lwFSCK: Light-weight Filesystem Check

Juwon Kim
KAIST
Republic of Korea

Dongeon Kim*
Presto Labs
Republic of Korea

Seungwon Yoo
KAIST
Republic of Korea

Myeongin Cheon
KAIST
Republic of Korea

Joontaek Oh*
University of Wisconsin-Madison
USA

Youjip Won
KAIST
Republic of Korea

ABSTRACT

The existing filesystem check and repair process (FSCK) requires tens of minutes or even hours with hundreds GByte memory to complete on petabyte-scale filesystems. To address this issue, we propose *lwFSCK*, a light-weight FSCK tool that can minimize the FSCK execution time and memory footprint. Our approach, called lwFSCK consists of two key technical ingredients: *Per-Thread Data Processing* and *TwinTree*. First, Per-Thread Data Processing partitions the set of metadata that needs to be examined into multiple partitions and allocates each of them to a separate thread which has its own per-thread data structure without lock contention. Second, lwFSCK introduces an adaptive data structure called TwinTree. TwinTree reduces the memory footprint that is required to hold the temporary information for filesystem check. lwFSCK reduces execution time by up to 1.4× compared to the state-of-the-art FSCK, pFSCK. lwFSCK reduces memory consumption by 6.8× in file-intensive filesystems that use only 1% of inodes.

CCS CONCEPTS

• **Software and its engineering → Filesystem Check and Repair Tool**.

KEYWORDS

Filesystem Check and Repair tool, Filesystem, Crash consistency

ACM Reference Format:
Juwon Kim, Dongeon Kim, Seungwon Yoo, Myeongin Cheon, Joontaek Oh, and Youjip Won. 2025. lwFSCK: Light-weight Filesystem Check. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25), March 31-April 4, 2025, Catania, Italy*. ACM, Sicily, Italy, 9 pages. https://doi.org/10.1145/3672608.3707806

1 INTRODUCTION

Crash consistency is paramount for ensuring reliability and data integrity in filesystems [7–9, 12, 19, 20, 25, 26, 45]. Filesystem Check and Repair (Crash and Repair) [14, 17, 18, 27, 30] tool is a widely adopted program that ensures the crash consistency of the filesystem. It traverses the entire filesystem metadata blocks such as inodes, directory entries, and bitmaps, verifies the consistency of the metadata blocks and corrects any detected errors. Representative filesystem Crash and Repair tools include e2fsck, a Crash and Repair tool for the Ext filesystem, and xfs_repair for the XFS filesystem [1, 6, 22, 37].

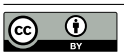
While the filesystem Check and Repair tools provide strong filesystem consistency, they take a huge amount of time on a large-scale storage. Since it traverses the entire filesystem, the latency of its filesystem verification process increases linearly as the number of files and directories or the filesystem size increases [15, 18]. Depending on the size of the filesystem, the total execution time can take from a few hours to a few weeks.

To mitigate the overhead of the filesystem Crash and Repair, there have been several research projects that propose to perform verification in parallel [4, 16, 21, 23, 28, 34]. Recently proposed pFSCK is a filesystem Crash and Repair tool that utilizes the CPU parallelization and the high bandwidth of recent storage devices [16]. In pFSCK, the multiple threads verify the consistency of the inode blocks and the directory blocks in parallel. pFSCK divides the filesystem verification process into several steps, and run each step in a pipelined manner, to avoid unnecessary waiting for the completion of previous passes. pFSCK has the advantage of applying parallelization to fine-grained units and being compatible with the existing filesystem without modification of the on-disk layout.

From our observation, the existing state-of-the-art work, pFSCK, has two limitations. First, pFSCK is not fully many-core scalable. pFSCK uses shared global data structures and global locks. We observe that pFSCK exhibits a limited performance due to its extensive use of the locks: the locks for bitmaps that maintain information about (i) the used inodes (ii) the used directories (iii) the file inodes (iv) the allocated blocks and (v) reference count of each inode. Second, pFSCK does not consider the memory pressure. The existing filesystem Check and Repair tools including pFSCK consume a huge amount of memory to maintain the information of the entire filesystem metadata blocks. The reason behind the huge memory footprint is that the existing FSCKs unnecessarily allocate memory for all metadata blocks, regardless of whether the metadata blocks are in use or not. This results in a significant amount of memory being wasted when only a small fraction of metadata blocks are in use. This memory overhead becomes severe as the storage capacity increases. For example, the existing filesystem Check and Repair tool renders 500 GByte memory footprint on the 1 PByte storage device.

To address this issue, we present a light-weight FSCK, lwFSCK. lwFSCK is redesigned to minimize the use of the locks and maximize the benefits of the existing FSCK optimization techniques, i.e. parallel filesystem scanning [3, 16] and the pipeline processing

*This work was done while the authors were graduate students at KAIST.



SAC '25, March 31-April 4, 2025, Catania, Italy

ACM ISBN 979-8-4007-0629-5/25/03.
https://doi.org/10.1145/3672608.3707806

[16]. lwFSCK consists of two technical ingredients, *per-thread data processing* and *TwinTree*. In per-thread data processing, we carefully redesign the filesystem scanning process, to make the process fully many-core scalable. lwFSCK maintains the data structures used for filesystem scanning in a per-thread manner, to get rid of any lock contention. In TwinTree, we design the data structure which is specialized to space-efficiently record the reference count of the inode. TwinTree reduces memory footprint by maintaining only in-use inodes' reference counts. TwinTree maintains two different data structures; one is for inodes with a single reference count, and the other is for inodes with multiple reference count. TwinTree adaptively uses one of those data structures to minimize the memory footprint.

By redesigning the filesystem Check and Repair tool to be fully many-core scalable, lwFSCK reduces the total execution time of FSCK by up to 1.4 × compared to pFSCK. Through the TwinTree data structure, lwFSCK reduces memory footprint by up to 6.8 × compared to the pFSCK.

2 BACKGROUND

2.1 Filesystem On-disk Layout

Filesystems abstract filesystem blocks into a hierarchy of files and directories. To support this abstraction, filesystems maintain various types of metadata. We categorize the filesystem metadata into two types: filesystem metadata and file metadata.

Filesystem Metadata. Filesystem metadata maintains the overall information of the filesystem. Filesystem metadata includes superblock and block allocation [41]. Superblock stores information such as the filesystem version, the filesystem partition size, the numbers of inodes in use and free inodes, the number of blocks in use, and the lists of orphan inodes and free blocks. The metadata for block allocation usually uses bitmaps [2, 32] or B+Trees [5] to track the allocation status (free or not) of each filesystem block.

File Metadata. File metadata maintains the information associated with individual files and the connectivity between files. We divide the file metadata into three categories. First is the metadata for individual files (or directories), such as type, timestamp, permissions, etc. UNIX-like filesystems, such as ext4 [15], XFS [22], and F2FS [32], employ *inode* to manage file's metadata. An inode block is a filesystem block dedicated to storing the file's metadata, such as file type, timestamp and several other information. Each file has its own inode block. Second is the file mapping. File mapping is metadata that maintains the location of data blocks of a file. The UNIX-like filesystem maintains file mapping in the form of extent [2], indirect mapping [2, 32], and B+Trees [5]. Third is the directory entry. The directory entry contains the file information, e.g. name, and location, included in the directory. The existing filesystems maintain directory entry by using lists [2] or hash tables [2, 32].

Filesystem Consistency. Filesystem consistency refers to ensuring a set of filesystem invariants [41]. When these invariants are violated, it can lead to problems like data corruption, security vulnerabilities, or file loss. For example, a filesystem invariant might require that every directory entry must point to an initialized inode. Filesystems like ext4 perform consistency checks during the FSCK process, verifying various aspects listed in Table 1.

Table 1: Checklist of e2fsck for each pass [21]

Pass #	Object	checklist
1	Inode	timestamp, file type, permission, block pointer. two inodes must not share the same block.
2	Directory	'.' and '..' point to itself and parent. each directory entry must point to a valid inode.
3	Connectivity	each directory must be connected to the root
4	Reference Count	reference count of file (or directory) inode must be equal to the # of hard links (or subdirectory entries)
5	Bitmap	fix inode and block bitmap

2.2 fsck: The Filesystem Check/Repair Tools

Numerous techniques have been proposed to ensure filesystem consistency. They include filesystem check and repair tools [1, 6, 34], journaling [15, 22, 43], copy-on-Write [32, 40], and soft-update [35]. While the journaling, copy-on-write, and soft-update are effective in maintaining filesystem consistency after a crash, they are insufficient for handling errors such as disk failures. In such cases, only check and repair (C/R) tools can be employed [10, 11, 29, 31, 38, 39, 42]. In environments like data centers, where hundreds of thousands of disks are in use, disk failures frequently occur [44] so the C/R tools are essential to guarantee a filesystem consistency from the disk failures.

In this work, we focus on one of the widely deployed C/R tool, e2fsck, that services for ext2, ext3, and ext4 filesystems. e2fsck runs total 5 passes as shown in Table 1.

Check and Repair in e2fsck. e2FSCK consists of five passes in total. Pass 1 traverses the on-disk inode table and checks the consistency of in-use inodes. For each inode, e2FSCK checks the correctness of timestamp, size, file type, permission, and several other attributes. Pass 2 traverses all the directory blocks and checks the consistency of the directory entries in each directory block. It checks the conflict between the length of the directory entry and the length of the corresponding file name. Pass 3 checks whether each directory can traverse to its upper directory via '..' entry. Pass 4 examines reference counts of file inode blocks and directory inode blocks. The reference count of a file inode indicates the number of hard links, including itself. The reference count in a directory inode is the count of subdirectory entries, including itself and the parent directory. Pass 4 counts the hard links of each file and subdirectory entries of each directory by traversing entire filesystem. Then it compares them with the reference count in the on-disk inodes. Pass 5 checks whether inode bitmap and block bitmap are correctly indicating in-use inodes and in-use data blocks, respectively.

Inter-Pass Serial Execution. Since each pass requires the results of the previous pass, e2fsck executes its five passes in a serial manner. For example, after the pass 1 checks the validity of each inode, pass 2 checks whether each directory entry points to a valid inode. After pass 2 verifies that all directory entries are correctly pointing valid inode, the pass 3 verifies whether each directory can reach the root directory by using '..' entry. Pass 4 sets the reference count for each inode based on the number of directory entries that reference it, which also requires the completion of pass 2. Finally, pass 5 checks the validity of the on-disk bitmap, which represents the allocation status of inodes and data blocks. The allocation status

of each inode and block is verified during pass 1, so pass 5 must also be executed after pass 1.

Intra-Pass Serial Execution. Current e2fsck is designed to run each pass in a serial manner with a single thread. In e2FSCK, there are lots of global in-memory and on-disk data structures that are referred to or updated during each pass. For example, when updating the validity of inodes during pass 1, the superblock and inode bitmap are also updated. The superblock records the number of inodes in use, and the inode bitmap sets the corresponding bit for each inode that is in use. These global data structures introduce significant lock contention, preventing e2fsck from being executed with multiple threads.

Memory Pressure. e2FSCK is designed to minimize disk I/Os by maintaining large in-memory data structure. Once e2FSCK reads on-disk metadata such as inode table and bitmap, e2FSCK keeps the information that will be potentially necessary for the subsequent passes on the memory. This is to avoid multiple disk read on the same metadata block. For example, when the pass 2 scans all directory block, it calculates reference counts of each inode pointed by the directory blocks. Then, e2FSCK maintains in-memory array of the [inode number, reference count]. This array is used in pass 4 that performs reference count check. The memory overhead of e2FSCK becomes severe as the storage capacity increases since e2FSCK maintains in-memory data structure to store the information of the entire filesystem blocks.

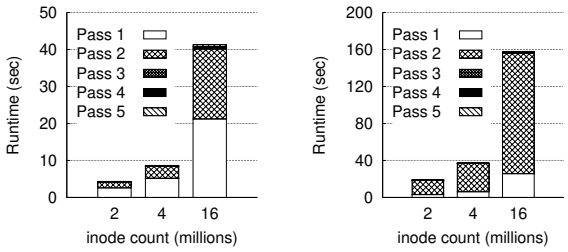
2.3 Scalability of fsck

As the size of a single filesystem reaches several terabytes, the execution time for fsck dramatically increases. Since the system that has crashed is likely to crash again in the near future, minimizing the fsck execution time is crucial. There have been numerous research projects [3, 13, 16, 17, 21, 24, 33, 34] to reduce the fsck execution time. They include exploiting (i) physical isolation, (ii) online check, (iii) data parallelism, and (iv) pipeline parallelism.

Physical Isolation. IceFS [33] and ChunkFS [24] divide the filesystem partition into multiple small, individually repairable fault-isolation domains [24]. With the physical isolation approach, the filesystem can be recovered in a fine-grained manner, minimizing the execution time of FSCK. However, this approach disables cross-domain references [33] and requires significant modification to the on-disk layout of the filesystem[24, 33].

Online Check. NoFS [13] and Recon [17] perform consistency checks while serving the file and directory abstraction. NoFS checks the consistency of only part of the partition and verifies the rest in the background [13]. Recon [17] verifies whether the journal transactions maintain filesystem invariants before committing them to the journal. However, NoFS requires modifying the on-disk layout [13], and Recon [17] must be used with journaling.

Data parallelsim. Wang et al. [3] and pFSCK [16] divide the inodes to be checked in pass 1 across multiple threads, enabling them to be checked in parallel. However, pFSCK shares five key data structures across all threads, limiting its scalability: (i) the bitmap that tracks in-use inodes, (ii) the bitmap that tracks directory inodes, (iii) the bitmap that tracks file inodes, (iv) the bitmap that tracks used logical blocks, and (v) the bitmap that tracks the inode reference counts.



(a) File-intensive scenario (b) Dir-intensive scenario

Figure 1: Execution time breakdown of e2fsck

Wang et al. allocate these five data structures per thread and merge them after pass 1. They also introduce per-thread data structures, except for certain data structures such as the disk read/write lock and block bitmap. However, Wang et al.'s approach introduces per-thread data structures only for pass 1, which limits its scalability.

Pipeline Parallelism. pFSCK pipelines pass 1 and pass 2 to enhance performance. pFSCK has two set of threads; one is for pass 1 and the other is for pass 2. When each thread for pass 1 scans the inode and finds a directory, it immediately delivers the directory block to the thread for pass 2. The thread for pass 2 checks the directory consistency in parallel with pass 1.

3 MOTIVATION

We evaluate two representative filesystem check and repair tools, e2FSCK [1] and pFSCK [16], and obtain three key observations.

3.1 e2FSCK

We examine the execution time of each pass in e2FSCK. We set the filesystem partition to 256 GByte. We vary the number of files and directories to 2, 4, and 16 million, which account for 12%, 24%, and 96% of total inodes in the filesystem, respectively. We evaluate two scenarios: dir-intensive case and file-intensive case. In the dir-intensive case, the ratio of file and directory is 50:50. In the file-intensive case, the ratio of file and directory is 99:1. The details of the experiment setting are shown in Sec. 5.

Fig. 1(a) and Fig. 1(b) depict the execution time of each pass in the file-intensive case and dir-intensive case, respectively. The total execution time of e2fsck increases with the number of in-use inodes. It is notable that the sum of the execution times of pass 1 and pass 2 accounts for more than 94% of the total execution time in both file-intensive and dir-intensive scenarios. Since there are more directories in the dir-intensive scenario, the execution time of pass 2 that checks directory consistency is significantly longer than that of file-intensive case.

Observation #1. Pass 1 and 2 account for most of execution time of filesystem check and repair. Thus, optimizing pass 1 and pass 2 is a key to reduce overall execution time of the filesystem check and repair. According to our analysis, Passes 1 and 2 involve disk read/write operations, while Passes 3, 4, and 5 involve significantly less disk I/O. In Pass 1, all inodes and file mappings must be read from the disk. In Pass 2, all directory blocks are read from the disk. During passes 1 and 2, e2fsck do the following with read inodes and

file mappings; (i) reconstruct the inode bitmap and block bitmap, (ii) record the reference count for each inode, and (iii) create a directory list. Thus, Pass 3 can minimize disk reads by utilizing the directory list created in Passes 1 and 2. Similarly, Pass 4 can reduce disk reads by using the [inode, reference count] pairs generated in Passes 1 and 2. Finally, Pass 5 relies on the bitmaps generated in Passes 1 and 2. Additionally, since the number of blocks occupied by the bitmaps is relatively small, disk I/O in this pass is minimal.

3.2 Manycore Scalability of Parellel FSCK

We examine pFSCK, the advanced e2FSCK that exploits multi-thread parallelism. To explore the manycore scalability of pFSCK. we run pFSCK with varying the number of threads on a 256 GByte filesystem partition. The evaluation runs with 16 million files and directories. We run file-intensive case and dir-intensive case, where the ratio of file and directory is 99:1 and 50:50, respectively. The detail of the experiment setting is shown in Sec. 5.

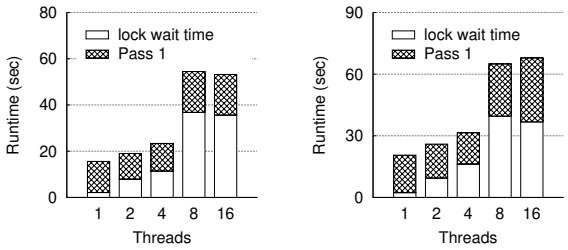
According to the result of the evaluation, pFSCK performance decreases as the number of threads increases. The root cause behind this is lock contention during pass 1. The pass 1 consists of two phases. In the first phase, the multiple threads check inode consistency in parallel. In the second phase, multiple threads collect inodes' information which will be used in subsequent passes. The inode information includes the type (file or directory) and the state (in-use or not) of the inode. pFSCK maintains global data structures to maintain the inode information: (i) the bitmap that tracks in-use inodes, (ii) the bitmap that tracks directory inodes, (iii) the bitmap that tracks file inodes, (iv) the bitmap that tracks used logical blocks, and (v) the bitmap that tracks the inode reference counts. When multiple threads update the global data structures, they suffer from severe lock contention, degrading the scalability of pFSCK. Fig. 2 depicts the execution time of pass 1 and lock wait time for the global data structure. In the file-intensive filesystem with 16 threads (Fig. 2(a)), the lock wait time is about 67% of the execution time of pass 1. In the dir-intensive case with 16 threads (Fig. 2(b)), the lock wait time accounts for about 60% of the execution time of pass 1.

Observation #2. Parallel FSCK is not fully parallel but suffers from lock contention. To fully exploit the manycore parallelism, the existing filesystem check and repair has to be redesigned.

3.3 Memory Consumption of Parallel FSCK

While pipeline parallelism enhances performance, it poses another problem: huge memory footprint. e2FSCK, which runs each pass serially, minimally allocates memory to keep the information necessary for subsequent passes in memory. For example, e2FSCK obtains the number of total in-use inodes when pass 1 finishes and shares this information with pass 2. When pass 2 creates the array of [inode number, reference count], it allocates memory for the array minimally to hold only the in-use inodes.

In pFSCK, however, pass 1 and pass 2 are run simultaneously, so pass 2 can not have the number of in-use inodes. Thus, pFSCK maximally allocates the array with the size of the total number of inodes, to prevent the record count array from being full. When the number of in-use inodes is low, pFSCK significantly wastes memory. When 1% of total inodes are in use in 512 TByte filesystem, for example, 64 GByte of memory is wasted.



(a) File-intensive scenario (b) Dir-intensive scenario

Figure 2: Execution time breakdown of Pass 1 in pFSCK

Observation #3. With pipeline parallelism, pass 2 can not obtain the complete outcome of pass 1. This results in an unnecessarily large memory footprint of pFSCK.

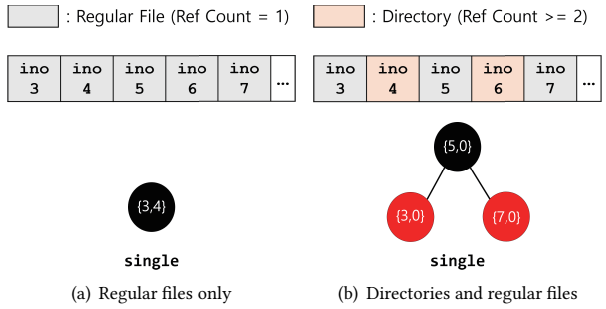
4 LIGHT-WEIGHT FSCK

4.1 An Overview

We present lwFSCK, a light-weight FSCK. The lwFSCK addresses two issues. First, it makes the accesses on the shared data structure in the filesystem check operation manycore scalable. Second, it minimizes the memory footprint of filesystem Check and Repair. The design of lwFSCK consists of two parts. First, lwFSCK adopts fully per-thread data structure to avoid lock contention overhead. Second, to reduce memory consumption, lwFSCK proposes to adaptively use two types of data structures in allocating the memory space for reference counts with respect to the number of inodes in use.

4.2 Per-thread Data Processing

Existing e2FSCK and pFSCK have low scalability due to the lock contention on global data structures. We replace all data structures used during filesystem check and repair to per-thread data structures. we adopt the per-thread data structure as proposed by Wang. et, al [3]. We allocate a set of block groups based on the number of threads and the total number of block groups as in Wang. et al [3]. For the data structure corresponding to the multiple block groups, e.g., superblock, we use delta record which represents the difference rather than the data field itself, as in [3]. With a per-thread data structure, the synchronization cost for global variables can be significantly reduced. Each pass consists of three phases. First, each thread allocates its own per-thread data structures used for the pass before the start of the pass. The per-thread data structures maintain the inode state (in-use or not) and inode type (file or directory). Then, each thread is assigned a portion of the block groups, calculated as the total number of block groups divided by the number of cores. Second, each thread checks the consistency of the metadata within the assigned block groups. During the inspection, the thread inserts inode type and inode state information into its own data structures. The threads do not suffer from lock contention since they have their own data structure. Finally, after the inspection is over, all data structures and the delta record from each thread are merged into the global structures. Compared to global data structures, there is a disadvantage in that additional



(a) Regular files only (b) Directories and regular files

Figure 3: Regular files only vs. Directories and Regular files

merging work is required, but there is no performance degradation due to synchronization through locks, so scalability is sufficiently high. Additionally, each thread executes the next pass even if the current pass is not done, e.g., inspecting the directory block even if the whole pass 1 is not cone as in pFSCK [16].

4.3 TwinTree

TwinTree is a data structure designed to space-efficiently maintain the reference count of inodes. TwinTree consists of two red-black trees, called `single` and `multiple`. `single` is for inodes with a reference count of 1. `multiple` is for inodes with a reference count of more than 1. Both `single` and `multiple` are red-black tree, but their entry structure are designed differently to minimize the memory footprint. The entry of `multiple` simply consists of the inode number and the reference count. The entry of `single` space-efficiently represents the range of consecutive inodes with a single reference count. The entry of `single` consists of the start inode number and the number of consecutive inodes. Since the `single` tree only maintains inodes with a reference count of 1, its entry does not include the reference count in the entry of `single`.

For example, assume that the i-th inode has a reference count of 1 and it is not consecutive with other inode. Then the corresponding `single` entry has inode number of i and number of consecutive inodes of zero. Then, when the $(i+1)$-th inode also has the reference count is 1, the $(i+1)$-th inode will be reflected to the `single`. Instead of creating a new entry and inserting it into the red-black tree, the count of the i-th inode entry is increased by one because the i-th and $(i+1)$-th are consecutive inodes.

Fig. 3(a) depicts the `single` tree when the most of files have a reference count of 1. Contiguous inodes can be represented by a single entry, providing efficiency in terms of memory usage. Plus, the advantage of having a small number of entries results in a faster search in the red-black tree. Fig. 3(b) shows `single` tree when the most of file have a reference count more than two. Since the continuity of inodes breaks, there are multiple entries in the `single` tree.

The way to increase, decrease, and search the reference count using TwinTree is as follows.

INCREMENT. To increase the reference count of the inode by 1, lwFSCK first checks if the inode exists in `single`. If it exists in `single`, it removes the corresponding entry from `single` and inserts a new entry with the inode number and reference count of 2 into `multiple`. If the inode does not exist in `single`, lwFSCK checks if the inode exists in `multiple`. If it exists in `multiple`, it increases the reference count by 1. Else, the inode is referenced firstly so lwFSCK inserts the corresponding inode into `single`.

DECREMENT. To decrease the reference count of the inode by 1, lwFSCK checks if the inode exists in `single`. If so, lwFSCK removes the corresponding entry from `single`. Otherwise, lwFSCK searches the inode in `multiple` and decreases the reference count of the inode by 1. If the reference count becomes 1, lwFSCK deletes the entry from `multiple` and inserts the corresponding inode into `single`. If the target inode does not exist in either `single` or `multiple`, it is an inode with a reference count of 0 so far and it is the first referenced inode. Therefore, the reference count cannot be decreased.

FETCH. To search for the reference count of the target inode, lwFSCK checks if the inode exists in `single`. If it exists, the reference count of the inode is 1. If it does not exist in `single`, lwFSCK checks if the inode exists in `multiple`. If the inode exists in `multiple`, it returns its reference count. If the inode exists in neither `multiple` nor `multiple`, it means the inode has not been referenced, so the reference count is 0.

In TwinTree, every operation such as insertion, search, and deletion has a maximum time complexity of O(log n). While TwinTree is not as fast as using an array with a time complexity of O(1), TwinTree can significantly reduces the memory footprint by space-efficiently maintaining reference counts of only in-use inodes.

4.4 Adaptive Object Allocation

To minimize memory footprint, lwFSCK adaptively switches the data structure to maintain the inode reference count based upon the utilization of the inode. It reduces the memory pressure associated with performing a filesystem check operation. Existing filesystem Check and Repair tools, e.g. e2fsck and pFSCK, statically determine the data structure for inode reference counters when the filesystem check procedure starts. pFSCK uses an array to represent a set of inode reference counters. The number of elements of the array corresponds to the number of total inodes in the filesystem no matter whether the inode is in use or not. When the filesystem utilization is low, pFSCK unnecessarily allocates a huge amount of memory.

lwFSCK estimates the memory footprint of arrays and TwinTree and adaptively chooses the data structure with minimal memory footprint. The memory footprint of each data structure can be estimated through the continuity of the file inode number and inode usage. The continuity of inode numbers for regular files is inversely proportional to the ratio of directory inodes in the filesystem and directly proportional to the ratio of regular file inodes in the filesystem. Using Eq. 1, the ratios of directory inodes (R_d) and regular file inodes (R_f) in the filesystem can be determined. N represents the number of block group descriptors, and d_i denotes the count of in-use directory inodes in the i-th block group. I_{used} represents the total count of in-use inodes in the filesystem. Eq. 1 calculates the counts of directory and regular file inodes using information about the number of directories and regular files in each block group descriptor.

$$R_d = \frac{\sum_{i=1}^{N} d_i}{I_{used}}, \quad R_f = \frac{\sum_{i=1}^{N} f_i}{I_{used}} \tag{1}$$

Eq. 2 and Eq. 3 represent estimated memory footprints of TwinTree ($Mem_{TwinTree}$) and array (Mem_{Arr}), respectively. S_{single} is the size of an entry of `single` tree. $S_{multiple}$ is the size of an entry of `multiple` tree. As the directory ratio increases and the inode usage rises, the memory footprint with the TwinTree structure also increases. S_{Arr} is the size of an element in the array. Since the array is always allocated with a size equal to the total number of inodes, the memory footprint of the array remains fixed.

$$Mem_{TwinTree} = I_{used} * R_d * (S_{single} + S_{multiple}) \tag{2}$$

$$Mem_{Arr} = I_{total} * S_{Arr} \tag{3}$$

lwFSCK calculates Eq. 2 and Eq. 3 before the start of Pass 1 and determines which data structure to use. If $Mem_{TwinTree}$ is smaller than Mem_{Arr}, it can be inferred that the TwinTree has lower memory usage than the array. In this case, lwFSCK uses the TwinTree. Otherwise, lwFSCK uses the array.

5 EVALUATION

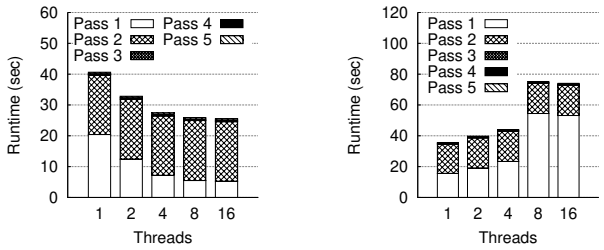
We use Dell PowerEdge R740 server with 40 cores (Intel Xeon Gold 6230), 512GB memory, and Samsung 970 pro NVMe SSD. We build lwFSCK on top of Linux kernel 5.18.18. We set the filesystem partition size to 256 GByte with 16 million inodes. We set the inode utilization to 96%, i.e. 96% of the inodes in the filesystem partition are in use. We compare lwFSCK with pFSCK, the state-of-the-art filesystem check and repair tools. We consider two scenarios: file-intensive case and dir-intensive filesystem. In the file-intensive case, each directory has 99 files. The ratio between the number of files and the number of directories is 99:1. In the dir-intensive case, each directory has one file in it. The ratio between the number of files and the number of directories is 50:50. We use FSMark to construct the filesystem images as in pFSCK [16]. Through the evaluation, we answer the following questions.

- How much execution time does lwFSCK reduce in performing the filesystem check?
- How much memory does lwFSCK save during the filesystem check?

5.1 Performance

Both lwFSCK and pFSCK have two types of parallelism techniques; data parallelism and pipeline parallelism. To explore the effect of each parallelism strategy, we run two experiments. In the first experiment, we activate only the data parallelism of FSCKs. Second, we apply both data parallelism and pipeline parallelism in running filesystem check.

Data Parallelism. We examine the effect of the per-thread data processing design in lwFSCK. lwFSCK adds per-thread data processing technique to the data parallelism of pFSCK. This approach eliminates unnecessary critical sections protected by locks. Fig. 4 illustrates the execution time breakdown of lwFSCK and pFSCK in file-intensive case. lwFSCK outperforms pFSCK by 2.94×. The execution time of pass 1 in lwFSCK decreases as the number of threads increases. This is because the per-thread data structure



(a) lwFSCK (b) pFSCK

Figure 4: Execution time breakdown of file-intensive case with data parallelism

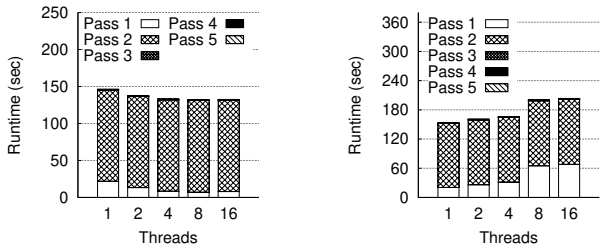
design enables to fully parallelize metadata scanning in pass 1. On the other hand, pFSCK suffers from severe lock contention, thus, its execution time of the pass 1 increases with the number of threads. Dir-intensive case shows a similar pattern as shown in Fig. 5. The per-thread data structure design results in lwFSCK to outperform pFSCK by 2.58 × in pass 1. Since data parallelism works only on pass 1, other passes do not change with the number of threads.

In both file-intensive and dir-intensive cases, the pass 1 execution time of lwFSCK no more decrease when the number of threads exceeds 8. This is the result of the small filesystem size, i.e, 256GB. In lwFSCK, each thread is assigned a set of block groups, and its size is not dramatically reduced if there are not enough block groups to scan. We find that if the filesystem size is 2TB, the pass 1 execution time is also reduced at 16 threads.

Notably, we observe that the merging time in lwFSCK itself is not the bottleneck of many-core scalability. The pass 1 of lwFSCK consists of two phases: checking metadata by multiple threads in parallel, and merging the checking results from multiple threads. The execution time of the first phase decreases as the number of threads increases, the second phase does not. From our observation, the merge time itself is less than 1% of the total execution time, which is too minimal to become a main bottleneck.

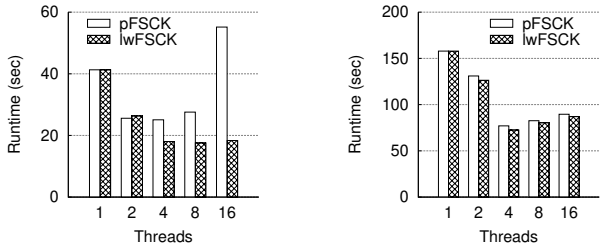
Data Parallelism and Pipeline Parallelism. We measure the execution time of pFSCK and lwFSCK. They use both the data parallelism and the pipeline parallelism in running the filesystem check. In the pipeline parallelism, both pass 1 and pass 2 run simultaneously in pipelined manner. Therefore, the total execution time for filesystem check is determined by the longer execution time between pass 1 and pass 2.

Fig. 6 illustrates the result with varying number of threads in pass 1 and pass 2. The number of threads used ranges from 1 to 16. The same number of threads are used for data parallelism and pipeline parallelism. In the file-intensive case as shown in Fig. 6(a), the execution time of lwFSCK is upto $\frac{1}{2.8}$ of the execution time of pFSCK. Since the lwFSCK is designed to fully exploit both data parallelism and pipeline parallelism, its execution time is enhanced as the number of threads increases. The execution time of pFSCK increases when the number of threads is more than 8. This is because the lock contention overhead during pass 1 becomes a bottleneck to the subsequent passes, increasing the overall execution time of pFSCK.



(a) lwFSCK (b) pFSCK

Figure 5: Execution time breakdown of dir-intensive case with data parallelism



(a) File-intensive case (b) Dir-intensive case

Figure 6: Execution time with data parallelism and pipeline parallelism

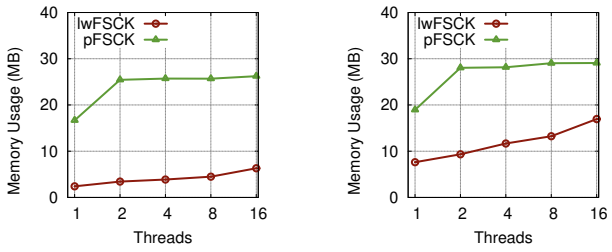
Fig. 6(b) compares the execution time of pFSCK and lwFSCK in a directory-intensive case. The data parallelism and pipeline parallelism reduces execution time of lwFSCK and pFSCK as increasing the number of threads by 4. pFSCK shows almost similar performance with lwFSCK, since it does not much suffer from lock contention during pass 1.

In dir-intensive case, the execution time of pass 1 of lwFSCK is shorter than that of pFSCK. However, the total execution time did not decrease significantly. In a directory-intensive filesystem, the directory ratio is 50%, so the execution time of pass 2 is 10 × more than the execution time of pass 1. As shown in Fig. 5, pass 2 accounts for about 90% of the total execution time. Thus, the lock contention overhead of pass 1 does not overshadow the total execution time of pFSCK. The decrease in the execution time of pass 1 does not have a big impact on the total execution time.

5.2 Memory Pressure

We examine how much lwFSCK can save memory footprint via TwinTree. We measure the memory footprint of lwFSCK and pFSCK with varying the number of threads. To examine the effect of TwinTree, we use the filesystem partition with low utilization. We use a 64GB filesystem parition with 40,000 inodes and set inode utilization to the 1%. We run two scenarios, file-intensive case and dir-intensive case.

Fig. 7(a) and Fig. 7(b) depict the memory footprint of file-intensive filesystem and dir-intensive filesystem, respectively. The memory



(a) File-intensive case (b) Dir-intensive case

Figure 7: Memory Footprint (filesystem partition size: 64 GByte, inode utilization: 1%)

footprint of lwFSCK slightly increases with the number of threads since lwFSCK maintains the per-thread data structure for many-core scalability. The memory footprint of lwFSCK is about $\frac{1}{6.8}$ and $\frac{1}{2.4}$ of pFSCK's in the file-intensive case and the dir-intensive case, respectively. This is because TwinTree of lwFSCK allocates memory only for in-use inodes while pFSCK does not. Since the ratio of in-use inode is about 1%, pFSCK wastes 99% of allocated memory.

In lwFSCK, the file-intensive case has much less memory footprint than dir-intensive case. The reason behind this is that TwinTree space-efficiently maintain the reference count of continuous inodes. Since the file-intensive case contains much more continuous inodes than dir-intensive case, TwinTree significantly reduces the memory footprint of lwFSCK in the file-intensive case.

When the utilization of inode is high (ex. 90%) or the number of threads is numerous (ex. 128), TwinTree of lwFSCK can have more memory footprint than pFSCK. To avoid this, lwFSCK adaptively uses the simple array instead of TwinTree as described in subsection 4.4.

5.3 Adaptive Object Allocation

Before starting pass 1, lwFSCK selects either TwinTree or array based on which data structure renders less memory pressure. As shown in Table 2, the array can have less memory footprint than TwinTree in certain cases. To decide between TwinTree and array, we calculate the ratios of directory inodes and regular file inodes in the filesystem. These ratios can be obtained from the superblock or group descriptor of each block group [36]. According to our experiments, the overhead of selecting the data structure is negligible.

Table 2: Comparison of Memory pressure in MByte (filesystem partition size : 256 GByte, 8 Threads)

inode utilization	1%	25%	50%	100%
Array	72.4	200.8	274.0	280.2
TwinTree	9.5	173.7	284.7	355.3

In a 2 TByte filesystem partition (RAID 5, 7+1 configuration with SATA Samsung SM883 SSDs, and a Dell PERC H730P Controller), the selection process takes only 159ms, while the total fsck execution time is 192 seconds. This is because the group descriptor size is very small—64 bytes per block group. Each group descriptor

block contains 64 group descriptors, and since the block group size is 128 MByte, a single group descriptor block covers 8 GByte of the filesystem. For a 2 TByte filesystem, only 256 blocks of 4 KByte each need to be read. Thus, the overhead is almost negligible.

6 RELATED WORK

As the size of the filesystem and the number of available inodes increase, the execution time of the filesystem Check and Repair tool linearly increases. This has led to research on improving the performance of filesystem Check and Repair tools. xfs_repair, a representative Check and Repair tool of the XFS filesystem, supports parallel processing. However, the unit of parallel processing is vast, such as disk, logical group, etc. If there is a large difference in the amount of data between the units performing parallel processing, it can lead to load imbalance, where some threads remain idle while others are overloaded. This imbalance reduces the overall efficiency of parallel processing, limiting the performance gains from multi-threading. Ffsck [34] is a filesystem Check and Repair tool that aims to reduce execution time by improving inefficient I/O patterns and changing the metadata structure of the existing filesystem. By changing the Ext filesystem layout considering the SEEK time of the hard disk, the inspection process has been improved to be processed as much as possible by sequential reading. However, because it requires modifications to the on-disk filesystem layout, such as changing the metadata structure and rearranging blocks for verification and recovery, it is challenging to be widely adopted in the industry. Chunkfs [24] aims at reducing the total execution time by dividing the entire filesystem partition into the multiple small, individually repairable fault-isolation domains and performing inspection on each domain in parallel. However, there can still be data imbalance between the partitioned domains, which reduces the overall efficiency of parallel processing. SQCK [21] performs filesystem verification and recovery work by using SQL queries. It proposes a technique to encapsulate 121 inspections performed in e2fsck into a set of SQL queries. Instead of e2fsck, which is composed of complex and incomplete code written in low-level languages, SQCK has the advantage of being able to perform recovery work simply and concisely through queries. However, SQCK hinders widespread adoption because it needs to build a new recovery tool.

7 CONCLUSION

We observe two limitations of the state-of-the-art filesystem Check and Repair tool, pFSCK: (i) severe lock contention on the global data structure, and (ii) unnecessarily excessive memory footprint to maintain inode reference counts. Based upon this observation, we propose the following two techniques. The first is per-thread data processing. In per-thread data processing, we carefully replace a global data structure where lock contention occurs with a per-thread data structures. The second is an adaptive data structure allocation. In order to minimize the memory footprint for maintaining the reference count of inodes, lwFSCK adaptively switches the data structure between the simple array and TwinTree. TwinTree is a red-black tree-based data structure which is designed to space-efficiently maintain the reference count of continuous inodes. lwFSCK estimates the memory footprints of simple array and TwinTree and adopts the data structure with a minimal memory footprint. lwFSCK reduces the execution time of pass 1 in a file-intensive filesystem by up to 2.94 × compared to pFSCK. lwFSCK reduces the entire execution time by up to 1.4 × compared to pFSCK. Through the TwinTree data structure, lwFSCK reduces memory footprint by 6.8 × in a file-intensive filesystem using only 1% of the inodes.

Acknowledgements We are grateful to the anonymous reviewers for their valuable comments and feedback. This work was supported by Development of Collection and Integrated Analysis Methods of Automotive Inter/Intra System Artifacts through Construction of Event-based experimental system Research Program (No. 2022-0-01022) through the Institute of Information & Communications Technology Planning & Evaluation (IITP), Korea.

REFERENCES

[1] e2fsck: fsck for ext4. https://linux.die.net/man/8/e2fsck.
[2] Ext4 on-disk layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.
[3] introduce parallel fsck to e2fsck pass1. https://patchwork.ozlabs.org/project/linux-ext4/list/?series=169193.
[4] Mtanski. Parallel XFS. https://github.com/mtanski/xfsprogs/tree/preadv2/repair.
[5] XFS on-disk layout. https://dubeyko.com/development/FileSystems/XFS/xfs_filesystem_structure.pdf.
[6] xfs_repair: fsck for xfs. https://linux.die.net/man/8/xfs_repair.
[7] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Correlated crash vulnerabilities. In *Proc. of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[8] Sultan Aldossary and William Allen. 2016. Data security, privacy, availability and integrity in cloud computing: issues and current solutions. *SAI International Journal of Advanced Computer Science and Applications (IJACSA)* 7, 4 (2016).
[9] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. 2007. Provable data possession at untrusted stores. In *Proc. of the 14th ACM Conference on Computer and Communications Security (CCS)*.
[10] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. 2008. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)* 4, 3 (2008), 1–28.
[11] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2014. OpLog: a library for scaling update-heavy data structures. *MIT CSAIL Technical Reports* (2014).
[12] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, and Yong Chen. 2018. PFault: A general framework for analyzing the reliability of high-performance parallel file systems. In *Proc. of the 32nd ACM International Conference on Supercomputing (ICS)*.
[13] Vijay Chidambaram, Tushar Sharma, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2012. Consistency without ordering. In *Proc. of the 10th USENIX Conference on File and Storage Technologies (FAST)*.
[14] Dong Dai, Om Rameshwar Gatla, and Mai Zheng. 2019. A performance study of lustre file system checker: Bottlenecks and potentials. In *Proc. of the 35th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*.
[15] Borislav Djordjevic and Valentina Timcenko. 2012. Ext4 file system in linux environment: Features and performance analysis. *International Journal of Computers (IJC)* 6, 1 (2012), 37–45.
[16] David Domingo and Sudarsun Kannan. 2021. {pFSCK}: Accelerating File System Checking and Repair for Modern Storage. In *Proc. of the 19th USENIX Conference on File and Storage Technologies (FAST)*.
[17] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying file system consistency at runtime. *ACM Transactions on Storage (TOS)* 8, 4 (2012), 1–29.
[18] Om Rameshwar Gatla, Mai Zheng, Muhammad Hameed, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojevic, Cyril Guyot, and Robert Mateescu. 2018. Towards robust file system checkers. *ACM Transactions on Storage (TOS)* 14, 4 (2018), 1–25.
[19] Haryadi S Gunawi. 2011. Improving File System Reliability and Availability with Continuous Checker and Repair. (2011).
[20] Haryadi S Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2007. Improving file system reliability with I/O shepherding. In *Proc. of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*.

[21] Haryadi S Gunawi, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2008. SQCK: A Declarative File System Checker. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[22] Christoph Hellwig. 2009. XFS: the big storage file system for Linux. *USENIX Magazine of USENIX & SAGE* 34, 5 (2009), 10–18.

[23] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. 2006. Reducing fsck time for ext2 file systems. In *Proc. of the 2006 USENIX Ottawa Linux Symposium (OLS)*.

[24] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. 2006. Chunkfs: Using Divide-and-Conquer to Improve File System Reliability and Repair. In *Proc. of the 2nd USENIX Workshop on Hot Topics in System Dependability (HotDep)*.

[25] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. 2019. Evaluating file system reliability on solid state drives. In *Proc. of the 2019 USENIX Annual Technical Conference (ATC)*.

[26] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. 2020. The reliability of modern file systems in the face of SSD errors. *ACM Transactions on Storage (TOS)* 16, 1 (2020), 1–28.

[27] Jerzy Kaczmarek and Michal Wrobel. 2008. Modern approaches to file system integrity checking. In *Proc. of the 1st IEEE International Conference on Information Technology (ICITS)*.

[28] Saisha Kamat, Abdullah Al Raqibul Islam, Mai Zheng, and Dong Dai. 2023. FaultyRank: A graph-based parallel file system checker. In *Proc. of the 37th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[29] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. {SpanFS}: A scalable file system on fast storage devices. In *Proc. of the 2015 USENIX Annual Technical Conference (ATC)*.

[30] Gene H Kim and Eugene H Spafford. 1994. The design and implementation of tripwire: A file system integrity checker. In *Proc. of the 2nd ACM Conference on Computer and Communications Security (CCS)*.

[31] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. 2021. {Z-Journal}: Scalable {Per-Core} journaling. In *Proc. of the 2021 USENIX Annual Technical Conference (ATC)*.

[32] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. {F2FS}: A new file system for flash storage. In *Proc. of the 13th USENIX Conference on File and Storage Technologies (FAST)*.

[33] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2014. Physical Disentanglement in a {Container-Based} File System. In *Proc. of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[34] Ao Ma, Chris Dragga, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Marshall Kirk Mckusick. 2014. Ffsck: The fast file-system checker. *ACM Transactions on Storage (TOS)* 10, 1 (2014), 1–28.

[35] Marshall K McKusick, Gregory R Ganger, et al. 1999. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proc. of the 1999 USENIX Annual Technical Conference (ATC)*.

[36] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. 1984. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)* 2, 3 (1984), 181–197.

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

[38] Juan Piernas, Toni Cortes, and José M García. 2002. DualFS: a new journaling file system without meta-data duplication. In *Proc. of the 16th ACM International Conference on Supercomputing (ICS)*.

[39] Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2005. Analysis and Evolution of Journaling File Systems. In *Proc. of the 2005 USENIX Annual Technical Conference (ATC)*.

[40] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

[41] Muthian Sivathanu, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Somesh Jha. 2005. A Logic of File Systems. In *Proc. of the 3rd USENIX Conference on File and Storage Technologies (FAST)*.

[42] Yongseok Son, Sunggon Kim, Heon Y Yeom, and Hyuck Han. 2018. {High-Performance} Transaction Processing in Journaling File Systems. In *Proc. of the 16th USENIX Conference on File and Storage Technologies (FAST)*.

[43] Stephen C Tweedie et al. 1998. Journaling the Linux ext2fs filesystem. In *Proc. of the 4th Southern California Annual Linux Expo (SCALE)*.

[44] Yuqi Zhang, Wenwen Hao, Ben Niu, Kangkang Liu, Shuyang Wang, Na Liu, Xing He, Yongwong Gwon, and Chankyu Koh. 2023. Multi-view feature-based {SSD} failure prediction: What, when, and why. In *Proc. of the 21st USENIX Conference on File and Storage Technologies (FAST)*.

[45] Yupu Zhang, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2010. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proc. of the 8th USENIX Conference on File and Storage Technologies (FAST)*.