Consistency Without Ordering

Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin, Madison

Abstract

Modern file systems use ordering points to maintain consistency in the face of system crashes. However, such ordering leads to lower performance, higher complexity, and a strong and perhaps naive dependence on lower layers to correctly enforce the ordering of writes. In this paper, we introduce the No-Order File System (NoFS), a simple, lightweight file system that employs a novel technique called backpointer-based consistency to provide crash consistency without ordering writes as they go to disk. We utilize a formal model to prove that NoFS provides data consistency in the event of system crashes; we show through experiments that NoFS is robust to such crashes, and delivers excellent performance across a range of workloads. Backpointer-based consistency thus allows NoFS to provide crash consistency without resorting to the heavyweight machinery of traditional approaches.

1 Introduction

One of the core problems in file systems research over the years has been the challenge of providing consistency in the presence of system crashes. There have been a number of solutions to tackle this problem: from the simple file-system check [20] of the Fast File System [18] to the complicated copy-on-write mechanism of ZFS [3]. Each approach has a different core technique: write-ahead logging [12], copy-on-write [15] or tracking dependencies among writes to disk [10].

Although these approaches all differ vastly in their details, they share one common trait: each uses a careful ordering of writes to implement its update protocol. Journaling file systems require that metadata and data are persisted before the commit record is written [2, 31, 41, 45]. Copy-on-write file systems require that the root block be updated only after the rest of the update is safely on disk [15, 32, 40, 48]. Soft updates is built entirely around the careful ordering of disk writes [10].

In the event of a crash, ordering points allow the file system to reason about which writes reached the disk and which did not, enabling the file system to take corrective measures, such as replaying the writes, to recover. Unfortunately, ordering points are not without their own set of problems. By their very nature, ordering points introduce waiting into the file-system code, thus potentially lowering performance. They constrain the scheduling of disk writes, both at the operating system level and at the disk driver level. They introduce complexity into the file-system code, which leads to bugs and lower reliability [25, 26, 49, 50]. The use of ordering points also forces file systems to ignore the end-to-end argument [34], as the support of lower-level systems and disk firmware is required to implement imperatives such as the disk cache flush. When such imperatives are not properly implemented [36], file-system consistency is compromised [29]. In today's cloud computing environment [1], the operating system runs on top of a tall stack of virtual devices, and only one of them needs to neglect to enforce write ordering [47] for file-system consistency to fail.

We can thus summarize the current state of the art in file-system crash consistency as follows. At one extreme is a lazy, optimistic approach that writes blocks to disks in any order (e.g., ext2 [4]); this technique does not add overhead or induce extra delays at run-time, but requires an expensive (and often prohibitive) disk scan after a crash. At the other extreme are eager, pessimistic approaches that carefully order disk writes (e.g., ZFS or ext3); these techniques pay a perpetual performance penalty in return for consistency guarantees and quick recovery. We seek to obtain the best of both worlds: the simplicity and performance benefits of the lazy approach with the strong consistency and availability of eager file systems.

We present the *No-Order file system* (NoFS), a simple, optimistic, lightweight file system which maintains consistency without resorting to the use of ordering. NoFS employs a new approach to providing consistency called *backpointer-based consistency*, which is built upon references in each file-system object to the files or directories that own it. We extend a logical framework for file systems [38] to prove that the incorporation of backpointer-based consistency in an order-less file system guarantees a certain level of consistency. We simplify the update protocol through *non-persistent allocation structures*, reducing the number of blocks that need to reach disk to successfully complete an operation.

Through reliability experiments, we demonstrate that NoFS is able to detect and handle a wide range of inconsistencies. We compare the performance of NoFS with ext2, an order-less file system with no consistency guarantees, and ext3, a journaling file system with metadata consistency. We show that NoFS has excellent performance overall, matching or exceeding the performance of ext2 and ext3 on various workloads. We also discuss the limitations of our approach.

2 Background

File systems use a number of data structures to keep track of the data on disk. These include allocation structures such as bitmaps, and metadata such as inodes. In order to do a single operation such as file creation, multiple data structures have to be updated on disk. For example, in the ext2 file system [4], in order to create an empty file, the inode bitmap, the parent inode, the parent directory, and the child inode all need to be updated and written to disk.

The problem of file-system consistency arises because the system may crash at any time, resulting in some of the updates persisting, and other updates being lost. File-system inconsistency manifests in different ways: a missing file, a file with garbage data, or in some cases, an unmountable file system. File systems have different solutions to this problem, with varying levels of consistency.

We first examine the different levels of consistency provided by file systems, describing the guarantees provided by each level. We then examine the techniques used in file systems to provide consistency and show that all of them (except the file-system check) have at least one ordering point in their update protocols. We discuss the disadvantages of having ordering points and motivate the design of our order-less file system.

2.1 File-system consistency

There are many levels of consistency in file systems, differing in terms of guarantees provided for data and metadata blocks. An inconsistency could be caused by many things: a hardware error, memory corruption, or a system crash. In this work, we are only concerned with inconsistencies occurring due to a system crash.

Metadata consistency: The metadata structures of the file system are entirely consistent with each other. There are no dangling files and no duplicate pointers. The counters and bitmaps of the file system, which keep track of resource usage, match with the actual usage of resources on the disk. Therefore a resource is in use if and only if the bitmaps say that it is in use. Metadata consistency does not provide any guarantees about data.

Data consistency: Data consistency is a stronger form of metadata consistency. Along with the guarantee about metadata, there is the additional guarantee that all data that is read by a file belongs to that file. In other words, a read of file A may not return garbage data, or data belonging to some file B. It is possible that the read may return an older version of the data of file A.

Version consistency: Version consistency is a stronger form of data consistency with the additional guarantee that the metadata version matches the version of the referred data. For example, consider a file with a single data block. The data block is overwritten, and a new block is added, thereby changing the file version: the old version had one block, and the new version has two blocks. Version consistency guarantees that a read of the file does not return old data from the first block and new data from the second block (since the read would return the old version of the data block and the new version of the file metadata).

2.2 Techniques for providing consistency

In this section, we review different approaches to providing consistency in file systems. We point out where ordering points are needed in each of the techniques, except for file-system checks. An ordering point signifies that some blocks need to be persistent on disk before other blocks. For example, an update protocol might require that all the file-system metadata reach the disk before all the data.

2.2.1 File-system check

The file-system check is the simplest solution to the consistency problem: let the system crash and become inconsistent, and upon reboot, fix the inconsistencies. This technique was used in the Fast File System [18, 20] and the ext2 file system [4]. No extra actions are required during runtime, allowing the file system to execute without any performance degradation. The simplicity comes with a high cost: the entire disk needs to be scanned before inconsistencies can be fixed in the file system. While this was acceptable for early file systems that were megabytes in size, scanning an entire disk (or worse, a large RAID volume [23]) would require hours in modern systems. Though several optimizations were developed to reduce the running time of the file-system check [13, 19, 24], it is still too expensive for large volumes, prompting the file-system community to turn to other solutions.

File systems that depend upon on the file-system check alone for consistency cannot provide data consistency, since there is no way for the file system to differentiate between valid data and garbage in a data block. Therefore file reads may return garbage after a crash. The state of every metadata structure is known after the disk scan, and hence duplicate resource allocation and orphan resources can be handled, ensuring metadata consistency.

2.2.2 Journaling

Journaling uses the idea of write-ahead logging [12] to solve the consistency problem: metadata (and sometimes data) is first logged to a separate location on disk, and when all writes have safely reached the disk, the information is written into its original place in the file system. Over the years, this technique has been incorporated into a number of file systems such as NTFS [21], JFS [2], XFS [41], ReiserFS [31], and ext3 [45, 46].

Journaling file systems offer data or metadata consistency based on whether data is journaled or not. Both journaling modes use at least one ordering point in their update protocols, where they wait for the journal writes to be persisted on disk before writing the commit block. Journaling file systems often perform worse than their order-less peers, since information needs to be first written to the log and then later to the correct location on disk. Recovery of the journal is needed after a crash, but it is usually much faster than the file-system check.

2.2.3 Soft updates

Soft updates involves tracking dependencies among in-memory copies of metadata blocks, and carefully ordering the writes to disk such that the disk always sees content that is consistent with the other disk metadata. In order to do this, it may sometimes be necessary to roll back updates to a block at the time of write, and roll-forward the update later. Soft updates was implemented for FFS, and enabled FFS to achieve performance close to that of a memory-based file system [10] . However, it was extremely tricky to implement the ordering rules correctly, leading to numerous bugs. Though the Featherstitch project [9] reduces the complexity of soft updates, the idea has not spread beyond the BSD distributions.

Soft updates provide metadata and data consistency at low cost. FFS with soft updates cannot tell the difference between different versions of data, and hence does not provide version consistency. Soft updates also provide high availability since a blocking file-system check is not required; instead, upon reboot after a crash, a snapshot of the file-system state is taken, and the file-system check is run on the snapshot in the background [19].

2.2.4 Copy-on-write

The copy-on-write technique, as the name suggests, directs a write to a metadata or data block to a new copy of the block, never overwriting the block in place. Once the write is persisted on disk, the new information is added to the file-system tree. The ordering point is in-between these two steps, where the file system atomically changes between the old view of the metadata to one which includes the new information. Copy-on-write has been used in a number of file systems [15, 32], with the most recent being ZFS [3] and btrfs [48].

Copy-on-write file systems provide metadata, data, and version consistency due to the use of logging and transactions. Modern copy-on-write file systems like ZFS achieve good performance, though at the cost of very high complexity. The large size of these file systems (tens of thousands of lines of code [35]) is partly due to the copy-on-write technique, and partly due to advanced features such as storage pools and snapshots.

2.3 Summary

Table 1 compares consistency techniques on complexity, performance, availability, and consistency guarantees provided. Observe that every technique that provides consistency and availability in file systems uses ordering points in its update protocol. Ordering points lead to complexity in the file-system code, paving the way for bugs and decreased reliability. File systems which use ordering points perform worse than order-less file systems on some workloads. The use of ordering points is built upon lower-level functionality such as the SATA flush command [43]; when disks do not reliably flush their cache [36], ordering points fail to enforce consistency and more complicated measures have to be taken [29]. Thus there is a need for a technique which provides consistency without sacrificing simplicity, availability, or performance. We believe that backpointer-based consistency fulfills this need.

	Consistency					
Technique	**Metadata**	**Data**	**Version**	**Complexity**	**Performance**	**Availability**
File-system check	√	×	×	L	H	L
Metadata journaling	√	×	×	M	M	H
Data journaling	√	√	√	M	M	H
Soft Updates	√	√	×	H	H	H
Copy-on-write	√	√	√	H	H	H
BBC	√	√	×	L	H	H

Table 1: **Consistency techniques.** *The table compares various approaches to providing consistency in file systems. Legend: L – Low, M – Medium, H – High. We observe that only backpointer-based consistency (BBC) provides data consistency with low complexity, high performance, and high availability.*

3 Design

We present the design of the *No-Order file system (NoFS)*, a lightweight, consistent file system with no ordering points in its update protocol. NoFS provides access to files immediately upon mounting, with no need for a file-system check or journal recovery.

In this section, we introduce *backpointer-based consistency (BBC)*, the technique used in NoFS for maintaining consistency. We use a logical framework to prove that BBC provides data consistency in NoFS. We discuss how BBC can be used to detect and recover from inconsistencies, and elaborate on why allocation structures are not persisted to disk in NoFS.

3.1 Overview

The main challenge in NoFS is maintaining consistency without ordering points. Consistency is closely tied to logical identity in file systems. Inconsistencies arise due to confusion about an object's identity; for example, two files may each claim to own a data block. If the block's true owner is known, such inconsistencies could be resolved. Associating each object with its logical identity is the crux of the backpointer-based consistency technique.

Employing backpointer-based consistency allows NoFS to detect inconsistencies on-the-fly, upon user access to corrupt files and directories. The presence of a corrupt file does not affect access to other files in any way. This property enables immediate access to files upon mounting, avoiding the downtime of a file-system check or journal recovery. A read is guaranteed to never return garbage data, though stale data may be returned.

We intentionally avoided using complex rules and dependencies in NoFS. We simplified the update protocols, not persisting allocation structures to disk. We maintain in-memory versions of allocation structures and discover data and metadata allocation information in the background while the file system is running.

3.2 Backpointer-based consistency

Backpointer-based consistency is built around the logical identity of file-system objects. The logical identity of a data block is the file it belongs to, along with its position inside the file. The logical identity of a file is the list of directories that it is linked to. This information is embedded inside each object in the form of a *backpointer*. Upon examining the backpointer of an object, the parent file or directory can be determined instantly. Blocks have only one owner, while files are allowed to have multiple parents. Figure 1 illustrates how backpointers link file-system objects in NoFS. As each object in the file system is examined, a consistent view of the file-system state can be incrementally built up.

Though conceptually simple, backpointers allow detection of a wide range of inconsistencies. Consider a block that is deleted from a file, and then assigned to another file and overwritten. If a crash happens at any point during these operations, some subset of the data structures on disk may not be updated, and both files may contain pointers to the block. However, by examining the backpointer of the block, the true owner of the block can be identified.

In designing NoFS, we assume that the write of a block along with its backpointer is atomic. This assumption is key to our design, as we infer the owner of the data block by examining the backpointer. Current SCSI drives allow a 520-byte atomic write to enable checksums along with each 512-byte sector [42]; we envision that future drives with 4-KB blocks will provide similar functionality.

Backpointers are similar to checksums in that they verify that the block pointed to by the inode actually belongs to the inode. However, a checksum does not identify the owner of a data block; it can only confirm that the correct block is being pointed to. Consistency and recovery require identification of the owner.

3.2.1 Intuition

We briefly provide some intuition about the correctness of using the backpointer-based consistency technique to ensure data consistency. We first consider what data consistency and version consistency mean, and the file-system structures required to ensure each level of consistency.

Data consistency provides the guarantee that all the data accessed by a file belongs to that file; it may not be garbage data or belong to another file. This guarantee is obtained when a backpointer is added to a data block.

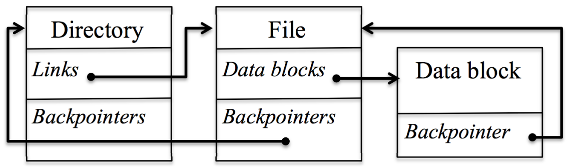

Figure 1: **Backpointers.** *The figure shows a conceptual view of the backpointers present in NoFS. The file has a backpointer to the directory that it belongs to. The data block has a backpointer to the file it belong to. Files and directories have many backpointers while data blocks have a single backpointer.*

Consider a file pointing to a data block. Upon reading the data block, the backpointer is examined. If the backpointer matches the file, then the data block must have belonged to the file, since the backpointer and the data inside the block were written together. If the data block was reallocated to another file and written, it would be reflected in the backpointer. Hence, no ordering is required between writes to data and metadata since the data block's backpointer would disagree in the event of a crash. Note that the data block could have belonged to the file at some point in the past; the backpointer does not provide any information about when the data block belonged to the file. Thus, the file might be pointing to an old version of the data block, which is allowed under data consistency.

Version consistency is a stricter form of data consistency which requires that in addition to belonging to the correct file, all accessed data must be the correct version. Stale data is not allowed in this model. Backpointers are not sufficient to enforce version consistency, as they contain no information about the version of a data block. Hence more information needs to be added to the file system. Each data block has a timestamp indicating when it was last updated. This timestamp is also stored in the inode containing the data block. When a block is accessed, the timestamp in the inode and data block must match. Since timestamps are a way to track versions, the versions in the inode and data block can be verified to be the same, thereby providing version consistency.

We decided against including timestamps in NoFS backpointers because updating timestamps in backpointers and metadata reduces performance and induces a considerable amount of storage overhead. Timestamps need to be stored with every object and its parent. Every update to an object involves an update to the parent object, the parent's parent, and so on all the way up to the root. Furthermore, doing so works against our goal of keeping the file system simple and lightweight; hence, NoFS provides data consistency, but not version consistency.

The full proof involves extending the logical framework of Sivathanu et al. [38] to prove that an order-less file system employing the backpointer-based consistency technique provides data consistency. We further prove that if the backpointer contains an update timestamp, the file system provides version consistency. The full proof can be found in Appendix A.

3.2.2 Detection and Recovery

In NoFS, detection of an inconsistency happens upon access to corrupt files or data. When a data or metadata block is accessed, the backpointer is checked to verify that the parent metadata block has the same information. If a file is not accessed, its backpointer is not checked, which is why the presence of corrupt files does not affect access to other files: checking is performed on-demand.

This checking happens both at the file level and the data block level. When a file is accessed, it is checked to see whether it has a backpointer to its parent directory. This check allows identification of deleted files where the directory did not get updated, and files which have not been properly updated on disk.

NoFS is able to recover from inconsistencies by treating the backpointer as the true source of information. When a directory and a file disagree on whether the file belongs to the directory or not, the backpointer in the file is examined. If the backpointer to the directory is not found, the file is deleted from the directory. Issues involving blocks belonging to files are similarly handled.

3.3 Non-persistent allocation structures

In an order-less file system, allocation structures like bitmaps cannot be trusted after a crash, as it is not known which updates were applied to the allocation structures on disk at the time of the crash. Any allocation structure will need to be verified before it can be used. In the case of global allocation structures, all of the data and metadata referenced by the structure will need to be examined to verify the allocation structure.

Due to these complexities, we have simplified the update protocols in NoFS, making the allocation structures non-persistent. The allocation structures are kept entirely in-memory. NoFS starts out with empty allocation structures and allocation information is discovered in the background, while the file system is online. NoFS can verify whether a block is in use by checking the file that it has a backpointer to; if the file refers to the data block, the data block is considered to be in use. Similarly, NoFS can verify whether a file exists or not by checking the directories in its backpointers. Thus NoFS can incrementally learn allocation information about files and blocks.

4 Implementation

We now present the implementation of NoFS. We first describe the operating system environment, and then discuss the implementation of the two main components of NoFS: backpointers and non-persistent allocation structures. We describe the backpointer operations that NoFS performs for each file-system operation.

Action	Backpointer operations
Create	Write backlink into new inode
Read	Translate offset
	Verify block backpointer in data block
Write	Translate offset
	Verify block backpointer in data block
Append	Translate offset
	Write block backpointer into data block
Truncate	No backpointer operations
Delete	No backpointer operations
Link	Write backlink into inode
Unlink	Remove backlink from inode
mkdir	Write directory entry backpointer into directory block
rmdir	No backpointer operations

Table 2: **NoFS backpointer operations.** *The table lists the operations on backpointers caused by common file system operations. Note that all checks are done in memory.*

4.1 Operating system environment

NoFS is implemented as a loadable kernel module inside Linux 2.6.27.55. We developed NoFS based on ext2 file-system code. Since NoFS involves changes to the file-system layout, we modified the `e2fsprogs` tools 1.41.14 [44] used for creating the file system.

Linux file systems cache user data in a unified page cache [6]. File reads (except direct I/O) are always satisfied from the page cache. If the page is not up-to-date at the time of read, the page is first filled with data from the disk and then returned to the user. File writes cause pages to become dirty, and an I/O daemon called `pdflush` periodically flushes dirty pages to disk. Due to this tight integration between the page cache and the file system, NoFS involves modifications to the Linux page cache.

4.2 Backpointers

NoFS contains three types of backpointers. We describe each of them in turn, pointing out the objects they conceptually link, and how they are implemented in NoFS. Figure 2 illustrates how various objects are linked by different backpointers. Every file-system operation that involves the creation or access of a file, directory, or data block involves an operation on backpointers. These operations are listed in Table 2.

4.2.1 Block backpointers

Block backpointers are $\{$*inode number, block offset*$\}$ pairs, embedded inside each data block in the file system. The first 8 bytes of every data block are reserved for the backpointer. Note that we need to embed the backpointer inside the data block since disks currently do not provide the ability to store extra data along with each 4K block atomically. The first 4 bytes denote the inode number of the file to which the data block belongs. The second 4 bytes represent the logical block offset of the data block

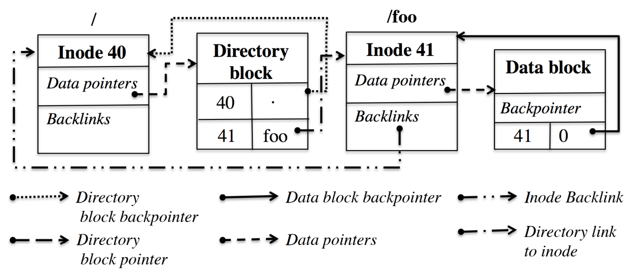

Figure 2: **Implementation of backpointers.** *The figure shows the different kinds of backpointers present in NoFS. foo is a child of the root inode /. This link is represented by a backlink from foo to /. Similarly, the data block is a part of foo, and hence has a backpointer to foo. Directory blocks also contain backpointers, in the form of dot entries to their owner's inode.*

within the file. Given this information, it is easy to check whether the file contains a pointer to the data block at the specified offset. Indirect blocks contain backpointers too, since they belong to a particular file. However, since the indirect block data is not logically part of a file, they are marked with a negative number for the offset.

Our implementation depends on the `read` and `write` system calls being used; data is modified as it is passed from the page cache to the user buffer and back during these calls. When these calls are by-passed (via `mmap`) or the page cache itself is by-passed (via direct IO mode), verifying each access becomes challenging and expensive. We do not support `mmap` or direct IO mode in NoFS.

Insertion: The data from a `write` system call goes through the page cache before being written to disk. We modified the page cache so that when a page is requested for a disk write, the backpointer is written into the page first and then returned for writing. The block offset translation was modified to take the backpointer into account when translating a logical offset into a block number.

Verification: Once a page is populated with data from the disk, the page is checked for the correct backpointer. If the check fails, an I/O error is returned. If this is the first time that the data block is accessed, the inode's attributes (size and number of blocks) are updated. Note that the page is not checked on every access, but only the first time that it is read from disk. Assuming memory corruption does not occur [51], this level of checking is sufficient.

4.2.2 Directory backpointers

The dot directory entry serves as the backpointer for directory blocks, as it points to the inode which owns the block. However, the dot entry is only present in the first directory block. We modified ext2 to embed the dot entry in every directory block, thus allowing the owner of any directory block to be identified using the dot entry.

Though the block backpointer could have been used in directory blocks as well, we did not do so for two reasons. First, the structured content of the directory block enables the use of the dot entry as the backpointer, simplifying our implementation. Second, the offset part of the block backpointer is unnecessary for directory blocks since directory blocks are unordered and appending a directory block at the end suffices for recovery.

Insertion: When a new directory entry is being added to the inode, it is determined whether a new directory block will be needed. If so, the dot entry in added in the new block, followed by the original directory entry.

Verification: Whenever the directory block is accessed, such as in `readdir`, the dot entry is cross-checked with the inode. If the check fails, an I/O error is returned and the directory inode's attributes (size and block count) are updated.

4.2.3 Backlinks

An inode's backlinks contain the inode numbers of all its parent directories. Every valid inode must have at least one parent. Hard linked inodes may have multiple parents.

We modified the file-system layout to add space for backlinks inside each inode. The inode size is increased from the default 128 bytes to 256 bytes, enabling the addition of 32 backlinks, each of size 4 bytes. The `mke2fs` tool was modified to create a backlink between the `lost+found` directory and the root directory when the file system is created.

Insertion: When a child inode is linked to a parent directory during system calls such as `create` or `link`, a backlink to the parent is added in the child inode.

Verification: At each step of the iterative inode lookup process, we check that the child inode contains a backlink to the parent. A failed check stops the lookup process and returns an I/O error. If this is the first time the inode is accessed via this particular path, the number of links for the inode is updated.

4.2.4 Detection

Every data block is checked for a valid backpointer when it is read from the disk into the page cache. We assume that neither memory nor on-disk corruption happens; hence, it is safe to limit checking to when a data block is first brought into main memory. It is this property that leads to the high performance of NoFS; because disk I/O is several orders of magnitude slower than in-memory operations, the backpointer check can be performed on disk blocks with very low overhead.

Inode backlink checking occurs during directory path resolution. The child inode's backlink to the parent inode is checked. Since both inodes are typically in memory during directory path resolution, the backlink check is a quick in-memory check, and does not degrade performance significantly, since a disk read is not performed to obtain the parent or child inode.

Note that the detection of inconsistency happens at the level of a single resource, such as an inode or a data

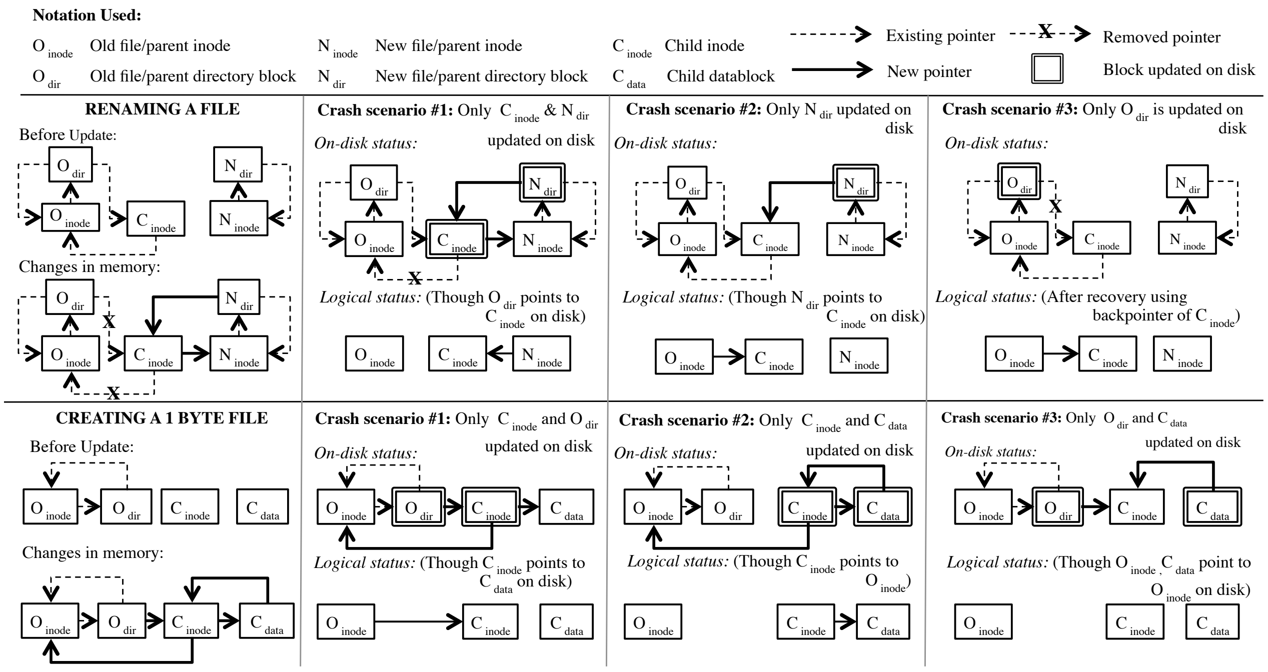
Notation Used:
O inode Old file/parent inode
O dir Old file/parent directory block
N inode New file/parent inode
N dir New file/parent directory block
C inode Child inode
C data Child datablock
Existing pointer
New pointer
Removed pointer
Block updated on disk
RENAMING A FILE
Before Update:
Changes in memory:
Crash scenario #1: Only C inode & N dir updated on disk
Crash scenario #2: Only N dir updated on disk
Crash scenario #3: Only O dir is updated on disk
On-disk status:
Logical status: (Though O dir points to C inode on disk)
Logical status: (Though N dir points to C inode on disk)
Logical status: (After recovery using backpointer of C inode)
CREATING A 1 BYTE FILE
Crash scenario #1: Only C inode and O dir updated on disk
Crash scenario #2: Only C inode and C data updated on disk
Crash scenario #3: Only O dir and C data updated on disk
Logical status: (Though C inode points to C data on disk)
Logical status: (Though C inode points to O inode)
Logical status: (Though O inode , C data point to O inode on disk)

Figure 3: **Handling crashes with backpointers.** *The figure presents three failure scenarios during the rename of a file, and the creation of a file with 1 byte of data. In each scenario, employing backpointers allows us to detect inconsistencies such as both the old and new parents claiming the child, and the child pointing to a data block that hasn't been updated.*

block. Verifying that a data block belongs to an inode can be done without considering any other object in the file system. The presence of corrupt files or blocks does not affect the reads or writes to other non-corrupt files. As long as corrupt blocks are not accessed, their presence can be safely ignored by the rest of the system. This feature contributes to the high availability of NoFS: a file-system check or recovery protocol is not needed upon mount. Files can be immediately accessed, and any access of a corrupt file or block will return an error. This feature also allows NoFS to handle concurrent writes and deletes. Even if many writes and deletes were going on at the time of a crash, NoFS can still detect inconsistencies by considering each inode and data block pair in isolation.

Let us illustrate this with an example. Upon mount, we run the command `cat /dir1/file1`, which involves several checks in the file system. First, the directory block for `dir1` is fetched, and checked whether it has a directory backpointer to the root directory. Similarly, when the `file1` inode is retrieved from disk, it is checked to see if it has a backlink to `dir1`. When the data block of `file1` is retrieved, it is checked to verify that the data block has a block backpointer to `file1`. If any of these checks fail, an error is returned to the user.

Figure 3 illustrates the detection of inconsistencies during different crash scenarios for two operations: renaming a file and creating a single byte file. The state of data structures in memory before and after the update is first shown. In each crash scenario, a different subset of the in-memory updates is successfully written to disk. The state of various pointers on disk after the crash is shown, followed by the consistent logical view that NoFS obtains after verification using back pointers. For example, during the rename, a crash may lead to the file being listed in both the old and new directories. However, the logical status shows that upon backpointer verification, the true owner of the child inode is found using the backlink.

4.2.5 Recovery

Having backlinks and backpointers allows recovery of lost files and blocks. Files can be lost due to a number of reasons. A rename operation consists of a unlink and a link operation. An inopportune crash could leave the inode not linked to any directory. A crash during the create operation could also lead to a lost file. Such a lost file can be recovered in NoFS, due to the backlinks inside each inode. Each such inode is first checked for access to all its data blocks. If all the data blocks are valid, it is a valid subtree in the file system and can be inserted back into the directory hierarchy (using the backlinks information) without compromising the consistency of the file system. When adding a directory entry for the recovered inode, it is correct to append the directory entry at the end of the directory, since directory entries are an unordered collection; there is no meaning attached to the exact offset inside a directory block where a directory entry is added.

In a similar fashion, it it possible to recover data blocks lost due to a crash before the inode is updated. A data block, once it has been determined to belong to an inode, cannot be embedded at an arbitrary point in the inode data. It is for this reason that the *offset* of a data block is embedded in the data block, along with the inode number. The offset allows a data block to be placed exactly where it belongs inside a file. Indirect blocks of a file do not have the offset embedded, as they do not have a logical offset within the file. Indirect blocks are not required to reconstruct a file; only data blocks and their offsets are needed.

Using reconstruction of files from their blocks on disk, files can be potentially "undeleted", provided that the blocks have not been reused for another file. We have not implemented undelete in NoFS. Block allocation would need to be tweaked to not reuse blocks for a certain amount of time, or until a certain free-space threshold is reached. Undelete might turn up stale data because NoFS does not support version consistency; the data block might have been part of an older version of the inode.

4.3 Non-persistent allocation structures

The allocation structures in ext2 are bitmaps and group descriptors. These structures are not persisted to disk in NoFS. In-memory versions of these data structures are built using the *metadata scanner* and *data scanner*. Statistics usually maintained in the group descriptors, such as the number of free blocks and inodes, are also maintained in their in-memory versions.

Upon file-system mount, in-memory inode and block bitmaps are initialized to zero, signifying that every inode and data block is free. Since every block and inode has a backpointer, it can be determined to be in use by examining its backlink or backpointer, and cross-checking with the inode mentioned in the backpointer. As every object is examined, consistent file-system state is built up and eventually complete knowledge of the system is achieved.

In the file system, a block or inode that is marked free could mean two things: it is free, or it has not been examined yet. Since all blocks and inodes are marked free at mount time, inodes need to be examined to check that they are indeed free; hence blocks or inodes that have not been examined yet cannot be allocated. In order to mark which inodes or blocks have been examined, we added a new bitmap each for inodes and data blocks called the *validity* bitmap. If a block or inode has been examined and marked as free, it is safe to use it. Blocks not marked as valid could actually be used blocks, and hence must not be used for allocation. The examination of inodes and blocks are carried out by two background threads called the metadata scanner and data scanner. The two threads work closely together in order to efficiently find all the used inodes and blocks on disk.

4.3.1 Metadata Scan

Each inode needs to be examined in order to find out if it is in use or not. The backlinks in the inode are found, and the directory blocks of the referred inodes are searched for a directory entry to this inode. Note that the directory hierarchy is not used for for the scan. The disk order of inodes is used instead, as this allows for fast sequential reads of the inode blocks.

Once an inode is determined to be in use, its data blocks have to verified. This information is communicated to the data scanner by adding the data blocks of the inode to a list of data blocks to be scanned. The inode information is also attached to the list so that the data scanner can simply compare the backpointer value to the attached value to determine whether the block is used. However, if the inode has indirect blocks, the inode data blocks are explored and verified immediately. An inode with indirect blocks may contain thousands of data blocks, and it would be cumbersome to add all those data blocks to the list and process them later; hence inode data is verified immediately by the metadata scanner. Each inode is marked valid after it has been scanned, allowing inode allocation to occur concurrently with the metadata scan.

4.3.2 Data Scan

Observe that a data block is in use only if it is pointed to by a valid inode which is in use; hence only data blocks that belong to a valid inode need to be checked, which reduces the number of blocks that need to be checked drastically.

The data block scanner works off a list of data blocks that the metadata scanner provides. Each list item also includes information about the inode that contained the data block. Therefore, the data scanner simply needs to read the inode off the disk and compare the backpointer inode to the inode information in the list item. The data block is marked valid after the examination is complete.

Since the data scanner only looks at blocks referred to by inodes, there may be plenty of unexamined blocks which are not referred and potentially free. These blocks cannot be marked as valid and free until the end of the data scan, when all valid inodes have been examined. While the scan is running, the file system may indicate that there are no free blocks available, even if there are many free blocks in the system. In order to fix this, we implemented another scanner called the sequential block scanner which reads data blocks in disk order and verifies them one by one. This thread is only started if no free blocks are found, and the data scanner is still running.

4.4 Limitations

The design of NoFS involves a number of trade-offs. We describe the limitations that arise from our design choices.

Recovery: NoFS was designed to be as lightweight as possible, avoiding heavy machinery for logging or copy-on-write. As a result, file-system recovery is limited. For example, consider a file that is truncated, and later written with new data. After a crash in the middle of these updates, the file may point to a block that it does not own. This inconsistency is detected upon access to the data block. However, the version of the file which pointed to its old data cannot be recovered easily. By utilizing logging, a file system like ext3 provides the ability to preserve data in the event of a crash.

Transactions: NoFS does not provide atomic transactions. Operations can be partially applied to different data structures. For example, if the file system crashes in the middle of a rename, it is possible that the file appears both in the old and new directories, as we do not validate directory entries during a `readdir`. Though the user will be able to access the file via only one directory, the 'old-or-new' aspect of transactions is not provided.

Accessing unverified objects: For large disks, it is possible that an object is accessed before the scan has verified it. Accessing such unverified objects involves a performance cost. The performance cost is felt during different system calls for inodes and data blocks.

Running the `stat` system call on an unverified inode may result in invalid information, as the number of blocks recorded in the inode may not match the actual number of blocks that belong to the inode on disk. In order to handle this, NoFS checks the inode status upon a `stat` call, and verifies the inode immediately if required, and then allows the system call to proceed. Since verification involves checking every data block referred to by the inode, the verification can take a lot of time. Running `ls -l` on a large directory of unverified files involves a large performance penalty arising from reading every file. For verified inodes, the `stat` will always return valid data, as the inode's attributes are updated whenever an error is encountered on block access. Note that NoFS does not check directory entries for correctness.

In the case of an unverified data block, no additional I/O is incurred during reads and partial writes since both involve reading the block off the disk anyway. However, in the case of a block overwrite, the block has to be read first to verify that it belongs to the inode before overwriting it. As a result, a write in ext2 is converted into a read-modify-write in NoFS, effectively cutting throughput in half. It should be noted that this happens only on the *first* overwrite of each unverified block. After the first overwrite, the block has been verified, and hence the backpointer no longer needs to be checked.

Thus it can be seen that accessing unverified objects involves a large performance hit. However, these costs are only incurred during the window between file-system mount and scan completion.

5 Evaluation

We now evaluate NoFS in two categories: reliability and performance. For reliability testing, we artificially prevent writes to certain sectors from reaching the disk, and then observe how NoFS handles the resulting inconsistency.

			ext2		NoFS	
System call	**Blocks dropped**	**Error**	**Detected?**	**Action?**	**Detected?**	**Action?**
mkdir	C^{inode}	P^{BD}, C^{OB}	×	–	√	R, C^{EI}
mkdir	C^{dir}	C^{BD}	√	C^{ED}	√	C^{ED}
mkdir	P^{dir}	C^{OI}, C^{OB}	×	–	√	R
mkdir	C^{inode}, C^{dir}	P^{BD}, C^{BD}	×	–	√	C^{EI}
mkdir	C^{inode}, P^{dir}	C^{OB}	×	–	√	R
mkdir	C^{dir}, P^{dir}	C^{OI}	×	–	√	R
link	C^{inode}	C^{HL}	×	–	√	C^{EN}
link	P^{dir}	C^{OI}	×	–	√	R
unlink	C^{inode}	C^{HL}	×	–	√	C^{EO}
unlink	O^{dir}	P^{BD}	×	–	√	C^{EI}
rename	N^{dir}	O^{BD}	×	–	√	C^{EI}
rename	O^{dir}	C^{OI}	×	–	√	R
write	C^{data}	C^{GD}	×	–	√	C^{EB}
write	C^{ind}	C^{GD}	×	–	√	C^{EB}
write	C^{inode}, C^{data}	C^{OB}	×	–	√	R
write	C^{inode}, C^{ind}	C^{OB}	×	–	√	R
write	C^{data}, C^{ind}	C^{GD}	×	–	√	C^{EB}
delete-create	O^{dir}	O^{BD}	×	–	√	C^{EO}
truncate-write	O^{inode}	O^{TP}	×	–	√	O^{EB}
unlink-link	O^{dir}	O^{BD}	×	–	√	C^{EO}

General Key

C	Child	*inode*	File inode
P	Parent	*dir*	Directory block
O	Old file/parent	*data*	Data block
N	New file/parent	*ind*	Indirect block

Key for Error		**Key for Action**	
BD	Bad dir entry	*R*	Block/inode reclaimed on scan
OB	Orphan block	*EI*	Error on inode access
OI	Orphan inode	*ED*	Error on data access
HL	Wrong hard link count	*EN*	Error on access via new path
GD	Garbage data	*EO*	Error on access via old path
TP	2 inodes refer to 1 block	*EB*	Error on block access

Table 3: **Reliability testing.** *The table shows how NoFS reacts to various inconsistencies that occur due to updates not reaching the disk. The behavior of ext2 is also shown. NoFS detects all inconsistencies and reports an error, while ext2 lets most of the errors pass by undetected.*

For performance testing, we evaluate the performance of NoFS on a number of micro and macro-benchmarks. We compare the performance of NoFS to ext2, an order-less file system with no consistency, and ext3 (in ordered mode), a journaling file system with metadata consistency.

5.1 Reliability

We test whether NoFS can handle inconsistencies caused by a file-system crash. When a crash happens, any subset of updates involved in a file-system operation could be lost. We emulate different system-crash scenarios by artificially restricting blocks from reaching the disk, and restarting the file-system module. The restarted module will see the results of a partially completed update on disk.

We use a pseudo-device driver to prevent writes on target blocks and inodes from reaching the disk drive. We interpose the pseudo-device driver in-between the file system and the physical device driver, and all writes to the disk drive go via the pseudo-device driver. The file system and the device driver communicate through a list of sectors. In the file system, we calculate the on-disk sectors of target blocks and inodes and add them to the black list of sectors. All writes to these sectors are ignored by the device driver. Thus, we are able to target inodes and blocks in a fine grained manner.

Table 3 lists the behavior of ext2 and NoFS when 20 different inconsistencies are caused by dropping some of the blocks involved in each file-system operation. For example, consider the *mkdir* operation. It involves adding a directory entry to the parent directory, updating the new child inode, and creating a new directory block for the child inode. We do not consider updates to the access time of the parent inode. In the reliability test, we would drop writes to different combinations of these blocks, and observe the actions taken by the file system. For instance, if the write to the new child inode is dropped, it creates a bad directory entry in the parent directory, and orphans the directory block of the new child inode. We observe whether the file system detects this corrupt directory entry, and whether the orphan block is reclaimed. Both these actions are performed successfully in NoFS, whereas ext2 allows the user to access a garbage inode, and the block remains an orphan until the next file-system check.

The table entries which have two system calls denote the second system call happening after the first system call. These particular combinations were selected because they share a common resource. For example, truncate-write explores the case when a data block is deleted from a file and reassigned to another file. If the write to the truncated file inode fails, both files now point to the same data block, leading to an inconsistency. Similarly unlink-link and delete-create may share the same inode.

Some inconsistencies, like a corrupt directory block, are detected by ext2. Many other inconsistencies, such as reading garbage data, are not detected by ext2. All inconsistencies are detected by NoFS, and an error is returned to the user. When blocks and inodes are orphaned due to a crash, they are reclaimed by NoFS when the file system is scanned for allocation information upon reboot. Some of the inconsistencies could lead to potential security holes: for example, linking a sensitive file for temporary access, and removing the link later. If the directory block is not written to disk, the file could still be accessed, providing a way to read sensitive information. These security holes are detected upon access in NoFS, and any operation on them leads to an error.

5.2 Performance

To evaluate the performance of NoFS, we run a series of micro-benchmark and macro-benchmark workloads. We also observe the performance of NoFS at mount time, when the scan threads are still active. We show that NoFS has comparable performance to ext2 in most workloads, and that the performance of NoFS is reasonable when the scan threads are running in the background. We also measure the scan running time when the file system is populated with data, the rate at which NoFS scans data blocks to find free space, and the performance cost incurred when the stat system call is run on unverified inodes.

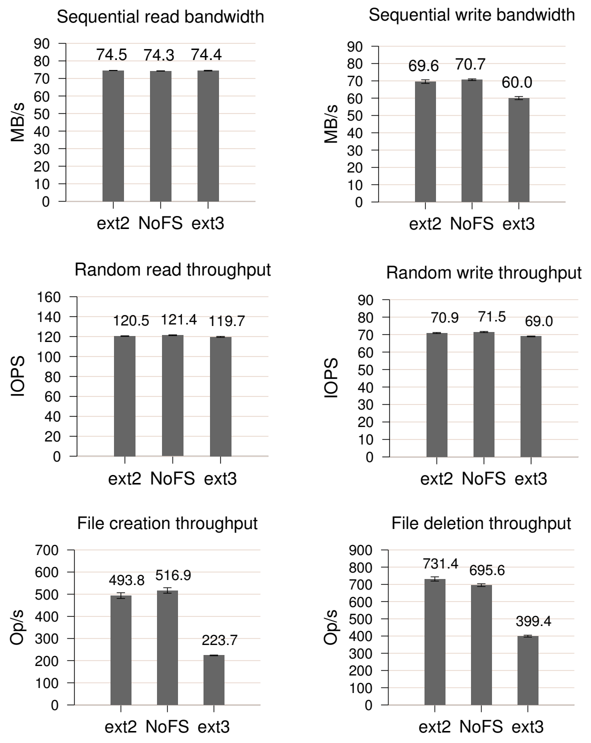

Figure 4: **Micro-benchmark performance.** *This figure compares file-system performance on various micro-benchmarks. The sequential benchmarks involve reading and writing a 1 GB file. The random benchmarks involve 10K random reads and writes in units of 4088 bytes (4096 bytes - 8 byte backpointer) across a 1 GB file, with a fsync after 1000 writes. The creation and deletion benchmarks involve 100K files spread over 100 directories, with a fsync after every create or delete.*

Our experiments were performed on a machine with a AMD 1 Ghz Opteron processor, and 1 GB of memory running Linux 2.6.27.55. The disk drive used in the experiment was a Seagate Barracuda 160 GB, which provides 75 MB/s read throughput and 70 MB/s write throughput. All experiments were performed on a cold file-system cache. The experiments were stable and repeatable. The numbers reported are the average over 10 runs.

5.2.1 Micro-benchmarks

We run a number of micro-benchmarks, focusing on different operations like sequential write and random read. Figure 4 illustrates the performance of NoFS on these workloads. We observe that NoFS has minimal overhead on the read and write workloads. For the sequential write workload, the performance of ext3 is worse than ext2 and NoFS due to the journal writes that ext3 performs.

The creation and deletion workloads involve doing a large number of creates/deletes of small files followed by fsync. This workload clearly brings out the performance penalty due to ordering points. The throughput of NoFS

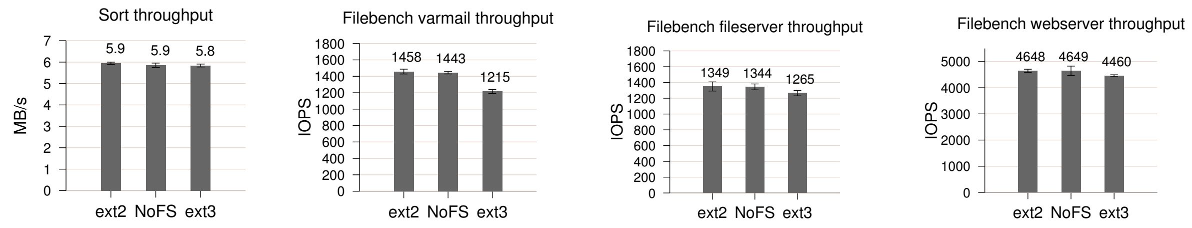

Figure 5: **Macro-benchmark performance.** *The figure shows the throughput achieved on various application workloads. The sort benchmark is run on 500 MB of data. The varmail benchmark was run with parameters 1000 files, 100K mean dir width, 16K mean file size, 16 threads, 16K I/O size and 16K mean append size. The file and webserver benchmarks were run with the parameters 1000 files, 20 dir width, 1 MB I/O size and 16K mean append size. The mean file size was 128K for the fileserver benchmark and 16K for the webserver benchmark. Fileserver benchmark used 50 threads while webserver used 100 threads.*

is twice that of ext3 on the file creation micro-benchmark, and 70% higher than ext3 on the file deletion benchmark.

5.2.2 Macro-benchmarks

We run the sort and Filebench [8] macro-benchmarks to assess the performance of NoFS on application workloads. Figure 5 illustrates the performance of the three file systems on this macro-benchmark. We selected the sort benchmark because it is CPU intensive. It sorts a 500 MB file generated by the *gensort* tool [22], using the command-line sort utility. The performance of NoFS is similar to that of ext2 and ext3, demonstrating that NoFS has minimal CPU overhead.

We run three workloads on Filebench: fileserver, webserver, and varmail. The fileserver workload emulates file-server activity, performing a sequence of creates, deletes, appends, reads, and writes. The webserver workload emulates a multi-threaded web host server, performing sequences of open-read-close on multiple files plus a log file append, with 100 threads. The varmail workload emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory.

We believe these benchmarks are representative of the different kind of I/O workloads performed on file systems. The performance of NoFS matches ext2 and ext3 on all three workloads. NoFS outperforms ext3 by 18% on the varmail benchmark, demonstrating the performance degradation in ext3 due to ordering points.

5.2.3 Scan performance

We evaluate the performance of NoFS at mount time, when the scanner is still scanning the disk for free resources. The scanner is configured to run every 60 seconds, and each run lasts approximately 16 seconds. In order to understand the performance impact due to scanning, we do two experiments involving 10 sequential writes of 200 MB each. The writes are spaced 30 seconds apart.

In the first experiment, we start the writes at mount time. The scanning of the disk and the sequential write is interleaved at 0s, 60s, 120s, and so on, leading to the write bandwidth dropping to half. When the sequential writes are run at 30s, 90s, 150s, and so on, the writes achieve peak bandwidth. In the second experiment, the writes were once again spaced 30s apart, but were started at 20s, after the end of the first scan run. In this experiment, the writes are never interleaved with the scan reads, and hence suffer no performance degradation. Graph (a) in Figure 6 illustrates these results.

Once the scan finishes, writes will once again achieve peak bandwidth. Running the scan runs without a break causes the scan to finish in around 90 seconds on an empty file system. Of course, one can configure this trade-off as need be; the larger the interval between scans, the smaller the performance impact during this phase, but the longer it takes to fully discover the free blocks of the system.

Graph (b) in Figure 6 depicts the time taken to finish the scan (both metadata and data) when the file system is increasingly populated with data. In this experiment, the scan is run without a break upon file-system mount. All the data in the file system are in units of 1 MB files. The running time of the scan increases slowly when the amount of data in the file system is increased, reaching about 140s for 1 GB of data. We also performed an experiment where we created a variable number of empty files in the file systems and measured the time for the scan to run. We found that the time taken to finish the scan remained the same irrespective of the number of empty files in the system. Since every inode in the system is read and verified, irrespective of whether it is actively used in the file system or not, the scan time remains constant.

During a file write, if there are no free blocks, the sequential block scanner is invoked in order to scan data blocks and find free space. The write will block until free space is found. Graph (c) illustrates the performance of the sequential block scanner. The latency to scan 100 MB is around 3 seconds, and 1 GB of data is scanned in around 30 seconds. The throughput is currently around 30 MB/s, so there is opportunity for optimizing its performance.

As mentioned in Section 4.4, when `stat` is run on an unverified inode, NoFS first verifies the inode by checking all its data blocks. We ran an experiment to estimate the cost of such verification. We created four identical directories, each filled with a number of 1 MB files. Every 140 seconds, `ls -li` was run on one directory, leading

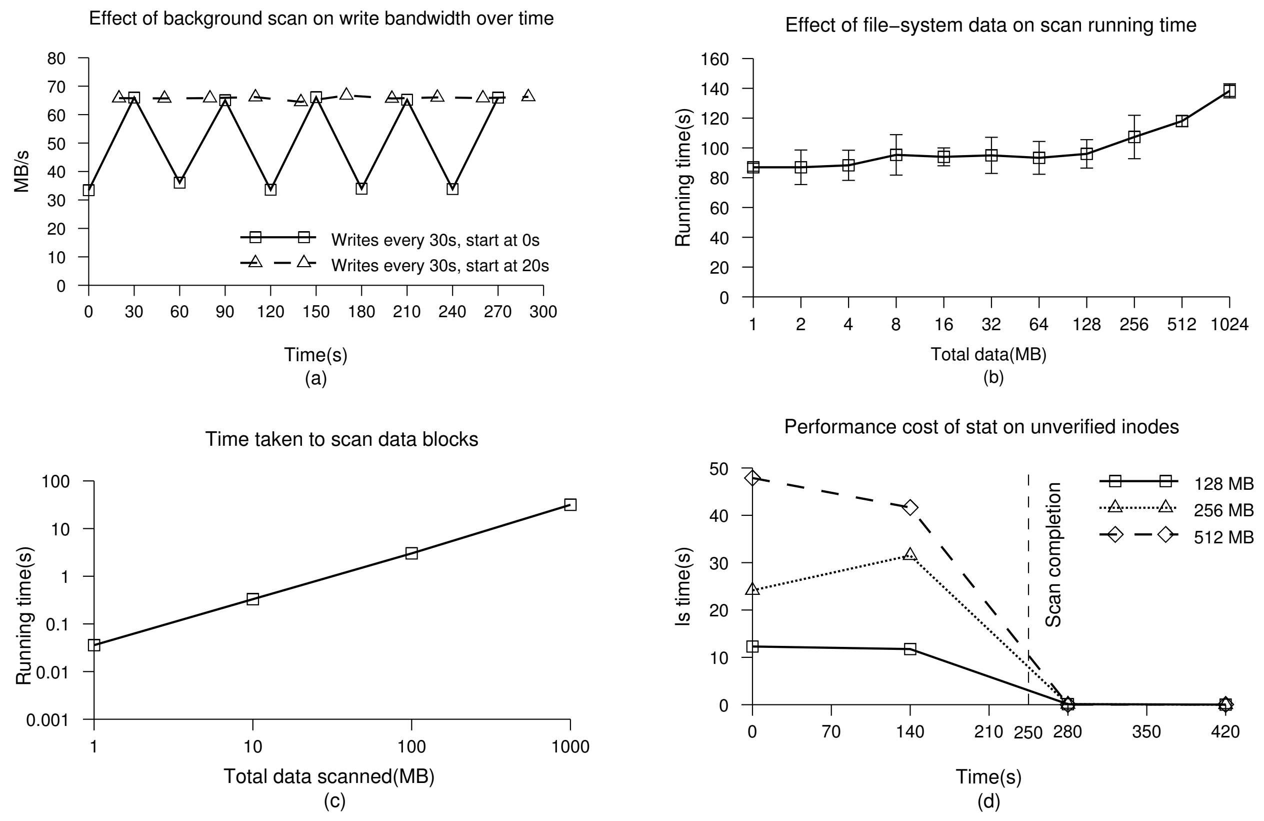

Figure 6: **Scan performance.** *Figure (a) depicts the reduction in write bandwidth when sequential writes interleave with the background scan. Figure (b) shows that the running of the scan increases slowly with the amount of data in the file system. Figure (c) illustrates the rate at which data blocks are scanned. Figure (d) demonstrates the performance cost incurred when the stat system call is run on unverified inodes.*

to a `stat` on each inode in the directory. The background scan started at file-system mount and finished at approximately 250 seconds. We varied the number of files from 128 to 512 and measured the time taken for `ls -li` in each experiment. Graph (d) illustrates the results. As expected, the time taken for `ls` to complete increases with the total data in the directory. After the scan completion at 250 seconds, all the inodes are verified, and hence `ls` finishes almost instantly.

6 Discussion

We have demonstrated that NoFS has better performance than journaling file systems such as ext3, while providing better consistency guarantees. However, it should be noted that NoFS differs from ext3 in two important aspects. First, it does not provide atomic transactions. Second, NoFS has no redundancy anywhere in the system. Part of the reason ext3 performs worse than NoFS is its extra log writes. By writing transaction updates to a log first, ext3 provides both metadata consistency, and the ability to preserve old data if the transaction fails before commit. NoFS only provides the former.

Given its current design, we feel an excellent use-case for NoFS would be as the local file system of a distributed file system such as the Google File System [11] or the Hadoop File System [37]. In such a distributed file system, reliable detection of corruption is all that is required, since redundant copies of data would be stored across the system. If the master controller is notified that a particular block has been corrupted in the local file system of a particular node, it can make additional copies of the data in order to counter the corruption of the block. Furthermore, such distributed file systems typically have large chunk sizes. As shown in section 5, NoFS provides very good performance on large sequential reads and writes, and is well suited for such workloads.

It should be noted that backpointer-based consistency could also be used to help ensure integrity in a conventional file system against bugs or data corruption. The simplicity and low overhead of backpointers makes such an addition to an existing file system feasible.

By eliminating ordering, backpointer-based consistency allows the file system to maintain consistency without depending upon lower-layer primitives such as the disk cache flush. Previous research has shown that SATA drives do not always obey the flush command [29, 36], which is essential for file systems to implement ordering. IDE drives have also been known to disobey flush commands [28, 39]. Using backpointer-based consistency allows a file system to run on top of such misbehaving disks and yet maintain consistency.

Potential users of NoFS should note two things. One, any application which requires strict ordering among file creates and writes should not use NoFS. Two, if there are corrupt files in the system, NoFS will only detect them upon access and not upon file-system mount. Some users may prefer to find out about corruption at mount time rather than when the file system is running. Such a use case aligns better with a file system such as ext3.

7 Related Work

The idea of using information inside or near the block to detect errors is not new. Cambridge File Server [7] used certain bits in each cylinder (cylinder map) to store the allocation status of blocks in that cylinder. Cedar File System [12] used 'labels' inside pages to check their allocation status. Embedding logical identity of blocks (inode number + offset) has been done in RAID to recover from lost and misdirected writes [16]. Transactional flash [27] embeds commit records inside every page to provide transactions and recovery. However, NoFS is the first work that we know of that clearly defines the level of consistency that such information provides and uses such information alone to provide consistency.

The design of the Pilot file system [30] is very similar to that of NoFS. Pilot employs self identifying pages and uses a scavenger to reconstruct the file system metadata upon crash. However, like the file-system check, the scavenger needs to finish running before the file system can be accessed. In NoFS, the file system is made available upon mount, and can be accessed while the scan is running in the background.

Pangaea [33] uses backpointers for consistency in a distributed wide area file system. However, its use of backpointers is limited to directory entry backpointers that are used to resolve conflicting updates on directories. Similar to NoFS, Pangaea also uses the backpointer as the true source of information, letting the backpointers of child inodes dictate whether they belong to a directory or not.

btrfs [48] supports back references that allow it to obtain the list of the extents that refer to a particular extent. Although back references are conceptually similar to NoFS backpointers, the main purpose of btrfs back references is supporting efficient data migration, rather than providing consistency. Other mechanisms such as checksums are used to ensure that the data is not corrupt in btrfs. Another key difference is that btrfs does not always store the back reference inside the allocated extent: sometimes the back references are stored as separate items close to the extent allocation records.

Backlog [17] also uses explicit back references in order to manage migration of data in write anywhere file systems. The back references in Backlog are stored in a separate database, and are designed for efficient querying of usage information rather than consistency. Backlog's back references are not used for incremental file-system checking or resolving ownership disputes.

While NoFS makes an order-less file system more available by eliminating the need for the file-system check, there have been other approaches to increasing availability such as doing the file-system check while the system is online. McKusick's background fsck [19] could repair simple inconsistencies such as lost resources by running fsck on snapshots of a running system. Chunkfs [14] is similar to our work, providing incremental, online file-system checking. Chunkfs differs from NoFS in that the minimal unit of checking is a chunk whereas it is a single file or block in NoFS. Chunkfs does not offer online repair of the file system, while it is possible in NoFS, due to backpointers and non-persistent allocation structures.

8 Conclusion

Every modern file system uses ordering points to ensure consistency. However, ordering points have many disadvantages including lower performance, higher complexity in file-system code, and dependence on lower layers of the storage stack to enforce ordering of writes.

In this paper, we demonstrate that it is possible to build an order-less file system, NoFS, that provides consistency without sacrificing simplicity, availability or performance. NoFS allows immediate data access upon mounting, without file-system checks. We show that NoFS has excellent performance on many workloads, outperforming ext3 on workloads that frequently flush data to disk explicitly.

Although potentially useful for the desktop, we believe NoFS may be of special significance in cloud computing platforms, where many virtual machines are multiplexed onto a physical device. In such cases, the underlying host operating system may try to batch writes together for performance, potentially ignoring ordering requests from virtual machines. NoFS allows virtual machines to maintain consistency without depending on the numerous lower layers of software and hardware. Removing such trust is key to building more robust and reliable storage systems.

A Proof of data consistency in NoFS

Sivathanu et al. [38] provided a logical framework for modelling file systems, and reasoning about the correctness of new features that are added. We show that that adding backpointers to data and metadata blocks in a file system ensures data consistency. Morever, by further adding timestamps, version consistency is achieved.

A.1 Notation

The main entities are *containers, pointers and generations*. A file system is simply a collection of containers, which can be freed and reused. They are linked to each

Symbol	Description
&A	set of entities that point to container A
A^x	the x^{th} version or epoch of A
A_k	the k^{th} generation of A
$g(A^x)$	the gen. of the xth epoch of container A
$\{A^x\}_M$	the x^{th} version of A in memory
$\{A^x\}_D$	the x^{th} version of A on disk
$A \dashrightarrow B$	A has a physical pointer to B
$A \dashleftarrow B$	B has a physical pointer to A
$A \rightarrow B$	A logically points to B
ts(A)	the time that A was last updated
$ts_c(B, A)$	the timestamp for A that is stored in B

Table 4: **Notation used.** *The table describes the symbols and operators used.*

other through pointers. The epoch of a container is incremented and its timestamp is changed every time the contents of a container change in memory. The generation of a container is incremented after each reallocation.

A state of the file system in memory or disk is represented by a belief. Beliefs denoted as $\{\}_M$ and $\{\}_D$ are memory beliefs and disk beliefs respectively. For example, $\{ A \dashrightarrow B\}_D$ indicates a belief that container A physically points to container B on disk.

We also use a special ordering operator called precedes. ($\prec$). Only a belief can appear to the left of a $\prec$ operator. A $\prec$ B means that belief A occurs before B. Table 4 lists these symbols and operators.

A.2 Axioms

In this subsection, we present the axioms that govern the transition of beliefs across memory and disk.

- If a version of a container exists on disk, it must first have existed in memory.

$$\{A^x\}_M \prec write(A) \Rightarrow \{A^x\}_D \tag{1}$$

- B points to A logically in memory (or disk) only if B has a pointer to A, and A has a pointer (backpointer) to B in memory (or disk).

$$\{B \rightarrow A\}_M \iff \{B \dashrightarrow A\}_M \wedge \{B \dashleftarrow A\}_M \tag{2}$$

$$\{B \rightarrow A\}_D \iff \{B \dashrightarrow A\}_D \wedge \{B \dashleftarrow A\}_D \tag{3}$$

- If A does not belong to any container in memory (or disk), it's backpointer does not point to any valid container in memory (or disk).

$$\{\&A = \phi\}_M \Rightarrow \forall c \neg \{c \dashleftarrow A\}_M \tag{4}$$

$$\{\&A = \phi\}_D \Rightarrow \forall c \neg \{c \dashleftarrow A\}_D \tag{5}$$

- Two versions of container B are different only if their timestamps are different.

$$x \neq y \iff ts(B^x) \neq ts(B^y) \tag{6}$$

A.3 Data Consistency

Data consistency guarantees that the data blocks of a file will not contain garbage data or data belonging to another file after a crash. Therefore, we need to prove if B points to A on disk, then the generation of A that is on disk is the same as the generation that was pointed to in memory. In other words, the crash did not end up with B pointing to something else other than the in-memory generation of A.

$$\Big(\{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A^z\}_D\Big) \Rightarrow (g(A^z) = k)$$

We assume that $g(A^z) \neq k$ and prove that this assumption leads to a contradiction.

$$\Big(\{B^x \rightarrow A_k\}_M \prec \{B^x \rightarrow A^z\}_D\Big) \wedge (g(A^z) \neq k)$$

Since the epoch of B 'x' is the same on disk and memory, B has not been changed until the disk write happened. $g(A^z) \neq k$ indicates that A has a new generation, therefore it has undergone reallocation. Block A must have been freed and written to disk. After the free, it was reallocated to B again.

This leads to two cases where the free could have happened - before the write of B (in memory) or after the write of B (on disk).

Case 1: Block A was freed and written to disk before Block B was written.

$$\Rightarrow \quad \Big((\{B^x \rightarrow A_k\}_M \ \wedge \ (\&A = \phi) \wedge write(a)) \prec \{B^x \rightarrow A^z\}_D\Big) \ \wedge \ (g(A^z) \neq k)$$

By (2), if B logically points to A, A has a backpointer to B.

$$\Rightarrow \quad \Big((\{B^x \dashleftarrow A_k\}_M \ \wedge \ (\&A = \phi) \wedge write(a)) \prec \{B^x \rightarrow A^z\}_D\Big) \ \wedge \ (g(A^z) \neq k)$$

By (4), a free container should not point to any other container.

$$\Rightarrow \Big((\{B^x \dashleftarrow A_k\}_M \wedge \neg\{B^x \dashleftarrow A_k\}_M) \prec \{B^x \rightarrow A^z\}_D\Big) \wedge (g(A^z) \neq k)$$

Combining the first two clauses, we arrive at a contradiction.

$$\Rightarrow \Big(false \prec \{B^x \rightarrow A^z\}_D\Big) \wedge (g(A^z) \neq k)$$

We have arrived at a contradiction (i.e a false belief), and hence this case cannot occur.

Case 2: Block A was freed and written to disk after Block B was written. The steps in the proof are similar to those in Case 1.

$$\Rightarrow \Big(\{B^x \rightarrow A_k\}_M \prec (\{B^x \rightarrow A^z\}_D \wedge(\&A = \phi) \wedge write(a))\Big) \wedge (g(A^z) \neq k)$$

$\Rightarrow$ Using (2),

$$\Big(\{B^x \rightarrow A_k\}_M \prec (\{B^x \dashleftarrow A^z\}_D \wedge(\&A = \phi) \wedge write(a))) \wedge (g(A^z) \neq k)$$

$$\Rightarrow \Big(\{B^x \rightarrow A_k\}_M \prec (\{B^x \dashleftarrow A^z\}_D \wedge\neg\{B^x \dashleftarrow A^z\}_D)\Big) \wedge (g(A^z) \neq k)$$

(Since $\{\&A = \phi\}_D \Rightarrow \forall c \neg\{c \dashleftarrow A\}_D$)

$$\Rightarrow \Big(\{B^x \rightarrow A_k\}_M \prec false\Big) \wedge (g(A^z) \neq k)$$

We have arrived at a contradiction, and hence this case cannot occur. Thus we have shown that data consistency holds given that the file system contains backpointers.

A.4 Version Consistency

Version consistency is a stricter version of data consistency. In version consistency, each data block has a timestamp indicating when it was last updated. This timestamp is also stored in the inode, with the pointer to the data block. When a block is accessed, the timestamp in the inode and the data block must match. This helps us detect lost updates to data blocks. This is reflected in the rule:

$$\{B^x \rightarrow A^y\}_D \Rightarrow \{B^x \rightarrow A^y\}_M \ll \{B^x \rightarrow A^y\}_D$$

For the L.H.S to hold on disk, writes to both B and A need to have happened. This could have happened in two ways, considering the two possible orderings of the writes to A and B:

$$\{B^x \rightarrow A^y\}_D \Rightarrow (\{B^x \rightarrow A^b\}_M \prec write(B)) \vee (\{B^a \rightarrow A^y\}_M \prec write(A))$$

where a and b are arbitrary epochs of containers B and A. The two cases are: B could have remained the same, with a new version of A being written from memory to disk, or A could have remained the same, with a new version of B being written from memory disk. We explore both cases.

Consider the first case:

$$\{B^x \rightarrow A^y\}_D \Rightarrow (\{B^x \rightarrow A^b\}_M \prec write(B))$$

Now, for the memory and on-disk copies of A to match, we need to prove that b = y:

$$\{B^x \rightarrow A^b\}_M \Rightarrow ts_c(B^x, A) = ts(A^b)$$
$$\{B^x \rightarrow A^y\}_D \Rightarrow ts_c(B^x, A) = ts(A^y)$$
$$\Rightarrow ts(A^b) = ts(A^y)$$
$$\Rightarrow b = y$$

Similarly, for Case 2, we can prove that a = x. Hence, when the file system uses back pointers with timestamps, we have shown that version consistency holds.

References

[1] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.

[2] Steve Best. JFS Overview. `www.ibm.com/developerworks/library/l-jfs.html`, 2000.

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

[4] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, 1994.

[5] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. Technical Report 1709, University of Wisconsin-Madison Computer Sciences, January 2012.

[6] Charles D. Cranor and Gurudatta M. Parulkar. The UVM Virtual Memory System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '99)*, Monterey, California, June 1999.

[7] Jeremy Dion. The Cambridge File Server. *SIGOPS Operating Systems Review*, 14:26–35, October 1980.

[8] Stony Brook University File system Storage Lab (FSL). Filebench Benchmark. `http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench`, 2011.

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

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

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing, New York, October 2003.

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

[13] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. Reducing Fsck Time For Ext2 File Systems. In *Ottawa Linux Symposium (OLS '06)*, Ottawa, Canada, July 2006.

[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 *IEEE 2nd Workshop on Hot Topics in System Dependability (HotDep '06)*, Seattle, Washington, November 2006.

[15] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '94)*, San Francisco, California, January 1994.

[16] Andrew Krioukov, Lakshmi N. Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[17] Peter Macko, Margo Seltzer, and Keith A. Smith. Tracking Back References in a Write-Anywhere File System. In *Proceedings of the 8th USENIX conference on File and storage technologies*, San Jose, California, February 2010.

[18] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for

UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[19] Marshall Kirk McKusick. Running 'fsck' in the Background. In *Proceedings of BSDCon 2002 (BSDCon '02)*, San Fransisco, CA, February 2002.

[20] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[21] Rajeev Nagar. *Windows NT File System Internals: A Developer's Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997.

[22] Chris Nyberg. gensort Data Generator. `http://www.ordinal.com/gensort.html`, 2009.

[23] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[24] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast Consistency Checking for the Solaris File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, pages 77–89, New Orleans, Louisiana, June 1998.

[25] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[26] 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 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

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

[28] R1Soft. Disk Safe Best Practices. `http://wiki.r1soft.com/display/CDP3/Disk+Safe+Best+Practices`, December 2011.

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

[30] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C.Purcell. Pilot: An Operating System for a Personal Computer. *Communications of the ACM*, 23(2):81–92, February 1980.

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

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

[33] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming Aggressive Replication in the Pangaea Wide-Area File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[34] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[35] Eric Schrock. UFS/SVM vs. ZFS: Code Complexity. `http://blogs.sun.com/eschrock/`, November 2005.

[36] Seagate Forums. ST3250823AS (7200.8) ignores FLUSH CACHE in AHCI mode. `http://bit.ly/xcSAUV`, September 2011.

[37] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST '10*, Incline Village, Nevada, May.

[38] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Somesh Jha. A Logic of File Systems. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 1–15, San Francisco, California, December 2005.

[39] SQLite. How To Corrupt Your Database Files. `http://www.sqlite.org/lockingv3.html`.

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

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

[42] Technical Committee T10. T10 Data Integrity Field standard. `http://www.t10.org/`, 2009.

[43] The Serial ATA International Organization. Serial ATA Revision 3.0 Specification. `http://www.sata-io.org/technology/6Gbdetails.asp`, June 2009.

[44] Theodore Ts'o. `http://e2fsprogs.sourceforge.net`, June 2001.

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

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

[47] VirtualBox Manual. Responding to guest IDE/SATA flush requests. `http://www.virtualbox.org/manual/ch12.html`.

[48] Wikipedia. Btrfs. `en.wikipedia.org/wiki/Btrfs`, 2009.

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

[50] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.

[51] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, California, February 2010.