ffsck: The Fast File System Checker

Ao Ma[*+], Chris Dragga[+], Andrea C. Arpaci-Dusseau[+], Remzi H. Arpaci-Dusseau[+]
Backup Recovery Systems Division, EMC Corporation[*] *University of Wisconsin, Madison*[+]

Abstract

Crash failures, hardware errors, and file system bugs can corrupt file systems and cause data loss, despite the presence of journals and similar preventive techniques. While consistency checkers such as fsck can detect this corruption and restore a damaged image to a usable state, they are generally created as an afterthought, to be run only at rare intervals. Thus, checkers operate slowly, causing significant downtime for large scale storage systems when they are needed.

We address this dilemma by treating the checker as a key component of the overall file system (and not merely a peripheral add-on). To this end, we present a modified ext3 file system, *rext3*, to directly support the fast file system checker, *ffsck*. The rext3 file system co-locates and self-identifies its metadata blocks, removing the need for costly seeks and tree traversals during checking. These modifications to the file system allow ffsck to scan and repair the file system at rates approaching the full sequential bandwidth of the underlying device. In addition, we demonstrate that rext3 performs competitively with ext3 in most cases and exceeds it in handling random reads and large writes.

1 Introduction

Data integrity is critically important for personal users and companies. Once data becomes lost or corrupted, it is expensive and challenging to restore [22]. As the file system plays a central role in storing and organizing data, system developers must carefully consider how to keep the file system robust and reliable.

Unfortunately, there are a variety of factors that can corrupt a file system. Unclean shutdowns and bugs in the file system or device drivers can easily corrupt metadata [13, 45]. Hardware failures such as memory corruption [24, 36, 52] and disk corruption [1, 4, 5, 7, 42, 46] can render the file system inconsistent. Since metadata corruption and file-system inconsistency can easily propagate to different parts of the file system, these errors can ultimately lead to significant data loss.

In the past several decades, file and storage systems have devised a variety of different techniques to handle corruption. Journaling [8, 33, 44, 47], copy-on-write [21, 34, 41, 48], and soft updates [17] can handle inconsistency in the event of a system crash. Bug finding tools [14, 49, 50] can identify and remove errors in file-system source code and prevent them from corrupting the file system. Checksums [6, 9, 40, 42] and scrubbing [37] can detect hardware failures and repair corrupted blocks with copies [29].

Unfortunately, while these tools and techniques can reduce the probability of system corruption and repair inconsistent file systems in some cases, they cannot protect against all faults. Even when combined with mechanisms to improve reliability, as seen in ZFS [9] and XFS [44], certain errors can still evade these measures and damage the file system [16, 51].

As a result, the offline file-system checker remains a last resort to protect the file system. A file-system checker restores a file-system image back to a consistent and usable state by scanning all the file-system metadata and using redundancy information to detect and repair inconsistencies [27]. Unlike the mechanisms previously described, file-system checkers are robust to nearly all types of failure (except those within the checker itself [11, 18]).

Despite the importance of file-system checkers, users and administrators are reluctant to run them, frequently complaining about bad experiences when doing so [2, 43]. Complaints such as "It's been almost 24 hours now and e2fsck -f -y -v /dev/sda1 is still running" are not uncommon [3]. Generally, most file-system checkers run extremely slowly, without providing a reliable indication of their progress or anticipated completion time. Because of this, system administrators often have to endure significant, unpredictable downtime when running a checker.

Addressing this dilemma and building an efficient checker for large scale storage systems requires a new approach that treats the file-system checker as more than an afterthought. Thus, we propose the *rext3* file system, a modified version of ext3 which sets fast handling of inconsistency as a principle design goal, providing direct support for the file-system checker in its implementation. To accompany this new file system, we develop a new fast file-system checker, *ffsck*.

While rext3 and ffsck are based on the widely-used Linux ext3 file system [47] and its default checker, e2fsck, respectively, they include several novel components that can be easily applied to other file systems and corresponding checkers. Specifically, rext3 enhances metadata density by co-locating all the file indirect blocks for each block group [10] and better supports ffsck by using backpointer-based data structures for these indirect blocks [12]. To fully utilize disk bandwidth, ffsck performs a disk-order scan of system metadata instead of a traditional inode-order scan. To reduce memory pressure, ffsck compresses in-memory metadata on the fly, in a fashion arising naturally from a disk order scan. Finally, ffsck employs a bitmap snapshot to avoid costly double scans that would otherwise occur when it encounters doubly-allocated data blocks.

Our measurements show that ffsck scans the file system significantly faster than e2fsck, nearing the sequential peak of disk bandwidth. Moreover, its speed is robust to file system aging, making it possible to estimate its running time beforehand and thus helping the system administrator make better decisions about running the checker.

We also find that, surprisingly, rext3 improves ordinary I/O performance in some cases. Specifically, rext3 performs up to 20% better in large sequential writes due to improvements in journaling performance and up to 43% faster in random reads due to better utilization of the segmented disk track buffer. In other cases, it remains competitive with ext3, with a worst-case degradation of 10% due to additional seeks caused by metadata co-location.

The rest of this paper is organized as follows. We first introduce related work (§2), and then provide an overview of the file-system checking policy and analyze traditional checker bottlenecks (§3). We next describe the design and implementation of rext3 and ffsck (§4), evaluate their performance (§5), discuss ffsck's and rext3's limitations (§6), and summarize our conclusions (§7).

2 Related Work

We introduce research closely related to our work in this section. In general, all of this research seeks to improve file system reliability and protect the user from potential data loss in the face of system crashes, file-system bugs, and hardware failures.

Mechanisms such as journaling [8, 19, 33, 44, 47], copy-on-write [21, 34, 41, 48], soft updates [17], and backpointer-based consistency [12] can handle file-system inconsistency caused by system crashes, but cannot rectify errors arising from file-system bugs and hardware failures [4, 7, 24, 52].

File-system bugs can be detected and fixed by bug-finding tools [14, 49, 50], which can significantly reduce the probability of file-system inconsistency resulting from coding errors. Despite their success, these tools have so far been unable to remove all bugs. For example, bugs that can cause metadata corruption are still being found in the widely-deployed ext3 file system, which has been in use for more than 10 years.

Hardware errors can be identified with checksums [6, 9, 40, 42] and corrected with redundant copies [29]. Though these mechanisms protect the file system from a wide range of disk faults [1, 4, 7, 46], they cannot handle errors coming from the file-system source code or hardware failures that happen before applying a checksum or creating a copy [51].

In general, these aforementioned mechanisms can only protect against a subset of the factors which corrupt file systems; none of them provide universal protection. Therefore, the checker remains an indispensable tool, serving as the last line of defense.

However, traditional checkers require a full scan of the entire file system, causing significant downtime. To make matters worse, while disk bandwidth and seek time have changed little in recent years, file systems have only grown larger, lengthening the time required for these scans [20]. Thus, a number of file systems have been developed with the intent of reducing this requirement.

Extent-based file systems, such as ext4 [23] and XFS [44] offer a straightforward improvement over the direct mapping of block pointers employed by file systems such as FFS and ext3. Extents can significantly compress metadata, reducing the amount that the checker has to scan if the average file size is large and most blocks are allocated contiguously. However, if the file system contains a large percentage of small and sparsely allocated files or the disk image is highly fragmented, checking (without costly defragmentation [35]) will suffer. In Section 3.2, we use ext4 as a case study to explore how these factors affect the file-system checker's performance.

In addition to using extents, ext4 further optimizes the file system checker by indicating which inodes in the inode table are currently in use, allowing e2fsck to skip unused inode blocks during its scan [23]. Unfortunately, this approach will grow less effective over time, as the file system's utilization increases. Furthermore, in the event that either the inode table or the checksum that protects the uninitialized inode table high-water mark becomes corrupted, the checker will still have to perform a full inode scan, rendering this heuristic ineffective.

Both Chunkfs and Solaris UFS provide an alternate method for addressing checking time, attempting to reduce it by dividing the file system into isolated chunks [20, 30]. Though the idea of fault isolation is appealing, it is difficult to create entirely separate chunks that can be checked independently. Moreover, it can be difficult to reliably determine that a partial check has found all errors in the system, especially since most failure modes give little indication of where an error may lie.

Finally, Abishek Rai's metaclustering patch [32] reduces the amount of time that fsck spends seeking between metadata and data by grouping indirect blocks together on disk. This closely resembles our approach in rext3; however, Rai focuses solely on the file system, presenting only a partial solution. By designing a file system checker in tandem with a new layout, we are able to realize larger improvements.

In contrast to the previous solutions, other work seeks to improve the repair process itself, rather than the underlying file system. Again, though, these works generally fail to provide a fully integrated solution.

Recon protects file system metadata from buggy file system operations by verifying metadata consistency at run-time [15]. Doing so allows Recon to detect metadata corruption before committing it to disk, preventing its propagation. However, Recon does not perform a global scan and thus cannot protect against errors resulting from hardware failures. The current implementation of Recon also relies on the proper functioning of the ext3 JBD, making the strong assumption that it is free of bugs.

McKusick proposes avoiding downtime during file checking by running the checker as a background process on a file-system snapshot [25]. While this does allow the primary system to continue running, assuming that file-system corruption is not catastrophic, it does not improve the overall time required to run the checker, and thus, some data may remain offline for long periods of time. It also requires the underlying file system to support soft updates, which may not be available.

Similarly, both WAFL [28], NetApp's file system, and ReFS [38], Microsoft's server-side successor for NTFS, provide mechanisms for the online removal of corrupt data. These are designed to remove the need for an offline checking tool; however, given the limited availability of technical details on these systems, it is difficult to evaluate their effectiveness.

Finally, SQCK takes a different approach to improving file-system checkers, focusing on reliability and correctness instead of execution time. To do so, it transforms the complicated rules used by most checkers into SQL, benefiting from the simplicity and compactness of a query language [18]. However, its simplicity comes at some cost; it executes more slowly than traditional checkers.

In summary, the file system checker provides the last chance to recover a damaged file system image, but the significant downtime it causes when scanning makes it costly. Though the mechanisms introduced above can accelerate the checking process, many of them sacrifice the checker's primary goal: thoroughly scanning the whole file system and guaranteeing complete freedom from inconsistency. Those that do avoid this pitfall only focus on part of the solution—either the checker or the file system—and fail to improve both.

3 Extended Motivation

Before developing a better file-system checker, one needs to thoroughly understand the approach current checkers use and clearly define their bottlenecks. In this section, we first introduce the overall file-system check and repair policy, focusing on the widely-used open-source checker e2fsck. We examine how well the checker performs under varying conditions, including file size and age, comparing both ext3 and ext4. Finally, we discuss the design trade-offs that account for poor checking performance.

Phase	Scan and checking task
1	Scan and check all inodes and indirect blocks. If blocks are multiply claimed, rescan all previously checked metadata to choose an owner.
2	Individually check each directory.
3	Check the directory connectivity.
4	Check the inode reference count and remove orphan inodes.
5	Update block and inode bitmaps if necessary.

Table 1: **Phases of e2fsck Operation.** *This table lists the main scanning and checking phases in e2fsck.*

3.1 Fsck Background

Though its reliability has improved over the past decade, file systems are still fragile and vulnerable to a variety of errors. When McKusick *et al.* designed and implemented the Fast File System [26], they also developed the fsck utility to restore a corrupt file-system image to a consistent and usable state [27]. At a high level, fsck scans all of the file system's metadata and uses redundant structural information to perform consistency checks. If an inconsistency is detected during the scanning process, the checker will repair it with best effort.

Below, we briefly describe how e2fsck uses these fields to verify the consistency of each type of metadata. E2fsck primarily executes its checking rules in five phases, described in Table 1. Once these phases are complete, the following rules will have been validated:

Superblock: e2fsck checks the values stored in the superblock, including the file-system size, number of inodes, free block count, and the free inode count. Although there is no way to accurately verify the first two numbers, because they are statically determined upon creation of the file-system disk image, fsck can still check whether these sizes are within a reasonable range.

Group Descriptor: e2fsck checks that blocks marked free in the data bitmap are not claimed by any files and that inodes marked free in the inode bitmap are not in use.

Directory: e2fsck applies several checking rules to each directory data block, including whether the directory inode numbers point to unallocated inodes, whether the directory inode numbers lie in a reasonable range, and whether the inode numbers of "." and ".." reference unallocated inodes.

Inode: on the most basic level, e2fsck verifies the consistency of the internal state of the inode, including its type and allocation status. In addition, it verifies the inode's link count and the number of blocks claimed by the inode. Finally, it ensures that all the blocks pointed to by the inode have valid numbers and are not held by any other inode.

Indirect Block: as with inodes, fsck checks that each block claimed by the indirect block has not been claimed by another and that each block number is valid. In addition, it records the number of data blocks that the indirect block references for later comparison with the total size claimed by the parent inode.

3.2 E2fsck Performance Analysis

In this section, we analyze the factors that affect fsck's performance, allowing us to accurately locate its bottlenecks and obtain hints for further optimization. We perform all of our experiments on a 2.2 Ghz AMD Opteron machine with 1 GB of memory and a 750 GB Seagate Barracuda 7200.12 testing disk with Linux 2.6.28 and e2fsprog 1.41.12. We create all file system images with their default settings, except where otherwise specified, and enable all features.

We first examine how e2fsck performs as the file system size grows. We initialize the file system image by creating one directory per block group, each of which contains a number of files with sizes chosen uniformly from a one to 512 block (4 KB-2 MB) range; we create files in this directory until it contains 25.6 MB (20% of the block group size). To increase the size of the file system to the desired amount, we then randomly create new files (4 KB-2 MB) or append one to 256 blocks (4 KB-1 MB) of data to existing files, choosing between the two operations uniformly. We display our results in Figure 1. The total run time of e2fsck grows quickly as the size of file system increases, indicating that e2fsck's performance does not scale to large file system images.

To determine which portion of the check dominates the scan time, Figure 1 also breaks down the total time by the amount spent in each phase. We find that phase 1 occupies more than 95% of the total checking time. During this phase, e2fsck scans all inodes and their corresponding indirect blocks, which comprise the largest portion of the file-system's metadata. Furthermore, since e2fsck has to execute this scan again if it detects multiply-claimed blocks, the actual total time may be even longer in the presence of errors.

To better understand the I/O behavior during this phase, we measure the time e2fsck spends reading each individual block during the 150 GB experiment. We show our results in Figure 2, which displays the cumulative time spent reading indirect and inode blocks, represented by Xs and circles, respectively. Accesses of indirect blocks overwhelmingly dominate I/O time. This behavior results from ext3's disk layout: inode blocks are stored contiguously, while indirect blocks are dynamically allocated and thereby scattered throughout the disk, requiring a separate seek for each block access.

Given this time spent reading indirect blocks, one might assume that an extent-based file system would reduce the time required to check the file system. However, extents are also likely to suffer under fragmentation resulting from regular use. To examine this effect, we measure the speed of e2fsck on a series of ext3 and ext4 file system images in increasing stages of aging [39].

We construct each image by initializing the file system as described previously and then performing a series of file creations, appends, truncations, and deletions, choosing uniformly between them. File creation and appending use the approaches described in our first experiment. Truncation chooses a random offset into the file and truncates it to that length. Finally, deletion chooses a file to delete at random from the current directory. As these operations can change the size of the final file system image dramatically, we discard any image with a capacity under 90% and generate a new one.

Figure 3 shows our results for this experiment. The x-axis shows the number of aging operations performed per directory and the y-axis depicts the throughput obtained by e2fsck, measured by bytes accessed (not by the total data in the file system). The results demonstrate that, while ext4 initially performs much better than ext3, it rapidly degrades due to increased fragmentation, performing only marginally better than ext3 under even moderate aging. Neither system ultimately performs well, achieving less than 10% of the underlying 100 MB/s disk bandwidth (calculated accounting for bandwidth differences between zones).

From these three experiments, we conclude that e2fsck does not scale well as the file system grows, that checking inodes and indirect blocks occupies the most time, and that e2fsck's performance degrades significantly as the file system ages. In addition, because ext4 shows little difference from ext3 in the presence of file-system aging, we focus the remainder of our discussion entirely on ext3.

3.3 File System Design Trade-offs

Based on our previous analysis, we observe two file-system design decisions that lead to e2fsck's poor performance. First, ext3 uses the same allocation strategies for data blocks and indirect blocks, storing them in a contiguous fashion to facilitate sequential access. However, this design causes indirect blocks to be scattered throughout the disk, growing increasingly further apart as the file system ages. Given the low density of these blocks, e2fsck has to pay a significant penalty to access them.

Second, ext3 relies on a tree structure to locate all of the indirect blocks, imposing a strict ordering of accesses when checking a file. For example, e2fsck can only locate a double indirect block by first traversing its inode and then its parent indirect block. This limitation prevents the checker from optimizing its accesses using disk locality.

Though batching several adjacent inodes and fetching all of their indirect blocks in an order sorted by their dou-

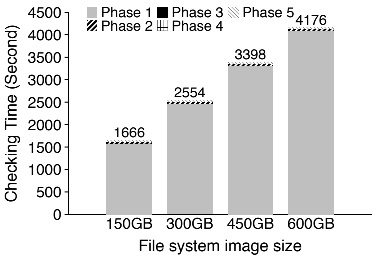

Figure 1: **e2fsck Execution Time By Size.** *This graph shows e2fsck's execution time for differently sized file-system images, broken down by time spent in each phase.*

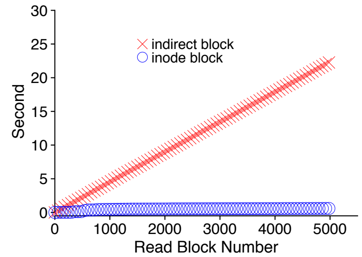

Figure 2: **Cumulative e2fsck Read Time In Phase 1.** *This figure shows the cumulative time to access each indirect block and inode block in phase 1 of e2fsck's scan.*

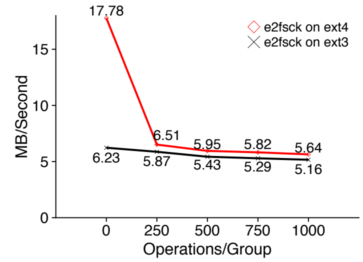

Figure 3: **e2fsck on Aging File System Images.** *This graph shows that e2fsck's speed degrades as the file system ages. The file system is on a 750 GB partition and has around 95% utilization.*

ble indirect blocks could ameliorate the I/O penalty to some extent, the dependency on a tree structure still limits the overall optimization across file boundaries. For example, if the checker knew the locations of all of the indirect blocks, it could access them with a sequential scan. Because it lacks this information, the checker is forced to seek frequently to look up the indirect blocks of a file.

From our analysis, we infer that fsck's poor performance results from a long-term focus on preventing errors, instead of fixing them. The checker is usually regarded as a peripheral addition to the file system to be used only as a matter of last resort, rather than an integral component. Despite this, factors in practical deployment that cause the file system to become corrupted and inconsistent may significantly exceed designers' expectations. For example, soon after SGI deployed XFS with "no need for fsck, ever," they added a checker for it [16].

4 Design and Implementation

In this section, we describe the design and implementation of both the rext3 file system and ffsck, the fast file system checker. While these systems are based on ext3 and e2fsck, respectively, they each contain a number of modifications designed to reduce the time required to fully check the file system. Rext3 improves metadata density by co-locating all the indirect blocks in a block group and uses a backpointer-based structure for indirect blocks to better support a fast checker. Ffsck then uses this modified layout to perform a disk-order scan that accesses all of the file-system metadata at once, significantly reducing the number of seek operations from that incurred by a traditional inode-order scan. To mitigate the memory pressure that storing all this data could incur, ffsck compresses the metadata that it caches on the fly. Finally, ffsck employs a bitmap snapshot to reduce the number of inode and indirect blocks it has to rescan when it encounters multiply claimed data blocks.

4.1 Goals

We expect rext3 and ffsck to meet the following criteria:

Fast scan speed: unlike e2fsck, which is limited to roughly 10% of the underlying disk bandwidth, we expect ffsck to scan the file system with the greatest possible efficiency. The ability to scan and repair quickly should be of paramount concern for file-system designers, as nobody wants to wait hours to bring a file system back online.

Robust performance despite file-system aging: e2fsck's speed drops quickly as the file system ages, which not only significantly increases its running time but also makes it impractical to estimate its completion time. We expect our new checker to scan the system at a constant speed, regardless of the aging that has occurred in the file system, allowing the system administrator to better decide when to execute the checker.

Competitive file-system performance: repairability cannot come at the expense of responsiveness and throughput, as these are critical in production environments. Therefore, we focus on ensuring that our repair-driven file system performs competitively with ext3.

4.2 Rext3 File System

Rext3 is developed atop ext3, the default file system for many popular Linux distributions. Rext3 inherits most of the mechanisms used in ext3, except two: the disk layout and the indirect block structure. This section details these new features and gives a basic overview of our implementation.

4.2.1 Rext3 Disk Layout

To reduce the time spent in phase one of file checking, rext3 decouples the allocation of indirect blocks and data blocks by reserving a block region immediately after the inode table in each block group, called the *indirect region*. This region stores all dynamically allocated metadata: specifically, indirect blocks and directory data

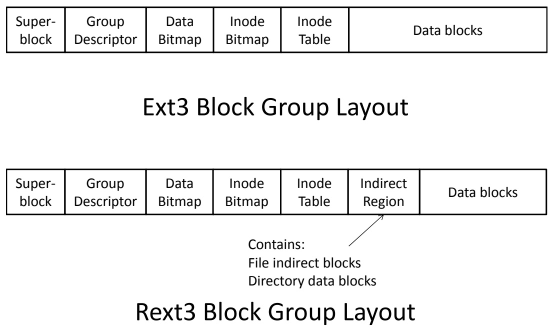

Figure 4: **ext3 and rext3 Disk Layout Comparison.** *This figure shows the disk layouts of both ext3 and rext3.*

blocks. When allocating metadata blocks for a file, rext3 first attempts to allocate from the indirect region in the same block group of the file's inode. If the current region is full, rext3 will iterate over the subsequent indirect regions until a free block is found. Ordinary data blocks are restricted to the free blocks outside the indirect region, but otherwise use the same allocation algorithm as ext3. The disk layout is shown in Figure 4.

By co-locating dynamically allocated metadata, we improve the I/O density during the checker's scan phase, because this metadata will no longer be scattered throughout the disk. Instead of having to perform a separate seek for each of these blocks, the indirect region of rext3 allows the fast checker to perform one sequential read of all metadata blocks for a given block group, including the block and inode bitmaps, the inode table, indirect blocks, and directory data blocks.

Initially, the indirect region seems to work against our goal of competitive file-system performance. Because we separate the allocation of indirect blocks and their corresponding data blocks, sequential access requires additional seek operations, apparently slowing ordinary use to accelerate the repair process. However, modern disks generally buffer entire tracks when reading individual blocks; thus, the disk will fetch several indirect blocks with a single I/O. Subsequent reads of these blocks will then return from the disk cache, which is usually an order of magnitude faster than the disk platter. This allows rext3 to perform as efficiently as ext3 in most cases, without the extensive manual prefetching used in other systems that allocate metadata and data separately, such as DualFS [31] and the ext3 metaclustering patch [32]. We verify this claim in Section 5.

4.2.2 Backpointer-based indirect structure

To allow ffsck to link related indirect blocks and perform some verification without referring to the indirect tree, rext3 uses a backpointer-based data structure. Specifically, the beginning of each indirect block will contain its inode number and level in the indirect tree structure. We discuss how ffsck uses these pointers in Section 4.3.2.

Because this global information is added when allocating a new indirect block for the file, it does not degrade performance. However, it does reduce the number of block pointers that each indirect block can store.

4.2.3 Rext3 Implementation

Rext3 is implemented in Linux 2.6.28. Our implementation removes the preallocation mechanism for indirect blocks [10], as all indirect blocks are written to designated indirect regions. In total, our modifications add 1357 lines to the ext3 codebase, most of which reside in inode.c and balloc.c.

The default size of each indirect region is 2 MB, which, given the default 4 KB block size, allows for 512 blocks; however, users can adjust this parameter based on the expected average file size. Tuning this parameter properly is of key importance: too large a value will lead to wasted disk space, while too small a value may cause the file system to run out of indirect blocks.

4.3 Fast File System Checker

As rext3 is based on ext3, ffsck is based on e2fsck, the default file system checker for ext2 and ext3. Ffsck inherits the same checking policy and phase structure employed by e2fsck; however, it features three new components. First, it performs a disk-order scan to fetch all file-system metadata into memory. Second, it compresses metadata on the fly to alleviate the memory pressure caused by the aforementioned scan. Third, it employs a bitmap snapshot that allows it to avoid a costly double inode scan when it encounters a doubly-allocated block. The following subsections detail each of these features individually.

4.3.1 Disk-order Scan

Ffsck loads all the file-system metadata into memory in a single sequential scan, referred to as a disk-order scan. Each metadata block is fetched in an order sorted by its disk address. To perform this disk-order scan, ffsck needs to know the location of each metadata item ahead of time. Statically-allocated metadata, such as the superblock, block descriptors, inodes, data bitmap, and inode bitmap, have a fixed location on disk, allowing ffsck to obtain their addresses with a simple calculation. Ffsck locates dynamically allocated metadata, such as the indirect and directory data blocks, using the portion of the data bitmap corresponding to the indirect region. Because the indirect region immediately follows the fixed-length inode table, ffsck can obtain its block range and fetch every block within it marked in-use in the data bitmap, removing nearly all seeks except those between block groups. Thus, ffsck is capable of scanning the file system with close to maximum efficiency under normal operation.

Moreover, the combination of the indirect region and the disk-order scan removes the negative influence of file-system aging on the check speed. Regardless of the file system's age, indirect and directory data blocks will still reside in the indirect region, ensuring that the scan speed remains nearly constant as the file system ages. Ffsck's execution time varies only with the total number of block groups, which is proportional to the size of the file system image. Thus, system administrators can accurately and easily evaluate the total checking time beforehand and infer how long downtime will last.

4.3.2 Self-check, Cross-check, and Compression

The disk-order scan is an efficient approach to maximizing the utilization of disk bandwidth. However, it leads to a new challenge: because ffsck accesses all of the file system metadata in a physical, rather than logical sequence, it cannot apply checking rules until all related metadata are cached in memory. Since the metadata size is directly proportional to the file-system data size, a disk-order scan can easily lead to memory saturation in large-scale storage systems. To prevent this, it will suffice to reduce the in-memory footprint of inodes and indirect blocks, since these comprise the majority of the file-system metadata.

All of the checking rules for inode and indirect blocks can be categorized into two groups. The first of these, self-check, only relies on the internal structure of the inode or indirect block; this group includes checks on inode types and on indirect block data pointer ranges, among others. The second, cross-check, requires data structures from multiple metadata items; this group includes more complicated checks, such as verifying the total number of blocks claimed by an inode and its file size.

Cross-check rules do not need all the information in each object; thus, we can save memory by removing unused fields of each metadata object. Ffsck self-checks each metadata block as soon as it is loaded into memory; ffsck then frees the memory it no longer needs, retaining only the data used for cross-check rules. Once all related metadata for a file are in memory and processed with the self-check rules, ffsck then executes the cross-check rules, removing the metadata entirely once the cross-check is complete for all files. With this method, we convert the memory saturation problem into a compression problem.

Self-check and Cross-check rules: The self-check of an inode includes checking the inode type, link count, and allocation state, and the self-check of an indirect block includes verifying its data block pointer range and its allocation state. These checks are performed when the block is fetched into memory.

The cross-check between each inode and its indirect blocks has two stages. First, it verifies that the number of blocks claimed by the inode agrees with the number of the actual blocks owned by the inode, which is the sum of the total number of indirect blocks owned by the inode and the number of non-zero block pointers in each of them. Second, it calculates the actual file size based on the last block's offset and compares that with the file size field of the inode.

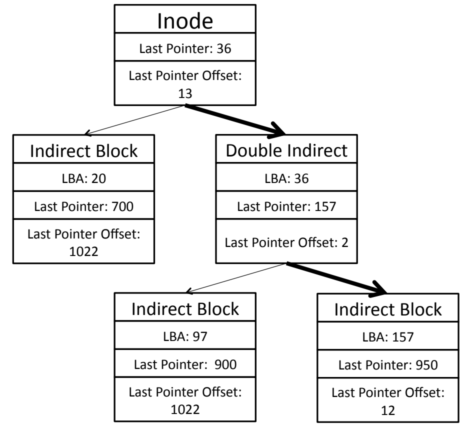

Figure 5: **File Size Verification.** *This figure shows the logical tree structure used to verify a file's size. The bold arrows indicate the path followed to retrieve and construct the final offset.*

For the first cross-check, ffsck links together blocks from the same file using the backpointers to the parent inode provided by rext3, allowing it to avoid the indirect tree structure. When self-checking an indirect block, ffsck also records the number of non-zero block pointers it contains, associating this with the inode number stored in its backpointer. Using this backpointer information, ffsck can then sum the block pointer counts to obtain the actual number of blocks associated with the inode.

The second cross-check verifies file size, requiring the file offset of the last data block pointer. To determine this, ffsck records the disk location of each indirect block, along with the address and offset of its last non-zero block pointer. This information allows ffsck to partially rebuild the indirect tree, find the last block offset, and calculate the actual file size. We provide an example of this procedure in Figure 5.

Because ffsck finds indirect blocks by using the portion of the data bitmap corresponding to the indirect region rather than by following pointers in metadata blocks, bit-flips or similar types of corruption in the data bitmap may cause ffsck to process obsolete indirect blocks or ignore current ones. Traditional checkers are not susceptible to this problem, since they directly traverse the indirect tree for each file; thus, we provide a mechanism to prevent this occurrence.

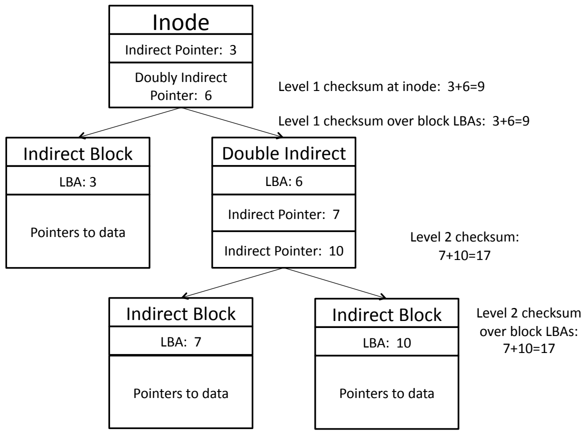

Figure 6: **Checksum of Each Indirect Layer.** *This figure shows how ffsck calculates and verifies the checksum of each layer in the tree of indirect blocks.*

For each file, ffsck calculates a series of checksums, two for each layer in the indirect tree. It calculates the first of these using the pointers in the inode (for the top level) or the indirect block (for the subsequent levels). It calculates the second of these using the logical block addresses of the blocks in the lower layer, which it identifies using the backpointer and layer data stored with each indirect block. If the two checksums are equal, ffsck assumes that the indirect blocks are up-to-date; otherwise, it manually traverses the file's indirect tree to obtain the correct metadata blocks. Currently, our checksum consists of a sum, though this could easily be upgraded to a more advanced function, such as a collision resistant hash, albeit at the cost of additional computation and memory. We provide an example of this procedure in Figure 6.

Compression: Thanks to our careful grouping of checking rules, we no longer need to retain all fields of a metadata item in memory after self-checking it. For inodes, we only store the inode number, file size, block number, last non-zero block pointer address, checksum of each layer, and data structures needed to link the inode with its corresponding indirect blocks. Similarly, indirect blocks only require their own address, the number of non-zero block pointers they contain, and the address and offset of their last block pointer.

Discarding data that will no longer be used significantly reduces ffsck's in-memory footprint. Specifically, the compression ratio of inodes is nearly 2:1 and the compression ratio of indirect blocks is nearly 250:1, substantially lowering the probability of memory saturation.

Figure 7 provides a quantitative comparison of the memory cost of storing the metadata for one gigabyte of data before and after compression. The x-axis represents the average file size, and the y-axis shows the memory cost of storing the inode and indirect blocks. Memory utilization peaks at 86 MB when the average file size is 49KB, at which size the average file has one indirect block and several data blocks; this ratio of metadata to data will not scale to large storage systems. However, by compressing the metadata, we lower memory consumption to 1 MB, alleviating much of the memory pressure.

4.3.3 Bitmap Snapshot

When a file-system checker detects a data block claimed by more than one inode, it must determine which inodes claim the data block and assign the data block to the correct one. Traditional checkers, such as e2fsck, have to rescan all of the inodes and indirect blocks encountered by that point to do so. Since scanning the inodes and indirect blocks comprises more than 95% of the total checking time, these double scans are very costly.

To accelerate this process, ffsck uses a list of bitmap snapshots, each of which shows which data blocks were allocated by a specific group of inodes. Each group has a predetermined size, allowing ffsck to easily determine which inodes correspond to which snapshot. These snapshots are created cumulatively; when a block is marked in the corresponding snapshot for its inode group, it is marked in all subsequent snapshots. Thus, ffsck can detect a doubly-allocated block if the bit in the current snapshot is already set. It can then find the inode group that first allocated the block by iterating through the snapshots until it finds the one in which the bit was first set. Once it has done so, it only needs to rescan that group to find the inode that first claimed the block, instead of rescanning all of the previous inodes and indirect blocks. This bitmap mechanism can be further optimized by batching multiple doubly-allocated blocks before rescanning.

An example is given in Figure 8. Bitmap snapshot 1 marks blocks 0, 3, and 4 in use, and bitmap snapshot 2 marks blocks 1 and 5 in use. When bitmap snapshot 3 marks blocks 2 and 5, it detects that block 5 has its use bit set, indicating that it has already been allocated. Iterating over its list of snapshots, ffsck will discover that snapshot 2 contains the inode that first pointed to the block. Ffsck can then find the inode that first claimed ownership by rescanning the block groups corresponding to snapshot 2.

Ffsck can configure the number of snapshots it takes dynamically based on the system's available memory; the more memory available, the more snapshots that ffsck can create, reducing the number of blocks ffsck has to rescan. Even two snapshots, however, can halve the time spent rescanning, time that will be further reduced by the use of a disk-order scan.

4.3.4 Ffsck Summary

The previous sections provided a detailed overview of ffsck's individual components; we now provide a step-by-step summary of its operation.

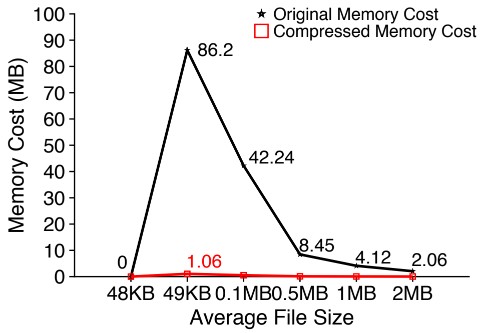

Figure 7: **Memory Cost Analysis.** *This figure compares memory cost of inodes and indirect blocks for 1 GB of data as average file size increases, before and after compression.*

Disk-order scan: First, ffsck scans the disk in ascending-address order. During the scan, ffsck:

1. Reads the inodes in each block group and self checks them, discarding fields unnecessary for the cross-check.
2. Reads the portion of the data bitmap corresponding to the indirect region; for each set bit, read the corresponding indirect block and self check it, again discarding unnecessary metadata.
3. For both inodes and indirect blocks, mark the appropriate bitmap snapshot for each data block allocated. If a data block is allocated twice, record it, its inode, and the conflicting inode group.

Cross-check: Next, ffsck performs a cross check using the data saved in memory. For each file, ffsck:

1. Verifies the total number of blocks.
2. Verifies the file size.
3. Verifies that the checksums in each layer agree.

Correct doubly-allocated blocks: Then, for each doubly allocated data block, ffsck scans the conflicting inode group and chooses to which inode to assign the block, using the same procedure as e2fsck.

Once these steps are completed, corresponding to phase 1 of e2fsck's operation, we then continue with the rest of e2fsck's phases. As these phases do not require much time to execute, we do not change their behavior.

4.3.5 Ffsck Implementation

We base ffsck on e2fsprog1.41.12, adding 2448 lines of source code to it. Most of these modifications occur in pass1.c, which implements the phase one scan and pass1b.c, which rescans all of the inodes and their corresponding indirect blocks to find which inodes have claimed doubly-allocated blocks.

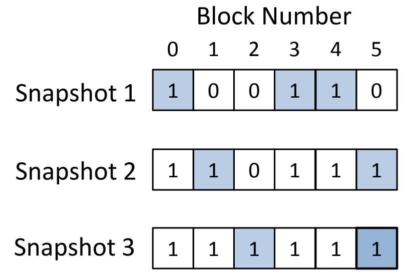

Figure 8: **Bitmap Snapshot Illustration.** *This figure shows an example of three bitmap snapshots taken during a file-system scan. A shaded bit indicates that the corresponding block was allocated in the scan of the inode group for that snapshot. The bold outline around block five in snapshot three indicates that that block has been allocated twice—once in snapshot two and once in snapshot three—and needs to be repaired.*

5 Evaluation

In this section we evaluate our prototype in three aspects: we compare the performance of e2fsck in the ext3 file system with that of ffsck in the rext3 file system, we compare the correctness of ffsck to that of e2fsck, and we measure the two file systems' relative performance. We execute our experiments in the environment described in Section 3.2 and employ the same techniques for creating and initializing file system images described there. In general, we demonstrate that rext3 provides comparable performance to ext3 during ordinary use, while allowing ffsck to operate significantly faster than e2fsck during recovery.

5.1 Checker Performance

To measure the relative performance of e2fsck and ffsck, we evaluate the time it takes to run them on different file system images of varying size and age. We create all of the images using the techniques described in Section 3.2. We generated none of these images with errors; thus, these results represent the best-case scenario.

We first compare e2fsck and ffsck by executing them on a series of increasingly large file-system images. Because the number of inodes and indirect blocks has the greatest impact on the checker's performance, we focus on file system images with large numbers of files, rather than those with complex directory trees. Figure 9 displays our results for file systems with 150-600 GB of data; the x-axis indicates the amount of data in the file system, and the y-axis indicates the checker's execution time. While e2fsck's total execution time grows with the size of the file system, ffsck's time remains roughly constant.

Next, we compare e2fsck's performance with that of ffsck on a series of aged, 750 GB file-system images, showing our results in Figure 10. The x-axis represents how many aging operations are performed on each block

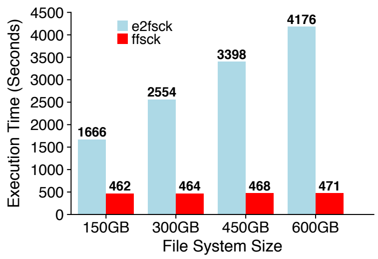

Figure 9: **E2fsck and ffsck Execution Time.** *Total running time of e2fsck and ffsck as the file-system size increases.*

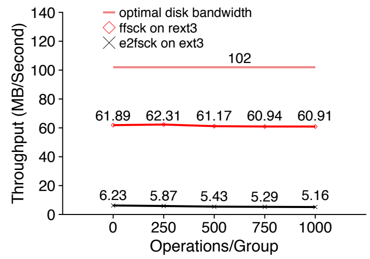

Figure 10: **E2fsck and ffsck Throughput Under Aging.** *Rate at which e2fsck and ffsck scan a 750 GB file system as the file system ages.*

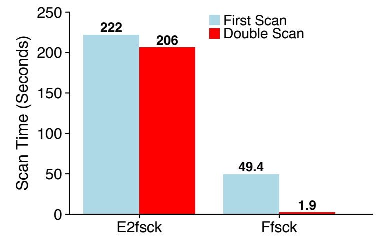

Figure 11: **Double Scan Execution Time.** *Total running time of e2fsck and ffsck in the presence of a doubly-allocated block.*

Metadata type	Corrupted fields
Superblock	magicNum
	inodeSize
	inodeNum
	blockNum
Group Descriptor	inodeBitmapAdd
	blockBitmapAdd
	inodeBitmapAdd
Inode	type
	linkCount
	blockNumber
	fileSize
	State
Indirect blocks	pointerRange
	blockNumber
	multiply-allocated block

Table 2: **Robustness Test.** *This table lists the types of corrupted metadata we use to test both checkers.*

group, and the y-axis depicts the rate at which the two programs process data (i.e., the total data read divided by total checking time). While e2fsck's throughput achieves roughly 5-6% of the disk bandwidth, our new checker uses roughly 61% of the hard disk bandwidth, nearly 10 times faster than e2fsck.

Moreover, ffsck scans the file system at a consistent speed regardless of the age of the file system. This occurs because rext3 co-locates its dynamically allocated metadata, allowing it to scan its metadata in disk-order and thus minimize both the number of disk head movements and the average seek distance.

5.2 Checker Correctness

We also test ffsck's robustness by injecting the 15 metadata corruptions listed in Table 2. We use one case for each superblock and group descriptor error, as our changes do not affect the validation of these objects. For each inode and indirect block error, we use three cases, injecting between one and three errors. In every circumstance, both e2fsck and ffsck detect and repair all errors, showing that ffsck provides the same basic correctness guarantees as e2fsck.

To see how ffsck performs in the most expensive error case, we test the performance of both checkers when they encounter doubly-allocated blocks, using a full 50 GB file system. We configure ffsck to create a bitmap snapshot for every ten block groups; since the file-system image is 50 GB, one snapshot occupies less than 2 MB. In our experiment, we create a single doubly-allocated block by randomly setting two block pointers in different files to point to the same data block. Figure 11 shows our results. When the doubly-allocated block is detected, ffsck only has to recheck the inode tables and indirect regions of 10 block groups, whereas e2fsck has to rescan the entire system. Thus, ffsck repairs the error much more quickly.

5.3 File System Performance

The main difference between ext3 and rext3 is that rext3 co-locates all the indirect blocks in each block group on disk, rather than placing them near their data. Despite this, rext3 achieves similar results to ext3's continuous allocation mechanism through better utilization of the disk track buffer. This section compares the performance of rext3 and ext3 and analyzes their differences.

Figure 12 compares the performance of ext3 and rext3 when reading files sequentially. The x-axis represents the target file size, ranging from 10 KB to 1 GB, and the y-axis depicts the average read throughput over a minimum of five runs. The difference between the two file systems is less than 3% in all cases except when the target file size is 100 KB. In this case, rext3 is 8.4% slower than ext3.

This discrepancy occurs because ext3 stores indirect blocks and data blocks contiguously on disk. Thus, there is no seek operation between an indirect block and its corresponding data blocks during sequential reads. In contrast, since rext3 places these blocks in different areas, an additional seek has to be performed between references to an indirect block and its data blocks.

While this seems like it could cause significant overhead, modern disks generally buffer entire tracks, as described in Section 4.2.1, causing most indirect blocks for

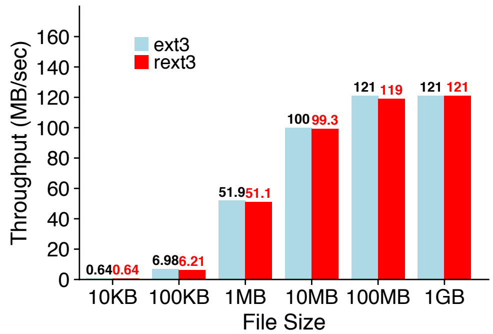

Figure 12: **Sequential Read Throughput.** *Sequential read throughput for a file of varying size.*

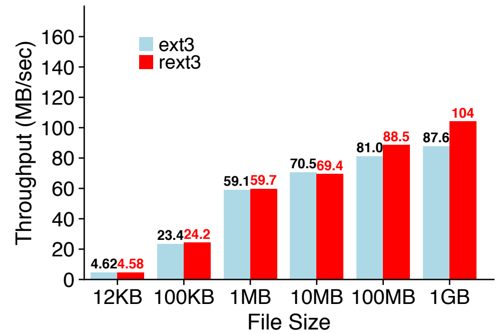

Figure 13: **Sequential Write Throughput.** *Sequential write throughput for a file of varying size.*

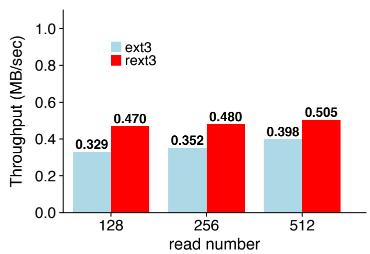

Figure 14: **Random Read Throughput.** *Throughput when performing varying numbers of random 4KB reads from a 2GB file.*

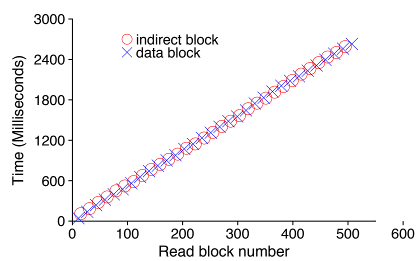

Figure 15: **Cumulative ext3 Strided Read Time.** *Cumulative time spent reading different block types in ext3 for the strided read benchmark.*

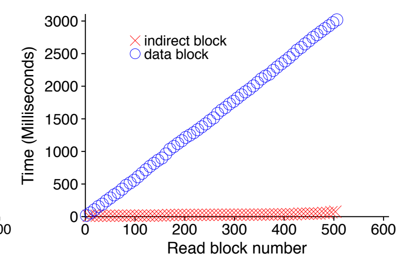

Figure 16: **Cumulative rext3 Strided Read Time.** *Cumulative time spent reading different block types in rext3 for the strided read benchmark.*

a file to be buffered in the disk cache after the first is read. Since the subsequent accesses to the indirect region will be then returned from the disk cache, the seek penalty becomes negligible for larger files, allowing rext3 to perform virtually identically to ext3. However, ext3 performs better when the target file is 100 KB, as the file has only one indirect block and a few data blocks, causing the extra seek time to materialize.

Figure 13 compares the sequential write performance of rext3 and ext3. The x-axis indicates the file size and the y-axis depicts the average access speed. Both systems perform almost identically for small files ranging between 12 KB and 10 MB. However, when the file size increases to 100 MB and 1 GB, rext3 demonstrates a 9.3% and 19% improvement, respectively.

This occurs because the indirect region aids ext3's ordered journaling mechanism. By default, ext3 checkpoints its journal every five seconds, causing a large write to be checkpointed several times before it completes. During these long writes, the file system performs interleaving I/Os from data blocks that are being flushed to disk and metadata blocks that are being checkpointed. Because it has to seek to write each indirect block, ext3 performs primarily random I/O when checkpointing; in contrast, rext3 can write the indirect blocks sequentially, improving overall write performance.

The benefits of the indirect region appear more readily when analyzing random read performance. To do so, we randomly read one 4 KB block from a 2 GB file multiple times and record the observed throughput. We show our results in Figure 14. The x-axis indicates the number of times we read from the file, and the y-axis shows the average read throughput over a minimum of five runs of the experiment in MB/s. The rext3 file system sees 43%, 39%, and 27% higher throughput for 128, 256, and 512 reads, respectively, a significant difference.

Since the main difference between rext3 and ext3 is the disk layout, we examine the performance of rext3 without the influence of the file system cache. To do so, we design a test using strided reads, which iterate over all the indirect blocks and periodically read a random data block to which the current indirect block points.

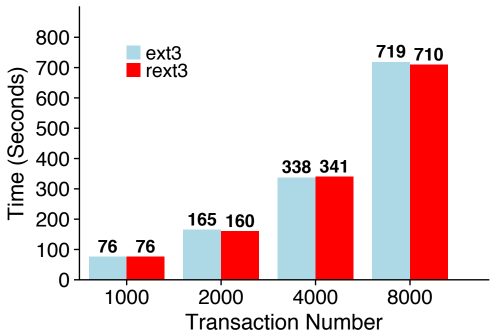

Figure 17: **Postmark Performance.** *Postmark Performance with ext3 and rext3.*

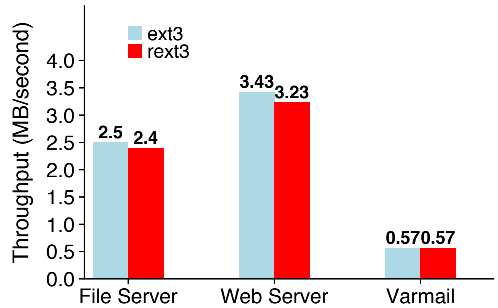

Figure 18: **Filebench Performance.** *Performance of fileserver, webserver and varmail macrobenchmarks with ext3 and rext3.*

Figures 15 and 16 display the cumulative time spent reading indirect blocks and data blocks during the strided read test, showing that rext3 outperforms ext3 by 68% in this experiment. This occurs because the first access to the indirect region caches it in the disk buffer, which is significantly faster than the disk platter. The rext3 file system will thus spend much less time fetching indirect blocks, doubling its read speed.

Finally, we use Postmark and Filebench as macrobenchmarks to compare rext3 and ext3. We use the default settings for Filebench and invoke Postmark with 5000 files between 4 KB and 4 MB in size, placed in 50 subdirectories, with 50/50 read/append and create/delete biases. Figures 17 and 18 show our results. In most cases, rext3 performs nearly identically to ext3, except when large numbers of small files are involved. In this case, rext3 performs 5% worse than ext3.

Given these performance measurements, we can conclude that rext3 performs competitively with ext3 in most cases, exceeding ext3 in its ability to handle random reads and large writes, and performing slightly worse when handling small reads.

6 Limitations

While rext3 and ffsck provide a number of substantial improvements over ext3 and fsck, they are not without some drawbacks that may make them unsuited for certain deployments. This section briefly discusses some of the areas where problems may occur or performance may suffer and some potential solutions for these problems.

As mentioned earlier, ffsck relies on the data bitmap to determine which indirect blocks are in use when performing its scan. Thus, corruption in the data bitmap can potentially cause ffsck to miss indirect blocks. While we construct checksums to guard against this kind of error, a mismatch in checksums will cause ffsck to revert to the behavior of traditional fsck, causing performance to suffer (though this will still benefit from metadata co-location, as demonstrated in the ext3 metaclustering patch [32]).

More significantly, our current checksum is simple and may, in some cases, miss certain errors due to collisions. In future work, this can likely be addressed by using a more sophisticated checksum for each file; many cryptographic hash functions provide strong enough guarantees to make the possibility of collision negligible. Doing so will consume additional memory during the scan, as the checksum will be larger, and may require backpointers to point to their immediate parent, rather than the inode; these changes should, however, be feasible.

Memory consumption comprises the other potential drawback of our checker. In the worst case, discussed in Section 4.3.2, metadata compression will not scale to very large file systems with tens or hundreds of terabytes of data, as this will require tens or hundreds of gigabytes of memory. This worst case behavior is, however, somewhat unusual, relying on large numbers of files with sizes near 49 KB, which is unlikely to arise in practice. In addition, this memory pressure could be partially alleviated by performing the cross-check on a file-by-file basis (rather than upon completion of the scan) and discarding the metadata for each file once all checks are complete for it; we have yet to evaluate this approach, however.

Metadata seek times comprise the largest barrier to rext3's efficient operation. While the disk track buffer mitigates nearly all of this, its effectiveness can be thwarted by fragmentation in the indirect region, causing seeks to distant block groups to fetch metadata for local files, or simply by disks that do not possess this technology. Fortunately, however, we have yet to observe the former in practice and the latter are likely to be rare.

The other potential concern with rext3 is the need to choose the size of the indirect region correctly, as described in Section 4.2.3; such misconfiguration could lead either to wasted space or premature exhaustion of indirect blocks for the file system. Future implementations of our file system may be able to allocate new indirect regions from unused groups of data blocks. This approach may warrant investigation if configuration proves difficult.

Finally, our techniques are most directly applicable to block-based file systems, such as ext3 and FFS. While it may be possible to apply them to extent based file systems, like ext4 and XFS, we have yet to analyze this in detail. Furthermore, they may be challenging to implement in copy-on-write file-systems, like btrfs, without adopting split partitions for data and metadata, as in DualFS [31].

7 Conclusion

While the file-system checker is ultimately the only mechanism that can repair all types of file-system damage, its design has been neglected since its first iteration. In some ways, this makes sense: in order to provide correctness, checkers have to examine all of the metadata in the file system, a process that will necessarily be slow. However, the layout of current file systems frequently makes this process excruciating due to the numerous random seeks needed to logically traverse the metadata tree. Furthermore, the erratic growth of this tree, which scatters indirect blocks throughout the disk, makes it virtually impossible to accurately estimate the run time of a given file system check after the file system has aged. Simply put, if your scan is tree-ordered, it will run slowly, and worse, unpredictably slowly.

To solve this problem, we place the correct, fast handling of file-system inconsistencies at the heart of the design of a new file system, rext3, a slight variant of the Linux ext3 file system. We design rext3 to explicitly support a fast file-system checker by co-locating all the indirect blocks in each block group into a single location per block group and by using backpointers to inodes in indirect and data blocks to eliminate the need to logically traverse the metadata tree during ordinary operation. In addition to this new file system, we build ffsck, a file-system checker capable of providing near optimal performance by scanning metadata in disk-order, rather than logically. While doing so could potentially exert substantial pressure on memory, as ffsck has to hold the entire contents of metadata in memory, it mitigates this by using a two-stage checking process that allows it to discard metadata it no longer needs, providing substantial compression. Finally, ffsck provides further optimizations over current checkers by using bitmap snapshots to track which data blocks have already been allocated, removing the need for a full rescan when it encounters already allocated blocks.

These innovations result in major improvements to checking behavior without sacrificing ordinary case file-system performance. During execution, ffsck manages to read metadata at rates nearing the sequential peak of disk bandwidth, operating 10 times faster than e2fsck in the optimal case, and scaling with sequential disk performance; further, it no longer suffers from file-system aging, allowing better prediction of time to completion.

The underlying file system, rext3, maintains performance competitive with ext3 in most cases, incurring only a small penalty of less than ten percent when dealing with files around 100 KB in size. However, rext3 actually outperforms ext3 by up to 20% for large sequential writes and up to 43% for random reads by facilitating journal checkpointing through metadata locality and by using the disk track buffer more efficiently.

While there are powerful reasons for a checker to be developed independently from the mainline file system it checks (i.e., in order to avoid making the same mistakes in each), some cooperation can be worthwhile. Such co-design may thus be worth considering in other domains.

Acknowledgments

We thank the anonymous reviewers and Dan Tsafrir (our shepherd) for their feedback and comments, which have substantially improved the content and presentation of this paper. We also thank the members of the ADSL research group, in particular Yupu Zhang, Yiying Zhang, and Vijay Chidambaram for their insight. The authors would also like to thank the Advanced Development Group and WAFL team at NetApp, especially Lakshmi Bairavasundaram, Minglong Shao, Prashanth Radhakrishnan, Shankar Pasupathy, and Yasuhiro Endo for their feedback during the early stage of this project, Yanpei Chen for discussing the disk-order scan, and Xiaoxuan Meng for suggestions on the kernel implementation.

This material is based upon work supported by the National Science Foundation under the following grants: CNS-1218405, CCF-0937959, CSR-1017518, CCF-1016924, as well as generous support from NetApp, EMC, and Google. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] D. Anderson, J. Dykes, and E. Riedel. More Than an Interface: SCSI vs. ATA. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[2] Anonymous. How long does it take fsck to run?! http://www.jaguarpc.com/forums/vps-dedicated/14217-how-long-does-take-fsck-run.html, 2006.

[3] Anonymous. e2fsck is taking forever. http://gparted-forum.surf4.info/viewtopic.php?id=13613, 2009.

[4] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, California, June 2007.

[5] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, pages 223–238, San Jose, California, February 2008.

[6] W. Bartlett and L. Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[7] R. Baumann. Soft errors in advanced computer systems. *IEEE Des. Test*, 22(3):258–266, 2005.

[8] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

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

[10] D. Bovet and M. Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[11] J. C. M. Carreira, R. Rodriguez, G. Candea, and R. Majumdar. Scalable Testing of File System Checkers. In *Proceedings of the EuroSys Conference (EuroSys '12)*, Bern, Switzerland, April 2012.

[12] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency Without Ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[13] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 57–72, Banff, Canada, October 2001.

[14] D. Engler and M. Musuvathi. Static Analysis versus Software Model Checking for Bug Finding. In *5th International Conference Verification, Model Checking and Abstract Interpretation (VMCAI '04)*, Venice, Italy, January 2004.

[15] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown. Recon: Verifying File Ssytem Consistency at Runtime. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, California, February 2012.

[16] R. Funk. fsck / xfs. http://lwn.net/Articles/226851/.

[17] G. R. Ganger and Y. 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.

[18] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. SQCK: A Declarative File System Checker. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI '08)*, San Diego, California, December 2008.

[19] R. 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.

[20] V. Henson, A. van de Ven, A. Gud, and Z. 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.

[21] D. Hitz, J. Lau, and M. 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.

[22] K. Keeton and J. Wilkes. Automating data dependability. In *Proceedings of the 10th ACM-SIGOPS European Workshop*, pages 93–100, Saint-Emilion, France, September 2002.

[23] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Ottowa Linux Symposium*, 2007.

[24] T. C. May and M. H. Woods. Alpha-particle-induced Soft Errors in Dynamic Memories. *IEEE Transactions on Electron Devices*, 26(1), 1979.

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

[26] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[27] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[28] NetApp. Overview of wafl_check. http://uadmin.nl/init/?p=900, September 2011.

[29] D. Patterson, G. Gibson, and R. 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.

[30] J. K. Peacock, A. Kamaraju, and S. 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.

[31] J. Piernas, T. Cortes, and J. M. Garcia. The Design of New Journaling File Systems: The DualFS Case. *Computers, IEEE Transactions on*, 56(2):267 –281, February 2007.

[32] A. Rai. Make ext3 fsck way faster [2.6.23.13]. http://lwn.net/Articles/264970/, January 2008.

[33] H. Reiser. ReiserFS. www.namesys.com, 2004.

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

[35] T. Sato. ext4 online defragmentation. http://ols.fedoraproject.org/OLS/Reprints-2007/sato-Reprint.pdf, 2007.

[36] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM Errors in the Wild: A Large-scale Field Study. In *Proceedings of the 2009 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance '09)*, Seattle, Washington, June 2009.

[37] T. J. Schwarz, Q. Xin, E. L. Miller, D. D. Long, A. Hospodor, and S. Ng. Disk Scrubbing in Large Archival Storage Systems. In *Proceedings of the 12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Volendam, Netherlands, October 2004.

[38] S. Sinofsky. Building the Next Generation File System for Windows: ReFS. http://blogs.msdn.com/b/b8/archive/2012/01/16/building-the-next-generation-file-system-for-windows-refs.aspx, January 2012.

[39] K. Smith and M. I. Seltzer. File System Aging. In *Proceedings of the 1997 Sigmetrics Conference*, Seattle, WA, June 1997.

[40] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying File System Protection. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

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

[42] R. Sundaram. The Private Lives of Disk Drives. http://partners.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html.

[43] V. Svanberg. Fsck takes too long on multiply-claimed blocks. http://old.nabble.com/Fsck-takes-too-long-on-multiply-claimed-blocks-td21972943.html, 2009.

[44] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[45] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[46] The Data Clinic. Hard Disk Failure. http://www.dataclinic.co.uk/hard-disk-failures.htm.

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

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

[49] J. Yang, C. Sar, and D. 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] J. Yang, P. Twohey, D. Engler, and M. 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.

[52] J. F. Ziegler and W. A. Lanford. Effect of Cosmic Rays on Computer Memories. *Science*, 206(4420):776–788, 1979.