Checking the Integrity of Transactional Mechanisms

Daniel Fryer, Mike Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, Ashvin Goel
University of Toronto

Abstract

Data corruption is the most common consequence of file-system bugs, as shown by a recent study. When such corruption occurs, the file system's offline check and recovery tools need to be used, but they are error prone and cause significant downtime. Previous work has shown that a runtime checker for the Ext3 journaling file system can verify that metadata updates within a transaction are mutually consistent, helping detect corruption in metadata blocks at commit time. However, corruption can still be caused when a bug in the file system's transactional mechanism loses, misdirects, or corrupts writes. We show that a runtime checker needs to enforce the atomicity and durability properties of the file system on every write, in addition to checking transactions at commit time, to provide the strong guarantee that every block write will maintain file system consistency.

In this paper, we identify the invariants that need to be enforced on journaling and shadow paging file systems to preserve the integrity of committed transactions. We also describe the key properties that make it feasible to check these invariants for a file system. Based on this characterization, we have implemented runtime checkers for a modified version of the Ext3 file system and for the Btrfs file system. Our evaluation shows that both checkers detect data corruption effectively, and they can be used during normal operation with low overhead.

1 Introduction

File systems contain bugs that are hard to detect even under heavy testing, as shown by researchers [25, 35] and painful real-world experiences [24]. These bugs can result in data corruption, data loss, or persistent application crashes. Today, most techniques that enhance the reliability of storage systems focus on recovery from crash failures, and a variety of storage hardware failures [12]. However, none of these methods address corruption caused by file system or operating system bugs, or random memory corruption [36]. For example, a mirror RAID offers no protection against a buggy file system write, which would be reliably replicated on multiple disks.

A comprehensive study recently showed that 40% of file system bugs have severe consequences, because they lead to in-memory or on-disk data corruption [18]. When a file system bug corrupts file-system metadata, the damage can propagate and thus the entire file system must be checked for possible corruption. This consistency check is typically performed offline, causing significant downtime for large storage systems [14]. Furthermore, repair is an error-prone process [13, 2].

To avoid downtime and data loss, file system corruption must be detected before it propagates to disk. To do so, the file system's write operations must be checked at *runtime*. Unlike a typical offline file system checker, such as fsck, that checks the consistency of the metadata already on disk, the Recon system [10] checks that metadata updates preserve the consistency semantics of the file system at runtime. These semantics are expressed as a set of invariants that are derived from the properties checked by the offline checker. When kernel bugs or memory corruption lead to metadata updates that violate these consistency invariants, a corruption is detected and the updates are prevented from reaching the disk.

Recon takes advantage of transactional methods, such as journaling [34, 7, 32] and shadow paging [15, 5, 26], used by modern file systems for providing crash consistency. In particular, it checks that metadata updates within a transaction are mutually consistent at transaction commit time. This approach is still vulnerable to file system corruption when the transactional mechanism is used incorrectly or has bugs. For example, the Recon checker for the Ext3 journaling file system verifies writes to the journal blocks, but it assumes that 1) all metadata writes first go to the journal, and 2) these writes are then checkpointed correctly. Any bugs that violate these assumptions, e.g., a lost or failed checkpointing write, will cause undetected corruption. In Section 2.2, we show that these bugs manifest in many different ways, such as lost, misdirected, out-of-order and corrupting writes,

making it difficult to detect them. Unfortunately, these types of bugs occur regularly [10], are hard to diagnose [11, 27], and can have serious impact [33].

In this paper, we describe the design and implementation of a runtime checking system that enforces correct usage and implementation of the crash consistency method used by the file system. Our system enforces the atomicity and durability properties of the file system at each block write, in addition to checking consistency at commit time, providing the strong guarantee that every block write will maintain file system consistency.

We express the atomicity and durability properties as invariants, called *location invariants* because they govern which blocks are written to given locations. We describe the location invariants that need to be enforced to preserve the integrity of committed transactions for journaling and shadow paging file systems, and the file system properties that make it feasible to check these invariants efficiently at the block layer.

We have implemented runtime checkers for the Btrfs file system and a slightly modified version of the Linux Ext3 file system by augmenting the Recon system. Our evaluation shows that the runtime checkers for both the file systems detect file-system corruption effectively, preventing any file system metadata inconsistency. We show that the Ext3 checker has low performance overhead, while the Btrfs checker overhead is higher due to increased metadata load. Checking location invariants in both checkers has negligible overheads.

2 Motivation

Our aim is to design a runtime checking system that can reliably detect file system and other operating system software bugs, and memory corruption errors, before they cause on-disk data corruption. Unlike an offline file system checker, a runtime checker does not detect file system corruption caused by I/O hardware failures, such as device controller failures or latent sector errors on disks. Instead, the runtime checker depends on hardware redundancy mechanisms, such as checksums and replication [25], implemented either in the storage system or in the file system [5, 9], to detect and recover from such failures when data is read from disk.

A runtime checking system can be deployed in either a development or a production setting. During development, a runtime checker can serve as a testing tool, catching subtle errors before the file system image becomes inconsistent, making it easier to determine the root cause of a bug. In production, the checker could trigger measures to preserve existing data, recover from the failure [31], or alert administrators to the problem. Our runtime checking system builds on the Recon system [10], and so this section starts by providing an overview of Recon. Then we motivate this work by discussing the types of bugs that Recon will fail to detect, leading to undetected data corruption.

2.1 The Recon System

The Recon system takes advantage of transactional methods, such as journaling and shadow paging, used by modern file systems for providing crash consistency. These transactional methods group writes to disk blocks from one or more operations (such as the creation of a directory and a file write) into transactions. When transactions are committed, the file system believes itself to be consistent. At this point, Recon checks that the contents of the blocks involved in the transaction are mutually consistent, thus detecting the effects of software bugs (or memory errors) that corrupt blocks within the transaction.

The consistency checks in Recon are derived from the consistency properties of the file system. These properties constrain the set of valid file system states that can be generated by an arbitrary sequence of file system operations. Typically, these properties are checked by the offline file system checker. For example, a consistency property in the Btrfs file system is that extents must not overlap. Checking this property requires a full scan of the extent tree, making it infeasible to perform at runtime. Instead, each consistency property is transformed into a local consistency invariant, which is an assertion that must hold for the transaction blocks to preserve consistency. In the Btrfs example, the consistency invariant is that when a new extent item is added to a tree, then the extent must not overlap with the previous or next extents in the extent tree. A runtime checker can enforce this consistency invariant by examining all updated extents and their adjacent extents.

The Recon system interposes at the block layer, and can be implemented in the host operating system, a hypervisor or a storage controller. The benefit of this approach is that the checker only depends on the the format and the consistency properties of the file system, rather than depending on the implementation of the file system, which may be buggy and cannot be trusted. File system formats and their consistency properties tend to be stable over time, even when the implementation changes significantly over time, or there are multiple different implementations of a particular file system.

The consistency invariants are expressed in terms of logical file-system data structures, such as the extent information in the Btrfs example. Since Recon interposes at the block layer, it uses an introspection approach, similar to semantically smart disks [30], to infer the types of blocks as they are accessed, and then interprets the block contents to derive the logical file-system data structures.

2.2 Problematic Bugs

Recon ensures that the blocks in a transaction are consistent, but it depends on the transaction mechanism being both implemented and used correctly. Below, we describe four classes of bugs that break these assumptions, and provide some examples of recent bugs in the Ext3, Ext4 and the Btrfs file system code deployed in "stable" Linux kernel releases.

Overwrite bugs: A write occurs to a location when it shouldn't have happened at all, either due to improper writing or flushing of buffers, or some other failure that causes a misdirected write. For example, Ext4 stores file system quota information as data in special quota files. The contents of these files are metadata, similar to directories, but they were overwritten in place without first writing to the journal, when the file system was used with certain mount options [16]. Recon's consistency invariants would not detect this problem because the journal would appear to be consistent. Similarly, a high profile bug was recently introduced in the Ext4 file system, in which the inode bitmap was modified without updating the journal, which could lead to occasional corruption [27]. Interestingly, after the corruption issue was reported, the developers at first mistakenly thought that the root cause was an incorrect update to the journal superblock [33]. This suggests that understanding, using and implementing the transactional mechanism is challenging and bug prone. In this case, if the file system is allowed to continue running, the transaction that was missing the inode bitmap update in the journal would commit, and the checkpoint of that transaction would bring everything back to a consistent state, with no one the wiser. Consistency problems only occur when an ill-timed crash forces recovery from the incomplete journal entries. When Recon is used in production, things actually become worse. Recon would detect that the journal contents are inconsistent, because the inode bitmap updates are missing (e.g., unallocated inodes would appear to change), and then discard the transaction and stop the file system. The inode bitmap, overwritten in place, would cause the file system to become inconsistent.

Lost write bugs: A write that should happen doesn't occur. For example, in a journaling file system, a lost checkpointing or recovery write will cause file system inconsistency even though the journal is consistent [12].

Write ordering bugs: The file system needs to enforce ordering of writes to disk at certain times. While the block layer may observe writes in the correct order, unless the correct disk barrier commands are sent, the disk or its controller may reorder writes, causing inconsistency of the on-disk state on a power loss. For example, Linux JBD2 journaling code maintains a pointer to the journal tail in a journal superblock. When the tail was updated, the journal superblock was not being flushed to disk before new transactions could reuse the newly freed journal space. On a power loss, the recovery code could replay old transactions containing blocks potentially overwritten in the journal by new transactions [17], including blocks from uncommitted transactions. Similarly, the Btrfs file system in multi-device setups (e.g., mirroring) would send barriers in the wrong order and not wait for all the barriers before writing the commit block [21]. These write ordering bugs would not be detected by Recon but they can cause serious file system inconsistencies.

Corrupting Write bugs: A write occurs to the correct location but its contents are corrupt. For example, the Ext3 journaling code modifies (escapes) its data blocks when they start with a magic code that identifies journal metadata blocks, to distinguish between the two types of blocks, similar to bit stuffing [1]. When Ext3 was used in data journaling mode, the recovery code had a bug that would unescape the wrong buffers, causing corruption of both the block that remains escaped, and the block that is wrongly unescaped [11]. This bug would not be caught by Recon's consistency invariants because the journal itself is not corrupt. However, blocks from committed transactions would be corrupt on disk following recovery.

3 Location Invariants

File systems that use transactional mechanisms for crash consistency provide atomicity and durability properties. Atomicity properties ensure that the file system will be able to roll back to a consistent state on a crash. Durability properties ensure that if a new version of a block is committed, it does not get rolled back or overwritten, except atomically as part of a subsequent transaction.

The problematic bugs described in Section 2.2 can cause corruption because they lead to violations of these properties. For example, a metadata overwrite that is not first committed to the journal violates atomicity, since we cannot roll back to the previous correct version of the block. Durability can be violated by either an omitted checkpoint write, or a write that corrupts a committed transaction in the journal, since updates that were successfully committed to the journal never reach the file system. Finally, in both journaling or shadow paging systems, a misdirected write that overwrites an allocated metadata block (e.g., a data block write that overwrites a metadata block) violates both atomicity and durability.

In this section, we first describe what is needed to detect violations of these properties, and then present the location invariants for journaling and shadow paging transactional mechanisms.

3.1 Enforcing Atomicity and Durability

The Recon runtime checker depends on the correctness of the file system's transactional mechanism to properly enforce the atomicity and durability of the metadata updates that it is checking. Unfortunately, in spite of Recon's distrust of buggy file systems, it assumes that the transactions themselves are implemented and used correctly. This assumption can be violated by several classes of bugs, as shown in Section 2.2. To detect these bugs, a runtime checker needs to enforce atomicity and durability invariants, in addition to consistency invariants. Consistency invariants apply to the contents of updated blocks; they need to be checked at transaction commit points because the file system does not guarantee that the updates are consistent until the commit. In contrast, the atomicity and durability invariants need to be checked on each block write, because they govern whether the write is permitted to the given location. Hence, we call them *location* invariants collectively. Rather than being derived from the offline checking tool, the location invariants are derived from the semantics of the transactional mechanism itself. In particular, they concern *overwrites* to the blocks, and the *ordering* of block write operations.

It is possible to enforce both atomicity and durability invariants on each write because they only depend on the correctness of committed metadata, which has already been checked using consistency invariants. Transactional techniques like journaling or shadow paging must first write metadata to unallocated blocks – for journaling, these are free blocks in the journal area, which must later be checkpointed back to the file system, while for shadow paging these may be any free blocks, which become part of the file system atomically at the commit point. To check that these properties are maintained, location invariants depend on information about block allocation and block type (data vs. metadata). The block allocation information must be based on committed metadata, since uncommitted changes to the allocation state may be rolled back following a crash. In particular, we must not permit a write to a block that has been freed in an uncommitted transaction, since we would not be able to recover the previous version of the block if the deallocation operation were rolled back.

As can be seen, correct checking of consistency and location invariants is interdependent. We begin from the assumption that the file system state on disk is consistent. Initially, this is the result of correct file system initialization, as is done by mkfs. Thereafter, each block write prior to a transaction commit is checked by the location invariants using the old, consistent, committed allocation and block type information. These checks ensure that the committed state is not corrupted. At the transaction commit point, the contents of the transaction are checked by the consistency invariants to ensure that the new file system state will be consistent. The location invariants then govern the write of the commit block itself, and the subsequent checkpoint writes to the file system, as well as the writes of blocks in the next transaction. By enforcing both consistency and location invariants, the runtime checker can provide the strong guarantee that the file system meets its consistency specification on every block write.[1]

As we will see in the next subsection, there are significant differences between the specific location invariants that apply to journaling and shadow paging mechanisms. However, both require the ability to infer block allocation information and the ability to distinguish between metadata and data blocks at the block layer.

3.2 Journaling Invariants

Journaling file systems use write-ahead logging to support failure atomicity. First, they write a consistent set of blocks and their final location information to a designated journal area. When all these blocks are durable in the journal, an atomic journal write signals a commit. After commit, the contents of the journal are flushed to their final locations. This flush to the final file system locations is called checkpointing in the Linux ext3/jbd terminology.

The journal area must be known to the runtime checker so that, on each write, it can distinguish between journal and non-journal writes. This distinction is necessary so that the correctness of both the journal writes and the checkpointing writes can be verified. Checkpointing of committed transactions occurs concurrently with new journal writes, but checkpointing writes must be directed to the non-journal area. Note that although we expect the journal to be a circular buffer, with writes occurring sequentially, at the block layer there is no guarantee of any particular ordering within a transaction.

The following four location invariants ensure that the journaling and checkpointing operations of the file system are correct:

1. Log invariant: A write to the journal area must be to a free block in the journal. A free journal block becomes allocated when it is written and free again when it has been checkpointed (see Checkpoint invariant below). This invariant checks that the allocated journal blocks are not overwritten.

2. Commit invariant: A write of a commit block, which marks a transaction as committed, is allowed to the journal area only after (1) all the blocks in

[1] While the checker implementation may have bugs that generate false alarms, it is unlikely that the checker will fail to detect file system corruption, unless its bugs are correlated with file system bugs [10].

the transaction are allocated in the journal, and (2) a barrier is issued to flush these transaction blocks to the disk. The transaction is considered to be committed (and hence, to be durable) only after the commit block is flushed to disk. When journal checksums are included in the commit block, as in IRON file systems [25], the write of the commit block can be concurrent with the writes of the transaction blocks, but a barrier is still needed to ensure that all these blocks are on disk before the transaction is deemed to be committed.

3. Flush invariant: A write to an allocated, non-journal location is permitted only when (1) the committed part of the journal contains a block that is destined for the same final location, and (2) the contents of this block in the journal matches the contents of the block being written. In other words, overwrites of allocated non-journal blocks are disallowed if the new content was not first committed to the journal. If the block exists only in the uncommitted portion of the journal, or the block does not appear in the journal at all, both atomicity and durability violations can occur. Atomicity is violated by writing new content into the file system ahead of the commit of the transaction that should contain it. Durability is violated by the loss of previously committed content that has been overwritten.

4. Checkpoint invariant: A write of a checkpoint record (e.g., in the journal superblock), which indicates that a set of blocks in the journal area are now free, is permitted only after all the journal blocks for the associated transaction have been either (1) flushed (see Flush invariant), or (2) superseded by a newer version of the corresponding block in a later committed transaction. If a newer version of a block exists in a later committed transaction in the journal, then this version does not need to be flushed before being freed. The affected journal blocks can only be considered free after the checkpoint record has been flushed to disk.

Metadata-only Journaling Since writing to the journal potentially doubles the total write traffic to disk, many file systems allow journaling only metadata blocks to reduce write traffic. The main complication with metadata-only journaling is that data writes are non-atomic, and while these writes must be allowed at any time, they must not overwrite metadata blocks. To accommodate non-journaled data writes, we refine the journaling flush invariant with an exception:

1. Data-flush exception: Any non-journal write that violates the flush invariant must be to a non-metadata (data or free) block location. The type of a block (metadata or not) is determined by the committed file-system state. The consequence of this exception is that data writes can overwrite data blocks unimpeded. Unfortunately, there is no way to tell if data writes are misdirected among each other.

The challenge with allowing this exception is that it must be possible to distinguish metadata blocks from non-metadata blocks on each write, but a file system may not provide this information easily. For example, the Ext3 file system uses allocation bitmaps that allow distinguishing between allocated blocks (which may be data or metadata) and free blocks. However, the file system does not provide an easy way to distinguish between dynamically allocated metadata (e.g., for directories and indirect blocks) and data blocks, other than by traversing the entire file system. We discuss this issue further in Section 4.3.

3.3 Shadow Paging Invariants

Compared to journaling, it is simpler to enforce location invariants for shadow paging systems because blocks are updated once per transaction and all these updates occur before commit. In a file system that uses shadow paging for all blocks, there are two atomicity invariants:

1. Flush invariant: All writes, other than to special non-shadow paged blocks, such as the super block, must be to unallocated blocks. This invariant follows from the basic copy-on-write properties of shadow paging systems. To enforce this invariant, the file system must provide an efficient method for determining the allocation status of a block. For example, the Btrfs file system maintains an extent allocation tree.

2. Commit invariant: The write of the commit block (usually a tree root) is flushed to disk only after both (1) all blocks referenced by the new tree have been updated, and (2) a barrier is issued to flush these blocks to disk. That is, there must be no dangling pointers to potentially uninitialized blocks, before the commit block is flushed.

Durability (e.g., a lost or corrupting update) is checked in modern shadow paging file systems using methods such as block checksums (ZFS) or generation numbers (Btrfs). This information is embedded in metadata blocks, and hence our Btrfs runtime checker uses consistency invariants to check the consistency of block headers and generation numbers for ensuring durability.

Metadata-only Shadow Paging Shadow paging can lead to fragmentation because the updated blocks are placed in new, possibly distant, physical locations. Fragmentation can be reduced with metadata-only shadow paging, with data writes being performed in place. To accommodate non-atomic data writes, we refine the flush invariant with an exception:

1. Data-flush exception: Any write that violates the flush invariant must be to a non-metadata (data or free) block location.

This exception requires being able to distinguish metadata and non-metadata blocks. Btrfs tracks whether an extent has metadata or data in its allocation tree, making it easy to enforce this invariant. Also, the default behavior of Btrfs is to separate metadata and data regions, making this identification even easier and more efficient.

4 Implementation

As explained in Section 3.1, location and consistency invariants are interdependent, and they need to be checked together. Hence, we have implemented location invariant checking for the Linux Ext3 (journal invariants) and Btrfs (shadow paging invariants) file systems by augmenting the Recon consistency checking system. Recon uses the block-layer Linux device mapper framework to interpose on block I/O, allowing location invariants to be checked on all writes. The block-layer approach ensures the independence of the checker and the file-system implementations. Next, we describe the requirements for implementing a runtime checker, and then discuss how these requirements are met in our implementation.

4.1 Runtime Checker Requirements

File system design impacts the capabilities and performance of a runtime checking system. In this section, we present the four types of information needed by a checker. The challenge is to obtain this information correctly and efficiently at the block layer. The more file system state that must be examined to do so, the higher the overhead of the checker.

Consistency Points: Runtime checking at the block layer requires being able to get a consistent picture of the file system state from outside the file system. Consistency points provide both a point in time to check consistency invariants and a consistent view of the file system when checking location invariants on each write.

Allocation Information: A checker needs to distinguish between allocated and unallocated blocks, particularly on the write path, to protect against accidental overwrites. Overwriting an unallocated block is harmless, but location invariants constrain when allocated metadata blocks can be overwritten.

Separate Metadata: The checker also needs to distinguish between metadata and data blocks on both the read and the write paths. Metadata blocks are cached to improve checker performance, since recently accessed metadata is likely to be relevant to invariant checking, while data blocks are ignored because they are not interpreted. Additionally, the location invariants may permit or forbid a write depending on whether the destination is a data or metadata block.

Block Identity: Finally, interpreting a metadata block requires knowing the *identity* of the block. The block identity determines the logical contents of the block in the file system. For example, suppose that the checker knows that some block is an inode block, and it identifies the block as the fourth inode block in the file system. If it knows that inode blocks contain 32 inodes, then it can determine that this block contains inodes with numbers 97-128. A runtime checker can then correlate these inodes with directory entries that reference them, with inode bitmaps that allocate them, and with the indirect blocks to which they point. Without knowing their specific identities, it would not be possible to make the associations between the data structures that are needed for enforcement of invariants.

4.2 Block-Layer Metadata Interpretation

In this section, we discuss two complementary approaches for determining block identity. The following sections describe how we apply them to interpret metadata in the Ext3 and Btrfs checkers.

Forward Pointers: File systems are tree structures or directed acyclic graph structures, with parent blocks containing some form of a pointer to child blocks. Thus, the easiest way to identify a block is if we are already traversing the parent block. For example, if the checker (or the file system) is looking up some specific metadata, starting from the root of the tree, it can traverse intermediate blocks to locate the desired block.

Back References: A back reference for a block is metadata that maps the block's physical location to blocks that reference the block [20], providing an efficient method for locating parent blocks. Back references are used for various tasks such as defragmentation and bad block replacement, in which the parent block containing the reference must be efficiently located and updated. The parent block has information to help type and identify the child block, and hence back references greatly simplify metadata interpretation. However, looking up a back reference may incur additional I/O operations.

4.3 Ext3 Implementation

Ext3 uses static block allocation bitmaps, making it easy for the checker to determine the allocation status of blocks. However, Ext3 does not provide any efficient method for distinguishing metadata blocks from other blocks, either on block writes or on block reads that violate pointer-before-block traversal. One option is for the checker to also create in-memory back references for all data blocks when the parent metadata blocks are traversed. This approach would greatly inflate the memory overhead of the checker. Instead, we have retrofitted the Ext3 file system with a metadata allocation bitmap which records whether a given block is metadata. The new metadata bitmap is stored alongside the block allocation bitmap. Using the metadata bitmap,[2] the checker can ensure that data blocks are never cached on either a read or a write, and the data flush exception, described in Section 3.2, can be implemented easily.

4.3.1 Interpreting Metadata

The Ext3 file system does not provide back references. Instead, we use the file system's forward pointer traversal to create in-memory back references dynamically. The file system needs to read the parent of a block at least once before it accesses the child block, which we call pointer-before-block traversal. When the parent block is read the first time, we create a back reference for each of the child blocks to which it points. For example, when an inode block is read by Ext3, we copy the block into the read cache, parse the inodes in the block, and then create back references for all child metadata blocks (e.g., indirect blocks) directly pointed to by the inodes. The back reference contains the block type, and for an indirect block, it contains the inode number and an offset that locates the indirect block. When the indirect block is read, its back reference will exist, and hence the block can be typed and identified. These back references are bootstrapped using the superblock, which exists at a known location.

The main drawback of in-memory back references is that they cannot be evicted because the file system may cache information from the parent block indefinitely, allowing it to access the child block directly at any time in the future. However, the in-memory references could be persisted by leveraging the backpointer-based consistency techniques developed in NoFS [6] and ffsck [19].

4.3.2 Location Invariants

The Ext3 location invariants require tracking the state of the journal. The checker maintains three data structures: a list of transactions currently present in the journal, an array containing information about the status of each block in the journal, including block checksums, and a hash table mapping from physical block numbers to versions of that block in the journal. A block in the journal can be in one of four states: logged, committed, flushed, and free. These four states correspond to the four journaling invariants described in Section 3.2. Note that a block stays allocated (as explained in the Log invariant) during the logged, committed, and flushed states.

Based on writes to different types of journal blocks (i.e., the descriptor blocks, metadata blocks, commit blocks, and the journal superblock) and non-journal blocks, the checker updates its data structures and the block states, and enforces the journaling invariants described in Section 3.2.

During a commit, if a new metadata pointer is found without a corresponding new metadata block in the journal, we detect a violation of the commit invariant (that all blocks should have been written before commit).

One complication with metadata-only journaling is that Ext3 uses *revoke* records to indicate that a metadata block has been freed, and could be reused as a data block that is updated non-atomically. As a result, any versions of this block in previous transactions should no longer be checkpointed or else the data block could be overwritten. The checker handles such revoked blocks by marking their status as checkpointed, so that the Checkpoint Invariant does not fail if the containing transaction is freed without seeing a write to that block.

4.4 Btrfs Implementation

Btrfs provides various features such as extent-based allocation (which allows a single allocation record to cover multiple blocks), back references (which help tasks like online defragmentation) and writable snapshots (which are isolated from the original version using copy-on-write semantics). Btrfs uses shadow paging for ensuring crash consistency, similar to the WAFL file system [15].

Btrfs uses multiple B-trees to store its metadata. A root B-tree contains pointers to the roots of other B-trees, including the main file system tree, snapshot trees, and an extent tree that records allocation information. Each B-tree consists of internal nodes and leaves. Internal nodes contain an array of key/block-pointer pairs, with the key representing the smallest key stored in the pointed-to node or leaf, and the block pointer helping locate the child node or leaf on disk. All Btrfs metadata blocks begin with a header that has a block checksum, a generation number, and the id of the tree containing the block.

We found that Btrfs can issue writes from concurrent transactions. For example, blocks from the next transac-

[2] Note that the consistency checker implements additional invariants for checking metadata bitmap consistency.

tion can be written to disk before the current transaction commit, but as expected, the next transaction blocks are unreachable from the current transaction. As a result, the Btrfs checker assumes that unreachable blocks belong to a future transaction and delays processing them.

4.4.1 Interpreting Metadata

Btrfs uses shadow paging, so that when a leaf node is updated, all its ancestor nodes are also updated. Because of this property, the checker can use forward pointer traversal on commit, starting from the superblock.

Btrfs uses an extent B-tree to store allocation information, which the checker also uses to determine the allocation status of blocks. Similarly, separating data and metadata blocks on both the read and write paths is relatively easy because Btrfs allocates separate large contiguous regions for data and metadata. However, if Btrfs is operating in a "mixed" region mode (not a common configuration), data extents can be distinguished from metadata extents by traversing the extent allocation tree and examining the per-extent flags.

Btrfs uses typed and self-identifying metadata blocks. Each metadata block has a header that stores the type (node or leaf) and level of the block in the tree, and the first key in the block is its identity, helping locate the block in the tree. Btrfs also supports back references to multiple snapshots, storing them with the allocation information in the extent tree.

Both back references and self-identifying metadata blocks can be used independently to type and identify blocks. We initially decided to implement a Btrfs runtime checker because we thought that both of these properties would be useful for the runtime checker. However, neither are necessary due to the forward pointer traversal enabled by shadow paging.

4.4.2 Location Invariants

The checker ensures atomicity and durability by checking that allocated blocks are never overwritten, which requires looking up the extent allocation tree on each write. For metadata-only shadow paging, a metadata flag in the extent record is checked to implement the data-flush exception. While checking a transaction for consistency, an invariant is tripped if a pointer to an unwritten block is encountered within the updated tree.

5 Evaluation

We evaluate our runtime checker in terms of its ability to detect violations of the location invariants, listed in Section 3, and the performance impact of checking location invariants in addition to consistency invariants for the Ext3 and Btrfs file systems. We have implemented the runtime checkers within the Linux kernel using the Recon framework, based on the approach described in the previous sections. Our Btrfs implementation is based on the Linux 2.6.35 kernel, which does not support passing disk barrier and flush requests through the device mapper, and so we cannot check for them. Recon for Ext3 is implemented and tested on Linux 3.8.11.

5.1 Correctness

We evaluate the ability of our runtime checker to detect the types of bugs described in Section 2.2. Specifically, we inject errors into write operations issued to the block layer that result in lost, misdirected, or corrupted writes. We refer to these injected errors as corruptions. If writes are correctly ordered, and no writes are lost, misdirected, or corrupt, then the transaction mechanism is working correctly. By deliberately altering writes to violate these properties, we can evaluate whether the location invariants can successfully protect the file system.

Our corruptor sits between the file system and the checking system, and has the opportunity to act before each write is visible to the checker. The actions the corruptor can take are: 1) discard a write (lost), 2) alter the destination of the write (misdirect), or 3) alter a range of bytes within the block being written (content). Because the location invariants distinguish between several different types of blocks, we perform corruption in a type-specific manner to increase our coverage of possible scenarios and to help explain any uncaught corruption. The type of a block is determined by its destination, and in the case of certain journal blocks, by the journal header stored at the beginning of the block.

5.1.1 Corrupting Ext3

The corruptor can target one of four types of journal blocks (journal metadata such as a descriptor block, revoke, and commit blocks, and journaled file system metadata block), or the two types of non-journal blocks (file system metadata and data). To misdirect writes, it must distinguish between free and allocated journal space, and data and metadata locations outside the journal. When the corruptor targets a non-journal metadata block write, it is emulating a bug that corrupts the checkpoint write of that metadata block. When the corruptor targets a data block write, it always misdirects the write to a non-journal metadata block. Lost write and content corruption types are not applied to data block writes.

Some corruptions may not violate location invariants immediately. Instead, they may lead to a future operation causing metadata corruption. For example, a lost write to the journal cannot be detected when it is dropped, and

Target Block Type	Corruption Type		
Journal Blocks	Lost	Misdirect	Content
Descriptor	10	10	8
Commit	10	10	4
Revoke	10	10	4
Metadata	10	10	2
Non-Journal Blocks			
Metadata	3	10	10
Data	N/A	10	N/A

Table 1: Corruptions detected by location invariants.

the resulting transaction may still be consistent, but the problem should be detected when the checkpoint write targets a metadata location that has not been committed to the journal. There are four distinct points in time when a corruption may be detected: during the corrupted write, at the next commit point, during the checkpoint of a corrupted transaction, and during transaction free. Any corruption that occurred in the past must be caught before a write harms the atomicity, durability, or consistency of metadata on disk.

There are a total of 16 combinations of target block types and corruption types, as shown in Table 1. We perform 10 corruptions per combination. Out of 160 corruptions, 131 were detected by the location invariants and 7 were detected by the consistency invariants (all were content corruptions of metadata blocks in the journal). We analyzed the remaining 22 corruptions that did not trigger any invariant violations. There are two situations in which we miss corruption events, but the "corruptions" do not affect file system integrity. In the case of random content corruptions to journal metadata, much of the space in the block is unused and corruptions to the unused area have no effect on the block semantics. Together, these cases account for 14 of the missed corruptions. Similarly, when unused space in a journaled metadata block is corrupted, which occurred in one case, no invariants are violated. We verified that the corrupted space was unused by logging the range of bytes corrupted and examining the target blocks. The final 7 missed corruptions were all lost checkpoint writes. In each case, we verified that these writes were safe to omit because there was already a newer version of the block committed to the journal. In all the 22 cases where we didn't catch the corruption, the e2fsck offline checker also reported that the file system was consistent.

5.1.2 Corrupting Btrfs

Testing the Btrfs location invariants is less involved, since the invariants are simpler, as described in Section 3.3. A buggy write in Btrfs can be redirected to overwrite an existing data or metadata block, lost, or redirected to the wrong free block. We simulated a metadata block being misdirected by the file system by changing the block's header to match the new, incorrect location, and updating the block checksum accordingly. Our checker always detected misdirections that cause overwrites of allocated data or metadata. Lost writes or writes that are misdirected to an incorrect free block are always detected by Recon during transaction processing, when a new pointer is found to a block that is missing from its write cache [10].

5.2 Performance

For benchmarking, we select three workload profiles with different behaviors from the Filebench workload generator. The varmail profile performs many small, synchronous writes. The webserver profile reads many small files concurrently (100 threads) in a large directory hierarchy (250,000 files), while appending to a log. The ms_nfs profile simulates a network file server, operating on a file system with a file size distribution from a study of Windows desktops [23].

All benchmarks were run on a dual-core 3.0 Ghz Xeon server with 4GB of RAM. The target disk for the benchmarks was a 250GB 7200rpm SATA drive. We allocated 256 MB of memory to the Recon caches [10]. The performance results account for Recon's memory usage because Linux implements a shared page cache, and so with Recon, this memory is not available to the file system cache.

Figures 1 and 2 show the benchmark throughput, and the time to initialize the benchmark's file system tree (setup time), averaged over 5 runs, for the Ext3 and the Btrfs file systems. Each graph shows the performance of the native file system, the file system with consistency checking enabled, and the file system with consistency and location checking enabled. These figures show that the overhead of checking location invariants is minimal compared to the existing overhead of checking consistency invariants in Recon. Even though the location invariants require a check on every write, this check is usually quick and takes advantage of the cached metadata.

Figure 1 shows that runtime consistency checking has relatively low overhead for the Ext3 file system. Figure 2 shows that the checking overhead is slightly higher for the Btrfs file system. The total amount of metadata is higher in Btrfs, due to its increased internal redundancy and larger data structures, putting more pressure on the Recon caches. The rapid reallocation of metadata blocks in a copy-on-write system makes it important to promptly evict blocks in the Recon caches that are no longer referenced.

The varmail and ms_nfs profiles show minimal overheads. Surprisingly, the most significant impact on per-

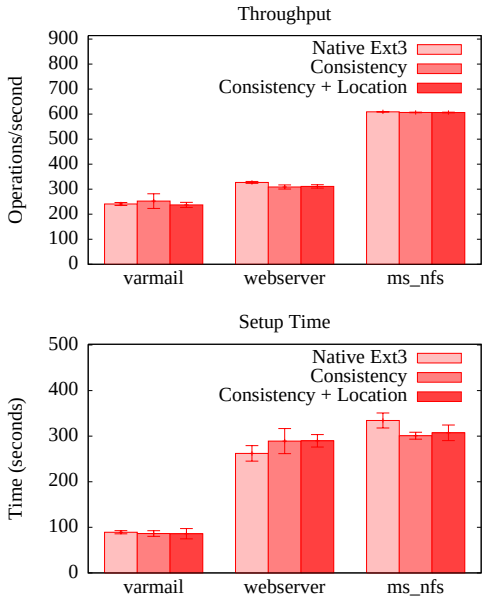

Figure 1: Performance on FileBench workloads for Ext3

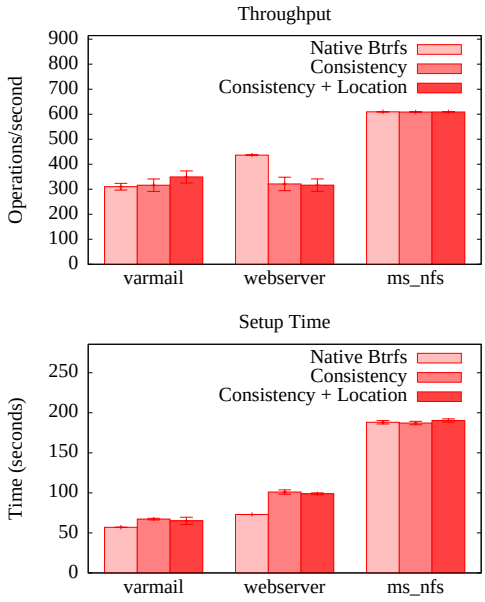

Figure 2: Performance on FileBench workloads for Btrfs

formance occurs for the webserver profile, which is a read-heavy benchmark. On closer analysis, we found that it generates significant metadata write traffic due to timestamp updates, which affects read performance. With the `noatime` mount option enabled, performance reaches roughly 90% of native performance.

6 Designing Checkable File Systems

Section 4.1 describes the four requirements of a runtime checker that make it feasible to check invariants efficiently at the block layer: 1) well-defined consistency points, 2) easily accessible allocation information, 3) easily distinguishable data versus metadata blocks, and 4) easily available block identity information. In this section, we describe how well various file systems meet these requirements. Table 2 provides a summary of our analysis. Then we recommend features that make file systems easily checkable at runtime.

6.1 Analysis of File System Design

No-Ordering FS: NoFS [6] aims to provide file system consistency in the face of poorly-behaved hardware that ignores ordering constraints and flush commands. They propose a novel commit-less approach to providing crash consistency by adding a backpointer to every block by using the out-of-band bytes provided by some devices, enabling atomic write of the block and its backpointer together. The backpointer makes it possible to identify the contents of blocks. NoFS performs block allocation based on an in-memory bitmap, thus avoiding any consistency issues between pointers and a persistent bitmap. Determining the allocation status at the block layer is expensive because it requires reading the block and its parent block to determine if a bidirectional pointer relationship exists between them. Unfortunately, NoFS does not provide any ordering guarantees by design, and thus lacks consistency points, and any consistency or location invariants. As a result, it is not possible to check any invariants in NoFS. Bugs in NoFS that cause data corruption would not be easily detectable by an offline checker as well.

FFS with Soft Updates: The soft updates mechanism provides crash consistency in an update-in-place file system without requiring journaling. Soft updates impose a partial order on writes and prevent cyclic dependencies between blocks by using a temporary in-memory rollback mechanism. Blocks and inodes can "leak" after a crash, but this problem is much less severe than blocks or inodes being overwritten while still in use. The ordering of writes allows some invariants to be checked (for example, you can't write a pointer to a newly-allocated block before you initialize the block). However, soft updates are not transactional and thus lack consistency points, and so most file system invariants cannot be checked because data might always be in flight.

Ext3: We have described the Ext3 file system properties in detail in Section 4.3. Ext3 provides consistency points and allocation information, but it mixes dynamically allocated metadata (directory data and indirect blocks) with data, thus requiring a full file system scan to distin-

	Consistency Points	Allocation Information	Separate Metadata	Block Identity
NoFS				X
Soft Updates		X		
Ext3	X	X		
REXt3	X	X	X	X
Btrfs	X	X	X	X

Table 2: Designing Checkable File Systems

guish data from metadata at the block layer. Instead, we distinguish the two types of blocks by retrofitting Ext3 with a metadata bitmap, as described in Section 4.3. In addition, the dynamically allocated metadata blocks cannot be easily identified, because they do not contain type information or information about the inode that points to them. We solve this problem by using in-memory back references in the checker.

REXt3: REXt3 [19] is a variant of ext3 that is optimized for a fast, offline file system checker called ffsck. Speeding up offline fsck involved two changes to the file system format, the co-location of metadata within metadata regions, and the addition of backpointers associating dynamically allocated metadata with their corresponding inodes. The separation of metadata and data into two regions makes it possible to distinguish between them with low overhead. With the addition of backpointers, the runtime checker for REXt3 will not need to use in-memory back references, thus reducing the memory overhead of the checker.

Btrfs: We have described the Btrfs properties in detail in Section 4.4. Btrfs provides consistency points, and it uses a separate extent tree to store allocation information. The extent records specify whether an allocated extent is data or metadata, and also record backpointers for the extent. Since Btrfs allows snapshots, some extents (both data and metadata) may have multiple parent blocks which point to them. A runtime checking system can identify metadata by its placement in a designated area, or by looking up the metadata flag in the extent tree. Furthermore, the contents of a metadata block can be identified based on the header structure shared by all metadata blocks. The shadow paging location invariants are easier to verify than their journaling equivalents because there is less state that needs to be tracked.

6.2 Design Recommendations

Based on our analysis of these file systems, we now suggest design features that enable efficient runtime checking of file systems. We expect that these same features will help implementing other file-system aware storage applications, such as differentiated storage services [22]. Consistency points are essential for runtime checking. While new file systems, possibly running on new hardware, may avoid providing consistency points, the resulting loss in protection is a serious issue. Easily accessible allocation information at the block layer, such as in bitmaps in fixed locations, allows enforcing location invariants efficiently. Other applications, like scrubbers and secure delete utilities, can also benefit from knowing the allocation status of a block. Separating data from metadata in well-defined regions allows distinguishing between them with low overhead because there is no need to lookup this information in bitmaps or trees. This approach also allows other policies, such as replication and placement, to be applied to contiguous metadata regions with ease. Fortunately, the mixing of metadata and data for performance reasons has been obsoleted by large disk caches [19]. Finally, backpointer information helps identify blocks at the block layer efficiently. This information is especially useful for dynamically allocated metadata in update-in-place file systems, because the checker may need to interpret an arbitrary block without knowing its position in the file system tree.

7 Related Work

We describe closely related work in the areas of runtime and offline file system consistency checking, and smart disk interfaces. Static bug finding tools [35] can reveal scores of bugs in file systems, but they can suffer from typical scalability issues, necessitating runtime checking. ZFS [5] uses a checksum-based runtime consistency checker for detecting and repairing file system corruption caused by storage hardware, e.g., latent sector errors, but it may not detect corruption caused by software bugs. Based on several requests, a check for location and some consistency invariants was added to Btrfs as a debugging tool [4]. These checks catch common errors, but they are embedded within the file system code itself, and so, for example, a file system bug could disable them. EnvyFS [3] uses N-version programming for detecting file system bugs at runtime. It uses the common VFS interface to pass each VFS-layer file system request to three child file systems and uses voting when returning results. The runtime overheads of this approach are high and subtle differences in file system semantics can make it hard

to compare results. HARDFS [8] detects software bugs in the Hadoop distributed file system (HDFS) at runtime by interposing on network messages and I/O, and verifies that the HDFS implementation behaves according to its operational specification. The verification state is compressed using bloom filters, significantly reducing the memory overhead. HARDFS can check certain end-to-end properties that a consistency checker cannot, such as whether a request was performed, but HARDFS does not attempt to catch all failures or guarantee that it will not raise false alarms.

Once a bug is detected at runtime, Membrane [31] proposes tolerating bugs by transparently restarting a failed file system. It assumes that file system bugs will lead to detectable, fail-stop crash failures. However, inconsistencies may have propagated to the on-disk metadata by the time the crash occurs. Our approach is complementary to Membrane, rather than waiting for the file system to crash, a restart could be initiated when a runtime checker detects an invariant violation.

Recently, there has been significant interest in improving the performance and robustness of offline consistency checkers. The rext3 file system [19] uses backpointers and collocates its metadata blocks, allowing its ffsck checker to scan the file system at rates close to the sequential bandwidth of the drive. Chunkfs [14] reduces the time to check consistency by breaking the file system into chunks that can be checked independent of each other. The SQCK offline consistency checker [13] expresses file system consistency properties declaratively, demonstrating that file system checks and repairs are more easily understood when expressed as SQL queries. It improves upon the repairs made by e2fsck by correcting the order in which certain repairs are performed and by using redundant information already provided by the file system. The SWIFT tool [2] tests the correctness of offline file system checker recovery code by leveraging the file system checker itself or by comparing the outputs of multiple checkers.

Our checker leverages ideas from semantically-smart disks [30], which use probing to gather detailed knowledge of file system behavior, allowing functionality or performance to be enhanced transparently at the block layer. Sivathanu et al. [29] provide a logic of file systems that helps reason about the correctness of smart disks. I/O shepherding [12] builds on smart disks, allows a file system developer to write reliability policies to detect and recover from a wide range of storage system failures. Unlike smart disks, a type-safe disk extends the disk interface by exposing primitives for block allocation [28], which helps enforce invariants such as preventing accesses to unallocated blocks.

8 Conclusion

We have presented the design of runtime file system checkers that can reliably detect file system bugs before they cause file system inconsistency. We show that the runtime checker needs to check location invariants on every write. These invariants enforce the atomicity and durability properties of the file system, helping preserve the integrity of committed transactions. Together with checking consistency properties on commit, the checker can provide the strong guarantee that every block write will preserve file system consistency.

We have implemented runtime checkers for the Ext3 journaling file system and the Btrfs copy-on-write file system. Our experimental results show that while consistency checking imposes some performance overhead, checking location invariants has almost no additional overhead. The Ext3 file system checker has low overhead but the Btrfs checker has higher overhead due to a higher metadata load. We are currently working on improving the Btrfs checker performance with better caching policies. Btrfs keeps a log to enable fast sync operations. We plan to implement our journaling invariants for this log. We expect that the checker overhead will be higher on faster storage devices, such as flash. We plan to evaluate this overhead in detail in the future.

We have shown that four file system features ease the design of runtime checkers, and enable checking invariants efficiently: 1) consistency points at which the file system is expected to be consistent on disk, 2) easily accessible allocation information at the block level, 3) distinguishable data versus metadata blocks at the block layer, and 4) backpointers for block typing and identification. We expect that these file system features will benefit other file-system aware storage applications as well.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Remzi Arpaci-Dusseau, for their detailed comments on this work. We also thank Andrei Soltan for designing and implementing the metadata bitmap for the Ext3 file system. We had many discussions and received feedback about this work from several members of the Computer Systems and Networking group and the SSRG group at the University of Toronto. Ali Hashemi provided invaluable system administration support. This research was supported by NSERC through the Discovery Grants and Graduate Scholarships programs.

References

[1] Bit stuffing. http://en.wikipedia.org/wiki/Bit_stuffing.

[2] Jo ao Carlos Menezes Carreira, Rodrigo Rodrigues, George Candea, and Rupak Majumdar. Scalable testing of file system checkers. In *Proceedings of the ACM SIGOPS European Conference on Computer Systems (Eurosys)*, pages 239–252, 2012.

[3] Lakshmi N. Bairavasundaram, Swaminathan Sundararaman, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Tolerating file-system mistakes with envyfs. In *Proceedings of the USENIX Technical Conference*, June 2009.

[4] Stephen Behrens. Btrfs: runtime integrity check tool, November 2011. http://lwn.net/Articles/466493.

[5] J. Bonwick and B. Moore. ZFS - The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf.

[6] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2012.

[7] Helen Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[8] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. HARDFS: Hardening HDFS with selective and lightweight versioning. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2013.

[9] Chris Mason et al. Btrfs. http://btrfs.wiki.kernel.org.

[10] Daniel Fryer, Kuei Sun, Rahat Mahmood, Tinghao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage*, 8(4):15:1–15:29, December 2012.

[11] Duane Griffin. jbd: correctly unescape journal data blocks, March 2008. http://kerneltrap.org/mailarchive/git-commits-head/2008/3/20/1206404/thread.

[12] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving file system reliability with I/O shepherding. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 293–306, 2007.

[13] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. SQCK: A declarative file system checker. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, December 2008.

[14] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep)*, 2006.

[15] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Technical Conference*, 1994.

[16] Jan Kara. ext4: Always journal quota file modifications, June 2010. http://www.kerneltrap.org/mailarchive/linux-ext4/2010/6/2/6884775.

[17] Jan Kara. jbd: Write journal superblock with WRITE_FUA after checkpointing, April 2012. https://git.kernel.org/cgit/linux/kernel/git/tytso/ext4.git/commit/?id=fd2cbd4dfa3db477dd6226d387d3f1911d36a6a9.

[18] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of Linux file system evolution. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2013.

[19] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. ffsck: The fast file system checker. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2013.

[20] Peter Macko, Margo Seltzer, and Keith A. Smith. Tracking back references in a write-anywhere file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[21] Chris Mason, November 2011. https://git.kernel.org/cgit/linux/kernel/git/tytso/ext4.git/commit/?id=387125fc722a8ed432066b85a552917343bdafca.

[22] Michael Mesnier, Feng Chen, Tian Luo, and Jason B. Akers. Differentiated storage services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 57–70, 2011.

[23] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[24] Rich Miller. Joyent Services Back After 8 Day Outage, January 2008. `http://www.datacenterknowledge.com/archives/2008/01/21/joyent-services-back-after-8-day-outage/`.

[25] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 206–220, 2005.

[26] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[27] Eric Sandeen. ext4: fix unjournaled inode bitmap modification, October 2012. `https://lwn.net/Articles/521819/`.

[28] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-safe disks. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 15–28, 2006.

[29] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A logic of file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[30] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *USENIX Conference on File and Storage Technologies (FAST)*, pages 73–88, 2003.

[31] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-dusseau, Remzi H. Arpaci-dusseau, and Michael M. Swift. Membrane: Operating system support for restartable file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.

[32] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the USENIX Technical Conference*, pages 1–14, 1996.

[33] Theodore Ts'o. Re: Apparent serious progressive ext4 data corruption bug in 3.6.3, October 2012. `https://lkml.org/lkml/2012/10/23/690`.

[34] Stephen C. Tweedie. Journalling the ext2fs filesystem. In *Proceedings of the 4th Annual Linux Expo*, May 1998.

[35] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, 2006.

[36] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end data integrity for file systems: a ZFS case study. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2010.