PerSeVerE: Persistency Semantics for Verification under Ext4

MICHALIS KOKOLOGIANNAKIS, MPI-SWS, Germany
ILYA KAYSIN, National Research University Higher School of Economics, JetBrains Research, Russia
AZALEA RAAD, Imperial College London, United Kingdom
VIKTOR VAFEIADIS, MPI-SWS, Germany

Although ubiquitous, modern filesystems have rather complex behaviours that are hardly understood by programmers and lead to severe software bugs such as data corruption. As a first step to ensure correctness of software performing file I/O, we formalize the semantics of the Linux `ext4` filesystem, which we integrate with the weak memory consistency semantics of C/C++. We further develop an effective model checking approach for verifying programs that use the filesystem. In doing so, we discover and report bugs in commonly-used text editors such as `vim`, `emacs` and `nano`.

CCS Concepts: • **Theory of computation → Program verification**; **Axiomatic semantics**.

Additional Key Words and Phrases: File Systems; Persistency; Weak Consistency; Model Checking

ACM Reference Format:
Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. 2021. PerSeVerE: Persistency Semantics for Verification under Ext4. *Proc. ACM Program. Lang.* 5, POPL, Article 43 (January 2021), 29 pages. https://doi.org/10.1145/3434324

1 INTRODUCTION

File I/O is one of the most fundamental concepts in computer science. Almost all applications interact with filesystems to store their configurations, while others (e.g., document editors) crucially depend on them for their core functionality. As such, there is a large body of work on designing and implementing filesystems; e.g., [Bonwick 2005; Park et al. 2017; Pillai et al. 2017; Rodeh et al. 2013; Son et al. 2017; Sweeney 1996]. Major deployments such as Linux's `ext4` [Ts'o et al. 2002] are sophisticated designs that incorporate a number of optimizations for good performance.

Nevertheless, most programmers have a simplistic view of filesystems and assume that their updates happen in the order specified by a program. For instance, if an application writes 'A' to file `a.txt` and then 'B' to `b.txt`, they would assume that if the computer were to crash at some point, it would not be possible for the 'B' update to have persisted without 'A' having also persisted. However, this assumption is violated by all modern filesystems, and programmers must insert system calls such as sync/fsync to ensure that updates on one file complete before those to another.

Due to programmer ignorance and the significant overhead of such system calls, programs often invoke these calls incorrectly, leading to critical bugs that thrash the persistent state and render the file/application useless. Despite their importance, these persistency bugs are hard to detect as they occur very rarely in cases of software/hardware crashes which are difficult to emulate.

Authors' addresses: Michalis Kokologiannakis, MPI-SWS, Saarland Informatics Campus, Germany, michalis@mpi-sws.org; Ilya Kaysin, National Research University Higher School of Economics, JetBrains Research, Russia, ilya.s.kaysin@gmail.com; Azalea Raad, Imperial College London, United Kingdom, azalea@imperial.ac.uk; Viktor Vafeiadis, MPI-SWS, Saarland Informatics Campus, Germany, viktor@mpi-sws.org.

2475-1421/2021/1-ART43
https://doi.org/10.1145/3434324

To ensure the correctness of file I/O programs, we make the following two contributions.

`ext4` *Formalization.* As our first contribution, we formalize the semantics of the `ext4` filesystem and study its use by text editors. We focus on `ext4` not only because it is the default filesystem in Linux and thus underpins a plethora of software, but also because it admits a number of peculiar program behaviours due to its optimizations.

Formalizing `ext4` involves several challenges. First, the operations supported are typically determined by the POSIX standard described informally in prose, which is often unclear, ambiguous, and occasionally even self-contradictory. Second, `ext4` is not POSIX-compliant: although it supports the POSIX operations, it does not always guarantee their POSIX-mandated semantics, and its documentation often does not discuss the exact discrepancies. Third, the documentation typically does not distinguish between the *consistency* and *persistency* behaviour of filesystems, where consistency describes the order in which file operations are made visible to concurrent threads/processes, while persistency describes the order in which they reach the disk and are observed upon crash recovery. Fourth, one must account for the interaction between the filesystem semantics and the (weak) consistency semantics of the underlying programming language or architecture.

To tackle these challenges, our formal model is based not only on our reading of the manuals and discussions with filesystem developers, but also on carefully consulting the `ext4` implementation and thorough experimental evaluation. Our model follows the style of the formal persistency models of architectures with non-volatile memory [Raad et al. 2018], which in turn follows the style of axiomatic weak memory models [Alglave et al. 2014]. As such, our model is very flexible: it is easy to integrate it into the existing (weak) memory models such as the C/C++ concurrency model, to extend it with additional constructs, and to adapt it to other filesystems.

Effective Model Checking. As our second contribution, we design and implement an effective model checking algorithm, PerSeVerE, for automatically verifying sequential or concurrent C/C++ programs that perform file I/O using the POSIX system calls. In essence, PerSeVerE enumerates all possible consistent executions of a given program and all its possible post-crash persisted states, and checks whether the supplied assertions/invariants hold. The novel major challenge in doing so is to combat the state space explosion arising from the filesystem semantics.

To see this, consider a sequential program with N independent file operations and no synchronization calls. These operations may persist to disk in any order (i.e., $N!$ ways) and any prefix of such orders may have completed before a crash (i.e., $N \times N!$ possible states). However, this naive enumeration of persistency ordering is far from optimal. A much better way is not to enumerate the orders in which operations persist, and instead to consider whether each of the N operations persisted before the crash (i.e., 2^N states). Moreover, it is typically the case that only $M \ll N$ of operations are relevant for the invariant in question, so it suffices to enumerate 2^M states. When there are synchronization calls, persistency of one operation implies persistency of all prior operations that are separated by a synchronization call, which further reduces the number of states.

Our key idea for exploring this vast state space efficiently is to model the assertions about the persisted state as a *recovery observer* that runs in parallel to the main program P and whose accesses are subject to different consistency axioms from those of P. By ensuring that our axioms do not require a total persistency order, our model checker never enumerates this order explicitly and thus significantly reduces the number of states to explore. Finally, following an axiomatic semantics enables us to integrate our approach into existing efficient algorithms for enumerating the (weak) behaviours of concurrent programs, thereby leveraging the state-of-the-art implementations.

Outline. The paper outline follows the aforementioned technical contributions of this work. We start with an intuitive example-driven description of `ext4` semantics in §2 and present its formal

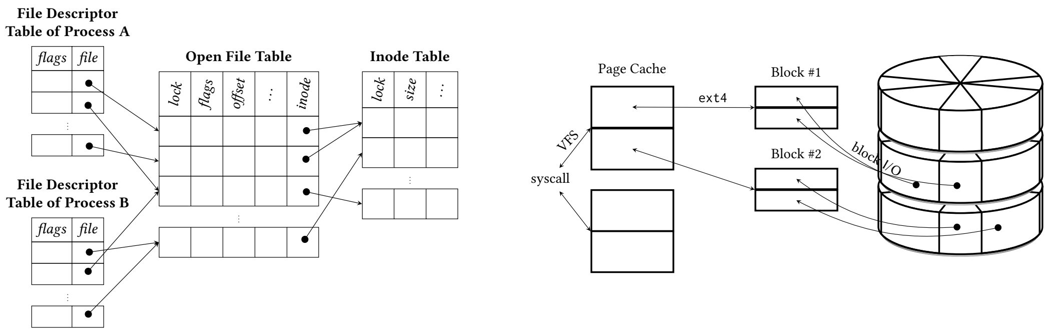


Fig. 1. File descriptors, descriptions and inodes (left); different I/O layers until data reaches disk (right)

model in §3. In §4, we describe our model checking algorithm and subsequently evaluate it in §5 on models of common text editors. In §6, we conclude with a discussion of related work.

2 THE SEMANTICS OF EXT4: AN INTUITIVE ACCOUNT

In order to perform file I/O, applications invoke system calls such as open and read to access the file data on disk. We describe the semantics of such system calls on Linux with the `ext4` filesystem. Wherever possible, we follow the naming conventions from the *Linux man pages* [2020] or the kernel source code. We note that while most of the terminology stems from POSIX [2018], the Linux/`ext4` behaviour often diverges from the POSIX standard.

Assumptions. For simplicity, we assume that all files reside in one directory and that all threads belong to the same process. We thus avoid modelling address translation and pathname lookup studied in [R. Chen et al. 2016; Ntzik et al. 2015]. We further do not model I/O failures and the direct memory access features of `ext4` (e.g., `O_DIRECT`). We proceed with a description of file operations.

2.1 File Operations

Opening a File. Before accessing a file, a process must first open it. This can be done using the open system call, which takes as arguments the file name as well as several flags describing how to open the file: e.g., its access mode (`O_RDONLY`, `O_WRONLY`, `O_RDRW`), whether the file is to be created (`O_CREAT`) or truncated (`O_TRUNC`), and whether subsequent I/O operations must use additional synchronization. A call to open returns a *file descriptor*, a small non-negative integer index into a per-process table maintained by the operating system. File descriptors may be duplicated (e.g., with dup) or shared with other processes (e.g., when passed through a socket or executing fork).

Each file descriptor is mapped to a *file description* entry in a system-wide table of open files, as shown in Fig. 1 (left). A file description stores its lock, file flags (e.g., those above), the current *offset* within the file (when reading/writing the file) and the file metadata (a.k.a. *inode*). This information is per open call and not per file: multiple file descriptions may be associated with the same file.

Reading/Writing a File. Once a file is opened, its contents can be read and written using read and write, respectively. These calls expect three arguments: a file descriptor referring to an open file, a buffer to store the data read/written, and the number of bytes to read/write. Given a file descriptor d_f, we write $r = \text{read}\,(d_f, \texttt{count})$ to read `count` bytes from d_f into buffer r, and $\text{write}\,(d_f, \texttt{buf})$ to write the whole buffer `buf` to d_f. For readability, we omit the error handling code.

The read/write calls access the file at the *offset* specified in the file description. This offset is initialized to 0 when the file is first opened, and is increased by the number of bytes read/written.

For example, given the "`foo.txt`" file containing the string "`foo`", the snippet in Fig. 2 reads the strings "`f`" and "`oo`" into r_0 and r_1, and sets the offset to the end of the file (EOF).

d_f = open ("`foo.txt`", O_RDWR);
r_0 = read (d_f, 1); *// reads "`f`"*
r_1 = read (d_f, 2); *// reads "`oo`"*

Fig. 2. Reading a file

POSIX also supports accessing a file at a given offset using pread and pwrite. These calls take the absolute offset as an additional argument, and do not change the offset stored in the file description. Attempting to read beyond EOF results in reading 0 bytes, while writing beyond EOF extends the file size and fills the gap between EOF and the offset where the new data is to be written with zeros[1].

The Linux-kernel, however, is not fully POSIX-compliant. For example, suppose that "`foo.txt`" above were opened with the O_APPEND flag, which sets the initial offset in the file description to EOF. The pwrite (d_f, "`bar`", 0) call (at absolute offset 0) would then update "`foo.txt`" to contain the string "`foobar`" rather than the expected string "`bar`". This is only one example of non-POSIX-compliant behaviour exhibited by the Linux kernel. In general, the Linux kernel exhibits multiple other non-POSIX-compliant behaviours, which we strive to model precisely.

Seeking in a File. Given a file descriptor d_f, the lseek (d_f, ...) system call updates the offset associated with d_f according to the given arguments: the offset may be set to an absolute value or to a value relative to predefined locations in the file (e.g., the file size or the current location). Indeed, lseek allows the offset to be set *beyond* EOF. Such a call does not alter the file size, but subsequent writes to the file will write at the offset specified by lseek and therefore increase the file size.

Note that if a file is opened with O_APPEND, lseek is of little use since (as per the POSIX standard) subsequent calls to write will reposition the offset to EOF before writing the data.

Closing a File. A file associated with a file descriptor d_f may be closed by calling close (d_f), which removes d_f from the file descriptor table of the calling process. If d_f is the only file descriptor associated with an open file description, then its resources are freed; otherwise, the file description is preserved so long as there are other file descriptors (in any process) associated with it.

Directory Operations. Several system calls, referred to as *directory operations* in our model, can be used to manipulate the file inode and the directory containing the file. Examples of such operations in our model are: (1) creat (nl), which creates a new file named nl (and is equivalent to open with O_CREAT|O_WRONLY|O_TRUNC flags); (2) link (nl_{old}, nl_{new}), which creates a new directory entry with name nl_{new} (if such entry does not already exist) referring to nl_{old}'s inode; (3) unlink (nl), which deletes the entry named nl; and (4) rename (nl_{old}, nl_{new}), which renames the entry named nl_{old} as nl_{new}. That is, rename (nl_{old}, nl_{new}) is similar to link (nl_{old}, nl_{new}) followed by unlink (nl_{old}).

Such operations exhibit interesting behaviours when interacting with open file descriptions. To see this, consider the following program, where "`bar.txt`" is unlinked immediately after creation:

d_f = creat ("`bar.txt`"); unlink ("`bar.txt`"); write (d_f, "`bar`");

The question is whether the subsequent write is valid. When a file is unlinked, if the removed entry is the last entry on the file, then the file must be deleted with its allocated space made available for reuse. However, if the file is still open, then unlink does not delete it immediately; instead, it returns an error, and the file is eventually deleted once all its associated file descriptions are closed. As such, close and unlink may execute in either order with respect to one another, and processes with open file descriptions on the file can read/write the file until they close their file descriptions.

Similar observations hold of the interaction of other directory operations with the I/O operations described thus far. In general, directory operations on a file with open file descriptions do not

[1]Most modern filesystems store *sparse files* more compactly, which typically avoids writing these zeros to disk.

hinder subsequent I/O calls that use these descriptions. They may, however, affect the outcome of subsequent I/O calls: e.g., if we call creat on an open file, creat will truncate the file size to 0.

2.2 The I/O Stack in Linux

We next present the basics of the kernel's I/O stack. We do not describe all aspects of the I/O stack; rather, we present an overview of the basic layers through which disk-bound data travels.

VFS and Page Cache. In Linux, data goes through several layers before reaching the disk, as shown in Fig. 1 (right). The first two layers are the *Virtual File System* (VFS) [Gooch 1999] and the kernel *page cache*. VFS is an abstract software layer in the kernel that provides a common API to different filesystem implementations; the page cache sits between VFS and the filesystem implementation (in our case `ext4`), and its purpose is to cache disk data in memory. The page cache comprises physical RAM pages, which in turn contain a number of disk *blocks* (see below).

In most cases, interacting with VFS/page cache does not imply interacting with the disk. For instance, a call to write does not guarantee that the written data is persisted to disk prior to returning, and a call to read does not necessarily fetch the desired data directly from the disk. Instead, both I/O operations simply manipulate the page cache; in fact, with the page cache, the VFS has to barely touch filesystem-specific code, if all the desired data is in the page cache already.

More concretely, upon a call to read, the kernel checks the page cache for the desired data, and *only if* (part of) this data is absent from the page cache, does the kernel fetch it directly from the disk. Similarly, upon a call to write, the kernel writes the data to the page cache and schedules the data to be persisted to disk *asynchronously* (i.e., it is a write-back cache). That is, upon returning from write, the written data may be pending to persist to disk. As such, to ensure that the written data reaches the disk, one must use special *flushing* system calls (see §2.4.3) to commit the persist-pending data to disk *synchronously*: simply closing the file does not commit the pending writes to disk.

Apart from being greatly beneficial for performance, the VFS and the page cache largely determine `ext4`'s consistency semantics, as we describe in §2.3.

Filesystems and Block I/O. We next describe what happens when a process tries to read data that is not in the page cache, or when a page in the page cache is to be written back to disk. VFS cannot handle such cases as each filesystem stores data on disk in a specific format. Therefore, filesystem-specific code is called in such cases to interact with the disk.

This interaction with the disk must account for the physical properties of the underlying hardware; see Fig. 1 (right). These properties are important as they affect how data is transferred to disk. The smallest addressable unit on a disk is a *sector*. It holds `4KiB` in modern disks [*Advanced Format* 2020] and `512B` in older disks. However, the filesystem communicates with the disk in logical units called *blocks*, denoting contiguous disk sectors (typically `4KiB` in Linux). For instance, to write a page from the page cache to disk, the OS issues a sequence of I/O requests over a number of blocks.

In general, these requests go through additional layers in the kernel; the part of the kernel that implements the interface between the filesystem and the hardware is called *the block I/O layer*. The block I/O layer is important for persistency properties: it can merge or reorder writes before passing them to lower layers; i.e., it affects the `ext4` persistency semantics, as we discuss in §2.4.

2.3 Consistency of File Operations

We now discuss the *consistency* semantics of file operations, describing the order they become visible to concurrent threads; we write c-atomic and c-ordering to refer to the atomicity and ordering guarantees at the level of consistency. Although POSIX [2018] requires that file operations be c-atomic, this is not honoured by the Linux kernel.

Writes. Linux provides strong c-atomicity guarantees for writes. Specifically, writing to a file involves acquiring the file's *inode lock*, ensuring that all writes to the same file are c-ordered with respect to one another. As each file is associated with a unique inode lock, mutual exclusion is guaranteed regardless of the file descriptor used to carry out the write. Moreover, a call to write (as opposed to pwrite) also acquires the *offset lock* in the file description, making the combination of the offset adjustment and the data write one big c-atomic step. For example, the pread operation in the program below reads either "bar" or "qux", since both pwrite operations acquire the inode lock:

$$\mathsf{pwrite}\,(d_f, \text{“bar”}, 0); \;\Big\|\; \mathsf{pwrite}\,(d_f, \text{“qux”}, 0); \\ r = \mathsf{pread}\,(d_f, 3, 0);$$

Reads and Overwrites. The c-atomicity guarantees of reads are more subtle: while read calls acquire the offset lock in the file description, pread calls acquire no locks. Therefore, a read call is c-ordered with respect to concurrent read/write calls on the *same file description* as they compete to acquire the offset lock. By contrast, concurrent read calls on different file descriptions and pread calls do not acquire a common lock and thus offer only *byte-level c-atomicity.* As such, if the file of d_f initially contains "foo", the pread call below may read "foo", "bar", "far", "fao", and so forth.

$$\mathsf{pwrite}\,(d_f, \text{“bar”}, 0); \;\Big\|\; r = \mathsf{pread}\,(d_f, 3, 0);$$

Reads and Appends. When racing with a write *appending* to a file, reads have stronger c-atomicity guarantees: reads consult the file size, which is modified by appends, resulting in stronger synchronization. In general, appends increase the file size not at once but incrementally: they write cache pages one at a time, increasing the file size after each page. Reads first read the file and then the data, and may thus observe the incremental size increases, at the granularity of the page size.

For example, assuming that each page is 3 bytes and that "foo.txt" containing "foo" is opened with O_APPEND, the pread in the program below can read "foo", "foobar" or "foobarqux".

$$\mathsf{write}\,(d_f, \text{“barqux”}); \;\Big\|\; r = \mathsf{pread}\,(d_f, 42, 0);$$

Note that when a read requests more data than available in the file, it reads as much data as it can.

This behaviour is also observable for lseek and appends: if one thread appends multiple pages to a file and another concurrently seeks to EOF, lseek may set the offset to an intermediate file size.

Directory Operations. Directory operations provide strong consistency guarantees: they are c-atomic against operations that manipulate the same inode (as they acquire the inode lock), as well as against other directory operations on the same directory (as they acquire the directory's inode lock). This gives strong consistency guarantees to file creation, linking, unlinking, etc.

One interesting exception is the rename (nl_{old}, nl_{new}) call when a file with name nl_{new} already exists. Recall that rename is analogous to link followed by unlink. While the link part is c-atomic in that nl_{new} always points to one of the two inodes, rename as a whole is not c-atomic because there is a window in which both nl_{old} and nl_{new} refer to the same inode [Linux man pages 2020].

Nevertheless, rename provides a mechanism for ensuring update c-atomicity as shown below:

```
d_b = creat ("foo.tmp");                 ||  d_f = open ("foo.txt", O_RDONLY);
write (d_b, "bar"); close (d_b);         ||  r = read (d_f, 3);     // reads "foo" or "bar"
rename ("foo.tmp", "foo.txt");           ||
```

As before, suppose "foo.txt" initially contains "foo". Regardless of whether open sees the new or the old version of "foo.txt", it can seamlessly read the data in the next step. Even if rename happens in between the open and read calls, the right thread still reads "foo" as the file description of d_f still points to the same inode even after rename. This inode, albeit no longer accessible via the "foo.txt" name after the rename, will not be deleted until all references to it are deleted.

2.4 Persistency of File Operations Under `ext4`

We next discuss the *persistency* semantics of file operations, describing the order their effects persist to disk, thus determining the observable disk states upon recovery from a crash (e.g., power loss or software crash). We write p-atomic and p-ordering to refer to the atomicity and ordering guarantees at the level of persistency. Although similar effects can be observed in other filesystems, the description here is `ext4`-specific. We proceed with a brief description of the journalling mechanism in `ext4`, which maintains the filesystem in a consistent state in case of a crash.

Journalling in `ext4`. As a crash can occur at any time during the program execution, including during a system call, a filesystem must guarantee data integrity. Such guarantees do not pre-empt data loss, but merely ensure that the filesystem can be restored to a consistent state after a crash. For example, when appending to a file, the file size update must not persist before the appended data: if a crash occurs right after the size update persists, then invalid data can be read upon recovery.

To ensure data integrity, `ext4` employs *write-ahead logging* or *journalling* [Tweedie 1998], which uses a transaction to record the intended changes in a *journal* (a designated disk area) before carrying out the changes. Once the transaction *commits*, the intended changes can be carried out in place. This way, if a crash occurs while enacting the changes, upon recovery one can simply replay the journal to bring the filesystem to a consistent state. By default, `ext4` journals only metadata, e.g., the on-disk file inode (see §2.5). As `ext4` stores the file metadata in a different place on disk than its data [Linux kernel 2020], this introduces dependencies between file data and metadata, leading to interesting persistency behaviours discussed below.

2.4.1 Persistency of I/O Operations. We proceed with the persistency guarantees of I/O operations. As file reads do not alter the persistent disk data[2], persistency guarantees are only meaningful for file writes and directory operations. The persistency guarantees of an `ext4` file write depend on whether it modifies the file size, and thus differ for *overwrites* and *appends*.

Overwrites. `ext4` provides very weak persistency guarantees for overwrites. First, it does not guarantee p-atomicity beyond what is provided by the underlying storage, which is typically sector-level p-atomicity for hard drives, but may only be byte-level p-atomicity for other persistent storage media (e.g., non-volatile memory). As such, many applications such as SQLite [*SQLite* 2020] do not assume such sector-level p-atomicity [*Atomic Commit In SQLite* 2020]. For a filesystem to guarantee p-atomicity e.g., at the block level, it must use techniques such as full *data journalling* (see §2.5) or copy-on-write [*Copy-on-write* 2020], which are not employed by `ext4` by default.

Second, even the order in which overwrites to different sectors are persisted is loosely constrained: writes to sectors within a block persist in order, whereas writes to different blocks may persist in an arbitrary order. That is, although a write issues I/O requests for writing blocks in order, these writes may be freely reordered both by the block I/O layer of the kernel and the disk itself (e.g., to minimize rotation), and by default `ext4` does not prevent this reordering. For example, consider the following overwrite on d_f associated with file "`foo.txt`" with initial contents "`foo`":

$$\mathsf{pwrite}\,(d_f, \text{“}\mathtt{bar}\text{”}, 0); \qquad (\textsc{ow-na})$$

Let us assume that the sector size is one byte and each block comprises 3 sectors. If a crash occurs during ow-na, then only a *prefix* of "`bar`" may persist to disk before the crash, guaranteeing *sector-level* p-atomicity. That is, in the post-crash disk state "`foo.txt`" may contain "`foo`", "`boo`", "`bao`" or "`bar`", but not "`fao`" or "`far`". This is because blocks are composed of contiguous sectors, and sectors within a block are written in a linear order. If, however, each block contains only one sector and a crash occurs, then outcomes such as "`fao`" and "`far`" are also possible.

[2] We do not model metadata potentially affected by reads such as file access times.

In our formal model, we keep the sector and block sizes as parameters; we treat sector writes as p-atomic and assume that sectors constituting a block are written in a linear order.

Appends. ext4 offers stronger persistency guarantees for appends: it guarantees *block-level* p-atomicity of append *prefixes* and that appends on the same file are p-ordered, as described below.

The block-level p-atomicity guarantees of appends are best seen with an example. Suppose that a file occupies n blocks on disk and a crash occurs while appending k more blocks to it. In the post-crash disk state the file may then contain $n + i$ blocks, where $0 \leq i \leq k$; i.e., a (potentially full) prefix of the appended blocks may persist to disk before the crash. For instance, had we opened "foo.txt" in OW-NA with the O_APPEND flag, assuming that the block size is 3B, in the post-crash disk state "foo.txt" would contain either "foo" or "foobar" – recall from § 2.1 that opening "foo.txt" with O_APPEND ignores the absolute offset of pwrite and simply appends to it.

At first glance, this may seem incompatible with the weak p-ordering guarantees of overwrites. If block writes are not p-ordered, how does ext4 ensure that the size update persists after the appended data? This is enabled by the *journal*: when ext4 journals the metadata updates of an append (e.g., its size update), it *binds* the commit of the transaction containing the metadata, with the persist of the associated data. That is, the size update transaction commits only once the appended data persists. As such, if a crash occurs before the transaction commits, the file size on disk will not have been updated, and thus the persisted append data (if any) will not be accessible. Note that in some cases a transaction has to commit before all data is persisted (e.g., because it has grown very large), thus allowing it to commit only with a *prefix* of the data persisted.

However, there is one notable case where these p-atomicity guarantees do not apply: when a file has preallocated blocks that are partially filled. For example, suppose that file "f" has its last block b_l already allocated on disk but not completely filled, and a crash occurs while a process performs an append to "f". Upon recovery, it is then possible to observe a disk state where the size of the file is partially increased (either to the end of b_l or to the size dictated by the append, whichever is smaller), but with '0's appended to the end instead of the data written, as shown in Fig. 3.

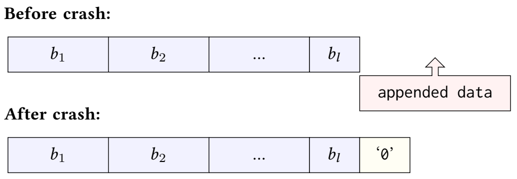


Fig. 3. ext4 p-atomicity violation: delayed allocation

This is because, as part of an optimization, ext4 does *not* bind the data persist of the *preallocated blocks* to the transaction commit. Before elaborating on this optimization, let us briefly discuss block allocation under ext4. Recall that writes happen asynchronously under ext4. This is to ensure that writes to the same page are merged, and that write blocks can be allocated more efficiently (a process called *delayed allocation*). Block allocation for an inode, however, requires that the inode be journalled to ensure filesystem consistency. Conversely, if a block is already preallocated when a write is issued, then the inode need not be journalled.

Thus, when extending files with preallocated blocks, ext4 optimizes the number of times an inode is journalled. Since the inode is already included in the transaction when a write to a preallocated block is issued (as the inode contains other metadata that are journalled), ext4 journals the inode with the updated size in the transaction, thus not journalling the same inode twice. However, it does so without binding the transaction commit to the data persist to avoid stalling the transaction. As such, this optimization decouples the on-disk size update from the data persist, thus allowing the transaction to commit before the data persists, leading to the behaviour shown in Fig. 3.

Finally, `ext4` guarantees that appends to the *same file* are p-ordered. This is because as discussed `ext4` updates the on-disk file size only after all earlier (i.e., c-ordered-before) writes have persisted to disk. However, appends to *different files* may not persist in the order they are issued. For instance, the append to "`b.txt`" in Fig. 4 may persist before that to "`a.txt`".

d_a = creat ("`a.txt`");
d_b = creat ("`b.txt`");
write (d_a, ...);
write (d_b, ...);

Fig. 4. (APP-DIFF-FILES)

2.4.2 Persistency of Directory Operations. `ext4` offers stronger persistency guarantees for directory operations than file operations. Specifically, all directory operations are p-atomic (including rename) and if a metadata-affecting operation persists, then all directory operations c-ordered before it will have persisted too. Metadata-affecting operations comprise all operations but overwrites, as overwrites do not affect the metadata we model (i.e., file size). Although not strictly a directory operation, similar guarantees are provided for file truncations (e.g., open with O_TRUNC).

Intuitively, these guarantees are achieved because directory operations only affect the metadata of the file inode (as well as the directory inode), and are thus *synchronously* journalled in the current transaction. In turn, this enforces p-ordering between directory operations and subsequent metadata-affecting operations, because any metadata-affecting operation that is c-ordered after a directory operation is either in the same transaction as the directory operation, or in a subsequent one (in contrast to overwrites, which are not bound to any transaction).

These persistency guarantees have often been the source of confusion and sparked many discussions among application and filesystem developers (e.g., [*Ext4 data loss* 2009; ext4 corruption 2015]). A common such discussion involves the "replace-via-rename" pattern, shown in Fig. 5. Assuming that "`foo.txt`" already exists, this pattern is used to update the contents of "`foo.txt`" (e.g., if "`foo.txt`" contains a log).

d_f = creat ("`foo.tmp`");
write (d_f, ...);
close (d_f);
rename ("`foo.tmp`", "`foo.txt`");

Fig. 5. (REPLACE-VIA-RENAME)

As the last instruction is responsible for renaming "`foo.tmp`" (thus updating "`foo.txt`"), developers commonly expect that upon crash recovery "`foo.txt`" will contain either its old data or the new data written by write. Unfortunately, however, this may not be the case. While directory operations are p-ordered with respect to *later* operations, they are not p-ordered with respect to *earlier* operations. As such, rename may persist before the write on "`foo.tmp`" (recall that appends happen asynchronously). Consequently, if a crash occurs after the rename persists but before the write does, then the new content will be lost upon crash recovery, leading to a zero-sized "`foo.txt`". Put differently, this is analogous to writing to different files: rename writes to the *directory inode*, while write writes to the file inode. As discussed in § 2.1, such writes are freely reordered under `ext4`, and one must explicitly flush the data to disk by using special instructions (see § 2.4.3).

When `ext4` was first introduced in Linux distributions, many users experienced data loss due to applications relying on this assumption [*Ext4 data loss* 2009]. This confusion has even led to the creation of specific programs aiming at the correct (i.e., p-atomic) renaming of files [*renameio* 2020]. Even though such guarantees were never made by POSIX, `ext4` or its predecessors (`ext2` and `ext3`), applications relied on `ext2` and `ext3` providing them, and inevitably suffered data loss when `ext4` became the default option in the kernel. This drove `ext4` developers to employ heuristics that detect patterns such as "replace-via-rename", which do minimize the chances of data loss but do not eliminate it. Indeed, as we show in §5, relying on such heuristics may itself lead to data loss.

2.4.3 Enforcing P-ordering. To enforce a particular p-ordering, developers can use special disk-flushing instructions or flags. Specifically, the sync and fsync system calls can be used to flush persist-pending (data and metadata) writes to disk. Concretely, sync flushes all persist-pending writes across the *entire filesystem synchronously*: it waits for I/O to complete before returning.

Analogously, given a file descriptor d_f, a call to fsync (d_f) synchronously flushes all persist-pending writes on d_f. For instance, we can avoid data loss in Fig. 5 by inserting fsync (d_f) before close (d_f).

The effects of fsync can be emulated by opening a file with the `O_SYNC` *flag*. Using `O_SYNC`, all blocks written to the file by a write/pwrite call are flushed immediately after the call. In practice, the guarantees of `O_SYNC` are slightly stronger in that *all* c-ordered-earlier writes to the file are flushed, including those of the write request. For instance, consider a scenario where a process A opens a file "`f.txt`" without `O_SYNC`, while a concurrent process B opens "`f.txt`" with `O_SYNC`. If first A and then B each write several blocks to "`f.txt`", then all persist-pending blocks written to "`f.txt`" up to and including those of process B are flushed to disk, including those of A.

2.5 Other Data Journalling Modes in ext4

The p-ordering guarantees described so far are those of the default journalling mode in `ext4` (i.e., `data=ordered`). Additionally, `ext4` offers two other journalling modes, prescribing how data and metadata are written to disk, further complicating its persistency guarantees.

The first mode, `data=journal`, provides stronger guarantees than the default mode, but greatly degrades system performance [ext4 benchmarks 2012; Prabhakaran et al. 2005]. It journals both the data and the metadata of writes before writing them to their final disk locations (compare this with `data=ordered` which only journals the metadata). As such, as everything goes through the journal with `data=journal`, all c-ordered writes (overwrites and appends) are p-ordered and block writes are also p-atomic. Writes in general, however, are not p-atomic. This is because the journal occupies finite disk space, and a write/pwrite call may write more data than the journal can accommodate.

The second mode, `data=writeback`, provides weaker guarantees than `data=ordered`: it only journals the metadata (as in `data=ordered`), but the metadata write is not bound to the data persist. As such, `data=writeback` only guarantees that (1) directory operations are p-ordered with respect to later metadata-affecting operations; and that (2) directory operations are p-atomic. This is because directory operations only affect an inode's metadata, and thus waiting on data is unnecessary. However, `data=writeback` may observe stale data when, e.g., appending to a file: if the new file size is written to the journal, the transaction commits, and then a crash occurs before the data persists, stale data in the space between the old and the new size can be observed. Therefore, while `data=writeback` offers better performance than `data=ordered`, it raises security concerns.

3 FORMAL MODEL

We now present our first contribution: a formal model of `ext4`'s semantics.

Programming Language. Our concurrent programming language is given in Fig. 6 (below). We assume a finite set Val of values, a finite set Tid of thread identifiers and any standard interpreted language for expressions, Exp, containing values. We use v as a metavariable for values, t for thread identifiers, and e for expressions. We model a multi-threaded program P as a function mapping each thread to its (sequential) program. We write $\mathsf{P}{=}\mathsf{C}_1 || \cdots || \mathsf{C}_n$ when $dom(\mathsf{P}){=}\{t_1 \cdots t_n\}$ and $\mathsf{P}(t_i){=}\mathsf{C}_i$. Sequential programs are described by the Comm grammar and include expressions (e) and *system calls* (c), as well as the standard constructs of sequential composition, conditionals and loops.

File Representation: Memory versus Disk. As discussed in §2.2, the page cache stores (a part of) a file in memory. As the page cache is written back to disk asynchronously (§2.3), at any point, there are two representations of a file, one in memory and one on disk, that may not agree with one another. Moreover, recall that under `ext4` the file metadata (e.g., size) is stored in a different location than its data (§2.4). We thus use the domains in Fig. 6 (above) to model the file representation.

Specifically, we define a set of *on-disk file locations*, Floc, storing one byte of file data: each file location is a pair $dl{=}(f, o)$, where $f \in$ Inode is the *file inode* and $o \in$ Offset is the *offset*, denoting

Disk Domains

$\mathsf{Floc} \triangleq \mathsf{Inode} \times \mathsf{Offset}$	file locations
$nl \in \mathsf{Dnameloc} \triangleq \mathsf{String}$	file name locations on disk
$ds_f \in \mathsf{Dsizeloc}$	file size locations on disk†
$dl \in \mathsf{Dloc} \triangleq \mathsf{Floc} \uplus \mathsf{Dnameloc} \uplus \mathsf{Dsizeloc}$	disk locations

Memory Domains

$ml \in \mathsf{Mloc}$	memory locations
$d_f \in \mathsf{Fd}$	file descriptors where $f \in \mathsf{Inode}$
$ms_f \in \mathsf{Mloc}$	file size locations in memory†
$ol_{d_f} \in \mathsf{Mloc}$	file offset locations in memory‡

General Domains

$f \in \mathsf{Inode} \triangleq \mathbb{N}$	file inodes
$id \in \mathsf{Id} \triangleq \mathbb{N}$	operation ids
$t \in \mathsf{Tid} \triangleq \mathbb{N}$	thread ids
$o \in \mathsf{Offset} \triangleq \mathbb{N}$	offsets
$\mathsf{Loc} \triangleq \mathsf{Mloc} \cup \mathsf{Dloc}$	locations
$v \in \mathsf{Val}$	byte values
$lck \in \mathsf{Lck} \triangleq \mathsf{Inode} \cup \mathsf{Fd} \cup \{dir\}$	lockables

$\dagger$: defined for all $f \in \mathsf{Inode}$
$\ddagger$: defined for all $d_f \in \mathsf{Fd}$

$\mathsf{Syscal} \ni \mathsf{c} ::= \mathsf{sync}\,(\,) \mid \mathsf{fsync}\,(d_f) \mid \cdots$ (see Fig. 7 for a full list of system calls)

$\mathsf{Comm} \ni \mathsf{C} ::= \mathsf{e} \mid \mathsf{c} \mid \mathsf{C};\mathsf{C} \mid \textbf{if}(\mathsf{e})\ \textbf{then}\ \mathsf{C}_1\ \textbf{else}\ \mathsf{C}_2 \mid \textbf{while}(\mathsf{e})\ \mathsf{C}$ $\quad\mathsf{Exp} \ni \mathsf{e} ::= v \mid \cdots$

$\mathsf{Prog} \ni \mathsf{P} ::= \mathsf{Tid} \xrightarrow{\text{fin}} \mathsf{Comm}$

Fig. 6. The PerSeVerE types and their metavariables (above); the PerSeVerE programming language (below)

that dl contains the o^{th} byte of the f data. We define a set of *on-disk* size locations, Dsizeloc, and a mapping from inodes to their on-disk size locations; for brevity, we omit this mapping and write ds_f for the on-disk size location of f. Recall that we assume a single-directory structure, which we model by the designated dir directory. We thus define a set of on-disk *file name locations*, $\mathsf{Dnameloc} \triangleq \mathsf{String}$, where each $nl \in \mathsf{Dnameloc}$ records the inode associated with the file name nl.

Similarly, we model the in-memory file representation via a set of *memory locations*, Mloc, and define a mapping from inodes to their in-memory size locations; we omit this mapping and write ms_f for the in-memory size location of inode f. We further define a set of *file descriptors*, Fd, and a mapping from file descriptors to their inodes; we omit this mapping and simply write d_f to denote that the descriptor d is associated with inode f. We further assume a mapping from file descriptors to their (in-memory) offset locations, and write ol_{d_f} for the offset location of file descriptor d_f.

Disk Sectors and Blocks. To keep our formalism general, we do not define an explicit size for disk sectors and blocks, as these are determined by the underlying filesystem. However, as discussed in §2, `ext4` provides certain guarantees on the same-sector/same-block accesses. To this end, we assume the existence of two *equivalence relations*, $\mathtt{ssec} \subseteq \mathsf{Dloc} \times \mathsf{Dloc}$ and $\mathtt{sblk} \subseteq \mathsf{Dloc} \times \mathsf{Dloc}$, relating the disk locations on the *same sector* and the *same block*, respectively.

Execution Events. Recall from § 2.3 that different system calls provide different c-atomicity guarantees, determined by the underlying synchronization mechanisms used (e.g., locks). That is, each system call may comprise several fine-grained instructions such as acquiring a lock.

As is common in the literature of declarative concurrency models, we model the traces generated by a program P as a set of *execution graphs*, where each graph node is an *event* corresponding to a fine-grained instruction (e.g., lock acquisition) associated with a system call. We shortly describe the mapping from system calls to their corresponding events (fine-grained instructions), and formally define the notion of execution graphs. We proceed with the formal definition of events.

An event e is a tuple of the form $\langle n, t, id, l\rangle$, where $n \in \mathbb{N}$ is an *event identifier* uniquely identifying e in an execution, $t \in \mathsf{Tid}$ denotes the thread associated with e, $id \in \mathsf{Id}$ is an *operation identifier* denoting the system call associated with e, and l is an event *label*, describing the event type as defined below in Def. 3.1. Note that the operation identifier allows us to track all events associated with a system call: all events corresponding to a system call c have the *same* operation identifier.

Definition 3.1 (Events). An *event* is a tuple $\langle n, t, id, l\rangle$, where $n \in \mathbb{N}$, $t \in \mathsf{Tid}$, $id \in \mathsf{Id}$ and l is an event *label* with one of the following forms: (1) $\mathtt{MR}(ml)$ for a memory read from ml; (2) $\mathtt{MW}(ml, v)$ for a memory write to ml with value v; (3) $\mathtt{DR}(dl)$ for a disk read from dl; (4) $\mathtt{DW}^{m}(dl, v)$ for a disk write to dl with value v and *mode* $m \in \{\mathsf{norm}, \mathsf{trunc}, \mathsf{zero}, \mathsf{rename}\}$; (5) $\mathtt{FS}(f)$ for a file sync on f; (6) $\mathtt{S}()$ for a sync; (7) $\mathtt{Open}(d_f, f)$ for opening f and yielding d_f; (8) $\mathtt{Close}(d_f, f)$ for closing d_f associated with f; (9) $\mathtt{L}(lck)$ for locking lck; (10) $\mathtt{U}(lck)$ for releasing lck.

The modes denote the event origin; e.g., rename for a disk write generated by rename (see §3.1). The set of *memory reads* is $\mathtt{MR} \triangleq \left\{\langle n, id, t, l\rangle \;\middle|\; l{=}\mathtt{MR}(.)\right\}$. The sets of *memory writes* ($\mathtt{MW}$), *disk reads* ($\mathtt{DR}$), *disk writes* ($\mathtt{DW}$), *syncs* ($\mathtt{S}$), *file syncs* ($\mathtt{FS}$), *locks* ($\mathtt{L}$) and *unlocks* ($\mathtt{U}$), as well as the sets $\mathtt{DW}^{\mathsf{trunc}}$, $\mathtt{DW}^{\mathsf{rename}}$, $\mathtt{DW}^{\mathsf{zero}}$ and $\mathtt{DW}^{\mathsf{norm}}$ are defined analogously. The sets of *reads* and *writes* are defined as $\mathtt{R} \triangleq \mathtt{MR} \cup \mathtt{DR}$ and $\mathtt{W} \triangleq \mathtt{MW} \cup \mathtt{DW}$, respectively; the set of *durable events* is defined as $\mathtt{D} \triangleq \mathtt{DW} \cup \mathtt{S} \cup \mathtt{FS}$. The set of all events is $\mathtt{Events}$.

Notation. Given an event $e{=}\langle n, t, id, l\rangle$, we write $\mathtt{eid}(e)$, $\mathtt{tid}(e)$, $\mathtt{id}(e)$ and $\mathtt{lab}(e)$ to project its components, respectively. Analogously, given a label l, we write $\mathtt{loc}(l)$, $\mathtt{val}(l)$ and $\mathtt{lck}(l)$, when applicable. For instance, when $l = \mathtt{MW}(ml, v)$ then $\mathtt{loc}(l){=}ml$ and $\mathtt{val}(l){=}v$. We lift $\mathtt{loc}(.)$, $\mathtt{val}(.)$ and $\mathtt{lck}(.)$ to events and write e.g., $\mathtt{loc}(e)$ for $\mathtt{loc}(\mathtt{lab}(e))$.

Given a relation r, we write $r^{?}$ and r^{+} for the reflexive and transitive closures of r, respectively. We write r^{-1} for the inverse of r, $r|_A$ for $r \cap (A \times A)$, $[A]$ for the identity relation $\{(a, a) \mid a \in A\}$, and $r_1; r_2$ for the composition of r_1 and r_2: $\{(a, b) \mid \exists c.\, (a, c) \in r_1 \wedge (c, b) \in r_2\}$. When A is a set of events, $x \in \mathsf{Loc}$ and $X \subseteq \mathsf{Loc}$, we define $A_x \triangleq \{e \in A \mid \mathtt{loc}(e){=}x\}$ and $A_X \triangleq \bigcup_{x \in X} A_x$. We also define $r_x \triangleq r|_{\mathtt{Events}_x}$ and $r_X \triangleq r|_{\mathtt{Events}_X}$.

3.1 Mapping System Calls to Events

We next describe the mapping from system calls to (sequences of) events. For clarity, we describe this correspondence algorithmically in Fig. 7, where each system call is mapped to an eponymous procedure that generates a (potentially singleton) sequence of events. In our algorithmic description we use several helper functions (e.g., freshFD) that do not generate any events; as such, we omit their algorithmic description and describe their behaviour intuitively, as necessary. Moreover, we assume that the file descriptors supplied as arguments are valid and forgo their validity checks.

We write $\rightsquigarrow l$ to denote generating an event $e{=}\langle n, t, id, l\rangle$ with label l, and omit the associated event, thread and operation identifiers. Instead, we assume that:

(1) events are generated with unique event identifiers in an increasing order;
(2) events corresponding to system calls by thread t are associated with t; and
(3) all events corresponding to a given system call have the same operation identifier.

For brevity, we write $\rightsquigarrow l_1; l_2$ as a shorthand for $\rightsquigarrow l_1$ followed by $\rightsquigarrow l_2$. Generating a read event r additionally yields the *value* read by r. We thus write $\rightsquigarrow l$ **in** v to denote that l reads value v.

For clarity and concision, we exclude several features discussed in §2 from our formalism here. Specifically, (1) we exclude the O_SYNC and O_APPEND flags when opening a file; (2) we focus on the default `ext4` journalling mode (`data=ordered`) and exclude `data=journal` and `data=writeback`. Nevertheless, we cover all these features in our implementation of PerSeVerE (§4).

We further assume that one page of the page cache accommodates exactly one block.

Open and Close. A call to open $(nl, flags)$ comprises three parts: lookup (Line 2), open (Lines 3–5) and truncate (Lines 6–8). The LOOKUP$(nl, flags)$ routine returns the inode f associated with nl (if any) and the (potentially updated) flags. The inode f returned is determined by the flags and whether a file named nl exists. If O_CREAT is not specified in the flags, LOOKUP reads the inode f associated with nl (Line 3) and returns f and the unchanged flags (Line 13). If O_CREAT is specified and nl does not exist, i.e., nl is not associated with an inode (Lines 6 and 7), then LOOKUP creates a

```
1:  procedure LOOKUP($nl$, $flags$)
2:    if O_CREAT $\notin$ $flags$ then
3:      $\rightsquigarrow$ DR($nl$) in $f$
4:    else
5:      $\rightsquigarrow$ L($dir$)
6:      $\rightsquigarrow$ DR($nl$) in $f$
7:      if $f = \bot$ then
8:        $f \leftarrow$ freshINode()
9:        $flags \leftarrow flags \setminus$ O_TRUNC
10:       $\rightsquigarrow$ DW$^{\text{norm}}$($ds_f$, 0) ; MW($ms_f$, 0)
11:       $\rightsquigarrow$ DW$^{\text{norm}}$($nl$, $f$)
12:     $\rightsquigarrow$ U($dir$)
13:   return ($f$, $flags$)

1:  procedure open($nl$, $flags$)
2:    ($f$, $flags$) $\leftarrow$ LOOKUP($nl$, $flags$)
3:    if $f = \bot$ then return
4:    $d_f \leftarrow$ freshFD($f$)
5:    $\rightsquigarrow$ MW($ol_{d_f}$, 0) ; Open($d_f$, $f$)
6:    if O_TRUNC $\in$ $flags$ then
7:      $\rightsquigarrow$ L($f$) ; DW$^{\text{trunc}}$($ds_f$, 0)
8:      $\rightsquigarrow$ MW($ms_f$, 0) ; U($f$)

1:  procedure lseek($d_f$, $o$)
2:    $\rightsquigarrow$ L($d_f$)
3:    $\rightsquigarrow$ MR($ol_{d_f}$) in $o'$
4:    $\rightsquigarrow$ MW($ol_{d_f}$, $o$)
5:    $\rightsquigarrow$ U($d_f$)

1:  procedure link($nl_{old}$, $nl_{new}$)
2:    $\rightsquigarrow$ L($dir$)
3:    $\rightsquigarrow$ DR($nl_{new}$) in $f'$  assert($f' = \bot$)
4:    $\rightsquigarrow$ DR($nl_{old}$) in $f$
5:    $\rightsquigarrow$ DW$^{\text{norm}}$($nl_{new}$, $f$)
6:    $\rightsquigarrow$ U($dir$)

1:  procedure unlink($nl$)
2:    $\rightsquigarrow$ L($dir$) ; DW$^{\text{norm}}$($nl$, $\bot$) ; U($dir$)

1:  procedure rename($nl_{old}$, $nl_{new}$)
2:    $\rightsquigarrow$ L($dir$)
3:    $\rightsquigarrow$ DR($nl_{old}$) in $f$
4:    $\rightsquigarrow$ DW$^{\text{rename}}$($nl_{new}$, $f$) ; DW$^{\text{rename}}$($nl_{old}$, $\bot$)
5:    $\rightsquigarrow$ U($dir$)

1:  procedure close($d_f$) $\rightsquigarrow$ Close($d_f$)

1:  procedure sync () $\rightsquigarrow$ S()

1:  procedure fsync($d_f$) $\rightsquigarrow$ FS($f$)

1:  procedure BUFFERREAD($f$, $buf$, $count$, $o$)
2:    $\rightsquigarrow$ MR($ms_f$) in $size$
3:    $m \leftarrow$ min($count$, $size - o$)
4:    for $i = 0$ to $m - 1$ do
5:      $\rightsquigarrow$ DR(($f$, $o + i$)) in $buf[i]$

1:  procedure pread($d_f$, $buf$, $count$, $o$)
2:    BUFFERREAD($f$, $buf$, $count$, $o$)

1:  procedure read($d_f$, $buf$, $count$)
2:    $\rightsquigarrow$ L($d_f$)
3:    $\rightsquigarrow$ MR($ol_{d_f}$) in $o$
4:    BUFFERREAD($f$, $buf$, $count$, $o$)
5:    $\rightsquigarrow$ MW($ol_{d_f}$, $o + count$)
6:    $\rightsquigarrow$ U($d_f$)

1:  procedure BUFFERWRITE($f$, $buf$, $count$, $o$)
2:    $\rightsquigarrow$ L($f$)
3:    $\rightsquigarrow$ MR($ms_f$) in $size$
4:    if isPreallocBlock($o$, $count$, $size$) then
5:      $end \leftarrow$ min($count + o$, getLastBlockEnd($f$))
6:      for $i = size$ to $end - 1$ do $\rightsquigarrow$ DW$^{\text{zero}}$(($f$, $i$), 0)
7:      $\rightsquigarrow$ DW$^{\text{zero}}$($ds_f$, $end$)
8:      $size \leftarrow end$
9:    if $o > size$ then
10:     for $i = size$ to $o - 1$ do $\rightsquigarrow$ DW$^{\text{norm}}$(($f$, $i$), 0)
11:   for $i = 0$ to $count - 1$ do
12:     $\rightsquigarrow$ DW$^{\text{norm}}$(($f$, $o + i$), $buf[i]$)
13:     if (isFstB($f$, $o$+$i$+1) $\vee$ $i$=$count$−1)) $\wedge$ $o$+$i$ > $size$
    then
14:       $\rightsquigarrow$ DW$^{\text{norm}}$($ds_f$, $o + i$) ; MW($ms_f$, $o + i$)
15:   $\rightsquigarrow$ U($f$)

1:  procedure pwrite($d_f$, $buf$, $count$, $o$)
2:    BUFFERWRITE($f$, $buf$, $count$, $o$)

1:  procedure write($d_f$, $buf$, $count$)
2:    $\rightsquigarrow$ L($d_f$)
3:    $\rightsquigarrow$ MR($ol_{d_f}$) in $o$
4:    BUFFERWRITE($f$, $buf$, $count$, $o$)
5:    $\rightsquigarrow$ MW($ol_{d_f}$, $o + count$)
6:    $\rightsquigarrow$ U($d_f$)
```

Fig. 7. Algorithmic description of mapping system calls to events where $\rightsquigarrow l$ generates an event with label l

fresh inode f (via freshINode), removes O_TRUNC from the flags as the file is just created, initializes the f size both in memory and on disk, associates nl with f and returns f and the updated flags. (Line 8–Line 13). If f (returned by lookup) is valid, the open part creates a fresh file descriptor d_f (via freshFD), initializes its offset and opens f with d_f. Finally, the truncate part sets the file size to zero (both in memory and on disk). A call to close (d_f) simply generates a single event to close d_f.

Sync, FSync and LSeek. A sync (resp. fsync (d_f)) call simply corresponds to a single event with label $\mathtt{S}()$ (resp. $\mathtt{FS}(d_f)$). As for the lseek system call, recall that lseek (d_f, o) updates the offset of the file descriptor d_f to o. Moreover, as with all system calls modifying the file descriptor offset, it does so by first acquiring its lock. As such, the associated algorithm generates a sequence of events to lock d_f, read the current offset o' of d_f, update it to o, and finally release the lock on d_f.

PRead and Read. A call to pread is handled by the BufferRead routine which reads the in-memory file size (Line 2) and generates a sequence of DR events for reading m bytes of file data at offset o, one byte at a time (Line 5), where m is the minimum of $count$ and $size - o$. Recall that the key difference between read and pread is that read updates the descriptor offset. As such, read generates analogous events to those of pread, and further include events for updating the descriptor offset (Lines 3 and 5) and thus acquiring/releasing its lock (Lines 2 and 6).

PWrite and Write. A call to pwrite (1) acquires the inode lock, (2) carries out the file data write one byte at a time, (3) increases the file size if necessary (in the case of appends) and (4) releases the inode lock. Analogously, the BufferWrite of pwrite generates the lock and unlock events in steps (1) and (4) on Lines 2 and 15, respectively, the events of step (2) on Lines 11–12, and those of (3) on Lines 13–14. Note that the size is updated both in memory and on disk (Line 14) after writing a full block (or the last sub-block of the write), provided that the file size changes (increases).

A pwrite must additionally account for cases where a file with preallocated blocks is appended, or the offset supplied is greater than the file size. The former case (see delayed allocation in §2.4) is handled on Lines 4–6, where the last file block is zeroed if it is preallocated (determined via `isPreallocBlock`)—we assume that a file has at most one preallocated block at the end that is not filled. The on-disk (but not in-memory) file size is then updated accordingly (Line 7); this ensures that the zeroed-out block is observable after recovery from a crash, but not by reads during the execution. The latter case is handled on Lines 9–10, where the bytes between $size$ and o are zeroed.

Lastly, the events generated by write are analogous to those of pwrite and additionally generate the events for updating the file descriptor offset as in read.

Link, Unlink and Rename. The link call acquires the lock on the (only) directory dir (Line 2) and inspects the new name nl_{new} specified (Line 3), ensuring that it is not already taken, i.e., holds $\perp$ (Line 3). It then determines the inode f associated with the old name (Line 4), updates the new name location to point to f (Line 5) and finally releases the directory lock (Line 6). The unlink (nl) call simply unlinks the name location nl from its associated inode by updating it to the designated $\perp$ value. Finally, the events generated by rename are those of link and unlink combined, except that unlike link, rename does not inspect the new name to check that it is available.

3.2 Executions

We next define the notion of an *execution graph* G, where its nodes are events and its edges denote the sundry *relations* on events describing e.g., the values read via the 'reads-from' relation.

Definition 3.2 (Executions). An *execution* $G \in \mathsf{Exec}$ is a tuple of the form $\langle \mathtt{E}, \mathtt{rf} \rangle$, where:

- $\mathtt{E}$ is a set of events (Def. 3.1) comprising a set of *initialization events*, $\mathtt{I} \subseteq \mathtt{E}$, by a designated thread t_0 such that (1) $\mathtt{I} \triangleq \left\{ e \in \mathtt{E} \cap (\mathtt{W} \cup \mathtt{U}) \;\middle|\; \mathtt{tid}(e) = t_0 \right\}$; (2) for each $ml \in \mathsf{Mloc}$, the set $\mathtt{I}$ contains a single event $w \in \mathtt{MW}$ on ml with value 0, i.e., $\mathtt{loc}(w)=ml$ and $\mathtt{val}(w)=0$; (3) for each $nl \in \mathsf{Dnameloc}$, the set $\mathtt{I}$ contains a single event $w \in \mathtt{DW}$ on nl with value $\perp$; and (4) for each $lck \in \mathsf{Lck}$, the set $\mathtt{I}$ contains a single unlock event u on lck (i.e., $\mathtt{lab}(u)=\mathtt{U}(lck)$).
- $\mathtt{rf} \subseteq (\mathtt{E}{\times}\mathtt{E}) \cap ((\mathtt{W}{\times}\mathtt{R}) \cup (\mathtt{U}{\times}\mathtt{L}))$ is the *reads-from* relation such that (1) for all $(w, r) \in \mathtt{rf} \cap (\mathtt{W}{\times}\mathtt{R})$: $\mathtt{loc}(w)=\mathtt{loc}(r)$; (2) for all $(u, l) \in \mathtt{rf} \cap (\mathtt{U} \times \mathtt{L})$: $\mathtt{lck}(u)=\mathtt{lck}(l)$; and (3) $\mathtt{rf}^{-1}$ is functional;

(4) $\mathtt{rf}$ is total on its range: for all $a \in \mathtt{E} \cap (\mathtt{R} \cup \mathtt{L})$, there exists a unique $b \in \mathtt{W} \cup \mathtt{U}$ such that $(b, a) \in \mathtt{rf}$; and (5) no two locks read from the same unlock: $\mathtt{rf}^{-1}; [\mathtt{U}]; \mathtt{rf} \subseteq [\mathtt{L}]$.

Given an execution G, we use the '$G.$' prefix to project its various components (e.g., $G.\mathtt{rf}$), including its implicit components (e.g., $G.\mathtt{I}$) and derived relations described below (e.g., $G.\mathsf{po}$). When the choice of G is clear from the context, we omit this prefix and simply write e.g., $\mathtt{I}$.

Derived Relations. Given an execution $G = \langle \mathtt{E}, \mathtt{rf} \rangle$, we define the *program-order* relation as:

$$\mathsf{po} \triangleq \mathtt{I} \times (\mathtt{E} \setminus \mathtt{I}) \ \cup \left\{ \begin{array}{l} \langle \langle n_1, t_1, id_1, l_1 \rangle, \\ \ \langle n_2, t_2, id_2, l_2 \rangle \rangle \end{array} \middle| \begin{array}{l} \langle n_1, t_1, id_1, l_1 \rangle, \langle n_2, t_2, id_2, l_2 \rangle \in \mathtt{E} \setminus \mathtt{I} \\ \wedge\ t_1 = t_2 \wedge n_1 < n_2 \end{array} \right\}$$

relating initialization events to all others, and the events of each thread in increasing order.

We also define the *same-operation* equivalence relation, $\mathtt{sid}$, on the events of the same operation: $\mathtt{sid} \triangleq \left\{ (a, b) \mid \mathtt{id}(a) = \mathtt{id}(b) \right\}$. The *same-file*, $\mathtt{sf}$, equivalence relation is defined analogously, relating the events on the same file (inode).

Finally, we lift $\mathtt{ssec}$ to events by defining $(a, b) \in \mathtt{ssec}$ to hold iff $(\mathtt{loc}(a), \mathtt{loc}(b)) \in \mathtt{ssec}$, and define the *block-sequence* relation, $\mathtt{bseq}$, prescribing the order in which a block is written as:

$$\mathtt{bseq} \triangleq \left\{ (w_1, w_2) \mid \mathtt{lab}(w_1) = \mathtt{DW}_{(f, o_1)} \wedge \mathtt{lab}(w_1) = \mathtt{DW}_{(f, o_2)} \wedge (o_1, o_2) \in \mathtt{sblk} \wedge o_1 < o_2 \right\}$$

Memory-Model Consistency. Note that in this initial stage, executions are unrestricted in that there are few constraints on $\mathtt{rf}$. Such restrictions are determined by the set of memory-model-specific *consistent executions.*

In order to keep our formalism general, we do not define a specific memory model and its associated set of consistent executions. Rather, we define a general formal framework that is *parametric* in the choice of the underlying memory model. We thus assume a *consistency predicate*, $\mathsf{cons}_{\mathcal{M}}(.)$, which determines whether an execution is $\mathcal{M}$-consistent (i.e., consistent under the $\mathcal{M}$ memory model). This way, our general framework can be *instantiated* for a desired memory model $\mathcal{M}$ by supplying it with the $\mathsf{cons}_{\mathcal{M}}(.)$ predicate.

$\mathcal{M}$-consistency is typically constrained via a *synchronizes-with* relation, $\mathtt{sw}$, prescribing the order induced by synchronization mechanisms, e.g., locks. As we do not restrict our framework to an explicit memory model, we accordingly keep its associated $\mathtt{sw}$ relation as a parameter, with the proviso that $\mathtt{sw}$ includes the $\mathtt{rf}$ edges on disk locations as well as the synchronization induced by lock acquisition. As is standard, we define the *happens-before* relation, $\mathtt{hb}$, as the transitive closure of po and $\mathtt{sw}$. Intuitively, $\mathtt{hb}$ denotes c-ordering: the order events become visible to other threads.

Parameter 1 ($\mathcal{M}$-consistency). Given an execution $\langle \mathtt{E}, \mathtt{rf} \rangle$, assume a *synchronizes-with* relation, $\mathtt{sw} \subseteq \mathtt{E} \times \mathtt{E}$, denoting a strict partial order such that $[\mathtt{DW}]; \mathtt{rf}; [\mathtt{DR}] \subseteq \mathtt{sw}$ and $[\mathtt{U}]; \mathtt{rf}; [\mathtt{L}] \subseteq \mathtt{sw}$.

Assume a consistency predicate, $\mathsf{cons}_{\mathcal{M}}(.) : \mathsf{Exec} \rightarrow \{\mathsf{true}, \mathsf{false}\}$, such that for all executions G, if $\mathsf{cons}_{\mathcal{M}}(G)$ holds, then (1) $\mathtt{hb} \triangleq (\mathsf{po} \cup \mathtt{sw})^+$ is irreflexive; and (2) $(\mathsf{po} \cup \mathtt{rf})^+$ is irreflexive.

The first requirement ensures that $\mathtt{hb}$ is a strict partial order; the second requirement precludes 'out-of-thin-air' behaviours [Lahav et al. 2017], and is required by GenMC [Kokologiannakis et al. 2019], the DPOR framework over which we build PerSeVerE (see §4).

Observe that all *disk writes* (in $\mathtt{DW}$) generated by the Fig. 7 algorithms are issued while holding a lock. As such, thanks to lock-induced synchronization (Parameter 1), all writes on the same disk location are related by $\mathtt{hb}$; i.e., $\mathtt{whb}_{dl}$ is total for all $dl \in \mathsf{Dloc}$, where $\mathtt{whb} \triangleq [\mathtt{DW}]; \mathtt{hb}; [\mathtt{DW}]$.

We define the *persists-before* relation, $\mathtt{pb} \subseteq \mathtt{D} \times \mathtt{D}$, as the least transitive relation such that:

$$(\mathtt{I} \cap \mathtt{D}) \times (\mathtt{D} \setminus \mathtt{I}) \subseteq \mathtt{pb} \qquad (\text{PB-INIT})$$

$$[\mathtt{DW}]; (\mathtt{hb} \cap \mathtt{ssec}); [\mathtt{DW}] \subseteq \mathtt{pb} \qquad (\text{PB-SECTOR})$$

$$[\mathtt{DW}]; (\mathtt{hb} \cap \mathtt{bseq}); [\mathtt{DW}] \subseteq \mathtt{pb} \qquad (\text{PB-BLOCK})$$

$$[\mathtt{DW}_{\mathsf{Floc}}]; (\mathtt{hb} \cap \mathtt{sf}); [\mathtt{DW}_{\mathsf{Dsizeloc}}] \subseteq \mathtt{pb} \qquad (\text{PB-META})$$

$$[\mathtt{S} \cup \mathtt{FS}]; \mathtt{hb}; [\mathtt{D}] \;\cup\; [\mathtt{D}]; \mathtt{hb}; [\mathtt{S}] \;\cup\; [\mathtt{DW}]; (\mathtt{hb} \cap \mathtt{sf}); [\mathtt{FS}] \subseteq \mathtt{pb} \qquad (\text{PB-SYNC})$$

$$[\mathtt{DW}_{\mathsf{Dnameloc}} \cup \mathtt{DW}^{\mathtt{trunc}}]; \mathtt{hb}; [\mathtt{D} \setminus \mathtt{DW}_{\mathsf{Floc}}] \subseteq \mathtt{pb} \qquad (\text{PB-DIROPS})$$

$$(\mathtt{atom}; \mathtt{pb}) \cup (\mathtt{pb}; \mathtt{atom}) \subseteq \mathtt{pb} \qquad (\text{PB-ATOM})$$

where $\mathtt{atom} \triangleq ([\mathtt{DW} \setminus \mathtt{DW}^{\mathtt{zero}}]; (\mathtt{ssec} \cap \mathtt{sid}); [\mathtt{DW} \setminus \mathtt{DW}^{\mathtt{zero}}]) \cup ([\mathtt{DW}^{\mathtt{rename}}]; \mathtt{sid}; [\mathtt{DW}^{\mathtt{rename}}])$.

Intuitively, pb denotes p-ordering: the order in which durable events (D) are persisted.

The PB-INIT axiom p-orders all initialization writes before all other durable events.

The PB-SECTOR axiom captures the hardware p-ordering guarantees: same-sector writes persist atomically and are never reordered. This ensures that $\mathtt{pb}_{dl} = \mathtt{whb}_{dl}$ for each disk location dl, as each disk location resides within one block. Note that as $\mathtt{whb}_{dl}$ is total, so is $\mathtt{pb}_{dl}$ for each dl.

The PB-BLOCK axiom models the assumption that sectors within a block are persisted in sequence. As a result, c-ordered same-block writes are also p-ordered, as long as their offsets match the order in which the block is written. Of course, this does not imply that same-block writes are p-ordered in general. For instance, a call to write/pwrite to a sector s_2 of a block b is not guaranteed to persist before a subsequent write/pwrite call to a previous sector s_1 of the same block, as such writes may be merged by the block I/O layer, and a crash may occur with only s_1 having persisted.

The PB-META axiom ensures that file data updates are p-ordered before their subsequent size updates, as required by the `data=ordered` journalling mode. Intuitively, this relates the event(s) on Line 6 of BufferWrite to that on Line 7, and those on Line 12 to that on Line 14.

The PB-SYNC axiom describes the p-ordering of sync/fsync: (1) all events that are hb-after a sync/fsync are p-ordered after it; (2) all events that are hb-before a sync are p-ordered before it; and (3) all disk writes to a file that are hb-before an fsync on the same file are p-ordered before it.

Recall from §2.4.2 that directory operations and file size updates due to truncation are p-ordered before all subsequent operations except overwrites. This is captured by PB-DIROPS, where $\mathtt{DW}_{\mathsf{Dnameloc}}$ denotes directory events (i.e., those that write to a name location) and $\setminus\mathtt{DW}_{\mathsf{Floc}}$ excludes overwrites.

Lastly, PB-ATOM ensures that renames and same-sector writes are p-atomic by requiring that pb be closed under composition with atom. The *atomic* relation atom relates the $\mathtt{DW}^{\mathtt{rename}}$ events of the same (rename) operation, as well as the same-sector $\mathtt{DW} \setminus \mathtt{DW}^{\mathtt{zero}}$ events of the same (write/pwrite) operation. Note that we exclude $\mathtt{DW}^{\mathtt{zero}}$ to model the (p-atomicity-violating) anomaly of delayed allocation (§2.4.1), thus allowing zero writes to persist without the corresponding data writes.

Given a memory model $\mathcal{M}$, an execution is consistent iff it is $\mathcal{M}$-consistent (Parameter 1) and its *persists-before* relation is a strict partial order.

Definition 3.3 (Consistency). An execution G is *consistent* according to a memory model $\mathcal{M}$ iff $\mathsf{cons}_{\mathcal{M}}(G)$ holds and $G.\mathtt{pb}$ is irreflexive.

3.3 Post-Crash Observable States

To determine the disk state upon recovery from a crashing execution G, we define a *p-snapshot*, $P \subseteq G.\mathtt{D}$, as a set of durable events whose effects have reached the disk before the crash.

More concretely, (1) P must include the initialization writes on disk locations ($\mathtt{I} \cap \mathtt{DW} \subseteq P$). Moreover, as pb prescribes the order in which writes are persisted to disk (2) P must be *downward-closed* with respect to pb: $dom(\mathtt{pb}; [P]) \subseteq P$. That is, if the effects of an event e have reached the disk (and thus $e \in P$), then the effects of all $\texttt{pb}$-earlier events must also have reached the disk. Similarly to ensure the p-atomicity of rename and the p-atomicity of same-sector writes (3) P must be downward-closed with respect to $\texttt{atom}$: $\mathit{dom}(\texttt{atom}; [P]) \subseteq P$.

Given a p-snapshot P of execution G, we define the *frontier* of P, written front_P, to contain *exactly one* write for each disk location dl, corresponding to the $G.\texttt{pb}_{dl}$-maximal write in P. That is, when P contains several writes on dl, its frontier contains the $\texttt{pb}_{dl}$-latest write in P ($\max(\texttt{pb}_{dl}|_P)$), capturing the last persisted write on dl, and thus the value observable for dl upon recovery. Note that as $\texttt{pb}_{dl}$ is total for each disk location dl, then $\max(\texttt{pb}_{dl}|_P)$ is uniquely defined.

Definition 3.4 (P-snapshot). A set P is a *p-snapshot* of an execution G iff (1) $(\mathtt{I} \cap \mathtt{D}) \subseteq P \subseteq \mathtt{D}$; (2) $\mathit{dom}(\texttt{pb}; [P]) \subseteq P$; and (3) $\mathit{dom}(\texttt{atom}; [P]) \subseteq P$.

Given a p-snapshot P of an execution G, the *frontier* of P is $\mathsf{front}_P \triangleq N \cup S \cup F$ where:

$$N \triangleq \left\{\max(G.\texttt{pb}_{dl}|_P) \;\middle|\; dl \in \mathsf{Dnameloc}\right\} \quad S \triangleq \left\{\max(G.\texttt{pb}_{ds_f}|_P) \;\middle|\; \exists w \in N.\, \mathtt{lab}(w) = \mathtt{DW}(nl_f, f)\right\}$$
$$F \triangleq \left\{\max(G.\texttt{pb}_{(f,o)}|_P) \;\middle|\; \exists w_1 \in N, w_2 \in S.\, \mathtt{lab}(w_1) = \mathtt{DW}(nl_f, f) \wedge \exists s.\, \mathtt{lab}(w_2) = \mathtt{DW}(ds_f, s) \wedge o < s\right\}$$

Note that N computes the p-snapshot for file name locations, while S and F do so for each file size location ds_f and data location (f, o) where f is accessible (mapped on to) by a file name written in N and o is within the f size written in S. This excludes inaccessible 'garbage' from p-snapshots.

4 PERSEVERE: MODEL CHECKING UNDER EXT4

An effective technique for verifying the consistency guarantees of concurrent programs is *Stateless Model Checking* (SMC) [Godefroid 1997; 2005; Musuvathi et al. 2008] coupled with *Dynamic Partial Order Reduction* (DPOR) [Abdulla et al. 2014; Flanagan et al. 2005; Kokologiannakis et al. 2019]. We next present our second key contribution: extending SMC and DPOR to verify persistency.

4.1 Effective Model Checking for Persistency

A key challenge of SMC is that a concurrent program may have a large number of executions to explore, typically exponential in the program size. To address this, existing literature includes several effective DPOR techniques that partition the executions into *consistency equivalence* classes (c-classes), aiming to explore exactly one execution per c-class. That is, all executions in a c-class have the same consistency guarantees, and thus it suffices to explore one execution from each.

However, although there exists a large body of work on different notions of c-classes and thus effective model checking for *consistency*, there is little to no work on model checking *persistency*. As such, to facilitate effective model checking for persistency, we develop several techniques to partition executions into *persistency equivalence* classes (p-classes). In what follows we formally describe our mechanisms for partitioning executions to p-classes and then present PerSeVerE.

Partial P-Ordering. The first partitioning mechanism we employ is representing p-ordering as a *partial* rather than a *total* relation: our definition of execution consistency (Def. 3.3) does not require the p-ordering relation $\texttt{pb}$ to be total and admits partial $\texttt{pb}$ orders. This is in contrast to the existing literature on persistency [Raad et al. 2018; 2019a; b], which requires p-ordering to be a *total* order. Modelling $\texttt{pb}$ as a partial order is highly effective in that it significantly reduces the number of executions to explore. To see this, consider the 3w+pb program below comprising three parallel writes, writing a single byte to offset 0 of files "a.txt", "b.txt" and "c.txt", respectively:

$$\begin{array}{l} d_{f_1} = \mathsf{open}\,(\text{“}\mathtt{a.txt}\text{”}); \\ \mathsf{pwrite}\,(d_{f_1}, \text{‘}1\text{’}, 0); \end{array} \;\Bigg\|\; \begin{array}{l} d_{f_2} = \mathsf{open}\,(\text{“}\mathtt{b.txt}\text{”}); \\ \mathsf{pwrite}\,(d_{f_2}, \text{‘}2\text{’}, 0); \end{array} \;\Bigg\|\; \begin{array}{l} d_{f_3} = \mathsf{open}\,(\text{“}\mathtt{c.txt}\text{”}); \\ \mathsf{pwrite}\,(d_{f_3}, \text{‘}3\text{’}, 0); \end{array} \qquad (\textsc{3w+pb})$$

Let us assume that file sizes are greater than one byte and thus all three writes are overwrites. Ignoring open, given the mapping in Fig. 7, each pwrite generates events for locking the respective inode, reading its size, writing (one byte) at offset 0 and unlocking the inode. That is, ignoring the non-disk events, if we assume that the inodes are respectively f_1, f_2 and f_3, an execution of 3W+PB comprises three disk-write events w_1, w_2 and w_3, with $\mathtt{lab}(w_i) = \mathtt{DW}((f_i, 0), i)$ for $i = 1 \cdots 3$.

Under representations that require a total p-ordering, there are six (3!) possible p-orderings, i.e., the number of $[w_1, w_2, w_3]$ permutations. Moreover, recall from our notion of p-snapshots (Def. 3.4) that a *prefix* of each p-ordering may have persisted prior to the crash. As such, since each p-ordering has three prefixes, the total number of explorations is 18 ($3 \times 3!$).

On the other hand, our partial pb definition does not order the three writes, since they are in different files; thus, when constructing our p-snapshot P, it suffices to consider whether (the effect of) each write has persisted, i.e., $P \in \mathcal{P}(\{w_1, w_2, w_3\})$, thus yielding eight ($2^3$) explorations corresponding to eight different p-snapshots. In the general case of 3W+PB with N parallel writes, this amounts to reducing the number of explorations from $N \times N!$ to 2^N.

Recovery Observer. Although keeping pb partial eliminates a significant number of explorations, it nevertheless includes redundancies and can be further improved. Consider the REC-OB example below where 3W+PB above runs in parallel with an independent write to “d.txt” with inode f' and crashes thereafter ($\lightning$); upon recovery (to the right of $\lightning$) the first byte of f' is read in b:

$$\textsc{3w+pb} \;\Big\|\; \begin{array}{l} d_{f'} = \mathsf{open}\,(\text{“}\mathtt{d.txt}\text{”}); \\ \mathsf{pwrite}\,(d_{f'}, \text{‘4’}, 0); \end{array} \quad \lightning \quad \begin{array}{l} d'_{f'} = \mathsf{open}\,(\text{“}\mathtt{d.txt}\text{”}); b = \mathsf{pread}\,(d'_{f'}, 1, 0); \\ \mathbf{assert}(b = \text{‘4’}); \end{array} \qquad (\textsc{rec-ob})$$

Since there are eight (2^3) possible p-snapshots for the locations $dl_i = (f_i, 0)$ (for $i = 1 \cdots 3$) and two (2^1) p-snapshots for $dl' = (f', 0)$, this amounts to 16 (8×2) possible explorations. However, note that although dl_1, dl_2 and dl_3 are all written by 3W+PB, they are never read upon recovery. By contrast, dl' is observed (read from) upon recovery, and we must therefore consider two explorations: one in which the pre-crash write on dl' persists and one in which it does not. In other words, it suffices to consider p-orderings only on those locations that are *read from* upon recovery.

We use this intuition to reduce the number of explorations further. More concretely, we model observable p-snapshots through a *recovery observer*, a designated thread that runs in parallel with the original program. This way, we can model observability by reads-from (rf) edges between the events of the original program and those of the recovery observer. To this end, we *instrument* our executions (Def. 3.2) to include recovery events, as we describe below.

Definition 4.1 (Instrumented execution). An *instrumented execution* is a tuple $\langle \mathsf{E}, \mathtt{rf} \rangle$ such that:

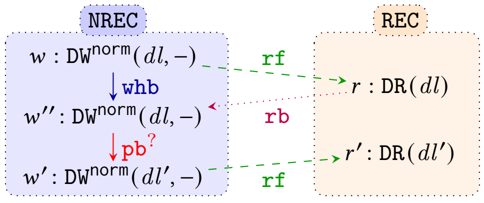
- $\langle \mathsf{E}, \mathtt{rf} \rangle$ is an execution (Def. 3.2); and
- the event set is partitioned, $\mathsf{E} = \mathtt{NREC} \uplus \mathtt{REC}$, into *non-recovery events*, NREC, and *recovery events*, REC, comprising disk reads by a designated thread t_r: $\mathtt{REC} \triangleq \{ e \in \mathsf{E} \cap \mathtt{DR} \mid \mathtt{tid}(e) = t_r \}$.

Note that instrumented executions are executions (Def. 3.2) that additionally include recovery events comprising disk reads by the designated recovery thread t_r. That is, we model programs such as REC-OB, by having t_r issuing *recovery* pread *calls* to inspect the disk after the crash, e.g., the pread to the right of $\lightning$ in REC-OB. The mapping from a recovery pread call is that of pread in Fig. 7, with the memory read on Line 2 of BufferRead replaced with an analogous disk read.

With instrumented executions in place, we no longer need a p-snapshot and its frontier to determine the observable values upon recovery. Instead, we simply constrain the set of observable values by requiring that the resulting instrumented execution be consistent, as defined below.

Definition 4.2 (Instrumented consistency). An instrumented execution G is *consistent* iff:

- $\langle \mathtt{NREC}, \mathtt{rf}|_{\mathtt{NREC}} \rangle$ is consistent according to Def. 3.3 (when $G = \langle \mathtt{NREC} \uplus \mathtt{REC}, \mathtt{rf} \rangle$); and (CON)
- $[\mathtt{REC}]; \mathtt{rb}; \mathtt{atom}^?; \mathtt{pb}^?; \mathtt{rf}; [\mathtt{REC}] = \emptyset$, where $\mathtt{rb} \triangleq \bigcup_{x \in \mathsf{Loc}} \mathtt{rf}_x^{-1}; \mathtt{whb}_x$ (REC)


NREC
REC
$w : \mathtt{DW}^{\mathtt{norm}}(dl, -)$
whb
$w'' : \mathtt{DW}^{\mathtt{norm}}(dl, -)$
pb?
$w' : \mathtt{DW}^{\mathtt{norm}}(dl', -)$
rf
rb
rf
$r : \mathtt{DR}(dl)$
$r' : \mathtt{DR}(dl')$

Algorithm 1 Main exploration algorithm

```
1: procedure VERIFY(P, P_R)
2:     ⟨$G, \Gamma$⟩ ← ⟨$G_0, \Gamma_0$⟩
3:     do
4:         VISITONE(P, $G, \Gamma$)
5:         RUNRECOVERY(P_R, $G, \Gamma$)
6:     while ⟨$G, \Gamma$⟩ ← pop($\Gamma$)
```

Fig. 8. An instrumented execution precluded by REC (left); the PERSEVERE main exploration algorithm (right)

The REC axiom *equivalently* enforces the conditions imposed by p-snapshots and frontiers, where the *reads-before* relation, rb, relates each read r to all writes that are hb-after the write r reads from.

Specifically, REC pre-empts scenarios such as that in Fig. 8 (left), where a recovery read r on dl reads from w which is later overwritten by w'': $(w, w'') \in \mathtt{whb}_{dl}$. As whb and pb agree for each location, we also have $(w, w'') \in \mathtt{pb}_{dl}$. Moreover, as r and r' respectively read from w and w', then w, w' must have persisted prior to the crash, i.e., they are in the p-snapshot. As w'' is pb-before w', for the p-snapshot to be pb-downward-closed, w'' must also be in the p-snapshot. As such, since $(w, w'') \in \mathtt{pb}_{dl}$ and w'' is in the p-snapshot, then w'' is the $\mathtt{pb}_{dl}$-maximal write in the p-snapshot and not w, violating the $\mathtt{pb}_{dl}$-maximality condition of the p-snapshot frontier.

Finally, in the theorem below we show that the two characterizations of the observable states on recovery are equivalent. We present the full proof in [Kokologiannakis et al. 2021].

THEOREM 4.3 (EQUIVALENCE). *For all consistent instrumented executions G, there exists a p-snapshot P of G such that $dom(\mathtt{rf}; [\mathtt{REC}]) \subseteq \mathsf{front}_P$. For all consistent executions $G = \langle \mathtt{E}, \mathtt{rf} \rangle$, p-snapshots P of G, relations $\mathtt{rf}'$ and recovery reads $\mathtt{REC} \subseteq \mathtt{DR}$ by t_r ($\forall r \in \mathtt{REC}.\ \mathtt{tid}(r) = t_r$), if $rng(\mathtt{rf}') = \mathtt{REC}$ and $dom(\mathtt{rf}') \subseteq \mathsf{front}_P$, then the instrumented execution $\langle \mathtt{E} \uplus \mathtt{REC}, \mathtt{rf} \uplus \mathtt{rf}' \rangle$ is consistent.*

4.2 PERSEVERE: DPOR for Persistency

We next present PERSEVERE, our persistency model checking algorithm. PERSEVERE can be built by extending any DPOR framework with persistency. For clarity, we build PERSEVERE as an extension of GENMC [Kokologiannakis et al. 2019], as it is parametric in the choice of its underlying memory model, thus allowing us to verify ext4's persistency properties under different $\mathcal{M}$-consistency models. We proceed with a brief description of the DPOR algorithm used by GENMC and then describe how we extend it with persistency. As PERSEVERE is based on GENMC, it inherits its correctness results. Specifically, PERSEVERE is *sound* (only explores consistent executions), *complete* (explores all consistent executions) and *optimal* (explores each execution exactly once).

GENMC in a Nutshell. As with other DPOR techniques, GENMC verifies a program P by exploring its executions one at a time, identifying alternative explorations on the fly and recording them in an *environment* Γ. Once an execution is fully explored, an alternative option is picked from Γ for further exploration. This high-level description is depicted in the VERIFY procedure of Algorithm 1. Ignoring the RUNRECOVERY function, VERIFY begins with an empty execution G_0 and environment Γ_0 (Line 2), and calls VISITONE to generate a full execution of P.

The crux of GENMC lies in VISITONE(P, G, Γ) (Algorithm 2), which extends the current execution G to a full execution of P and extends Γ with alternative explorations along the way. To this end, at each step it extends G by one event a provided that G is $\mathcal{M}$-consistent ($\mathsf{cons}_{\mathcal{M}}(G)$ holds). If there are no such events a then G is a full execution and VISITONE returns. If a denotes an error, then the error is reported and VISITONE terminates (Line 3). Otherwise, a is added to G (Line 4).

Algorithm 2 Exploration of one execution

```
1:  procedure VisitOne(P, $G, \Gamma$)
2:      while $\mathsf{cons}_{\mathcal{M}}(G) \land a \leftarrow \mathsf{next}_{\mathsf{P}}(G)$ do
3:          if $a \in \mathtt{error}$ then exit("error")
4:          $G.\mathsf{E} \leftarrow G.\mathsf{E} +\!\!+ [a]$
5:          if $a \in \mathtt{R}$ then
6:              $W \leftarrow G.\mathsf{E} \cap \mathtt{W}_{\mathtt{loc}(a)}$
7:              let $\{w_0\} \uplus ws = W$
8:              $G \leftarrow \mathsf{SetRF}(G, w_0, a)$
9:              $A_s \leftarrow \{\mathsf{SetRF}(G, w, a) \mid w \in ws\}$
10:             $\Gamma \leftarrow \mathsf{push}(\Gamma, A_s)$
11:         if $a \in \mathtt{W}$ then CalcRevisits$(G, \Gamma, a)$
```

Algorithm 3 Exploration of recovery routine

```
1:  procedure RunRecovery($\mathsf{P_R}$, $G, \Gamma$)
2:      while $\mathsf{cons}(G) \land a \leftarrow \mathsf{next}_{\mathsf{P_R}}(G)$ do
3:          if $a \in \mathtt{error}$ then exit("error")
4:          $G.\mathsf{E} \leftarrow G.\mathsf{E} +\!\!+ [a]$
5:          assert($a \in \mathtt{DR}$)
6:          $W \leftarrow G.\mathsf{E} \cap \mathtt{W}_{\mathtt{loc}(a)}$
7:          let $\{w_0\} \uplus ws = W$
8:          $G \leftarrow \mathsf{SetRF}(G, w_0, a)$
9:          $A_s \leftarrow \{\mathsf{SetRF}(G, w, a) \mid w \in ws\}$
10:         $\Gamma \leftarrow \mathsf{push}(\Gamma, A_s)$
```

If a is a read event, then the rf component of G must be accordingly extended for a. To this end, the set of all possible reads-from options for a (i.e., all write events in G on the same location) are computed in W (Line 6) and one write w_0 is picked from W (Line 7) as the rf option for a (Line 8). Moreover, for each remaining write option in ws, an alternative execution is constructed with the corresponding rf edge (Line 9) and added to Γ as an alternative future exploration (Line 10).

If a is a write event, then it may constitute an alternative reads-from option for existing reads in G on the same location, thus inducing additional alternative explorations. This is computed by the CalcRevisits function (Line 11) which extends Γ with such alternative explorations. We omit the details of CalcRevisits and refer the reader to Kokologiannakis et al. [2019] for more details.

PerSeVerE. As discussed in §4.1, PerSeVerE determines the post-crash states of a program P via a recovery observer program $\mathsf{P_R}$ that is run in parallel with P. As such, after generating one execution G of P through VisitOne, PerSeVerE extends G to a full execution of $\mathsf{P}||\mathsf{P_R}$ by calling RunRecovery on Line 5 of Verify, where $\mathsf{P_R}$ denotes the recovery observer program.

The RunRecovery algorithm (Algorithm 3) is very similar to VisitOne: it extends the current execution G with the events of $\mathsf{P_R}$ towards a full execution, and extends the environment with alternative explorations en route. Similarly, after each step RunRecovery ensures that the instrumented execution is consistent according to Def. 4.2, written $\mathsf{cons}(G)$, thereby checking both the $\mathcal{M}$-consistency axioms and the rec persistency axiom. That is, the consistency check used is that of instrumented executions which encodes checking persistency as a single axiom rec.

In contrast to events added by VisitOne, all events added by RunRecovery are disk reads (cf. REC of Def. 4.1). As explained in §4.1, this is ensured by having the recovery observer program issue pread calls that read the file size with a disk read instead of a memory read. Therefore, unlike VisitOne, RunRecovery never calls CalcRevisits.

In effect, the key idea behind PerSeVerE lies in how it explores the possible post-crash disk states through a recovery observer. This encodes checking the persistency of an execution as checking its consistency instead (on Line 2), which in turn enables us to use existing effective DPOR techniques (e.g., GenMC). In particular, checking the consistency of an instrumented execution (Def. 4.2) amounts to checking its $\mathcal{M}$-consistency which is standard, and checking its persistency as encoded by the single axiom rec, which is similar in form to existing $\mathcal{M}$-consistency axioms.

4.3 PerSeVerE: An Example

We next describe how PerSeVerE works through an example.

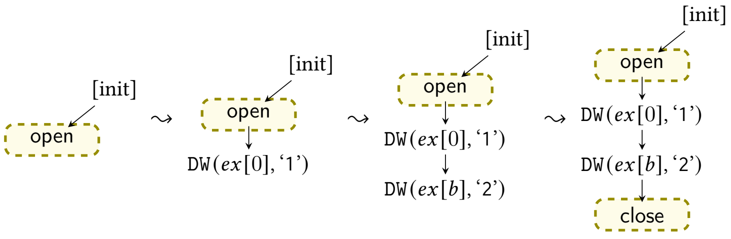


Fig. 9. Exploration of the main program for REC-WW+RR.

Consider the program below manipulating the "ex.txt" file which comprises multiple blocks of the '0' character on disk:

$$\begin{array}{l} d_f = \mathsf{open}\ (\text{“ex.txt”}); \\ \mathsf{pwrite}\ (d_f, \text{‘1’}, 0); \\ \mathsf{pwrite}\ (d_f, \text{‘2’}, b); \\ \mathsf{close}\ (d_f); \end{array} \quad ↯ \quad \begin{array}{l} d_{f'} = \mathsf{open}\ (\text{“ex.txt”}); \\ c_1 = \mathsf{pread}\ (d_{f'}, 1, b); \\ c_2 = \mathsf{pread}\ (d_{f'}, 1, 0); \\ \mathsf{close}\ (d_{f'}); \\ \mathbf{assert}(\neg(c_1 = \text{‘2’} \land c_2 = \text{‘0’})); \end{array} \qquad (\text{REC-WW+RR})$$

Let b denote the starting offset of the file's second block. As before, the code to the right of ↯ denotes the recovery observer, inquiring whether it is possible upon recovery to see the second write but not the first; i.e., $c_1 = \text{‘2’} \land c_2 = \text{‘0’}$. As discussed in §2.4.1, this is indeed possible; we next show how PERSEVERE generates all possible outcomes of REC-WW+RR including one where $c_1 = \text{‘2’} \land c_2 = \text{‘0’}$.

Let P denote the original program (to the left of ↯). Starting from the main exploration algorithm (Algorithm 1), PERSEVERE generates a full execution of P by calling VISITONE (Line 4) which adds the P events one at a time, as depicted in Fig. 9, where → denote po edges. For brevity, we omit the events of open/close and the lock/unlock events of pwrite (as there is only one thread in P). In addition, note that both pwrite calls leave the file size unchanged as they are overwrites; we thus omit the read event reading the in-memory file size and represent each pwrite with a disk write.

Observe that none of the events have an alternative option and thus no exploration is added to Γ (Line 10, Algorithm 2), and no revisiting is performed (Line 11, Algorithm 2). Moreover, there is no pb edge between the two DW events as they write to different disk blocks.

Continuing from Line 5 of Algorithm 1, PERSEVERE adds the recovery observer events as shown in Fig. 10 (top), where ⇢ denote rf edges. When the DR($ex[b]$) event of $c_1 = \mathsf{pread}\ (d_{f'}, 1, b)$ is added, it can read two different values: '2' written by P or the initial data at b. PERSEVERE explores both options, leading to executions ② and ③, respectively. However, these executions are not both generated and stored in memory simultaneously. As described in §4.2, PERSEVERE continues with one option (Algorithm 2, Line 8) and extends it to a full execution, and pushes the other option to the environment (Algorithm 2, Line 10), and *backtracks* to it at a later time for further exploration.

Let us assume that PERSEVERE continues with ②. PERSEVERE next adds the DR event corresponding to $c_2 = \mathsf{pread}\ (d_{f'}, 1, 0)$, which can similarly read from two values: value '1' written by P or the initial value on disk. As before, PERSEVERE explores both options, leading to executions ㉑ and ㉒ depicted in Fig. 10 (middle). Assuming that PERSEVERE continues with ㉑ and that ㉒ is pushed to the environment, PERSEVERE finally adds the close events (not depicted), at which point the resulting execution is complete, thus concluding the first exploration.

PERSEVERE next explores an alternative execution (Algorithm 1, Line 6). As is standard with DPOR techniques, PERSEVERE explores alternative executions in DFS-style, i.e., it considers the

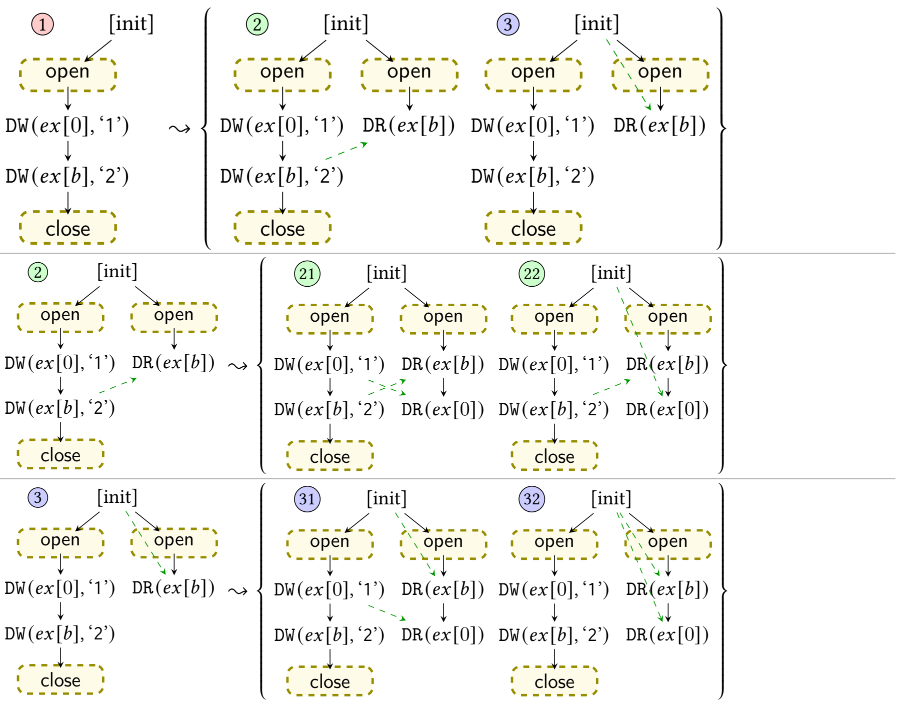


Fig. 10. Exploration of the REC-WW+RR recovery observer: step 1 (top), step 2 (middle) and step 3 (bottom)

latest alternative. As such, it next backtracks to (22) to explore it further. Once again, after adding the close events, the resulting execution is complete, thus concluding the second exploration.

Finally, since there are no more alternative reads-from options for DR(ex[0]), PerSeVerE backtracks to execution (3) as shown in Fig. 10 (bottom), and explores it further in a similar manner to the exploration starting from (2): the exploration will continue with either one of (31) and (32), and eventually backtrack to the unexplored option, thus concluding the third and fourth explorations.

We conclude with an observation. Because of the absence of pb edges in execution (1), the reads-from options of the recovery observer of REC-WW+RR are unrestricted, and so four executions were generated. This would be different if the two writes of P were to the same block (i.e., if b pointed to a disk location in the first block), as there would be a pb edge between the two writes because of PB-BLOCK. Specifically, execution (22) where DR(ex[0]) reads '0' would be deemed inconsistent as it would create an edge in [REC]; rb; pb; rf; [REC], thus violating REC. Intuitively, since blocks are written linearly (see § 2.4.1), writes to the same block are guaranteed to persist in order. So, if the recovery thread observes a disk write to a certain block, it must also observe all previous, smaller-offset writes to the same block.

5 EVALUATION

Implementation. We develop PerSeVerE on top of GenMC [Kokologiannakis et al. 2019], a publicly available SMC tool for concurrent C programs. GenMC employs an effective DPOR algorithm, and can verify programs under both RC11 [Lahav et al. 2017] and IMM [Podkopaev et al. 2019] memory models. We build PerSeVerE by extending GenMC with the numerous I/O system calls; this requires significant engineering effort. PerSeVerE can thus check the *consistency* of programs that use I/O calls under both RC11 and IMM, without checking persistency.

Table 1. Sequential benchmarks used to evaluate PerSeVerE

	P-Total		P-Partial		PerSeVerE	
	Execs	Time	Execs	Time	Execs	Time
pjnl-app-mfl	🕒	🕒	15	57.39	9	0.02
pjnl-crowr-ord	1323	7.84	18	0.04	6	0.02
pjnl-owr-at	36	0.03	5	0.02	5	0.02
pord-owr-N	🕒	🕒	243	308.26	2	0.02
pord-app-mbl	12	0.03	5	0.03	3	0.02
pord-app-sfl	900	1.90	13	0.03	3	0.02
pord-MP+fsync	8724	496.63	3	0.03	3	0.02
pord-trapp-dfl-ord	4044	47.22	5	0.03	3	0.02
pord-crapp-ooo	810	1.38	21	0.03	3	0.02
pord-MP+osync	8724	596.20	3	0.03	3	0.02
pord-owrapp-ord	450	0.39	10	0.03	4	0.02
pord-owrapp-ord2	3780	26.69	21	0.03	5	0.02
pord-owr-sbl	592	0.26	25	0.02	6	0.02
pord-rnm-at2	696	0.14	9	0.06	5	0.02
pord-rnmtrapp-ord	3240	5.87	18	0.05	6	0.02
pord-app-N	🕒	🕒	🕒	🕒	5	0.03
nano-backup-old	🕒	🕒	🕒	🕒	46	0.08
nano-backup-fix	🕒	🕒	🕒	🕒	10	0.04

PerSeVerE also supports *persistency* and can be configured to fine-tune the `ext4` behaviour. Specifically, (1) the block size can be tuned as desired; (2) all journalling modes of `ext4` are supported; and (3) under `data=ordered`, the user can decide whether delayed allocation is enabled (§ 2.4.1).

Methodology. We evaluate PerSeVerE in two ways. First, we run PerSeVerE on various litmus tests to show the scalability of our approach (§ 5.1). As a baseline, we implement two DPOR algorithms for persistency: (1) P-Total, that checks all prefixes of all total extensions of pb, and (2) P-Partial, that uses a partial p-ordering for pb that does not take into account which locations are read upon recovery, as in § 3.3. We show that PerSeVerE (using the approach in § 4.1) explores exponentially fewer executions than both approaches, and is thus exponentially faster.

Second, we study the persistency guarantees of several text editors (§ 5.2). The purpose of this study is twofold: (1) show that ensuring correct persistency is difficult as outlined in §1; and (2) show how PerSeVerE can help developers when designing algorithms that interact with persistent storage. As a result of this study, we found and confirmed bugs in commonly-used editors such as `nano` [GNU Nano 2019], `vim` [Vim 2019] and `emacs` [GNU Emacs 2019] using both stress testing and PerSeVerE. We reported these bugs to the developers and proposed fixes that are now merged [Kokologiannakis 2020]. We validated our fixes using stress testing and PerSeVerE.

Experimental Setup. We ran all tests on a Dell PowerEdge M620 blade system with two Intel Xeon E5-2667 v2 CPUs (8 cores @3.3 GHz) and 256GB of RAM, running a custom Debian-based distribution. We used LLVM-7 for GenMC, and ran the tool under RC11. All reported times are in seconds. We set the timeout limit to twenty minutes.

5.1 Litmus Tests

To evaluate PerSeVerE against the baseline implementations, we use both synthetic benchmarks as well as benchmarks extracted from the code of the editors in § 5.2. Our evaluation is split into two parts: a comparison on sequential benchmarks (cf. Table 1) and a comparison on concurrent benchmarks (cf. Table 2).

Sequential Benchmarks. Our results for sequential benchmarks are summarized in Table 1. We note that all these benchmarks are simple litmus tests, with the exceptions of `nano-backup-old` and `nano-backup-fix`, which are extracted directly from `nano`'s code base, and verify the persistency guarantees of `nano`'s backup procedure (~900LoC). However, as we discuss below, these simple litmus tests provide us with a good intuition on how the different approaches scale.

We highlight two main points about the results of Table 1. First, PerSeVerE and P-Partial explore exponentially fewer executions than P-Total, which unsurprisingly results in a runtime difference: it takes PerSeVerE less than a second to run all tests of Table 1, while it takes P-Total anywhere from less than a second to a few hours to run a single test. This is expected since PerSeVerE and P-Partial both employ a partial p-ordering for pb. Additionally, PerSeVerE

Table 2. Concurrent benchmarks used to evaluate PerSeVerE

	P-Total		P-Partial		PerSeVerE	
	Execs	Time	Execs	Time	Execs	Time
pord-wr+rdwr+fsync	🕒	🕒	206	0.30	6	0.03
pord-wr+wr-N	🕒	🕒	🕒	🕒	6240	2.13
pord-wr+wr-N-RR	🕒	🕒	🕒	🕒	17	0.72
pord-wr+wr-N-unord	🕒	🕒	🕒	🕒	22 680	27.74
pord-wr+wr-N-join-main	🕒	🕒	🕒	🕒	216	1.19
pord-wr+wr-N-join-thr	🕒	🕒	🕒	🕒	2052	37.00
pord-rd-wr+wr-N-cont	🕒	🕒	🕒	🕒	12 888	61.99
pord-rd-wr+wr-N-join	🕒	🕒	🕒	🕒	216	3.76

depends solely on the recovery observer, namely the number of recovery reads and their locations, and thus always explores fewer executions than the baseline approaches.

Second, P-Partial performs similarly to PerSeVerE in terms of executions for most benchmarks. This is due to the nature of the benchmarks: most benchmarks in Table 1 manipulate a part of a single file and the recovery observer checks whether these changes persist. In these cases, PerSeVerE does not gain much in terms of executions over P-Partial, since the recovery observer reads from most disk locations. However, when more than one file is manipulated (e.g., `pjnl-app-mfl`, `nano-backup-fix`), PerSeVerE greatly outperforms P-Partial. In the case of `pjnl-app-mfl`, even though the two implementations explore a similar number of executions, PerSeVerE is much faster as it does not calculate all valid p-snapshots of pb ($O(2^N)$), but rather checks for instrumented consistency on-the-fly, as it runs the recovery observer.

Concurrent Benchmarks. Our results for concurrent benchmarks are summarized in Table 2. With the exception of `pord-wr+rdwr+fsync`, these benchmarks aim to model realistic workloads where many worker threads are used to concurrently process one or more files. Thus, in addition to the persistency properties checked, these benchmarks also exercise the consistency aspect of our semantics, and therefore all tools have to deal with a significantly larger state space.

The observations here are similar to the ones for Table 1. More specifically, only PerSeVerE manages to terminate within the time limit for all of the benchmarks, as the state space quickly becomes intractable for P-Total and P-Partial. This is expected as all these benchmarks, in addition to the main and the recovery threads, involve 3 to 4 worker threads manipulating shared resources and files. The only benchmark in which one of the baseline implementations (P-Partial) manages to terminate within the time limit is `pord-wr+rdwr+fsync`, which is a simple message-passing litmus test. Even for `pord-wr+rdwr+fsync`, however, P-Total times out, since it has to calculate all total extensions of pb, a calculation that becomes extremely expensive when multiple files are involved, even for simple benchmarks.

5.2 Text Editors

In this study we focus on `emacs` [GNU Emacs 2019], `vim` [Vim 2019], `nano` [GNU Nano 2019], and `joe` [JOE 2018]. We chose `emacs` and `vim` as they are ubiquitous, and `nano` and `joe` because they are commonly available, and often used for editing files over ssh. As we describe shortly, these editors are *not crash-safe* under all circumstances.

Arguably, the most crucial functionality of editors lies in saving files. When a user edits a buffer, regardless of whether it originates from an existing file, the contents of the buffer reside in memory. Thus, saving the buffer contents is absolutely critical, especially when editing existing files, since saving the buffer implies the *p-atomic replacement* of an existing file (or, at least, the intent of doing so). We next describe the behaviour of the editors in our study when saving a buffer.

`nano` *and* `joe`. The procedure followed by `nano` and `joe` when saving an open buffer BUF as "`f.txt`" is shown in Fig. 11: "`f.txt`" is first truncated (if it exists), and then the contents of

```
d_f = open ("f.txt", O_WRONLY|O_CREAT|O_TRUNC);
write (d_f, BUF);
close (d_f);
```

Fig. 11. (EDITOR-SAVE-PROCEDURE)

BUF are written with one write system call. Surprisingly, however, there is no call to fsync after write: the editor may claim that the file is saved and even exit (when the user exits after saving) without the data persisted to disk. This is misleading from the point of view of users, who naturally expect that saving a file implies persistency.

One then may wonder why the absence of fsync does not lead to frequent data loss in such editors, especially given that writes happen asynchronously (§2.3). The reason is that this writing pattern, known as "replace-via-truncate", is one of the heuristics (§2.4.2) that is detected by `ext4` and remedied by flushing the data. Specifically, if (1) a file f is truncated (2) data is written to f and (3) f is subsequently closed, then `ext4` allocates and flushes all data written in step 2 after step 3.

However, there is a caveat: the flushing described above is not synchronous: close does *not* wait for the data to be flushed before returning. This then allows a race window where it is possible to see the truncated file without seeing the data that is subsequently written to it. This can happen, e.g., if the transaction containing the truncation commits before the data is flushed, and a crash occurs after the commit but before the flush. Moreover, this flushing does not extend to the disk's cache (as it is very expensive); as such, a similar race could occur even if the flush were synchronous.

Such a race, although rare, is definitely possible. We managed to reproduce it systematically with both litmus tests and PerSeVerE, and showed that data loss is possible in `nano` or `joe` under `ext4`.

Of course, data loss could occur even if fsync were used, since a crash could occur during fsync. There are, however, two observations to note. First, users commonly expect a file to be safely persisted to disk once the editor reports it as "saved". If a crash occurs during an fsync, i.e., while saving, data will be lost, but the user would expect this as saving is incomplete. Having an editor exiting gracefully, only to learn later that the file is corrupted is counter-intuitive, to say the least.

Second, to avoid data loss in such cases where crashes occur while saving, editors should make temporary data backups. We thus investigated the backup strategy of `nano` and `joe`, and found them both to be buggy, in that they are insufficient to prevent data loss. Indeed, `nano` does not even create backups by default, unless the `-B` option is used when editing a file.

For both editors, the problem lies in how they create and save the backup. As shown in Fig. 12, they read the contents of the original file ("`f.txt`") and write them to a backup file ("`f.txt~`"), which is truncated if it exists.

d_f = open ("`f.txt`", O_RDONLY);
d_b = open ("`f.txt~`", O_WRONLY|O_CREAT|O_TRUNC);
b = read (d_f); write (d_b, b);
close (d_b); close (d_f);

Fig. 12. (UNSAFE-BACKUP-STRATEGY)

However, this backup strategy has the very problem suffered by the save procedure: it follows "replace-via-truncate" which does not guarantee that data persists once a file is closed. Once again, without fsync the backup may not be safely persisted to disk before the save procedure starts. It is therefore possible to obtain a corrupted file *and* a corrupted backup. We have validated this behaviour using both litmus tests and PerSeVerE. This backup procedure can be made crash-safe by adding an fsync after write. As such, if a crash occurs while saving the original file, the backup copy would be available on disk. We reported these bugs to the developers, and proposed fixes that are now merged [Kokologiannakis 2020].

`emacs` *and* `vim`. The save procedures of `emacs` and `vim` are as in Fig. 11 with an fsync after write, ensuring that file data has safely persisted to disk once the save is complete. Moreover, they use a different backup strategy that, for most cases, does not suffer from the problems discussed above. Specifically, when a file is first modified in a session, `emacs` typically creates a backup as follows:

rename ("`f.txt`", "`f.txt~`");

`vim` follows a similar strategy and we omit it for brevity. The above procedure with a simple rename call is crash-safe under `ext4`. More concretely, if a crash occurs before the rename commits, then the original file will be intact. As the file truncation from save is c-ordered after rename (i.e., the backup), it will also be p-ordered after it (PB-DIROPS, §3.2). Therefore, even if a crash occurs during fsync leading to a truncated file, the backup will have persisted, thus avoiding data loss.

In certain cases, however, emacs follows a different backup strategy than the one mentioned above. One such case is when the original file has incoming hard links, in which case renaming the original file would undesirably direct the hard links to the backup's name. As such, emacs *copies* the original file instead, with a procedure similar to that of nano in Fig. 12. In particular, as the emacs backup strategy in such cases does not use fsync, it is *not crash-safe* as discussed above.

We note that the editors in our study are old, and the problems discussed would not be observed with older filesystems such as ext2 and ext3. Nevertheless, not observing such bugs does not make these editors crash-safe, especially since neither ext2/ext3 nor POSIX ever offered any guarantees of files persisting without the explicit use of fsync. Our study thus underlines the need for formal semantics and tools that can help developers correctly implement applications handing user data.

6 RELATED AND FUTURE WORK

Formal specification and verification of filesystems is an active research area (e.g., [H. Chen et al. 2015; Joshi et al. 2007; Kang et al. 2008; Keller et al. 2013; Ridge et al. 2015; Schellhorn et al. 2014; Sigurbjarnarson et al. 2016]). To our knowledge, PERSEVERE is the first project that encompasses both consistency and persistency guarantees in one (axiomatic) framework, presenting the first DPOR extension that can check persistency violations.

Several tools have been successful in finding persistency bugs in applications under different filesystems (e.g., [Cui et al. 2013; Mohan et al. 2018; Pillai et al. 2014; Rubio-González et al. 2009; Yang et al. 2006; Zheng et al. 2014]). The models used by these tools are products of empirical studies and thorough testing. Unfortunately, however, they do not come with formal semantics. On the other hand, such tools can be usually used under many different filesystems.

A notable exception to the above (and most closely related to our work) is the work of Bornholt et al. [2016], providing a framework for specifying and synthesizing the persistency semantics of different filesystems, as well as the FERRITE tool which exhaustively enumerates the persistency behaviours of litmus tests against the models of different filesystems. In contrast to PERSEVERE, FERRITE leverages SMT techniques to execute litmus tests symbolically against filesystem specifications, and may explore all prefixes of a given pb relation to check whether a given safety property holds. Moreover, FERRITE does not model the consistency guarantees of different filesystems, and only focuses on persistency. Among the filesystems they model is ext4, but their model is not precise and it is not clear which aspects of ext4 are covered, e.g., data=writeback or data=ordered.

Finally, Raad et al. [2019a,b] use axiomatic semantics to model persistency guarantees of NVM. We believe that PERSEVERE can be extended to verify the persistency guarantees of NVM programs.

Apart from NVM, we plan to extend our work in several directions in the future. First, we will formalize the semantics of other filesystems such as zfs [Bonwick 2005] or btrfs [Rodeh et al. 2013], and compare them with ext4. Second, we will formalize the consistency semantics of pathname lookup in recent kernels, and study how it affects the ext4 consistency/persistency guarantees. Finally, we will use PERSEVERE to verify the implementations of various applications (e.g., databases), and check if they provide sufficient consistency and persistency guarantees.

ACKNOWLEDGMENTS

We thank Luc Maranget and the POPL 2021 reviewers for their valuable feedback. Ilya Kaysin was supported by RFBR (grant number 18-01-00380). Azalea Raad was supported in part by a European Research Council (ERC) Consolidator Grant for the project "RustBelt", under the European Union Horizon 2020 Framework Programme (grant agreement number 683289).

REFERENCES

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas (2014). "Optimal dynamic partial order reduction." In: *POPL 2014*. New York, NY, USA: ACM, pp. 373–384. DOI: 10.1145/2535838.2535845.

Advanced Format (2020). URL: https://en.wikipedia.org/wiki/Advanced_Format (visited on May 20, 2020).

Jade Alglave, Luc Maranget, and Michael Tautschnig (July 2014). "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory." In: *ACM Trans. Program. Lang. Syst.* 36.2, 7:1–7:74. DOI: 10.1145/2627752.

Jeff Bonwick (Oct. 2005). *ZFS: The Last Word in Filesystems*. Library Catalog: blogs.oracle.com. URL: https://blogs.oracle.com/bonwick/zfs%3A-the-last-word-in-filesystems (visited on June 17, 2020).

James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang (2016). "Specifying and Checking File System Crash-Consistency Models." In: *ASPLOS 2016* 44.2, pp. 83–98. DOI: 10.1145/2980024.2872406.

Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich (2015). "Using Crash Hoare logic for certifying the FSCQ file system." In: *SOSP 2015*. the 25th Symposium. Monterey, California: ACM Press, pp. 18–37. DOI: 10.1145/2815400.2815402.

Ran Chen, Martin Clochard, and Claude Marché (2016). "A Formal Proof of a Unix Path Resolution Algorithm." In: *HAL* hal-01406848. URL: https://hal.inria.fr/hal-01406848/document (visited on Nov. 16, 2020).

Copy-on-write (2020). URL: https://en.wikipedia.org/wiki/Copy-on-write (visited on May 20, 2020).

Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang (2013). "Verifying Systems Rules Using Rule-Directed Symbolic Execution." In: *ASPLOS 2013*. Houston, Texas, USA: ACM, pp. 329–342. DOI: 10.1145/2451116.2451152.

GNU Emacs (2019). *GNU Emacs: An extensible, customizable, free/libre text editor — and more.* URL: https://www.gnu.org/software/emacs/ (visited on June 15, 2020).

ext4 benchmarks (2012). *EXT4 File-System Tuning Benchmarks*. URL: https://www.phoronix.com/scan.php?page=article&item=ext4_linux35_tuning&num=1 (visited on May 20, 2020).

Ext4 data loss (2009). URL: https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781 (visited on May 20, 2020).

ext4 Linux kernel (2020). *ext4 Data Structures and Algorithms*. URL: https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html (visited on May 20, 2020).

ext4 corruption (2015). *ext4: Filesystem corruption on panic*. URL: https://bugs.chromium.org/p/chromium/issues/detail?id=502898 (visited on May 20, 2020).

Michalis Kokologiannakis (July 2020). *files: improve the backup procedure to ensure no data is lost*. URL: https://git.savannah.gnu.org/cgit/nano.git/commit/?id=a84cdaaa50a804a8b872f6d468412dadf105b3c5 (visited on July 9, 2020).

Cormac Flanagan and Patrice Godefroid (2005). "Dynamic partial-order reduction for model checking software." In: *POPL 2005*. New York, NY, USA: ACM, pp. 110–121. DOI: 10.1145/1040305.1040315.

Patrice Godefroid (1997). "Model Checking for Programming Languages using VeriSoft." In: *POPL 1997*. Paris, France: ACM, pp. 174–186. DOI: 10.1145/263699.263717.

Patrice Godefroid (Mar. 2005). "Software Model Checking: The VeriSoft Approach." In: *Form. Meth. Syst. Des.* 26.2, pp. 77–101. DOI: 10.1007/s10703-005-1489-x.

JOE (2018). *JOE - Joe's Own Editor*. URL: https://joe-editor.sourceforge.io (visited on June 15, 2020).

Rajeev Joshi and Gerard Holzmann (June 11, 2007). "A Mini Challenge: Build a Verifiable Filesystem." In: *Formal Asp. Comput.* 19, pp. 269–272. DOI: 10.1007/s00165-006-0022-3.

Eunsuk Kang and Daniel Jackson (2008). "Formal Modeling and Analysis of a Flash Filesystem in Alloy." In: *ABZ 2008*. Ed. by Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca. Vol. 5238. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 294–308. DOI: 10.1007/978-3-540-87603-8_23.

Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser (2013). "File systems deserve verification too!" In: *PLOS 2013*. Farmington, Pennsylvania: ACM Press, pp. 1–7. DOI: 10.1145/2525528.2525530.

Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis (Jan. 2021). "PerSeVerE: Persistency Semantics for Verification under Ext4 (Supplementary Material)." In: URL: https://plv.mpi-sws.org/persevere.

Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis (2019). "Model Checking for Weakly Consistent Libraries." In: *PLDI 2019*. New York, NY, USA: ACM. DOI: 10.1145/3314221.3314609.

Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer (2017). "Repairing Sequential Consistency in C/C++11." In: *PLDI 2017*. Barcelona, Spain: ACM, pp. 618–632. DOI: 10.1145/3062341.3062352.

Linux man pages (2020). URL: http://www.man7.org/linux/man-pages/index.html (visited on May 20, 2020).

Richard Gooch (1999). *Overview of the Linux Virtual File System*. URL: https://www.kernel.org/doc/html/latest/filesystems/vfs.html (visited on May 20, 2020).

Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram (2018). "Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing." In: *OSDI 2018*. Carlsbad, CA, USA: USENIX Association, pp. 33–50. URL: https://www.usenix.org/system/files/osdi18-mohan.pdf (visited on Nov. 16, 2020).

Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu (2008). "Finding and Reproducing Heisenbugs in Concurrent Programs." In: *OSDI 2008*. USENIX Association, pp. 267–280. URL: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf (visited on Nov. 16, 2020).

GNU Nano (2019). *The GNU Nano homepage*. URL: https://nano-editor.org (visited on June 15, 2020).

Gian Ntzik and Philippa Gardner (Oct. 23, 2015). "Reasoning about the POSIX file system: local update and global pathnames." In: *OOPSLA 2015*. Pittsburgh, PA, USA: Association for Computing Machinery, pp. 201–220. DOI: 10.1145/2814270.2814306.

Daejun Park and Dongkun Shin (2017). "iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call." In: pp. 787–798. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/park.

Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau (Oct. 27, 2017). "Application Crash Consistency and Performance with CCFS." In: *ACM Trans. Storage* 13.3, pp. 1–29. DOI: 10.1145/3119897.

Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau (Oct. 2014). "All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications." In: *OSDI 2014*. Broomfield, CO: USENIX Association, pp. 433–448. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai.

Anton Podkopaev, Ori Lahav, and Viktor Vafeiadis (Jan. 2019). "Bridging the Gap Between Programming Languages and Hardware Weak Memory Models." In: *Proc. ACM Program. Lang.* 3.POPL, 69:1–69:31. DOI: 10.1145/3290382.

POSIX (2018). *The Open Group Base Specifications Issue 7*. URL: https://pubs.opengroup.org/onlinepubs/9699919799/ (visited on May 20, 2020).

Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau (2005). "Analysis and Evolution of Journaling File Systems." In: p. 16. URL: https://www.usenix.org/legacy/events/usenix05/tech/general/full_papers/prabhakaran/prabhakaran.pdf.

Azalea Raad and Viktor Vafeiadis (Oct. 2018). "Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model." In: *Proc. ACM Program. Lang.* 2.OOPSLA. DOI: 10.1145/3276507.

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis (Dec. 20, 2019a). "Persistency semantics of the Intel-x86 architecture." In: *Proc. ACM Program. Lang.* 4 (POPL), 11:1–11:31. DOI: 10.1145/3371079.

Azalea Raad, John Wickerson, and Viktor Vafeiadis (Oct. 10, 2019b). "Weak Persistency Semantics from the Ground Up." In: *Proc. ACM Program. Lang.* 3 (OOPSLA), 135:1–135:27. DOI: 10.1145/3360561.

renameio (2020). URL: https://github.com/google/renameio (visited on May 20, 2020).

Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell (2015). "SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems." In: *SOSP 2015*. Monterey, California: ACM Press, pp. 38–53. DOI: 10.1145/2815400.2815411.

Ohad Rodeh, Josef Bacik, and Chris Mason (Aug. 1, 2013). "BTRFS: The Linux B-Tree Filesystem." In: *ACM Trans. Storage* 9.3, 9:1–9:32. DOI: 10.1145/2501620.2501623.

Cindy Rubio-González, Haryadi S. Gunawi, Ben Liblit, Remzi H. Arpaci-Dusseau, and Andrea C. Arpaci-Dusseau (June 15, 2009). "Error propagation analysis for file systems." In: *SIGPLAN Not.* 44.6, pp. 270–280. DOI: 10.1145/1543135.1542506.

Gerhard Schellhorn, Gidon Ernst, Jörg Pfähler, Dominik Haneberg, and Wolfgang Reif (2014). "Development of a Verified Flash File System." In: *ABZ 2014*. Vol. 8477. Berlin, Heidelberg, pp. 9–24. DOI: 10.1007/978-3-662-43652-3_2.

Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang (2016). "Push-Button Verification of File Systems via Crash Refinement." In: *OSDI 2016*. Savannah, GA, USA: USENIX Association, pp. 1–16. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-sigurbjarnarson.pdf.

Seongbae Son, Jinsoo Yoo, and Youjip Won (2017). "Guaranteeing the Metadata Update Atomicity in EXT4 File system." In: *APSys 2017*, pp. 1–8. DOI: 10.1145/3124680.3124722.

SQLite (2020). URL: https://sqlite.org/index.html (visited on May 20, 2020).

Atomic Commit In SQLite (2020). URL: https://sqlite.org/atomiccommit.html (visited on May 20, 2020).

Adam Sweeney (1996). "Scalability in the XFS file system." In: *USENIX ATC 1996*, pp. 1–14. URL: https://www.usenix.org/legacy/publications/library/proceedings/sd96/sweeney.html.

Theodore Y Ts'o and Stephen Tweedie (2002). "Planned Extensions to the Linux Ext2/Ext3 Filesystem." In: pp. 235–243. URL: http://www.usenix.org/publications/library/proceedings/usenix02/tech/freenix/tso.html.

Stephen C Tweedie (1998). "Journaling the Linux ext2fs Filesystem." In: *LinuxExpo 1998*. URL: http://e2fsprogs.sourceforge.net/journal-design.pdf (visited on Nov. 16, 2020).

Vim (2019). *Vim - the ubiquitous text editor*. URL: https://vim.org (visited on June 15, 2020).

Junfeng Yang, Can Sar, and Dawson Engler (Nov. 6, 2006). "EXPLODE: a lightweight, general system for finding serious storage system errors." In: *OSDI 2006*. Seattle, Washington: USENIX Association, pp. 131–146. URL: https://www.usenix.org/legacy/event/osdi06/tech/full_papers/yang_junfeng/yang_junfeng.pdf (visited on June 17, 2020).

Mai Zheng, Joseph Tucek, Dachuan Huang, Elizabeth S Yang, Bill W Zhao, Feng Qin, Mark Lillibridge, and Shashank Singh (2014). “Torturing Databases for Fun and Profit.” In: *OSDI 2014*. Broomfield, CO: USENIX Association, pp. 449–464. URL: https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-zheng_mai.pdf (visited on Nov. 16, 2020).