rename heuristic does not persist file truncates, only two vulnerabilities will be fixed; if the `O_TRUNC` heuristic also acts on new files, an additional vulnerability will be fixed.

Safe file flush. An `fsync()` on a file does not guarantee that the file's directory entry is also persisted. Most file systems, however, persist directory entries that the file is dependent on (e.g., directory entries of the file and its parent). We found that this behavior is required by three applications for maintaining basic consistency.

Durability

We find vulnerabilities in seven applications resulting in durability loss. Of these, only two (GDBM and Mercurial) are affected because an `fsync()` is not called on a file. Six applications require `fsync()` on directories: three are affected by *safe file flush* discussed previously, while four (HSQLDB, SQLite, Git, and Mercurial) require other `fsync()` calls on directories. As a special case, with HSQLDB, previously committed data is lost, rather than data that was being committed during the time of the workload. In all, only four out of the twelve vulnerabilities are exposed when full ordering is promised: many applications do issue an `fsync()` call before durability is essential, but do not `fsync()` all the required information.

Summary

We believe our study offers several insights for file-system designers. Future file systems should consider providing ordering between system calls, and atomicity within a system call in specific cases. Vulnerabilities involving atomicity of multiple system calls seem to have minor consequences. Requiring applications to separately flush the directory entry of a created and flushed file can often result in application failures. For durability, most applications explicitly flush some, but not all, of the required information; thus, providing ordering among system calls can also help durability.

Application	ext3-j	ext3-o	ext3-w	ext4-o	btrfs	default
Leveldb1.10	1	1	3	2	4	10
Leveldb1.15	1	1	2	2	3	6
LMDB						1
GDBM	2	3	3	3	4	5
HSQLDB					4	10
Sqlite-Roll	1	1	1	1	1	1
Sqlite-WAL						0
PostgreSQL						1
Git	2	2	2	2	5	9
Mercurial	3	3	4	6	8	10
VMWare						1
HDFS					1	2
ZooKeeper		1	1	1	1	4
Total	10	12	16	17	31	60

Table 4.4: **Vulnerabilities on Current File Systems.** *The table shows the number of vulnerabilities that occur on current file systems. The final column compares this against the default* ALICE *APM: all applications are vulnerable under future file systems.*

4.6 Impact on Current File Systems

Our study thus far has utilized an abstract (and weak) file system model (i.e., APM) in order to discover the broadest number of vulnerabilities. We now utilize the file-system APMs specified in Table 3.5 to understand how modern protocols would function atop a range of modern file systems and configurations. Thus, we focus on Linux ext3 (including writeback, ordered, and data-journaling mode), Linux ext4, and btrfs.

Table 4.4 shows the vulnerabilities reported by ALICE for each file system. Note that a significant number of vulnerabilities are exposed on *all*

examined file systems. However, some applications show no vulnerabilities on any of the considered real-world APMs; the flaws we found in these applications do not manifest on today's file systems (but may do so on future systems).

Ext3 with journaled data exposes the fewest vulnerabilities and hence is the safest of the considered file systems (however, it has low performance, as discussed in the next chapter). This is to be expected, since it provides the most intuitive and strongest crash behavior to applications: operations are never re-ordered, directory operations are always atomic, and data operations are atomic at 4K-granularity. The only vulnerabilities exposed are those requiring atomicity across system calls (all four mentioned in Section 4.5) and the four durability vulnerabilities that occur even when full ordering is promised (one each in GDBM, SQLite-Rollback, Git, and Mercurial). All exposed durability vulnerabilities are known to application developers: GDBM does not guarantee any consistency, the discussion with SQLite developers was previously mentioned, and Git and Mercurial do not issue *any* `fsync()` with their default options (though we configure them with stronger options).

Ext3 with ordered data behaves similar to ext3 with journaled data, except that it can re-order file overwrites. This relaxation results in two additional vulnerabilities being exposed: one in GDBM, and one in ZooKeeper. Ext3 with writeback data further relaxes crash behavior: it does not provide content atomicity during appends, and hence additionally exposes two vulnerabilities in LevelDB-1.10, one in LevelDB-1.15, and one in Mercurial.

Ext4 with ordered data is the default file system prevalent in current Linux systems, and has crash behavior similar to ext3 with ordered data. However, ext4 can re-order file appends. Such re-ordering causes ext4 to expose an additional vulnerability each in LevelDB-1.10 and LevelDB-1.15, and three more in Mercurial, when compared to ext3 with ordered data.

Btrfs is a modern file system with a significantly different CoW-based

design compared to the ext3 and ext4, and is being adopted currently in some Linux distributions (such as SUSE). It can re-order all operations, including directory operations, but implements the *safe renames* and *safe file flush* heuristics discussed previously. The extensive re-ordering results in 31 vulnerabilities being exposed.

Summary. Application vulnerabilities are exposed on many current file systems. The vulnerabilities exposed vary between file systems. Hence, testing applications on only a few file systems is not sufficient: multiple file systems (or ALICE's strategy of a hypothetical weak file system) should be considered when testing application correctness.

4.7 Discussion

We now consider why crash vulnerabilities occur commonly even among widely used applications. We find that application update protocols are complex and hard to isolate and understand. Many protocols are layered and spread over multiple files. Modules are also associated with other complex functionality (e.g., ensuring thread isolation). This complexity leads to issues that are obvious with a bird's eye view of the protocol: for example, HSQLDB's protocol has 3 consecutive `fsync()` calls to the same file (increasing latency). Logical representations of update protocols as in Figures 4.1–4.6 (obtained using ALICE) can help solve the problem.

Another factor contributing to crash vulnerabilities is poorly written, untested recovery code. In LevelDB, we find vulnerabilities that should be prevented by correct implementations of the documented update protocols. Some recovery code is non-optimal: potentially recoverable data is lost in several applications (e.g., HSQLDB, Git). Mercurial and LevelDB provide utilities to verify or recover application data; we find these utilities hard to configure and error-prone. For example, LevelDB's recovery command works as expected only when (seemingly) unrelated configuration

options (*paranoid checksums*) are set up properly, and sometimes ends up *further* corrupting the data-store. We believe these problems are a direct consequence of the recovery code being infrequently executed and insufficiently tested; it is imperative that developers use tools such as ALICE to extensively test recovery code.

Convincing developers about crash vulnerabilities is sometimes hard: there is a general mistrust surrounding such bug reports. Usually, developers are suspicious that the underlying storage stack might not respect `fsync()` calls [71], or that the drive might be corrupt. We hence believe that most vulnerabilities that occur in the wild are associated with an incorrect root cause, or go unreported.

Unclear documentation of application guarantees contributes to the confusion about crash vulnerabilities. During discussions with developers about durability vulnerabilities, we found that SQLite, which proclaims itself as fully ACID-complaint, does not provide durability (even optionally) with the default storage engine, though the documentation suggests it does. Similarly, GDBM's `GDBM_SYNC` flag *does not* ensure durability. When suspicious, users should employ tools such as ALICE to determine guarantees directly from the code, bypassing the problem of bad documentation.

4.8 Summary

In the real world, application-level crash consistency is dangerously dependent upon file-system behavior, and the correctness of applications varies even among different widely-used file systems. Among the 11 applications analyzed, we find 60 vulnerabilities, of which more than half can be exposed even under current file systems. Some of the vulnerabilities result in severe consequences like corruption or data loss. In the next chapter, we look at how the situation can be addressed.

5

C^{2}FS

The work described in our thesis so far, as well as other research elsewhere [109], show that widely used applications written by experienced developers (such as Google's LevelDB and Linus Torvalds's Git) have crash-consistency vulnerabilities. Correctly implementing crash-consistency protocols has proved to be difficult for a variety of reasons. First, as described in Chapter 2, the correctness inherently depends on the exact semantics of the system calls in the update protocol with respect to a system crash. Because file systems buffer writes in memory and send them to disk later, from the perspective of an application the effects of system calls can get re-ordered before they are persisted on disk.

Second, the recovery protocol must correctly consider and recover from the multitude of states that are possible when a crash happens during the update protocol. Application developers strive for update protocols to be efficient, since the protocols are invoked during each modification to the data store; more efficient update protocols often result in more possible states to be reasoned about during recovery.

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

across applications.

In this chapter, we argue that it is practical to construct a file system that automatically improves application crash consistency. We base our arguments on the following hypotheses:

The Ordering Hypothesis: Existing update and recovery protocols (mostly) work correctly on an ordered and weakly-atomic file system (the exact definition of these terms is explained subsequently).

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

The initial section of this chapter briefly explains the ordering hypothesis, providing a summary of the related results from the previous chapter. It also describes the challenges involved in constructing an efficient file system that satisfies the ordering hypothesis and our reasoning behind the efficiency hypothesis. The rest of this chapter describes the design and implementation of a file system, c^2fs, that we believe satisfies both hypotheses, and an evaluation that validates our belief.

5.1 The Ordering Hypothesis

We hypothesize that most vulnerabilities that exist in application-level update protocols depend on two specific file-system guarantees. File systems that provide these guarantees, therefore, automatically mask application vulnerabilities. The first guarantee, and the major focus of our work, is that the effect of system calls should be persisted on disk in the order they were issued by applications; a system crash should not produce a state where the system calls appear re-ordered. The second (minor) guarantee is that, when an application issues certain types of system calls, the effect of the system call should be atomic across a system crash. Such *weak atomicity* is specifically required for system calls that

	Time (s)	Seeks	Median seek distance (sectors)
Re-ordered	25.82	23762	120
FIFO	192.56	38201	2002112

Table 5.1: **Seeks and Order.** *The table shows the number of disk seeks incurred and the total time taken when 25600 writes are issued to random positions within a 2GB file with a HDD. Two different settings are investigated: the writes can be re-ordered or the order of writes is maintained using the FIFO strategy. The number of seeks incurred in each setting and the LBA seek distance shown are determined from a block-level I/O trace. We use a Intel® Core™ 2 Quad Processor Q9300 machine with 4 GB of memory running Linux 3.13, and a Toshiba MK1665GSX 160 GB HDD.*

perform directory operations, such as file creation and file deletion. Weak atomicity also includes stipulations about writes to files, but only at a sector granularity (i.e., there is generally no need to guarantee that arbitrarily large writes are atomic). This stipulation affects both writing to existing portions of a file (i.e., overwriting) and appending data to the end of a file. However, appending data to the end of a file includes an additional condition: both increasing the file size and the writing of data to the newly appended portion of the file should be atomic together (but the append can be broken down into chunks at sector boundaries).

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

For quantitative evidence, consider the 60 vulnerabilities discussed in the previous chapter: 16 are tolerated by maintaining weak atomicity alone,

while 27 are tolerated by guaranteeing order. Of the remaining, 12 are durability vulnerabilities; however, 8 of these 12 will be masked if the file system guarantees order, as described in the previous chapter (Section 4.5). Thus, in all, 50 of the 60 vulnerabilities are addressed by maintaining order and weak atomicity; the remaining 10 have minor consequences and are readily tolerated or have already been fixed.

5.2 Order: Bad for Performance

Most real-world deployed file systems (such as btrfs) already maintain the weak atomicity required to mask application-level crash-consistency vulnerabilities. However, all commonly deployed file-system configurations (including ext4 in metadata-journaling mode, btrfs, and xfs) re-order updates, and the re-ordering only seems to increase with each new version of a file system (e.g., ext4 re-orders more than ext3 [65]; newer versions of ext4 re-order even more [102], as do newer systems like btrfs [65]). While maintaining update order is important for application crash consistency, it has traditionally been considered bad for performance, as we now discuss.

At low levels in the storage stack, re-ordering is a fundamental technique that improves performance. To make this case concrete, we created a workload that issues writes to random locations over a disk. Forcing these writes to commit in issue order takes roughly eight times longer than a seek-optimized order (Table 5.1). Approaches that constrict write ordering are insufficient for both hard drives [76] and SSDs [39].

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

```
write(f1);
write(f2);
fsync(f2);
truncate(f1);
```

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

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

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

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

5.3 Order with Good Performance

We believe it is possible to address the overheads associated with maintaining order in practice. To reduce disk-level scheduling overheads, a variety of techniques have been developed that preserve the *appearance* of ordered updates in the event of a crash while forcing few constraints on disk scheduling. For example, in ext4 data journaling, all file-system updates (metadata and data) are first written to a journal. Once committed there, the writes can be propagated (checkpointed) to their in-place final locations. Note that there are no ordering constraints placed upon the checkpoint writes; they can be re-ordered as necessary by lower layers in the storage stack to realize the benefits of low-level I/O scheduling. Further, by grouping all writes into a single, large transaction, writes are effectively committed in program order: if a write to f_1 occurs before a write to f_2, they will either be committed together (in the same transaction), or the write to f_2 will commit later; never will f_2 commit before f_1.

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

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

Thus, we are left with two open questions. Can a metadata-only journaling approach be adopted that maintains order but with high performance? Second, can a new abstraction eliminate false dependence? We answer

these questions in the affirmative with the design of c^{2}fs.

5.4 Crash-Consistent File System

In this section, we describe c^{2}fs, a file system that embraces application-level crash consistency. C^{2}fs has two goals: preserving the program order of updates and weak atomicity, and performance similar to widely-used re-ordering file systems. So as to satisfy these goals, we derive c^{2}fs from the ext4 file system. Ext4 is widely used, includes many optimizations that allow it to perform efficiently in real deployments, and includes a journaling mechanism for internal file-system consistency. In c^{2}fs, we extend ext4's journaling to preserve the required order and atomicity in an efficient manner without affecting most optimizations already present in ext4.

The key idea in c^{2}fs is to separate each application into a *stream*, and maintain order only within each stream; writes from different streams are re-ordered for performance. This idea has two challenges: metadata structures and the journaling mechanism need to be separated between streams, and order needs to be maintained within each stream efficiently. C^{2}fs should solve both without affecting existing file-system optimizations. In this section, we first explain the streams abstraction, how streams are separated, and how order is maintained within a stream.

Streams

C^{2}fs introduces a new abstraction called the *stream*; each application usually corresponds to a single stream. Writes from different streams are re-ordered for performance, while order is preserved within streams for crash consistency. We define the stream abstraction such that it can be easily used in common workflows; as an example, consider a text file f1 that is modified by a text editor while a binary file f2 is downloaded from

the network, and they are both later added to a VCS repository. Initially, the text editor and the downloader must be able to operate on their own streams (say, A and B, respectively), associating f1 with A and f2 with B. Note that there can be no constraints on the location of f1 and f2: the user might place them on the same directory. Moreover, the VCS should then be able to operate on another stream C, using C for modifying both f1 and f2. In such a scenario, the stream abstraction should guarantee the order required for crash consistency, while allowing enough re-ordering for the best performance possible.

Hence, in c^{2}fs, streams are transient and are not uniquely associated with specific files or directories: a file that is modified in one stream might be later modified in another stream. If two streams perform operations that affect logically related data (such as the same offsets of a file), the file system takes sufficient care so that the temporal order between those operations is also maintained. We loosely define the term *related* such that related operations do not commonly occur in separate streams within a short period of time; if they do, the file system might perform inefficiently. For example, separate directory entries in a directory are not considered related (since it is usual for two applications to create files in the same directory), but the creation of a file or directory is considered related to the creation of its parent. Overall, the abstraction is flexible: while we expect most applications to use a single stream, if needed, applications can also use separate streams for individual tasks.

The stream interface allows all processes and threads belonging to an application to easily share a single stream, but also allows a single thread to switch between different streams if necessary. Specifically, we provide a `setstream(`*`s`*`)` call that creates (if not already existing) and associates the current thread with the stream *s*. All future updates in that thread will be assigned to stream *s*; when forking (a process or thread), a child will adopt the stream of its parent. By default, the *init* process is assigned

an *init* stream. We expect most applications (whose write performance are user visible) to issue a single `setstream()` call in the beginning of the application, but to not make any other changes to their code. The API is further explained in Section 5.5.

Separating Multiple Streams

In c^{2}fs, the basic idea used to separately preserve the order of each stream is simple: c^{2}fs extends the journaling technique to maintain multiple in-memory running transactions, one corresponding to each stream. Whenever a synchronization system call (such as `fsync()`) is issued, only the corresponding stream's running transaction is committed. All modifications in a particular stream are associated with that stream's running transaction, thus maintaining order within the stream (optimizations regarding this are discussed in the next section).

Using multiple running transactions poses a challenge: committing one transaction without committing others (i.e., re-ordering between streams) inherently re-orders the metadata modified across streams. However, internal file-system consistency relies on maintaining a global order between metadata operations; indeed, this is the original purpose of ext4's journaling mechanism. It is hence important that metadata modifications in different streams be logically independent and be separately associated with their running transactions. We now describe the various techniques that c^{2}fs uses to address this challenge while retaining the existing optimizations in ext4.

Hybrid-granularity Journaling

Ext4's journaling mechanism (described in Chapter 2) works at block-granularity: entire blocks are associated with running transactions, and committing a transaction records the modified contents of entire blocks.

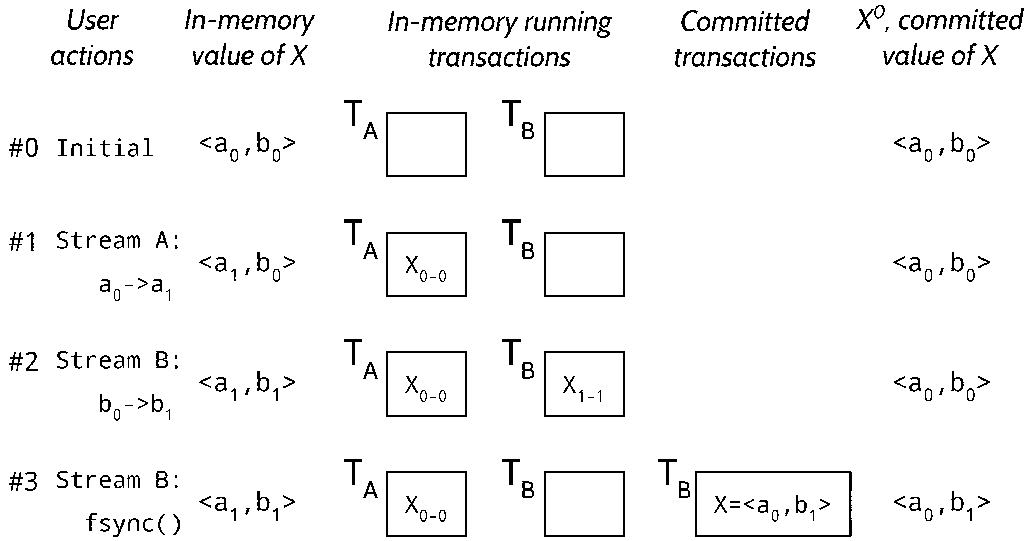


Figure 5.1: **Hybrid-granularity Journaling.** *Timeline showing hybrid-granularity journaling in c²fs. Block* X *initially contains the value* $\langle a_0, b_0 \rangle$, T_A *and* T_B *are the running transactions of streams* A *and* B*; when* B *commits,* X *is recorded at the block level on disk.*

C²fs uses *hybrid-granularity journaling*, where byte-ranges (instead of entire blocks) are associated with the running transaction, but transactional commits and checkpointing still happen at block-granularity.

C²fs requires byte-granularity journaling because separate metadata structures modified by different streams might exist in the same file-system block. For example, a single block can contain the inode structure for two files used by different applications; in block-granularity journaling, it is not possible to associate the inodes with the separate running transactions of two different streams.

Block-granularity journaling allows many optimizations that are not easily retained in byte-granularity. A major optimization affected in ext4 is data coalescing during checkpoints: even if multiple versions of a block are committed, only the final version is sent to its in-place location. Since

the buffer cache and storage devices manage data at block granularity, such coalescing becomes complicated with a byte-granularity journal.

To understand hybrid-granularity journaling, consider the example illustrated in Figure 5.1. In this example, block X initially contains the bytes $\langle a_0 b_0 \rangle$. Before allowing any writes, c^{2}fs makes an in-memory copy (say, X^0) of the initial version of the block. Let the first byte of X be modified by stream A into a_1; c^{2}fs will associate the byte range X_{0-0} with the running transaction T_A of stream A (X_{i-j} denotes the i^{th} to j^{th} bytes of block X), thus following byte-granularity. Let stream B then modify the second byte into b_1, associating X_{1-1} with T_B; the final in-memory state of X will be $\langle a_1 b_1 \rangle$. Now, assume the user calls `fsync()` in stream B, causing T_B to commit (T_A is still running). C^{2}fs converts T_B into block-granularity for the commit, by super-imposing the contents of T_B (i.e., X_{1-1} with the content b_1) on the initial versions of their blocks (i.e, X^0 with content $\langle a_0 b_0 \rangle$), and committing the result (i.e., $\langle a_0 b_1 \rangle$). When T_B starts committing, it updates X^0 with the value of X that it is committing. If the user then calls `fsync()` in A, X_{0-0} is super-imposed on X^0 ($\langle a_0 b_1 \rangle$), committing $\langle a_1 b_1 \rangle$.

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

Delta Journaling

In addition to simply associating byte ranges with running transactions, c^{2}fs allows associating the exact changes performed on a specific byte range (i.e., the *deltas*). This technique, which we call *delta journaling*, is required when metadata structures are actually shared between different streams (as opposed to independent structures sharing the same block). For example, consider a metadata tracking the total free space in the file system: all streams need to update this metadata.

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

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

Pointer-less Data Structures

Metadata in file systems often use data structures such as linked lists and trees that contain internal pointers, and these cause metadata operations in one stream to update pointers in structures already associated with another stream. For example, deleting an entry in a linked list will require updating the *next* pointer of the previous entry, which might be associated with another stream. C^{2}fs eliminates the need to update pointers across streams by adopting alternative data structures for such metadata.

Ext4 has two metadata structures that are of concern: directories and the *orphan list*. Directories in ext4 have a structure similar to linked lists, where each entry contains the relative byte-offset for the next entry. Usually, the relative offset recorded in a directory entry is simply the size of the entry. However, to delete a directory entry d_i, ext4 adds the size of d_i to the offset in the previous entry (d_{i-1}), thus making the previous entry

point to the next entry (d_{i+1}) in the list. To make directories pointer-less, c^{2}fs replaces the offset in each entry with a *deleted* bit; deleting an entry sets the bit, and the insert and scan procedures are modified appropriately.

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

Order-less Space Reuse

C^{2}fs carefully manages the allocation of space in the file system such that re-ordering deallocations between streams does not affect file-system consistency. For example, assume stream A deletes a file and frees its inode, and stream B tries to create a file. The allocation routines in ext4 might allocate to B the inode that was just freed by A. However, if B commits before A, and then a crash occurs, the recovered state of the file system will contain two unrelated files assigned the same inode.

Ext4 already handles the situation for block allocation (for reasons of security) by reusing blocks only after the transaction that frees those blocks has fully committed. In c^{2}fs, we extend this solution to both inode and directory-entry reuse. Thus, in our example, B will reuse A's freed inode only if A has already been committed.

Maintaining Order Within Streams

We saw in the previous section how to separate dependencies across independent streams; we now focus on ordering the updates within the same stream. Ext4 uses metadata-only journaling: ext4 can re-order file appends and overwrites. Data journaling, i.e., journaling all updates, preserves application order for both metadata and file data, but significantly reduces

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

Figure 5.2: **Order-preserving Delayed Allocation.** *Timeline of allocations performed, corresponding to a system-call sequence.*

performance because it often writes data twice. A hybrid approach, selective data journaling (SDJ) [12], preserves order of both data and metadata by journaling only overwritten file data; it only journals the block pointers for file appends. Since modern workloads are mostly composed of appends, SDJ is significantly more efficient than journaling all updates.

We adopt the hybrid SDJ approach in c^{2}fs. However, the approach still incurs noticeable overhead compared to ext4's default journaling under practical workloads because it disables a significant optimization, *delayed allocation*. In our experiments, the createfiles benchmark results in 8795 ops/s on ext4 with delayed allocation on a HDD, and 7730 ops/s without (12% overhead).

Without delayed allocation, whenever an application appends to files, data blocks are allocated and block pointers are assigned to the files immediately, as shown in the second column of Figure 5.2. With delayed allocation (third column), the file system does not immediately allocate blocks; instead, allocations for multiple appends are delayed and done together. For order to be maintained within a stream, block pointers need to be assigned immediately (for example, with SDJ, only the order of allocations is preserved across system crashes): naive delayed allocation inherently violates order.

C^{2}fs uses a technique that we call *order-preserving delayed allocation* to

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

However, the running transaction's size threshold poses a challenge: at commit time, what if we cannot add all batched allocations to T_i? C^{2}fs solves this challenge by reserving the space required for allocations when the application issues the appends. Order-preserving delayed allocation thus helps c^{2}fs achieve ext4's performance while maintaining order. For the createfiles benchmark, the technique achieves 8717 ops/s in c^{2}fs.

5.5 Implementation

We describe our implementation of c^{2}fs in this section. C^{2}fs changes 4,500 lines of source code (ext4 total: 50,000 lines). Of these, preserving order required only 500 lines while implementing multiple streams was more complicated and involved the rest of the changes. Overall, most of the changes (3,000 lines) are related to the journaling code within ext4.

Stream API. The `setstream()` call takes a *flags* parameter along with the stream. One flag is currently supported: `IGNORE_FSYNC` (ignore any `fsync()` calls in this stream). We provide a `getstream()` call that is used, for example, to find if the current process is operating on the *init* stream or a more specific stream. A `streamsync()` call flushes all updates in the current stream.

Separating Multiple Streams. Our prototype implementation of streams

incurs CPU overhead, because the journaling code manages each byte range separately, and we do not use optimized in-memory data structures. We implemented two optimizations to reduce overhead. First, contiguous byte ranges in the same stream are merged and tracked together under common scenarios. Second, we maintain an (in-memory) pointer from the metadata structure stored in each byte-range to the corresponding byte-range in running transactions.

We applied the pointer-less data structures technique only for ext4's linear directories and the inode orphan list; this proved sufficient for our purposes. The technique can also be adopted for ext4's hashed directories and extended attributes if the related features of the file system are found necessary. Also, our implementation currently limits the maximum size of directories: a fixed number of blocks are allocated to each directory during its creation, and inserted entries are contained within those blocks. This limitation can be removed by applying the pointer-less technique to the extent map of directories.

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

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

new version must be journaled in T_{i+1}. C^{2}fs handles these cases correctly.

5.6 Evaluation

This section answers the following questions. The first question validates the ordering hypothesis, and the rest of the questions evaluate the efficiency hypothesis:

- Does c^{2}fs improve application crash consistency?
- Does c^{2}fs effectively use streams to eliminate the overhead of write dependencies?
- How does c^{2}fs perform in standard file system benchmarks run in a single stream?
- What is the performance effect of maintaining order on real application workloads?

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

Reliability

Improvements in application crash consistency under an ordered, weakly atomic file system can be understood from the study in the previous chapter. We now re-examine the results, by using ALICE to compare ext4 (APM used in the previous chapter) and c^{2}fs (APM with system calls weakly atomic and in-order) with five applications that exhibit crash inconsistencies on ext4: LevelDB, SQLite, Git, Mercurial, and ZooKeeper. Different from the previous chapter, we use newer versions of the applications that

Application	ext4	c²fs
LevelDB	1	0
SQLite-Roll	0	0
Git	2	0
Mercurial	5	2
ZooKeeper	1	0

(a) Vulnerabilities found

Application		ext4	c²fs
LevelDB	Images	158 / 465	427 / 427
	Time (s)	24.31 / 30	30 / 30
Git	Images	84 / 112	96 / 96
	Time (s)	9.95 / 40	40 / 40

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

Table 5.2: **Consistency Testing.** *The first table shows the results of model-based testing using Alice, and the second shows experimental testing with BoB. Each vulnerability reported in the first table is a location in the application source code that has to be fixed. The* Images *rows of the second table show the number of disk images reproduced by the BoB tool that the application correctly recovers from; the* Time *rows show the time window during which the application can recover correctly from a crash (*x */* y*: *x *time window,* y *total workload runtime). For Git, we consider the default configuration, unlike the previous chapter that considers a safer configuration, since the safer configuration results in bad performance (§5.6).*

have fixed some of the vulnerabilities discovered in the previous chapter. Furthermore, we do not check durability in Git and Mercurial, since they never call `fsync()` under the configurations we test here.

The results of our testing are shown in Table 5.2(a). Ext4 results in multiple inconsistencies: LevelDB fails to maintain the order in which

key-value pairs are inserted, Git and Mercurial can result in repository corruption, and ZooKeeper may become unavailable. With c^{2}fs, the only inconsistencies were with Mercurial. These inconsistencies are exposed on a process crash with any file system, and therefore also occur during system crashes in c^{2}fs; they result only in *dirstate corruption*, which can be manually recovered from and is considered to be of minor consequence [55]. Thus, our model-based testing reveals that applications are significantly more crash consistent on c^{2}fs than ext4.

We used the BoB tool [65] to test whether our implementation of c^{2}fs maintains weak atomicity and ordering, i.e., whether the implementation reflects the model used in the previous testing. BoB records the block-level trace for an application workload running on a file system, and reproduces a subset of disk images possible if a crash occurs. BoB generates disk images by persisting blocks in and out of order; each image corresponds to a time window during the runtime where a crash will result in the image. These windows are used to measure how much time the application remains consistent.

We used Git and LevelDB to test our implementation; both these applications have crash vulnerabilities that are easily exposed on a re-ordering file system. We ran the applications atop both c^{2}fs and ext4 with BoB. Note that, our workloads for Git and LevelDB (same as in the previous chapter) finish quickly (within a second) and also dirty only a small amount of data (less than 4 MB), but the dirty data remains buffered in memory when the workload finishes. For BoB to correctly work, it needs to observe all the dirtied data getting flushed to the disk; furthermore, for BoB to correctly calculate the consistency time windows, the dirtied data should be naturally flushed to disk by the background buffer-cache flushing daemon. To illustrate, consider a workload that dirties two files, first A and then B, and requires them to be sent to disk in-order to maintain consistency. If B is flushed immediately by the application before exiting, but A remains in

the buffer cache and is flushed by the background daemon only 30 seconds later, the application had a inconsistent window of 30 seconds.

Hence, for our measurements with BoB to be correct, we let our Git and LevelDB workloads sleep for a time period upon completing the workload, until we are certain that all dirtied data have been flushed from the buffer cache (30 seconds for LevelDB, 40 for Git). Table 5.2(b) shows our results.

With ext4, both applications can easily result in inconsistency if a system crash happens during the workload. For example, LevelDB on ext4 is consistent only on 158 of the 465 images reproduced; a system crash can result in being unable to open the datastore after reboot, or violate the order in which users inserted key-value pairs. Similarly, with Git on ext4, Git will not recover properly if a crash happens during 30.05 seconds of the 40 second runtime of the workload. Note that the time windows shown in Table 5.2(b) depend mainly on the frequency of the background flushing daemon, and hence are similar for both our HDD and SSD (Table 5.2 reports numbers where BoB runs with the HDD).

However, with c^{2}fs, we were unable to reproduce any disk state in which LevelDB or Git are inconsistent. We hence conclude that our implementation of c^{2}fs provides the desired properties for maintaining application consistency. To summarize both our ALICE and BoB results, c^{2}fs noticeably improves the state of application crash consistency. We next evaluate whether this is achieved with good performance.

Multi-stream Benefits

Maintaining order causes write dependence during `fsync()` calls and imposes additional overheads, since each `fsync()` call must flush all previous dirty data. In the simplest case, this results in additional `fsync()` latency; it can also prevent writes from being coalesced across `fsync()` calls when data is overwritten, and prevent writes from being entirely avoided when

the previously written data is deleted. We now evaluate if using separate streams in c^{2}fs prevents these overheads.

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

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

We ran the benchmarks on three file-system configurations: ext4, which re-orders writes and does not incur additional overheads, c^{2}fs using a single stream (c^{2}fs-1), and c^{2}fs with modifications of A and B in separate streams (c^{2}fs-2). Table 5.3 and Figure 5.3 show our results.

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

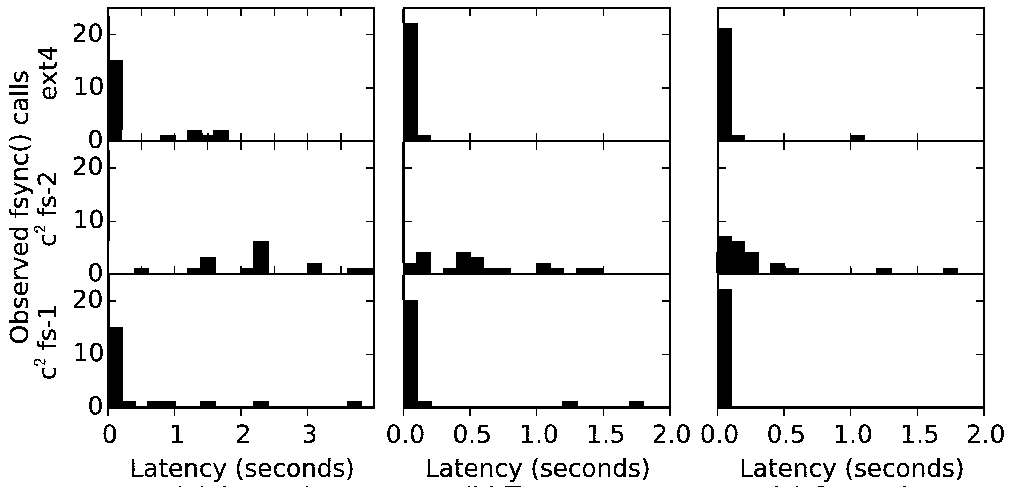
With truncates, ext4 and c^{2}fs-2 never send file A's data to disk, but

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

Table 5.3: **Single-`fsync()` Experiments.** *`fsync()` latencies in the first column correspond to the data written by the `fsync()` shown in the second column, while the total data shown in the third column affects the available device bandwidth and hence performance in more realistic workloads.*

c^{2}fs-1 sends the 100 MB during `fsync()`, resulting in higher latency and more disk writes. Most repeated `fsync()` calls in ext4 and c^{2}fs-2 are fast, as expected; they are slow in c^{2}fs-1, but still quicker than the append benchmark because the background thread would have just truncated *A* before some of the `fsync()`.

With overwrites, in both ext4 and c^{2}fs-2, only the final version of *A*'s data reaches the disk: in c^{2}fs-2, SDJ considers the second modification of *A* an append because the first version of *A* is not yet on disk (this still maintains order). In c^{2}fs-1, the first version is written during the `fsync()`, and then the second version (overwrite) is both written to the journal and propagated to its actual location, resulting in 300 MB of total disk writes. Repeated `fsync()` calls are slow in c^{2}fs-1 but quicker than previous benchmarks because of fewer disk seeks: with this version of the benchmark, since *A* is constantly overwritten, data is only sent to the



(a) Append (b) Truncate (c) Overwrite

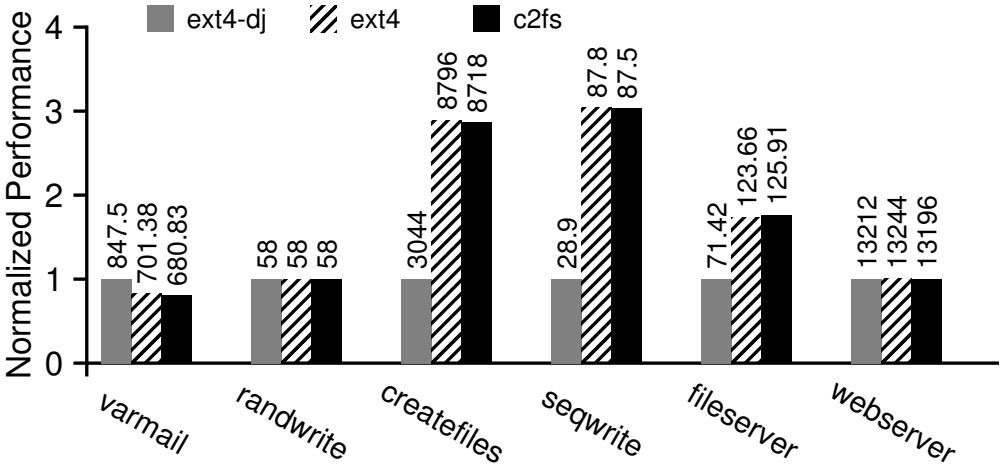
Figure 5.3: **Repeated `fsync()` Experiments.** *Histogram of user-observed foreground latencies in our multi-`fsync()` experiments. Each experiment is run for two minutes on a HDD.*

journal in c^{2}fs-1 and is never propagated to its actual location.

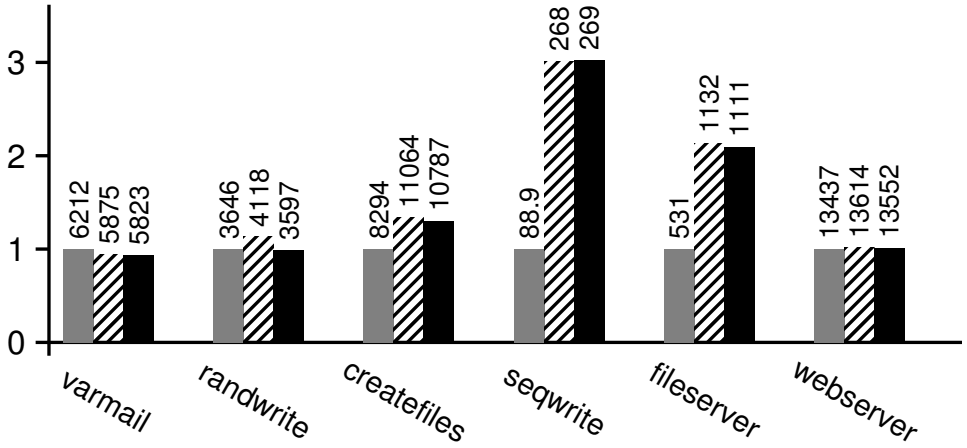
These results show that c^{2}fs is effective at avoiding write dependence overheads when multiple streams are used. The results also show that, within a stream, write dependence can cause noticeable overhead. For certain applications, therefore, it is possible that dividing the application into multiple streams is necessary for performance. The subsequent sections show that the majority of the applications do not require such division.

Single-stream Overheads

The previous experiments show how c^{2}fs avoids the performance overheads across streams; we now focus on performance within a stream. The performance effects of maintaining order within a stream are affected by false dependencies between updates within the stream, and hence depend significantly on the pattern of writes. We perform our evaluation using the Filebench [24, 95] suite that reflects real-world workload patterns and



(a) HDD Performance



(b) SSD Performance

Figure 5.4: **Single-stream Overheads: Performance.** *Throughput under standard benchmarks for c²fs, ext4, and ext4 under the data=journal mode (ext4-dj), all normalized to ext4-dj. Varmail emulates a multithreaded mail server, performing file creates, appends, deletes, reads, and* `fsync()` *in a single directory. Randwrite does 200K random writes over a 10 GB file with an* `fsync()` *every 100 writes. Webserver emulates a multithreaded web server performing open-read-close on multiple files and a log file append. Createfiles uses 64 threads to create 1M files. Seqwrite writes 32 GB to a new file (1 KB is considered an op in (c)). Fileserver emulates a file server, using 50 threads to perform creates, deletes, appends, and reads, on 80K files. The fileserver, varmail, and webserver workloads were run for 300 seconds. The numbers reported are the average over 10 runs.*

	Writes (KB/op)			CPU (µs/op)	
	ext4-dj	**ext4**	**c²fs**	**ext4**	**c²fs**
varmail	3.42	2.91	2.98	59.9	67.2
randwrite	8.1	4.1	8.1	16.8	24.4
createfiles	12.22	5.53	5.49	89.1	94.4
seqwrite	2.0	1.0	1.0	0.9	2.4
fileserver	1093	321	327	1040	2937
webserver	0.49	0.24	0.15	74.4	75.5

Table 5.4: **Single-stream Overheads: Data Written and CPU usage.** *The table shows the total writes and CPU usage with a HDD, corresponding to Figure 5.4(a).*

microbenchmarks, and compare performance between ext4 (false dependencies are not exposed) and c²fs (false dependencies are exposed because of ordering within streams). Another source of overhead within streams is the disk-level mechanism used to maintain order, i.e., the SDJ technique used in c²fs. Hence, we compare performance between ext4 (no order), c²fs (order-preserving delayed allocation and SDJ), and ext4 in the data=journal mode (*ext4-dj*, full data journaling). We compare performance both with a HDD (disk-level overheads dominated by seeks) and an SSD (seeks less pronounced).

The overall results are shown in Figure 5.4 and Table 5.4; performance is most impacted by overwrites and `fsync()` calls. We now explain the results obtained on each benchmark.

The *varmail* benchmark performs appends, deletes, and `fsync()` calls. Since each append is immediately followed by an `fsync()`, there is no additional write dependence due to ordering. Performance is dominated by seek latency induced by the frequent `fsync()` calls, resulting in similar performance across ext4 and c²fs. Ext4-dj issues more writes but incurs less seeks (since data is written to the journal rather than the in-place

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

Randwrite overwrites random locations in an existing file and calls `fsync()` every 100 writes. Since the `fsync()` calls always flush the entire file, there is no additional write dependence due to ordering. However, the overwrites cause both c^{2}fs (SDJ) and ext4-dj (full journaling) to write twice as much data as ext4. In the HDD, all file systems perform similarly since seeks dominate performance; in the SSD, additional writes cause a 12% performance decrease for c^{2}fs and ext4-dj.

Createfiles and *seqwrite* keep appending to files, while *fileserver* issues appends and deletes to multiple files; they do not perform any overwrites or issue any `fsync()` calls. Since only appends are involved, c^{2}fs writes the same amount of data as ext4. Under the HDD, similar performance is observed in c^{2}fs and in ext4. Under SSDs, createfiles is 4% slower atop c^{2}fs because of delayed allocation in the `T_LOCKED` state, which takes a noticeable amount of time (an average of 132 ms during each commit); this is an implementation artifact, and can be optimized. For all these benchmarks, ext4-dj writes data twice, and hence is significantly slower. *Webserver* involves mostly reads and a few appends; performance is dominated by reads, all file systems perform similarly.

Table 5.4 compares the CPU usage of c^{2}fs and ext4. For most workloads, our current implementation of c^{2}fs has moderately higher CPU usage; the significant usage for *fileserver* and *seqwrite* is because the workloads are dominated by block allocations and de-allocations, which is especially CPU intensive for our implementation. This can be improved by adopting more optimized structures and lookup tables (§5.5). Thus, while it does not noticeably impact performance in our experiments, reducing CPU usage is an important future goal for c^{2}fs.

Overall, our results show that maintaining order does not incur any inherent performance overhead for standard workloads when the workload

is run in one stream. False dependencies are rare and have little impact for common workloads, and the technique used to maintain order within streams in c^{2}fs is efficient.

Case Studies

Our evaluation in the previous sections shows the performance effects of maintaining order for standard benchmarks. We now consider three real-world applications: Git, LevelDB, and SQLite with rollback journaling; we focus on the effort required to maintain crash consistency with good performance for these applications in c^{2}fs and ext4. For ext4, we ensure that the applications remain consistent by either modifying the application to introduce additional `fsync()` calls or using safe application configuration options. All three applications are naturally consistent on c^{2}fs when run on a single stream.

Single Application Performance. We first ran each application in its own stream in the absence of other applications, to examine if running the application in one stream is sufficient for good performance (as opposed to dividing a single application into multiple streams). Specifically, we try to understand if the applications have false dependencies. We also consider their performance when `fsync()` calls are omitted without affecting consistency (including user-visible durability) on c^{2}fs.

The results are shown in Table 5.5. For Git, we use a workload that adds and commits the Linux source code to an empty repository. While Git is naturally consistent atop c^{2}fs, it requires a special option (*fsyncobjectfiles*) on ext4; this option causes Git to issue many `fsync()` calls. Irrespective of this option, Git always issues 242 MB of appends and no overwrites. In c^{2}fs, the 242 MB is sent directly to the device and the workload completes in 28.9 seconds. In ext4, the `fsync()` calls needed for correctness causes many disk flushes and 1.4 GB of journal commits, and the workload takes 2294 seconds (80$\times$ slower).

For SQLite, we insert 2000 rows of 120 bytes each into a table. SQLite issues `fsync()` calls frequently, and there are no false dependencies in c^{2}fs. However, SQLite issues file overwrites (31.83 MB during this workload), which causes data to be sent to the journal in c^{2}fs. Sending the overwritten data to the journal improves the performance of c^{2}fs in comparison to ext4 (1.28×). Because SQLite frequently issues an `fsync()` after overwriting a small amount (4 KB) of data, ext4 incurs a seek during each `fsync()` call, which c^{2}fs avoids by writing the data to the journal. SQLite can also be heavily optimized when running atop c^{2}fs by omitting unnecessary `fsync()` calls; with our workload, this results in a 685× improvement.

For LevelDB, we use the fillrandom benchmark from the db_bench tool to insert 250K key-value pairs of 1000 bytes each. Atop ext4, we needed to add additional `fsync()` calls to improve the crash consistency of LevelDB. LevelDB on c^{2}fs and the fixed version on ext4 have similar write avoidance, as can be seen from Table 5.5. Since LevelDB also does few file overwrites, it performs similarly on c^{2}fs and ext4. With c^{2}fs, existing `fsync()` calls in LevelDB can be omitted since c^{2}fs already guarantees ordering, increasing performance 5×.

Thus, the experiments suggest that false-dependency overheads are minimal within an application. In two of the applications, the ordering provided by c^{2}fs can be used to omit `fsync()` calls to improve performance.

Multiple Application Performance. We next test whether c^{2}fs is effective in separating streams: Figure 5.5 shows the performance when running Git and SQLite simultaneously. The situation in current real-world deployments is exemplified by the *ext4-bad* configuration in Figure 5.5: both applications are run on ext4, but Git runs without the *fsyncobjectfiles* option (i.e., consistency is sacrificed). The *c^{2}fs-2* configuration is the intended use case for c^{2}fs: Git and SQLite are in separate streams on c^{2}fs, achieving consistency while performing similar to *ext4-bad*. (SQLite performs better

		Throughput		User-level Metrics			Disk-level Metrics		
				fsync()	*Append (MB)*	*Overwrite(kB)*	*Flushes*	Data (MB)	
								Journal	*Total*
Git	ext4	17	*files/s*	38599	242	0	77198	1423	1887
	c²fs	1351		0	242	0	10	18	243
	c²fs+	1351		0	242	0	10	18	243
SQLite	ext4	5.23	*ops/s*	6000	31.56	31.83	12000	70	170
	c²fs	6.71		6000	31.56	31.83	12000	117	176
	c²fs+	4598		0	0.32	0	0	0	0
LevelDB	ext4	5.25	*MB/s*	598	1087	0.01	1196	16.3	1131
	c²fs	5.1		523	1087	0	1046	16.2	1062
	c²fs+	25.5		0	199	0	2	0.074	157

Table 5.5: **Case Study: Single Application Performance.** *The table shows the performance and observed metrics of Git, LevelDB, and SQLite-rollback run separately under different file-system configurations on HDD. C²fs+ denotes running c²fs with unnecessary* `fsync()` *calls omitted; in both c²fs configurations, the application runs in a single stream. The user-level metrics characterize each workload; "appends" and "overwrites" show how much appended and overwritten data needs to be flushed by* `fsync()` *calls (and also how much remain buffered when the workload ends). Overhead imposed by maintaining order will be observed by* `fsync()` *calls in the c²fs configuration needing to flush more data. The disk-level metrics relate the characteristics to actual data written to the device.*

under *c²fs-2* because c²fs sends some data to the journal and reduces seeks, as explained previously.) Thus, c²fs achieves real-world performance while improving correctness.

The *c²fs-1* configuration demonstrates the overhead of global order by running Git and SQLite in the same stream on c²fs; this is *not* the intended use case of c²fs. This configuration heavily impacts SQLite's performance because of (false) dependencies introduced from Git's writes. Running applications in separate streams can thus be necessary for acceptable performance.

The *ext4* configuration re-iterates previous findings: it maintains correctness using Git's *fsyncobjectfiles* on ext4, but Git is unacceptably slow due to `fsync()` calls. The *c²fs+* configuration represents a secondary use case for c²fs: it runs the applications in separate streams on c²fs with unneeded `fsync()` calls omitted, resulting in better SQLite performance (Git is moderately slower since SQLite uses more disk bandwidth).

Thus, running each application in its stream achieves correctness with good performance, while global order achieves correctness but reduces performance.

Developer Overhead. Achieving correctness atop c²fs (while maintaining performance) required negligible developer overhead: we added one `setstream()` call to the beginning of each application, without examining the applications any further. To omit unnecessary `fsync()` calls in c²fs and improve performance (i.e., for the c²fs+ configuration), we used the `IGNORE_FSYNC` flag on the `setstream()` calls, and added `streamsync()` calls to places in the code where the user is guaranteed durability (one location in LevelDB and two in SQLite).

Correctness with ext4 required two additional `fsync()` calls on LevelDB and the *fsyncobjectfiles* option on Git. The changes in ext4 both reduced performance and were complicated; we carefully used results from the Alice study to determine the additional `fsync()` calls necessary for correctness. Note that, while we happened to find that Git's *fsyncobjectfiles* makes it correct on ext4, other changes are needed for other file systems (e.g., btrfs).

Thus, developer effort required to achieve correctness atop c²fs while maintaining performance is negligible; additional effort can improve performance significantly.

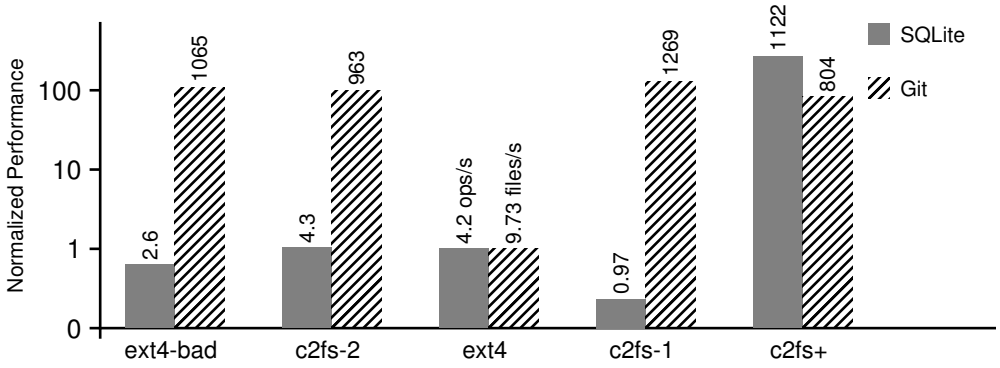


Figure 5.5: **Case Study: Multiple Application Performance.** *Performance of Git and SQLite-rollback run simultaneously under different configurations on HDD, normalized to performance under* `ext4` *configuration.* `Ext4-bad` *configuration runs the applications on ext4 with consistency sacrificed in Git.* `C2fs-2` *uses separate streams for each application on c²fs.* `Ext4` *uses ext4 with consistent Git.* `C2fs-1` *runs both applications in the same stream on c²fs.* `C2fs+` *runs applications in separate streams without unnecessary* `fsync`*. Workload: Git adds and commits a repository 25 times the size of Linux; SQLite repeatedly inserts 120-byte rows until Git completes.*

5.7 Summary

This chapter presents the stream abstraction as a practical solution for application-level crash consistency. We describe the stream API and the c²fs file system, an efficient implementation of the API. The effectiveness of streams is based on two hypotheses. We use real applications to validate consistency atop c²fs, thus validating the ordering hypothesis. We then compare performance with ext4, finding that c²fs maintains (and sometimes signficantly improves) performance while improving correctness, thus validating the efficiency hypothesis. Our results also suggest that developer effort for using the streams API is negligible and practical.

6 Related Work

This chapter presents research work related to each part of this thesis. We begin with tools and techniques that help in finding application crash vulnerabilities and in modeling file-system crash behavior, and then describe studies that examine crash vulnerabilities in real applications. Finally, we explain proposed solutions to application-level crash consistency that form an alternative to the Stream API, and various research work that is related to our design of c^{2}fs.

6.1 Tools to Find Crash Vulnerabilities

EXPLODE [109] has a similar flavor to ALICE: the authors use *in-situ* model checking to find crash vulnerabilities on different storage stacks. ALICE differs from EXPLODE in four significant ways. First, EXPLODE requires the target storage stack to be fully implemented; ALICE only requires a model of the target storage stack, and can therefore be used to evaluate application-level consistency on top of proposed storage stacks, while they are still at the design stage. Second, EXPLODE requires the user to carefully annotate complex file systems using *choose()* calls; ALICE requires the user to only specify a high-level APM. Third, EXPLODE reconstructs crash states by tracking I/O as it moves from the application to the storage. Although it is possible to use EXPLODE to determine the root cause of a vulnerability,

we believe it is easier to do so using ALICE since ALICE checks for failures associated with each system call, which is further associated directly with a particular source code line and call stack. Fourth, EXPLODE stops at finding crash vulnerabilities; by helping produce protocol diagrams, ALICE contributes to understanding the protocol itself.

Zheng et al. [111] describe a tool to find crash vulnerabilities in databases. They contribute a standard set of workloads that stress databases and check ACID properties; the workloads and checkers can be used with ALICE. Unlike ALICE, Zheng et al. do not systematically explore vulnerabilities of each system call. Their tool resembles BOB [65] (described in Section 5.6) but with additional heuristics, and is hence limited by the re-orderings and non-atomicity exhibited by a particular (implemented) file system during a single workload execution. Thus, their work is more suited for finding vulnerabilities that are *commonly* exposed under a given file system.

ALICE is influenced by SQLite's internal testing tool [91]. The tool works at an internal wrapper layer within SQLite, and also only deals with a subset of system calls, which are used by SQLite. It also does not allow being configured with a specific file-system model. Hence, unlike ALICE, it is not helpful for generic testing.

Jiang et al. [37] describe C^3, another tool that finds crash vulnerabilities (their work follows ALICE), and evaluate the tool with applications such as text editors that use simpler crash-consistency protocols than those tested using ALICE. The core method used by C^3 to reproduce crash states is the same as BOB (described in Section 5.6). To deal with the low state-space coverage implicit in this method, the authors develop a technique of automatically inducing timing-changes for the given input workload. C^3 also does not use an invariant checker, instead simply detecting whether a crashed file-system state is similar to the file-system state observed while running the input workload. The invariant-checking methodology used by C^3 does not work for the complex applications that ALICE targets, while

generating alternative workloads with varying timing effects is unnecessary (does not improve coverage) with ALICE. However, C^3's technique of using BOB with generated timing-differing workloads might be advantageous than ALICE in that an APM is not required when validating atop pre-existing storage stacks.

Following ALICE, Bornholt et al. [6] implement a model checker, FERRITE, that operates on the crash-behavior specifications of file systems (the specifications are similar to ALICE's APM). Since FERRITE is built using Rosette [100], it allows program synthesis and verification based on SAT and SMT solvers, and hence allows discovering crash vulnerabilities. To our knowledge, the tool neither considers the presence of an explicit recovery protocol (instead requiring checking-invariants be described in terms of the file-system state), and hence is not practical for complex applications, nor has been applied on any real-world applications.

Koskinen and Yang [41] recently defined application-level crash recoverability in terms of a formal model that allows crash consistency to be thought of similar to reachability. This allows the authors to build ELEVEN82, a tool that finds recoverability bugs or proves their absence using a symbolic search through the input and crash state space; the authors use the tool to analyze seven of the applications that ALICE investigated. ELEVEN82 offers a better solution than ALICE in theory, but currently has practical concerns; for instance, it does not support nested directory structures.

Woodpecker [20] can be used to find crash vulnerabilities when supplied with suspicious source code patterns to guide symbolic execution. Woodpecker can both have false positives, since a given pattern might appear in a non-vulnerable part of the source code, and will have false negatives, since it is impossible to determine all patterns that can result in vulnerabilities. However, Woodpecker can be used to quickly check a given source code and warn the developer about previously known vulner-

ability patterns appearing in the code. Woodpecker is complementary to ALICE's methodology; ALICE is fundamentally different from Woodpecker's approach, as it does not require prior knowledge of patterns in checked applications.

RacePro [43], a testing tool for concurrency bugs, records system calls and replays them by splitting them into small operations, similar to micro operations in ALICE. Unlike ALICE, it does not test crash consistency.

Chen et al. [10, 11] focus on crash consistency at the file-system layer, defining formal logic (termed Crash Hoare logic) and designing a file system with crash behavior proved to correspond to a specification. Alagappan et al. [1] broadly explore specifying crash behavior across different layers of the storage stack, and describe techniques to improve both correctness and performance using the specifications. Both these works are complementary to ALICE, focusing on the same end goal of improving the crash reliability of storage, but at different layers.

6.2 Vulnerabilities Survey

To our knowledge, there are only few studies on the crash consistency of real-world applications; none are as extensive as ours. EXPLODE [109] broadly studies multiple layers of the storage stack, including file systems, RAID, and applications. It investigates four applications (BerkeleyDB, CVS, Subversion, and an unnamed version control system) and finds 11 failures across them. Of these, the authors are able to accurately identify the source error locations for only 3 failures, one each in CVS, Subversion, and the other version control system; the authors are unable to suggest the error locations for 6 failures observed in BerkeleyDB. Hence, the data presented by EXPLODE is not sufficient to understand the file-system behaviors associated with the vulnerabilities or identify common patterns present across applications.

Zheng et al. [111] focus on databases, and use a standard set of workloads to test 8 databases atop the ext4 and xfs file systems. They report whether each of the tested databases hold the A, C, and D parts of the ACID property. The methodology used by Zheng et al. makes it cumbersome to identify the source error locations corresponding to an observed failure, and the authors report (as case studies) on the error locations corresponding to only three of the failures discovered. Thus, similar to EXPLODE, the data is not sufficient to understand associated file-system behaviors or common patterns.

Koskinen and Yang [41], whose work follows ours, examine a subset of the same applications as Chapter 4 and observe failures, but do not describe or comment further on the failures observed. C^3 [37], which also follows our work, examines text editors, the GNU make utility, and other applications that only implement simple forms of crash consistency; they find failures, but do not further examine the data.

6.3 Atomicity Interfaces

Transactional file systems have a long history [77] and allow applications to delegate most crash-consistency requirements to the file system. They give applications easy-to-use transactional interfaces with ACID semantics, and aim to remove complexity within applications and thus reduce vulnerabilities. Hence, they provide an alternative to the Stream API for achieving application-level crash consistency.

Recent work in this space includes Amino [108], Valor [85], and Windows TxF [56]. More recent research has recognized that the ACID semantics of transactional file systems affect both flexibility [62] and performance [57] by imposing a specific isolation and concurrency control mechanism on their users. Hence, researchers have proposed atomicity-only file systems (i.e., they do not provide isolation when compared to

transactional file systems), by Vermat et al. [106], Park et al. [62], and CFS [57]. There also exist OS-level transaction support as advocated by TxOS [68].

Such interfaces allow adding crash consistency easily to applications which do not already implement them, and help heavily optimized applications that trade portability for performance [51]. File systems implementing atomicity interfaces take advantage of the close interaction possible with other kernel subsystems (such as the buffer cache) to improve performance. If the storage medium has atomic capabilities, then these file systems might transparently use the capabilities to improve performance.

For applications with existing consistency implementations, proponents of atomicity interfaces and transactional file systems advocate replacing the existing implementation with the interface provided by the file system. This is not trivial to achieve (though perhaps much easier than writing a new consistency implementation). For instance, consider the SQLite database, and assume that we want to replace its consistency implementation using a straightforward *begin_atomic()–end_atomic()* interface provided by the file system.

A simple attempt at achieving this would be to switch off SQLite's journaling (i.e., its consistency mechanism) and add a *begin_atomic()* when the user requests a transaction start and *end_atomic()* when the user requests a commit. This attempt does not work for two reasons. First, it removes (does not implement) SQLite's ROLLBACK command (i.e., abort transaction) and the SAVEPOINT command (which allows an aborted transaction to continue from a previous point in the transaction). Second, unless the file system provides isolation (which recent research argues against), it requires re-implementing isolation and concurrency control, since SQLite's isolation mechanism is inherently tied to its consistency mechanism (for both rollback journaling and WAL).

In addition to converting the application to use the file-system pro-

vided atomicity interface, in practice, a programmer has to maintain two versions of application code, since the application will often be run in environments without the atomic file system. Also note that we have discussed an ideal atomic interface so far, and there are usually further concerns. For instance, if the file system guarantees the durability of all atomic updates, and the application does not desire durability, performance will be lost. If durability is optional, and the file-system instead promises global ordering between atomic updates, performance degradation due to false dependencies will arise. If neither ordering nor durability is guaranteed, applications would often require additional measures to achieve consistency. If stream-ordering is provided between atomic updates, the effort required to use the atomicity interface would be strictly greater than that to use streams (assuming an already existing consistency implementation).

To summarize, adopting atomicity interfaces to overcome vulnerabilities is nonoptimal in applications with existing consistency implementations. One challenge is simply the changes required: CFS [57], with arguably the most user-friendly atomic interface, requires changing 38 lines in SQLite and 240 lines in MariaDB; although the number of lines changed might appear small compared to the total size of the application, much developer effort resides in identifying the places to be changed (and to be left unchanged), i.e., the changes themselves are more complicated. Another challenge is portability: until the interfaces are widely available, the developer must maintain both the existing consistency protocol and a protocol using the atomic interface; this has deterred such interfaces in Linux [19]. Finally, the complexity of data structures and concurrency mechanisms in modern applications (e.g., LSM trees) are not directly compatible with a generic transactional interface; Windows TxF, a transactional interface to NTFS, is being considered for deprecation due "*due to its complexity and various nuances which developers need to consider as part of application development*" [56].

In contrast, the Stream API focuses on masking vulnerabilities in existing application-level consistency implementations. C^{2}fs advocates a single change to the beginning of applications. The applications can then be run without further modification on both stream-enabled and stream-absent file systems, gaining correctness with stream-enabled file systems.

6.4 Fine-grained Ordering interfaces

Much previous work has proposed new interfaces that allow the application developer to express ordering dependencies among file-system updates. Burnett [7] and Featherstitch [25] propose an interface to express the exact ordering required as directed acyclic graphs. OptFS [12] introduces the `osync()` call to express ordering dependencies, and compares it against the `fsync()` call, which imposes both ordering and durability. These interfaces allow applications to achieve system-crash consistency efficiently, with less complexity than with the current file-system interface.

Such interfaces allow better performance, but require developers to specify the exact ordering required, and as such are not optimal for fixing existing protocol implementations. The interfaces are instead best suited for applications that are willing to trade off programmer overhead to get micro-managed performance gains. In contrast, the Stream API aspires to mask vulnerabilities in existing applications while involving the programmer minimally.

6.5 Stream and Global Ordering

Streams resemble *dependency queues* from HP's classical MPE XL operating system [40]. However, the internal implementation of dependency queues, their application-level usage, and applicability to current operating systems are unclear. Also very related to the Stream API is recent work by

Pelley et al. [63], which separates and studies the write ordering imposed for persistence and memory consistency in NVRAMs. But, Pelley et al.'s work does not deal with file systems or the file system interface.

Some file systems provide global ordering as a side effect of their internal consistency mechanisms, including specific modes of ext3 and ext4. LinLogFS [21] is an example that is explicitly designed for providing global ordering. All such generic file systems that provide global ordering suffer from the performance penalities described in Chapter 5 (Section 5.2). BPFS [18] provides efficient global ordering and atomicity, but is designed to be run on NVRAM with transactional primitives. Xsyncfs [60] provides global ordering and avoids performance effects due to false dependencies by buffering user-visible outputs; this approach is complementary to our approach of reducing false dependencies.

6.6 C^{2}FS Implementation

C^{2}fs builds upon seminal work in database systems [30, 58] and file-system crash consistency [17, 23, 26, 27, 31, 73, 75, 78], but is unique in assembling different techniques required to efficiently implement the stream API. Specifically, c^{2}fs uses journaling [17, 31] for order within a stream, but applies techniques similar to soft updates [26, 27, 78] for separating streams. Log-structured approaches [73, 75] can be an alternate method for order within streams, consisting of different performance trade-offs, but have not been explored in c^{2}fs.

Thus, c^{2}fs uses a design with different classic consistency techniques, one each for ordering within streams and for separating ordering between streams. This approach is necessary: using soft updates directly for a long chain of dependent writes ordered one after the other (as c^{2}fs promises within a stream) will result in excessive disk seeks.

In principle, one should be able to easily construct a stream-ordered

file system atop a fine-grained ordering interface. However, the direct implementation of ordering in Featherstitch [25] uses the soft-updates approach, which is incompatible as described previously. The `osync()` interface used by OptFS [12] does not allow separating the effect of false dependencies between streams, and is hence insufficient for implementing streams. C^{2}fs uses the SDJ technique from OptFS but optimizes it; the original relies on specialized hardware (durability notifications) and decreased guarantees (no durability) for efficiency. Block-level guarantees of atomicity and isolation, such as Isotope [80] and TxFlash [70], can simplify c^{2}fs' separation of streams, but techniques in Section 5.4 are still needed.

7 Conclusion

The reading and writing of data, one of the most fundamental aspects of any Von Neumann computer, is surprisingly subtle and full of nuance. For example, consider access to a shared memory in a system with multiple processors. While a simple and intuitive approach, known as "strong consistency", is easiest for programmers to understand and reason about [44], many weaker models exist and are widespread (e.g., x86 total store ordering [79]); such approaches improve system performance, but at the cost of making reasoning about system behavior more complex and error prone. Fortunately, a great deal of time and effort has gone into thinking about such memory models [84], and, as a result, most multiprocessor applications are not caught unaware.

In many ways, this thesis simply tries to understand similar subtelities involved in local file systems, i.e., the systems that manage data stored in your desktop computer, on your cell phone [38], or serve as the underlying storage beneath large-scale distributed systems such as HDFS [81]. We first study the easiest model for the users of these file systems (i.e., applications), designing a tool for understanding applications in the process, and then propose a model that is most aligned to the ideal model while still providing performance. In this chapter, we first summarize each part of this thesis (§7.1), discuss the various lessons we learnt through the course of this thesis (§7.2), and possible future work (§7.3).

7.1 Summary

In this section, we discuss the summary of each part of this thesis.

ALICE

In the first part of this thesis, we looked at the Alice tool which helps understand the crash-consistency protocols of applications and finds vulnerabilities in the protocols. Alice works by analyzing the system-call trace of the application produced from a user-supplied workload. It determines and reconstructs various crash states that might result from the trace, and then runs a user-supplied checker on each reconstructed state. Among all possible crash states for the obtained system-call trace, Alice targets a specific subset designed to find more vulnerabilities while examining few states. Results are presented to the user as static crash vulnerabilities, each vulnerability corresponding to a location in the source code that causes a violation of the application's crash consistency.

Crash states are determined in Alice by first representing the initial file-system state of the application workload as a set of logical entities, and converting the system-call trace into file-system-agnostic logical operations operating on these entities. The logical operations are then converted into micro operations; this conversion differs between file systems, and depends on the crash behavior of the file system that the user is interested in. The user hence supplies the crash behavior to Alice as an APM (abstract persistence model) of the file system. The APM also specifies ordering constraints between the resulting micro operations, which are taken into account when determining crash states.

We also evaluated the usability of Alice by considering the effort required to write workloads and checkers. We evaluate the running time for Alice and the effectiveness of targeting only a subset of crash states for a specific application (LevelDB). A more encompassing (but less quantita-

tive) evaluation of ALICE is left to the next chapter of this thesis.

Vulnerabilities Study

In the next part of this thesis, we first examined two applications, SQLite and LevelDB, to understand developer and user attitudes towards crash consistency in the real world. For both applications, we examined previous mailing list discussions and bug reports. We considered configuration options in the applications that deal with file-system crash behavior and with the crash guarantees of the application. We also studied the source code and the system-call trace of these applications, and tried to manually find crash vulnerabilities in them.

With LevelDB, we found that one vulnerability had been previously reported, and we discovered four other vulnerabilities; we also found that LevelDB did not formally document its crash guarantees. With SQLite, three vulnerabilities had previously been reported and fixed, and we did not find any more, perhaps because SQLite has an internal testing tool for crash consistency. However, SQLite has multiple configuration options that can improve performance if the crash-behavior of the underlying file system is known.

We then examined 11 applications using ALICE, and found 60 vulnerabilities in them. Chapter 4 reports on the update protocol observed in each application and the interesting vulnerabilities found in them. We explained the results in terms of the common patterns observed across applications, consolidating the vulnerabilities both based on their failure consequences as well as the associated file-system behavior: vulnerabilities lead to corruption and unavailability, and can be associated with crash behavior that is becoming more common in modern file systems.

We analyzed the vulnerabilities exposed for different modern file systems. We observed that more and more vulnerabilities are exposed by each file system in the following sequence: ext3 under the data-journaled

mode, ordered mode, writeback mode, ext4 under the ordered mode, and btrfs. We also commented on our interaction with developers when reporting the vulnerabilities: crash vulnerabilities can easily be misreported and understood to be device failures, and application developers are misinformed about file-system crash behavior.

C^{2}FS

In the final part, we proposed c^{2}fs, a solution to application-level crash consistency. We first explained the file-system behavior that is most compatible with the application requirements for correctness: full ordering and weak atomicity. Next, we explained why full ordering is bad for performance: false write dependencies might occur, reducing coalescing, avoidance, and other overheads depending on the storage stack.

We then introduced the hypothesis that forms the basis of c^{2}fs: false dependencies that have noticeable performance impact only occur between applications; so long as full ordering is provided only within each application, performance overhead can be avoided. Based on this hypothesis, we introduced the Stream API. With minimal developer effort, the API can be used to distinguish between write requests from different applications at runtime; a file system can thus ensure application correctness by providing the required ordering without impacting performance. With slightly more developer effort, the API also supports increasing application performance by orders of magnitude while maintaining correctness.

We explained the design for the c^{2}fs file system. The file system is derived from ext4, but changes the journaling methodology to separately ensure ordering within each stream (but not across streams). C^{2}fs takes many measures to ensure that performance optimizations in ext4 are not affected, while ordering (and hence application correctness) is provided.

We evaluated c^{2}fs on many aspects. We evaluated whether our implementation of c^{2}fs correctly provides full ordering by using the BoB tool,

and whether full ordering provides correctness when applications are configured practically (unlike in Chapter 4, wherein the safest configuration is used). We evaluated whether our implementation of c^{2}fs re-orders between streams to avoid performance impact under false dependencies. These experiments show that c^{2}fs provides the required ordering, full ordering improves application-level crash consistency, and that c^{2}fs re-orders sufficiently between streams.

We then used the Filebench suite to evaluate whether running each benchmark in a single stream introduces any impactful false dependencies; we found that it does not, as hypothesized. We also use SQLite, LevelDB, and Git to evaluate whether false dependencies are introduced within each application, whether running the applications simultaneously in different streams provides good performance, and the developer effort required to run applications in separate streams. We found c^{2}fs provides good performance, and the required developer effort is trivial.

7.2 Lessons Learnt

We now discuss the various lessons we learnt throughout the course of this thesis.

Manual studies are important

We first focused on manually studying crash consistency protocols and vulnerabilities in two applications before using any automated tools. In hindsight, this was important, especially because there was little previous research insight into this area: only Explode previously reported on application crash consistency. The manual study contributed many important design decisions for the Alice tool, and enabled us to appropriately interpret the results of our study. From the manual study, we determined that a tool that intercepts at the device layer and records a block trace, and

then determines crash states based on the trace, is not sufficient for finding application crash vulnerabilities. This is because many vulnerabilities are often hidden by non-deterministic flushing actions by the file system at the buffer-cache layer. Hence, enumerating possible crash states at the block layer will not produce those states that would have resulted if the buffer cache were flushed in a different order by the file system.

Similarly, the manual study also revealed that the application's recovery protocol needs to be considered to determine crash vulnerabilities (instead of simply considering the file-system state after the crash) and that associating the vulnerabilities with their source lines is important for understanding them. These aspects of ALICE are significantly different from other tools because of the manual study, as described in Chapter 6, and are important for the effectiveness of ALICE. We also interpreted our results correctly because of the manual study; this is important because of the inherent difficulty involved in reporting vulnerabilities and in describing file-system behavior to developers.

Specification is important

Throughout this thesis, we observe many situations where the ultimate reason for incorrectness and failure is the absence of clear specification. At the application layer, the guarantees provided in application documentation might be confusing (or simply absent); an example is SQLite's absence of a durability guarantee by default, and GDBM's absence of any crash consistency guarantees (despite a synchronous flag). Indeed, only some application developers have a clear view of the exact guarantees provided, and many offer only probabilistic guarantees.

On the other hand, most application developers are confused about the crash behavior of file systems. Some developers believe that POSIX requires file systems to have good crash behavior, such as requiring that directory operations are sent to the disk in-order. Unfortunately, there is

little clarity as to what exactly POSIX defines with regards to crashes [3, 105], leading to much debate and little consensus.

Consider practical runtime environments

The Stream API and the c^{2}fs file system is based on the hypothesis that, within an application, there are few false dependencies that cause performance overhead. We also verified this quantitatively by running standard benchmarks. Assuming that the hypothesis is true, note that the subset of experiments we used to validate the hypothesis does not require the Stream API nor most techniques in c^{2}fs for good performance. Instead, a much simpler file system that maintains global ordering while avoiding the journaling overhead of writing data twice will perform sufficiently with these experiments. This is because these experiments, based on standard file-system benchmarks, do not consider multiple applications running simultaneously: the benchmarks themselves only represent a single application.

Hence, if we had only considered standard benchmarks that are used commonly in file-system research, we would have falsely concluded that false dependencies never arise, and that the simpler global-ordering file system provides good performance while improving correctness. The requirement for both the Stream API and an efficient implementation of the API is borne out of the requirement of considering a practical deployment environment, in which multiple applications are run together.

Thus, considering practical environments can present challenges and opportunities for further research. We designed both the Stream API and c^{2}fs to separate updates between applications, and verified both our hypothesis and the performance of our implementation by using customized benchmarks mimicking a practical environment.

Intuitive semantics must be the default

When designing any interface, a trade-off occurs between hiding implementation details and providing stronger semantics. The default file-system interface takes the approach of weak semantics about crash behavior, and leaves the exact semantics to the file-system implementation. Many file systems, taking advantage of this, provide only non-intuitive behavior to applications, arguing both for performance and simplicity of the file system.

In Chapter 5, we derived the file-system crash behavior that is most compatible with application crash consistency from the data in the previous chapter. However, one should note that the behavior required is simply the most intuitive from an application developer's perspective. Given the impact that providing such an intuitive behavior has on application consistency, we believe that the approach taken by the file-system interface happens to be wrong. Ideally, other than situations where it is impossible to provide durable file-system behavior (such as in a file system running on volatile main memory), the file-system interface would offer intuitive and strong crash semantics by default, but let the application choose weaker semantics if it desires performance.

7.3 Future Work

We now discuss the different directions for future work.

Crash Vulnerabilities in Distributed Applications

This thesis focuses on crash consistency for single-node applications. While Chapter 4 considers HDFS and ZooKeeper, they are run in the local mode and the single-node configurations, respectively. Hence, while the discovered vulnerabilities in HDFS and ZooKeeper affect their crash

consistency, we do not consider situations such as a single node crashing while other nodes are still running.

Alagappan et al. [1] have studied the crash consistency of distributed systems by considering a broader *correlated crash* scenario than our study: they run the distributed system on multiple nodes, and consider the situation where all nodes crash at the same time. Much of the challenge in investigating correlated crashes stems from the fact that each node can possess its own crash state after the crash; the possible crash states of the entire system is equivalent to the valid cross product of possible individual crash states. However, correlated crashes do not consider the situation where each individual node can crash independent of the other nodes, and thus does not examine an important set of crash states.

Investigating the crash consistency of distributed systems is interesting for two reasons. First, since a part of the system is running while another part crashes, there are chances of relevant information being modified while crash recovery is still in progress. Second, distributed systems might have more ways of detecting and recovering from crashes, since data is replicated. Studying distributed crash consistency is also not straightforward (provided ALICE) due to the first reason. Hence, we find it imperative to study distributed crash consistency in the future.

Workload Characterization for Stream API

The idea behind the Stream API is that overhead-inducing false dependencies do not commonly occur within an application. We have provided evidence for this hypothesis by considering standard benchmarks and a few applications. However, future work can investigate the hypothesis further, since our experiments lack some interesting details.

The first aspect that future work can improve upon, when considering this hypothesis, is to simply inspect more real-world applications. Second, future work should characterize applications, perhaps based on

the number of false dependencies occurring within an application, and how each of these dependencies can impact performance. One can also imagine characterizing each false dependency, such as whether they affect write coalescing, write avoidance, or other file-system optimizations like delayed allocation. Such studies will help design file systems where little overhead is observed for any prevalent false dependencies in common applications. They will also provide insight into whether separating applications into multiple streams can satisfy performance goals with unexpected application workloads, and the developer effort involved.

The final aspect that future work should consider is the interaction between multiple applications. We claimed that false dependencies occurring between different applications might impact performance, and provided evidence for this with some experiments. However, we did not characterize whether some applications can be affected more by false dependencies because of other applications, or whether some applications can affect other applications more. Such a study would reveal if it is possible to club multiple applications into the same stream; if so, this would allow streams (and fully ordered file systems) to be easily adapted.

Application Crash Consistency under Persistent Memory

The default abstract persistence model in Chapters 3 and 4 models a file system that guarantees atomicity of data writes only at the byte level, mainly in consideration of file systems atop persistent memory. We also find some vulnerabilities where applications require sector-level atomicity.

However, our results only indicate that studying crash consistency under persistent memory is interesting; future work should investigate further details. Future work can investigate crash consistency using applications that are optimized for persistent memory, to understand the common patterns that occur in such applications. It can also consider the APM of realistic persistent-memory file systems, and devise techniques

to check crash consistency when new storage interfaces (instead of the standard file-system interface) are used for accessing persistent memory. Future work can also propose solutions for improving crash consistency in practice under persistent-memory environments: the trade-offs involved will be different from those considered for the Stream API and c^{2}fs.

7.4 Closing Words

Application-level crash consistency is hard to implement efficiently. This thesis, in accordance with anecdotal evidence, concurs that many implementations are incorrect. On the one hand, application developers blame such incorrectness on non-intuitive crash guarantees from the file system. On the other, most file-system developers and researchers either assume that applications run correctly with traditional file-system guarantees (which are often undefined or misunderstood), or consider it entirely the application developers' burden.

Similar problems have been faced before in other areas of computer systems, in the domains of multiprocessor shared memory and distributed systems. Those problems have been overcome by creating new abstractions, understanding various trade-offs, and and even thinking about the problem with analogies to baseball [96]. In this thesis, we present the stream abstraction and the c^{2}fs file systems as practical solutions for application-level crash consistency; we find that c^{2}fs maintains (and sometimes signficantly improves) performance while improving correctness.

Bibliography

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

[2] Apache. Apache Zookeeper. `http://zookeeper.apache.org/`.

[3] Austin Group Defect Tracker. 0000672: Necessary step(s) to synchronize filename operations on disk. `http://austingroupbugs.net/view.php?id=672`.

[4] Steve Best. JFS Overview. `http://jfs.sourceforge.net/project/pub/jfs.pdf`, 2000.

[5] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. `http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf`, 2007.

[6] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Lan-*

guages and Operating Systems (ASPLOS 21), Atlanta, Georgia, April 2016.

[7] Nathan C. Burnett. *Information and Control in File System Buffer Management*. PhD thesis, University of Wisconsin-Madison, Oct 2006.

[8] Donald D Chamberlin, Arthur M Gilbert, and Robert A Yost. A history of system r and sql/data system. In *VLDB*, pages 456–464, 1981.

[9] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 205–218, Seattle, Washington, November 2006.

[10] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP '15)*, Monterey, California, October 2015.

[11] Haogang Chen, Daniel Ziegler, Adam Chlipala, M. Frans Kaashoek, and Eddie Kohler Nickolai Zeldovich. Specifying crash safety for storage systems. In *The Fifteenth Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, May 2015.

[12] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[13] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic Crash Consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[14] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, pages 101–116, San Jose, California, February 2012.

[15] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[16] Chris Davies. Fake hard-drive has short-term memory not 500GB. `http://www.slashgear.com/fake-hard-drive-has-short-term-memory-not-500gb-08145144/`.

[17] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode File System. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 43–60, San Francisco, California, January 1992.

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

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

[20] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, pages 329–342. ACM, 2013.

[21] Christian Czezatke and M. Anton Ertl. LinLogFS: A Log-structured Filesystem for Linux. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, San Diego, California, June 2000.

[22] Jeff Dean and Sanjay Ghemawat. Leveldb. `https://rawgit.com/google/leveldb/master/doc/index.html`.

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

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

[25] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 307–320, Stevenson, Washington, October 2007.

[26] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2), May 2000.

[27] Gregory R. Ganger and Yale N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[28] GNU. GNU Database Manager (GDBM). `http://www.gnu.org.ua/software/gdbm/gdbm.html`, 1979.

[29] Google. LevelDB. `https://code.google.com/p/leveldb/`, 2011.

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

[31] Robert Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, Austin, Texas, November 1987.

[32] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[33] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach (4th Edition)*. Morgan Kaufmann, 2006. With contributions by Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Krste Asanovic, Robert P. Colwell, Thomas M. Conte, Jose Duato, Diana Franklin, David Goldberg, Wen-mei W. Hwu, Norman P. Jouppi, Timothy M. Pinkston, John W. Sas, David A. Wood.

[34] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter*

Technical Conference (USENIX Winter '94), San Francisco, California, January 1994.

[35] HyperSQL. HSQLDB. `http://www.hsqldb.org/`.

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

[37] Yanyan Jiang, Haicheng Chen, Feng Qin, Chang Xu, Xiaoxing Ma, and Jian Lu. Crash consistency validation made easy. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '16)*, Seattle, WA, USA, November 2016.

[38] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting Storage for Smartphones. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[39] Jaeho Kim, Jongmoo Choi, Yongseok Oh, Donghee Lee, Eunsam Kim, and Sam H. Noh. Disk Schedulers for Solid State Drives. In *EMSOFT*, Grenoble, France, October 2009.

[40] Alan J. Kondoff. The MPE XL Data Management System Exploiting the HP Precision Architectures for HPâŁ™s Next Generation Commercial Computer Systems. In *IEEE Compcon Proceedings*, San Fransisco, California, 1988.

[41] Eric Koskinen and Junfeng Yang. Reducing Crash Recoverability to Reachability. In *The 43rd SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*, St. Petersburg, FL, USA, January 2016.

[42] Dean Kuo. Model and verification of a data manager based on aries. *ACM Trans. Database Syst.*, 21(4):427–479, December 1996.

[43] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, , and Jason Nieh. Pervasive Detection of Process Races in Deployed Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[44] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, September 1979.

[45] Butler Lampson. Computer Systems Research – Past and Present. SOSP 17 Keynote Lecture, December 1999.

[46] LevelDB. LevelDB Issues List. `http://code.google.com/p/leveldb/issues/list`.

[47] Linus Torvalds. Git. `http://git-scm.com/`, 2005.

[48] Linus Torvalds. Git Mailing List. Re: what's the current wisdom on git over NFS/CIFS? `http://marc.info/?l=git&m=124839484917965&w=2`, 2009.

[49] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[50] LWN. That massive filesystem thread. `http://lwn.net/Articles/326471/`.

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

[52] Cris Pedregal Martin and Krithi Ramamritham. Toward formalizing recovery of (advanced) transactions. In *Advanced Transaction Models and Architectures*, pages 213–234. Springer, 1997.

[53] Chris Mason. The Btrfs Filesystem. `oss.oracle.com/projects/btrfs/dist/documentation/btrfs-ukuug.pdf`, September 2007.

[54] Matt Mackall. Mercurial. `http://mercurial.selenic.com/`, 2005.

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

[56] Microsoft. Alternatives to using Transactional NTFS. Retrieved April 2016 from https://msdn.microsoft.com/en-us/library/windows/desktop/hh802690(v=vs.85).aspx.

[57] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight Application-Level Crash Consistency on Transactional Flash Storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC '15)*, Santa Clara, CA, July 2015.

[58] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[59] C Mohan, Bruce Lindsay, and Ron Obermarck. Transaction management in the r* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[60] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the Sync. In *Proceedings of the 7th Symposium on*

Operating Systems Design and Implementation (OSDI '06), pages 1–16, Seattle, Washington, November 2006.

[61] Patrick OâŁ™Neil, Edward Cheng, Dieter Gawlick, and Elizabeth OâŁ™Neil. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[62] Stan Park, Terence Kelly, and Kai Shen. Failure-Atomic Msync (): a Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the EuroSys Conference (EuroSys '13)*, Prague, Czech Republic, April 2013.

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

[64] Thanumalayan Sankaranarayana Pillai. Ordering of directory operations maintained across system crashes in Btrfs? Retrieved Nov 2016 from https://www.spinics.net/lists/linux-btrfs/msg32217.html, March 2014.

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

[66] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency. *Communications of the ACM*, 58(10), October 2015.

[67] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency: Rethinking the Fundamental Abstractions of the File System. *ACM Queue*, 13(7), July 2015.

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

[69] Postgres. PostgreSQL: Documentation: 9.1: WAL Internals. `http://www.postgresql.org/docs/9.1/static/wal-internals.html`.

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

[71] Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Coerced Cache Eviction and Discreet-Mode Journaling: Dealing with Misbehaving Disks. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '11)*, Hong Kong, China, June 2011.

[72] Hans Reiser. ReiserFS. `www.namesys.com`, 2004.

[73] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[74] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[75] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '93)*, pages 307–326, San Diego, California, January 1993.

[76] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '90)*, pages 313–324, Washington, D.C, January 1990.

[77] Margo I. Seltzer. *File System Performance and Transaction Support*. PhD thesis, EECS Department, University of California, Berkeley, Jun 1993.

[78] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 71–84, San Diego, California, June 2000.

[79] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. X86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Communications of the ACM*, July 2010.

[80] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. Isotope: Transactional Isolation for Block Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, California, February 2016.

[81] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST '10)*, Incline Village, Nevada, May 2010.

[82] Chuck Silvers. UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, San Diego, California, June 2000.

[83] Stewart Smith. Eat My Data: How everybody gets file I/O wrong. In *OSCON*, Portland, Oregon, July 2008.

[84] Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 2011.

[85] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proceedings of the 7th USENIX Symposium on File and Storage Technologies (FAST '09)*, San Francisco, California, February 2009.

[86] SQLite. Atomic Commit In SQLite. `http://sqlite.org/atomiccommit.html`.

[87] SQLite. Creation and deltions of files should fsync() directory. `https://www2.sqlite.org/cvstrac/tktview?tn=410`.

[88] SQLite. Database corruption following power-loss in WAL mode. `http://www.sqlite.org/src/info/ff5be73dee`.

[89] SQLite. How SQLite Is Tested. `http://www.sqlite.org/testing.html`.

[90] SQLite. Missing call to xSync() following rollback. http://www.sqlite.org/src/info/015d3820f2.

[91] SQLite. SQLite transactional SQL database engine. http://www.sqlite.org/.

[92] Sun Microsystems. ZFS: The last word in file systems. www.sun.com/2004-0914/feature/, 2006.

[93] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[94] Symas. Lightning Memory-Mapped Database (LMDB). http://symas.com/mdb/, 2011.

[95] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A Flexible Framework for File System Benchmarking. *;login: The USENIX Magazine*, 41(1), June 2016.

[96] Doug Terry. Replicated Data Consistency Explained Through Baseball. MSR Technical Report, October 2011.

[97] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Joo-young Hwang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. Towards Efficient, Portable Application-Level Consistency. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep '13)*, Farmington, PA, November 2013.

[98] The Open Group. POSIX.1-2008 IEEE Std 1003.1. http://pubs.opengroup.org/onlinepubs/9699919799/, 2013.

[99] The PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org/.

[100] Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, June 2014.

[101] Theodore Ts'o. Don't fear the fsync! `http://thunk.org/tytso/blog/2009/03/15/dont-fear-the-fsync/`.

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

[103] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[104] Stephen C. Tweedie. EXT3, Journaling File System. `olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html`, July 2000.

[105] Valerie Aurora. POSIX v. reality: A position on O_PONIES. `http://lwn.net/Articles/351422/`.

[106] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Srivilliputtur Mannarswamy, Terence P. Kelly, and Charles B. Morrey III. Failure-Atomic Updates of Application Data in a Linux File System. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, California, February 2015.

[107] VMWare. VMWare Player. `http://www.vmware.com/products/player`.

[108] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID Semantics to the File System Via Ptrace. *ACM Transactions on Storage (TOS)*, 3(2):1–42, June 2007.

[109] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[110] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services. In *HotStorage '13*, San Jose, California, June 2013.

[111] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S Yang, Bill W Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.