pFSCK: Accelerating File System Checking and Repair for Modern Storage

David Domingo and Sudarsun Kannan, *Rutgers University*

https://www.usenix.org/conference/fast21/presentation/domingo

This paper is included in the Proceedings of the
19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Open access to the Proceedings
of the 19th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

pFSCK: Accelerating File System Checking and Repair for Modern Storage

David Domingo, Sudarsun Kannan
Rutgers University

Abstract

We propose and design pFSCK, a parallel file system checking and recovery (C/R) tool designed to exploit compute and storage parallelism in modern storage devices. pFSCK enables fine-grained parallelism at the granularity of inodes and directory blocks without impacting the C/R's correctness. pFSCK first employs data parallelism by identifying functional operations in each stage of the checking logic and then isolating dependent operations and shared data structures. However, full isolation of shared structures is infeasible and requires serialized updates. To reduce serialization bottlenecks, pFSCK introduces pipeline parallelism, allowing multiple stages of C/R to run concurrently without impacting correctness. Further, pFSCK provides per-thread I/O cache management, dynamic thread placement across C/R stages, and a resource-aware scheduler to reduce the impact of C/R on other applications sharing CPUs and the file system. Evaluation of pFSCK shows more than 2.6x gains over e2fsck (Ext file system C/R) and more than 1.8x over XFS's C/R that provides coarse-grained parallelism.

1 Introduction

Modern ultra-fast storage devices such as SSDs, NVMe, and byte-addressable NVM storage technologies offer higher bandwidth capabilities and lower latency than hard-disks providing better opportunities for exploiting CPU parallelism. While the I/O access performance has increased, storage hardware and software errors have continued to grow coupled with newer and exploratory high-performance designs impacting file system reliability [10, 12, 18, 21, 27, 46]. For decades, file system checking and repair tools (referred to as C/R henceforth) have played a pivotal role in increasing the reliability of software storage and availability of systems by identifying and correcting file system inconsistencies [41].

A significant body of prior work has shown that, in the event of a system crash or failure in data centers, C/Rs are typically used as the first remedial solution to system recovery. Prior work [21, 27] and discussions with file system maintainers and IT teams of organizations show that C/Rs are run across various scenarios. This includes problems during reboot due to hardware or software errors [11, 21, 27], periodic maintenance, or during mandatory security upgrades [37]. When C/Rs are run on a disk partition in an offline fashion, the partition's data is unavailable. Some C/Rs support online checking, but it is crucial that they do not interfere with other applications that use the same device. Thus, improving C/R performance and flexibility is critical for system availability and reducing performance impact on other applications.

File system C/R tools work by identifying and fixing the structural inconsistencies of file system metadata. The inconsistencies could be in inodes, data and inode bitmaps, links, or directory entry structures. Well-known and widely used tools such as e2fsck (file system checker for Ext4) [2] divide C/R across multiple stages (commonly referred to as passes), with each pass responsible for checking a file system structure (e.g., directories, files, links). However, C/Rs are known to be notoriously slow, showing a linear increase in C/R time with an increase in file and directory count [24, 38–41], at times lasting hours [37] or even weeks [11]. Although modern flash and NVM technologies provide lower latency and bandwidth, current C/R tools fail to exploit such hardware capabilities or multi-core CPU parallelism. While modern C/Rs have attempted to increase parallelism, they adopt coarse-grained approaches, such as parallelizing C/R across logical volumes or logical groups, which are insufficient to accelerate C/R on file systems with data imbalance across logical groups [20, 24, 39, 42].

To overcome such limitations, we propose **pFSCK**, a parallel C/R that exploits CPU parallelism and modern storage's high bandwidth to accelerate file system C/R in offline and online forms, thereby reducing system downtime and improving data (and system) reliability and availability [10, 20, 21, 39]. While pFSCK borrows ideas from prior task parallelism research [35, 45], it must solve several challenges specific to C/R, which includes increasing scalability in the presence of complex file system layouts and shared file system structures (e.g. universal bitmaps) without impacting correctness, adapting to various file system configurations, and reducing C/R impact on other applications. pFSCK introduces fine-grained parallelism, i.e., parallelism at the granularity of inodes and directory blocks, resulting in a significantly faster execution than traditional C/Rs. pFSCK first employs **data parallelism** by breaking up the work done at each pass, redesigning data structures for scalability, and allowing multiple threads to perform checks in parallel. Although data parallelism accelerates checking, updates to global data structures (e.g., bitmaps) within each pass are designed to match the file system's layout (e.g., block bitmap in an Ext4 file system) and must be synchronized to ensure checking correctness. As a result, with increasing thread counts, the cost of synchronization and serialization can quickly outweigh the performance gains. Hence, pFSCK introduces **pipeline parallelism** to parallelize C/R along with the logical flow (i.e. across multiple passes).

Supporting data and pipeline parallelism within pFSCK requires addressing several challenges. First, certain consistency checks must be ordered for correctness. For example, a

directory cannot be certified to be error-free by the directory checking pass until all its files are verified as consistent by the inode checking pass. To address these ordering constraints, we take inspiration from modern hardware processors that support out-of-order execution but with in-order instruction commit. We isolate the global data structures and perform all necessary operations in parallel but certify correctness only when the results are merged. Second, static partitioning of CPU threads across different C/R passes is suboptimal because the time to process different metadata (e.g., file, directory, links) varies significantly (e.g., checking a directory can take substantially longer than a file). Hence, we propose **pFSCK scheduler**, a dynamic thread scheduler that monitors progress across different passes of pFSCK and uses the pending work ratio for thread assignment.

Third, I/O optimizations such as I/O caching and read-ahead mechanisms in current C/Rs are not designed for multi-threaded parallelism, which we address by designing thread-aware I/O caching, thereby substantially reducing I/O wait-times. Finally, to exploit multi-core parallelism in ways that do not affect the performance of other co-running applications that share CPUs or access the same disks checked by C/R (online checking), we design a **resource-aware pFSCK scheduler** that dynamically scales the C/R threads across passes by monitoring the total CPU utilization of the system.

The combination of pFSCK's above techniques significantly reduces C/R runtime. For example, using pFSCK's data parallelism and pipeline parallelism on a 1TB NVMe (and 2TB SSD) reduces C/R runtime for a file- and directory-intensive disk configurations by up to 2.6x and 1.6x, respectively compared to widely used e2fsck and by up to 1.8x over the XFS C/R tool. Further, pFSCK's scheduler increases gains by 1.1x. When sharing the CPUs between pFSCK and RocksDB, the resource-aware mechanism minimizes pFSCK performance degradation to 1.07x and limits RocksDB's performance overheads by 1.05x. The online C/R performance improves by up to 1.7x over the vanilla e2fsck. Finally, pFSCK improves I/O throughput by up to 2.7x with a nominal increase in memory use by 1.3x over e2fsck (from 2.7 GB in e2fsck to 3.5 GB) to manage task structures for threads.

2 Background and Motivation

We first give a brief background on current hardware trends, C/R tools, and then discuss prior approaches that accelerate C/R and their limitations.

2.1 Hardware and Software Trends

Modern ultra-fast storage devices such as SSDs and NVMe provide not only high bandwidth (8-16 GB/s) but also two orders of reduction in storage access latency (< 20µsec) compared to traditional hard drives [31, 49]. At the other end, fast storage class memories such as Intel's DC Optane [5] and other byte-addressable persistent memory technologies are evolving with access latency < 1µsec. In recent years, several new file systems have evolved to exploit these hardware benefits. A huge body of prior and ongoing research is developing optimized file systems to support fast storage hardware. This includes file systems for SSDs [34], NVMes [44], open-source efforts to optimize traditional Ext4 and XFS file systems for NVMs [48], and other research efforts [30, 33, 50]. However, reducing data corruption and errors with these file systems would require a few years of production use [9, 28]. While file system C/R tools will play a crucial role in these file systems, they are yet to be optimized to extract hardware storage benefits and multi-core parallelism.

2.2 File System Checking and Repair

Since the dawn of file systems, consistency has always been an issue. Though storage mechanisms such as journaling, copy-on-write, log-structured writes, and soft updates have been developed to mitigate potential file system inconsistencies, they are limited as they cannot fix errors that arise from software bugs or corruptions manifested in the past by events such as a failing disk, bit flips, overheating, or correlated crashes [13–15, 29, 51]. In these cases, popular C/R tools, such as e2fsck and xfs_repair [42], are used to detect and fix corruptions and errors by traversing the file system's layout and checking for inode consistency, directory consistency, file and directory connectivity, directory entry consistency, and consistent reference counts of inodes and blocks

C/R usage. The frequency and resulting runtime of file system C/Rs vary significantly in the real-world setting. While there is a lack of well-documented C/R best practices, our discussions with file system C/R maintainers, infrastructure teams, and other public discussions show that C/R tools such as e2fsck and xfs_repair remain critical for data reliability in current large-scale and personal computing systems as they are generally run during after system errors [7, 21, 27, 29], hardware or kernel upgrades, or even after mandatory security updates. Infrequent C/R and storage maintenance can increase system downtime to as high as three hours [37] and, in extreme cases, weeks on petabyte-scale file systems [11].

2.3 Related Work

Increasing disk capacities, overall file system size, and file counts have made C/Rs notoriously run longer, leading to longer downtime and forcing developers and users to reduce C/R usage at the risk of data loss [1, 7, 8, 25, 36]. We next discuss the state-of-the-art C/R optimizations for offline (unmounted file systems) and online C/Rs and their limitations.

Offline C/Rs. Widely used open-source tools such as Ext4 file system's e2fsck parallelize C/R across multiple disks (e2fsck), where XFS file system's xfs_repair parallelizes C/R across disk and volume logical groups. Other approaches like Ffsck [41] and Chunkfs [26] have proposed accelerating C/R speed up by modifying file systems to provide a better balance across logical groups. For example, Chunkfs utilizes disk bandwidth by partitioning the file system into smaller, isolated groups that can be repaired individually and in parallel. In contrast, Ffsck [41] rearranges metadata blocks

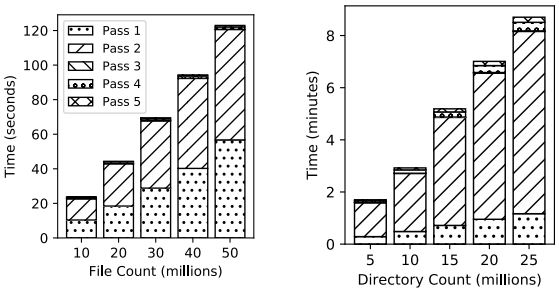
to reduce the seek cost and optimize file system traversal. SQCK [19] enhances C/R by utilizing declarative queries for consistency checking across file system structures. While prior approaches have advanced C/R innovations, they suffer from several weaknesses. First, most prior techniques fail to exploit multi-core parallelism and high storage bandwidths. Second, prior parallelization efforts are mostly coarse-grained (e2fsck, xfs_repair, Chunkfs). For instance, as we show in Section § 6, for an XFS file system that parallelizes across logical groups, imbalance in the number of files across logical groups and lack of parallelism for directory metadata checking leads to high overheads. Finally, techniques such as SQCK and Ffsck demand intrusive changes to file system metadata, block placement, or the need to rebuild C/R, hindering widespread adoption.

Online C/Rs. The last decade has seen an active push to develop online C/R techniques to identify and fix errors while applications concurrently use the file system in order to reduce system downtime and allow for the proactive identification of potentially harmful corruptions. Proprietary online C/Rs such as WAFL file system's Iron [32] (a NetApp-based C/R tool for WAFL file system) performs incremental live C/R by applying invariants such as checking all blocks before any software use and checking ancestor blocks (directory) before any data or metadata block (inode block). To scale C/R to petabytes, WAFL-Iron expects the presence of block-level checksums, RAID, and, most importantly, good storage practices by customers. Alternatively, Recon protects file system metadata from buggy operations by verifying metadata consistency at runtime [16]. Doing so allows Recon to detect metadata corruption before committing it to disk, preventing error propagation. Recon does not perform a global scan and cannot identify or fix errors originating from hardware failures. C/Rs such as e2fsck, traditionally meant for offline use, allow for partial online checking by utilizing LVM-based snapshotting and scanning for errors on the snapshot while the file system is still in use [3]. However, if errors need to be fixed, the C/R must be used offline. In this paper, we also study and evaluate open-source and widely-used online e2fsck against online pFSCK that exploit storage and compute parallelism.

C/R Correctness. To ensure the correctness and crash-consistency of C/Rs itself and recover more reliably in light of system faults, Rfsck-lib [17] provides C/Rs with robust undo logging. pFSCK's fine-grained parallelism goals are orthogonal to Rfsck-lib; however, incorporating Rfsck-lib can further improve pFSCK's reliability.

3 Motivation and Analysis

In the pursuit of accelerating C/Rs, we first decipher the performance bottlenecks of the widely-used Ext4 file system's e2fsck C/R tool. We first provide an overview of e2fsck and then examine e2fsck's runtime for different file system configurations. For brevity, we study xfs_repair in Section § 6.



(a) File Count Sensitivity: Runtime of e2fsck as total file count increases

(b) Directory Count Sensitivity: Runtime of e2fsck as total directory count increases

Figure 1: Runtime of C/R for an 1TB file system with varying counts of files or directories

3.1 e2fsck Overview

E2fsck uses five sequential passes for C/R: the first pass (referred to as Pass-1) checks the consistency of inode metadata; Pass-2 checks directory consistency; Pass-3 checks directory connectivity; Pass-4 checks reference counts; finally, Pass-5 checks data and metadata bitmap consistency.

3.2 Setup and Runtime Analysis

To analyze and decipher the cost of C/R runtime, we run e2fsck on file systems with varying configurations. We conduct our analysis on a 64-core Dual Intel® Xeon Gold 5218 running at 2.30GHz, 64GB of DDR memory, a 1TB NVMe, and a 2TB Micron 5200 SATA SSD running Ubuntu 18.04.1. We fill the file system using *fs_mark*, an open-source, file system benchmark tool [47]. We mainly focus on file systems without corruptions but study images with corruptions in Section § 6. To get a finer understanding of how e2fsck scales with file system configurations, we study the sensitivity of C/R's runtime for multiple file system variables such as file count and directory count.

File-intensive file systems. First, to understand how the file count affects runtime, we generate multiple file-intensive file system configurations with a 95 : 1 file to directory ratio. Pass-1, which checks the consistency of inodes structures, dominates e2fsck runtime, followed by Pass-2, which checks directory block consistency. Figure 2 shows the function-wise breakdown in Pass-1 that checks the consistency of file inodes as well as tracks directory blocks encountered to be examined in the next pass. We notice a function `dcigettext` (a seemingly innocuous) language translator used for error handling gets (incorrectly) used for every inode check, causing notable C/R slowdown. [1] Other steps such as `check_blocks` that checks blocks referenced by an inode, `next_inode` that reads next inode blocks from disk, `mark_bitmap` that updates global bitmaps to track the metadata encountered, and `icount store` that stores inode references also increase in runtime. Despite fewer directories, Pass-2's (directory checking) runtime increases because the number of directory blocks to

[1] We reported this to e2fsck developers, and the fix has been upstreamed.

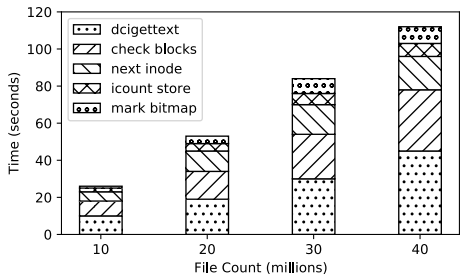


Figure 2: e2fsck Pass-1 Time Breakdown. Time spent within inode checking pass (Pass-1) as the total file count increases.

store directory entries increases. For all file counts, time spent in Pass-1 and Pass-2 account for over 95% of the runtime. Finally, for a small directory count, Pass-3, which checks connectivity and reachability of directories from the root, has lower runtime. *Increasing the file size when keeping file count constant does not increase C/R runtime significantly* (not shown for brevity).

Directory-intensive file system. In Figure 1b, we decipher the runtime of a directory-heavy file system configuration with 1 : 1 file to directory ratio. To ensure that each directory requires the same amount of work, we create a single file in each directory. First, the cost of fetching, identifying directory blocks, and adding them to a global block list (*db_list*) increases Pass-1's cost along with reference counting. As expected, with an increase in directory count, Pass-2's runtime significantly increases due to an increase in the number of directory blocks. Additionally, the directory blocks store checksums, and the checksum is recomputed and verified against the one stored in the directory blocks for consistency. Similar to the file-intensive file systems, for all directory counts, over 90% of the runtime is spent within Pass-1 and Pass-2.

3.3 Compute time and I/O utilization

To understand the computational vs. I/O bottlenecks, we analyzed the time spent by e2fsck on compute time vs. the time spent waiting for I/O to complete. Our analysis shows that in modern storage devices like NVMe, the I/O wait time for file-intensive and directory-intensive configurations are only 3% and 20% of the overall execution time. We also notice poor storage throughput utilization, which is just 270 MB/s on an NVMe device with 2 GB/s and 512 MB/s sequential and random read bandwidth, respectively. This clearly shows that (1) computation is the main bottleneck as it dominates overall execution time, and (2) C/Rs such as e2fsck fail to benefit from modern storage device bandwidths.

Summary. To summarize, our analysis of widely-used C/R tools such as e2fsck (and xfs_repair later in § 6) show high runtime overheads mainly due to single-threaded or lack of fine-grained parallelism. The linear complexity of C/R runtime is unsuitable as disk capacities and file system size trends upward, potentially taking hours, or even days, to check datacenter-scale file systems. Besides, C/R's repair during file system inconsistencies could further increase C/R runtime.

4 pFSCK Goals and Insights

pFSCK aims to address the limitations of current file system C/Rs by exploiting fine-grained multi-core parallelism and higher disk bandwidth. We first discuss our goals and insights, followed by pFSCK's design and implementation.

4.1 Goals

The main goal is to make the file system C/R faster. We want to increase the speed at which file system metadata is scanned and inconsistencies are identified without compromising repairing capabilities. In this pursuit, pFSCK strives to achieve the following goals: (1) adapt to different file system configurations, regardless of file system size, utilization, or configurations, such as a file-intensive or directory-intensive file system, (2) support efficient C/R for both online and offline forms, and finally, (3) adapt to varying system resource utilization to reduce the performance impact on any concurrently running applications.

4.2 Design Insights

Insight 1: Maximize potential bandwidth through multiple cores and data parallelism. To overcome the bottlenecks of current C/R tools that employ serial or coarse-grained parallelization techniques at the disk, volume, or logical group-level, pFSCK introduces fine-grained data parallelism. As shown in Section § 3.2, since Pass-1 and Pass-2 account for over 90% of the runtime for both file- and directory-intensive file systems, pFSCK focusses its efforts on these two passes. Our approach divides finer file system structures such as inodes, directory blocks, and dirents across a pool of worker threads in a single pass that performs C/R concurrently. While seemingly simple, achieving data parallelism requires data structure isolation across threads to reduce synchronization bottlenecks.

Insight 2: Enable pipeline parallelism by reducing inter-pass dependencies. Though data parallelism accelerates C/R, each pass (e.g., directory checking) must wait for the previous pass (e.g., inode check) to complete. Specifically, in C/R, several inter-pass global data structures are used to build a consistent view of the file system and identify inconsistencies (ex. bitmaps). As a consequence, updates to the shared global structures must be serialized, thereby increasing contention to shared structures with increasing thread count and limiting the CPU scalability. To reduce serialization overheads, pFSCK designs pipeline parallelism that breaks the rigid wall across passes allowing multiple passes to be executed concurrently. pFSCK manages per-pass thread pools, isolates inter-pass shared structures using divide and merge approaches, delineates checking from actual certification of an inode, and reduces I/O wait times.

Insight 3: Adapt to file system configurations with dynamic thread scheduling. Both data and pipeline parallelism requires assigning threads across different passes. Static or equal partitioning of CPU threads is suboptimal due to a lack of information about metadata types (files, directories, links)

and work across passes. Simple checks such as information about the number of files vs. directory inodes are insufficient because directory processing is complex and time-consuming (see § 3). To overcome the above challenge, we design a C/R thread scheduler that dynamically assigns and migrates threads across passes to process different file system objects as they are discovered.

Insight 4: Reduce system impact through resource utilization awareness. File system C/Rs could potentially run with other applications sharing CPUs while performing checking on separate disks. Given pFSCK's goal is to exploit available CPUs, it could potentially impact other co-running applications. Similarly, C/R could run on disks that are also actively used by other applications to store data. To reduce the overall system impact on co-running applications as well as pFSCK, we equip pFSCK's scheduler with resource awareness to dynamically identify the number of cores to use at any single point in time to minimize the potential impact on other co-running applications and pFSCK's performance.

5 Design and Implementation

We discuss pFSCK's design and implementation of data parallelism, pipeline parallelism, dynamic thread scheduler, and resource-aware scheduling. We aim to extend the widely used e2fsck without requiring file system layout changes and reuse features such as snapshot-based online C/R.

5.1 Data Parallelism

pFSCK's data parallelism aims to divide work across worker threads in each pass at the granularity of inodes for parallelizing C/R. However, pFSCK must ensure efficient parallelism without compromising file system integrity or correctness through a functional separation of C/R passes and per-thread contexts that isolate global data structures for reducing synchronization cost.

Fine-grained Inode-level Parallelism. For fine-grained inode parallelism, pFSCK uses the superblock information to identify the total number of inodes in a file system and evenly divides the inodes across C/R workers. To reduce worker management costs, we use a thread-pool framework from our prior work to assign tasks across workers [31].

Functional Parallelism for Reducing Synchronization Overheads. Only dividing inodes for C/R across workers is insufficient. To benefit from fine-grained parallelism, reducing synchronization cost across workers in each pass without impacting correctness is critical.

pFSCK breaks each C/R pass into four main functional steps that comprise 95% of the work and reduces synchronization across these steps. These steps include (1) file system metadata checks, (2) global file system metadata update, (3) accounting, and finally, (4) intermediate result sharing. The metadata check verifies the integrity of metadata structures (for example, inode checksums and block references). Next, the global file system metadata updates include changes to file system-level bitmaps that maintain the checker's view of the file system in order to detect inconsistencies. For example, the block bitmap as shown in Table 1 is marked to track which block references have been seen to detect duplicate block references where more than one inode claims the same block(s). Third, C/R-level accounting involves updating counters that track statistics such as file types. Finally, intermediate result sharing involves creating data structures and lists that hold information to be processed by the next pass. Such structures include directory info lists and directory block lists that store directory information and the location of their blocks so it can be checked in the directories pass.

Structure	Role	pFSCK's access
dir_info_list	maintains directory information/relationships	splits into per-thread lists
db_list	maintains directory blocks for Pass-2 to check	splits into per-thread lists and merges
inode_used_bitmap	tracks valid inodes	using locks
inode_dir_bitmap	tracks directory inodes	using locks
inode_reg_bitmap	tracks regular file inodes	using locks
block_found_bitmap	tracks used data blocks	using locks
icount	tracks inode reference counts	using locks

Table 1: Global structures, their role, and access method.

While synchronization between file system metadata checks (step 1) and global metadata update steps (step 2) is essential, synchronization between the first two steps and the last two steps, C/R counter/statistics update (step 3) intermediate result sharing (step 4) can be avoided by allowing threads to maintain per-thread stats and generate data structures in isolation. The results of steps 3 and 4 can be aggregated before the next pass, reducing synchronization costs significantly.

Thread Contexts for Isolation. In current C/Rs such as e2fsck (and xfs_repair), we find significant use of global data structures inside and across passes. To reduce sharing and increase concurrency, we introduce per-thread contexts similar to OS thread contexts. These contexts store information that allow threads to operate in parallel. First, per-thread block caches, buffers (heap allocations), and iterators of file system objects allow for parallel file system traversal. Second, per-thread intermediate data structures and counters are used for gathering high-level file system state in parallel. For example, each inode pass (Pass-1) thread has its own `db_list` (directory block list) and `dir_info` list (directory information list) that gathers information about directories and exports these to the directory pass (Pass-2) threads for processing. Lastly, per-thread counters are used to track file type statistics in parallel. However, we observe that due to frequent access of global bitmaps across passes (for almost every operation), pFSCK is forced to use synchronization through locking. While disaggregating these bitmaps into per-thread structures is feasible, it would demand significant changes to the e2fsck framework. Table 1 shows the shared structures and their role in e2fsck.

Thread Colocation for Improved Locality. To increase locality and better utilize processor cache state, in each pass, pFSCK attempts to co-locate threads within the same pass to

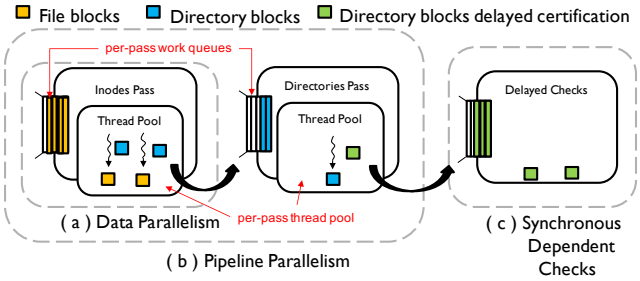


Figure 3: Parallelism in pFSCK. (a) Threads within each pass allows for data to be operate in parallel (*data parallelism*). (b) Multiple thread pools allows each pass of pFSCK to operate simultanously (*pipeline parallelism*). (c) Any dependent checks needed to be carried out synchronously is delayed within its own logical pass.

the same cores and memory sockets to avoid bouncing lock variables and shared structures across processor caches. To enable thread colocation, pFSCK maintains the CPU number each thread has used and a list of cores used by a particular pass. pFSCK first attempts to place the thread to the previously used core (if available), and if unavailable, uses other available cores which were used in the same pass.

Reducing I/O Wait Time with Per-thread Prefetchers and Cache. Though current C/Rs such as e2fsck cache and prefetch file system blocks, the caching and prefetching mechanisms are inflexible and lack thread awareness. First, e2fsck uses a small, static fully-associative LRU-based cache (with 8 blocks) and prefetches just inode blocks. E2fsck does not prefetch directory data blocks, which could be non-contiguous, unlike inode blocks. Second, because threads access blocks at different offsets, sharing a cache across threads results in conflicting evictions, increasing I/O overheads.

To overcome such limitations, we design and implement a per-thread caching mechanism to avoid false eviction of cache entries across threads. For avoiding the non-contiguity problem of directory blocks, we implement an adaptive prefetching mechanism (similar to Linux filesystem prefetching) – decrease prefetching window if previously prefetched directory blocks are not used due to lack of sequential access.

5.2 Pipeline Parallelism

While data parallelism achieves concurrency for processing file system objects within a pass, fully isolating per-pass shared data structures and global data structures is not feasible without substantial changes to either the file system layout or the C/R. As a result, data parallelism does not fully benefit from increasing the CPU count. As our results show, the benefits can considerably degrade performance at higher core counts due to increasing synchronization overheads.

To reduce synchronization time and increase CPU effectiveness, pipeline parallelism breaks the limitation that C/R passes must be sequentially executed. pFSCK's pipeline parallelism allows a subsequent C/R pass ($Pass_{i+1}$) to start even before the completion of an earlier pass ($Pass_i$) in a **pipelined** fashion (i.e., checking directories in directory checking pass even before the inode checking pass has completed).

5.2.1 Per-Pass Thread Pools and Work Queues.

First, to facilitate concurrent execution of passes, we use *per-pass thread pools*. As shown in Figure 3, the inode and directory checking passes maintain a separate thread pool and a dedicated work queue filled with file system objects needing to be checked. As each pass operates, any intermediate work generated is placed in the next pass's work queue. For example, the inode checking pass, when encountering an inode representing a directory, queues its blocks to the directory checking pass's work queue to enable concurrent C/R.

5.2.2 Delayed Certification for Concurrency.

Allowing multiple passes to run in parallel using pipeline parallelism requires reordering logical checks for correctness. For example, with pFSCK's pipeline parallelism, the directory data blocks can be checked by the directory checking pass (Pass-2) in parallel with inode checking pass (Pass-1) checking all the inodes (files and subdirectories) in the directory. While the two passes can proceed in parallel, a directory can be marked as consistent only after the inode checking pass verifies the consistency of its subdirectories and files. There are two main constraints for certifying a directory by the C/R: (1) all inodes referenced by the dirents of this directory are valid, and (2) the parent directory referenced by this directory is valid.

Providing Ordering Guarantee. To address the challenge of ordering guarantee, pFSCK delays certain checks until the prior pipeline pass is complete. For example, the inode checking pass within the pipeline is responsible for creating directory structures used in the directory C/R pass. The directory pass examines subdirectories and checks whether the subdirectory's parent (represented by double dot ..) maps back to the directory. However, because the inode and directory checking passes run in parallel, not all the inodes of the subdirectories would have been checked when the parent directories are checked. For handling the scenarios above, pFSCK delays certification by encapsulating the relationship needing to be verified into task structures that are added to a separate work queue. This task queue is then processed only after all inodes have been checked (e.g., after the inode checking pass completes) as shown in Figure 3. Delayed certifications are infrequent in file-intensive configurations and frequent in the directory-intensive configuration. pFSCK's delayed certification increases concurrency between Pass-1 and Pass-2, consequently improving performance.

To summarize, combining pipeline with data parallelism reduces I/O wait time and improves pFSCK's performance across different file system configurations, as our results show in Section § 6.

5.3 Dynamic Thread Scheduler

C/R runtime can vary significantly depending on the configuration of the file system. For example, C/R on a file system with a larger ratio of smaller files could result in a substantially

longer runtime compared to a file system with few-but-larger files due to more metadata needing to be checked. Similarly, heterogeneity in terms of inode types (files, directories, links) can impact runtime, and the exact configuration remains unknown until the inodes are iterated over in the inode checking pass (Pass-1). Additionally, each pass within C/R has differing degrees of access to shared structures. Therefore, statically assigning threads across each pass could be ineffective. Hence, to adapt to file system configurations, pFSCK implements a C/R-aware scheduler, **pFSCK-sched**, supported by extending the thread pools to allow for migration of threads between the passes. Also, pFSCK-sched maintains an idle thread pool to hold any threads not scheduled to run for any of the passes.

Thread Assignment and Migration of Worker Threads. In pFSCK, we enable dynamic assignment of threads across each pass by implementing a scheduler that actively monitors progress and migrates threads across the passes. The scheduler periodically scans through the work queues of each pass to identify the work distribution ratio across the pipelined passes and uses this ratio to assign threads across them.

Figure 4 shows an example of pFSCK-sched across the first two passes. Initially, all the CPU threads are assigned to the first pass (inode checking pass) given that pFSCK only knows total inodes from the file system superblock and not the types of inodes. When an inode C/R thread identifies a group of directory inodes, it places the directory inodes and their corresponding directory blocks to the work queue of directory C/R pass. If no threads are present in the thread pool used for the directory pass, threads from the inode pass are migrated to the directory pass. To calculate the number of threads to be reassigned, a dedicated scheduler thread finds the total work to be done across all passes using the following model.

Let W_{total} be the amount of work needing to be done. Let q_i be the length of the work queue for pass i. Let n_i be the number of discrete elements needing to be processed for each entry in the work queue. Let w_i be some weight that normalizes the work to be done for each element in pass i. Let C be the core budget and t_i be the number of threads to assign for pass i.

$$W_{total} = \sum_{i=0}^{N} q_i n_i w_i \quad (1)$$

$$t_i = C \cdot q_i n_i w_i \cdot \frac{1}{W_{total}} \quad (2)$$

As shown in Equation (1), the total work to be done is a summation of outstanding work across each pass, which is a product of the work queue length (q_i), the number of objects encapsulated within each queue entry (n_i), and a normalizing weight (w_i). As shown in Equation (2), with the total amount of work needing to be done, the scheduler can determine the ideal number of threads to assign to a pass (t_i) based on the total core budget (C) and the relative amount of work calculated for each pass. Note that the normalizing weights are essential for accounting for the differences in the time to process different file types (directories vs. inodes). Our analysis of different file system configurations (discussed in § 3) shows that in pFSCK (and in e2fsck), the average CPU cycles spent on processing one directory is 1.8x - 2.3x higher than processing an inode, mainly due to directory checksum calculations. Overall, we find it is beneficial to use higher weights for directory checking queues.

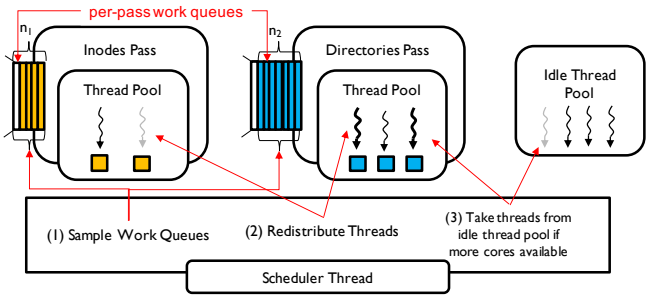


Figure 4: Dynamic Thread Scheduling. A dedicated scheduler thread periodically samples work queues among all the passes and redistributes threads based on the proportion of outstanding work.

5.4 System Resource-Aware Scheduling

File system C/Rs could potentially coexist or even share CPUs with other applications using the same or another file system (or disk). In the pursuit of exploiting parallelism, pFSCK must reduce the impact on other applications. To address this goal, we introduce **pFSCK-rsched**, a system resource-aware pFSCK scheduler.

5.4.1 Efficient CPU Sharing

First, we discuss a case where the C/R runs alongside other applications but performs C/R on a separate, unmounted disk. To reduce the impact of C/R overheads on other applications, pFSCK-rsched maintains a scheduler thread. Initially, the pFSCK-rsched workers are scheduled with SCHED_IDLE priority that mostly schedules a process on any idle CPUs [6]. As the scheduler periodically runs, pFSCK-rsched first determines a CPU core budget to identify the maximum number of threads it could use at any point in time by identifying the number of CPUs in active use across the system, the number of idle cores available, and the number of cores used by pFSCK-rsched. Based on the effective number of cores being used by pFSCK-rsched, pFSCK-rsched increases pFSCK's core budget if it was utilizing less than the available idle cores or shrinks pFSCK's core budget if pFSCK uses more than the idle cores, reducing contention with other applications. After determining the core budget, the scheduler identifies the work ratio across the passes using the per-pass work queues and redistributes an ideal number of threads across each pass. When adding threads to a pass, threads are taken from the idle thread pool and assigned to the per-pass thread pool. If threads need to be removed due to a decrease in the core budget, threads are signaled and reassigned to the idle thread pool. In § 6, we discuss the performance benefits and implications of pFSCK-rsched when co-running and sharing CPUs with another application (RocksDB).

5.4.2 Efficient CPU and File System Sharing

Given the renewed focus for supporting online C/R [32], C/R tools like e2fsck, originally intended for offline use, can be used online with the help of Linux's Logical Volume Manager (LVM). LVM's snapshot feature captures file system state by employing a copy-on-write approach to preserve the original version of modified blocks. [23]. This enables C/R tools to be used in a proactive manner, scanning for pre-existing errors without having to bring the system down.

Towards online C/R with LVM, an empty snapshot volume is first initialized. If any blocks are modified by another application, LVM copies the original blocks to the snapshot before updating the blocks in place on the original volume. When C/R reads blocks that are found to have been modified, the reads are redirected to the snapshot which holds the original blocks. Reads of unmodified blocks are redirected to the original volume. While snapshot initialization is inexpensive, applications incur extra overhead of synchronous data copying and I/O redirection reducing the available storage bandwidth for C/R. This is especially the case when file system blocks are being frequently modified the other application.

In the case of pFSCK, fine-grained parallelism accelerates online C/R even when applications share the same file system (and disks). Futher, pFSCK's resource awareness reduces the impact on co-running applications by reducing CPU (and I/O contention), allowing the application to run faster. We further discuss the benefits of pFSCK for online C/R in § 6.

5.5 Verifying Correctness and Optimizations

Correctness. To ensure the correctness of the C/R, pFSCK with fine-grained parallelism employs a series of steps. First, although the checks are done in parallel, an inode is not marked complete unless prior passes in the pipeline are complete (e.g., a directory inode is marked complete only after all the child inodes (directory entries) are checked. Second, the C/R threads synchronize upon detecting errors. The thread that detects an inconsistency notifies other threads to stall and attempts to fix errors with (e.g., incorrect inode, blocks claimed by multiple inodes) or without user input (e.g., inconsistent bitmap), after which parallel execution is resumed. While tools such as C/Rs allow partial and full checks and checkpoint intermediate states, more robust tools could be added to increase C/R crash-consistency [17].

Optimizations. As additional optimizations to both e2fsck and pFSCK, we restrict the overheads of language localization as discussed in § 3.2, utilize Intel's hardware acceleration for checksum calculations, as well as improve the cache-readahead mechanism. We evaluate the benefits of these optimizations in § 6 (referred to as e2fsck-opt in graphs).

pFSCK support for other file systems and C/R tools. While pFSCK currently extends e2fsck (on Ext file system), the fine-grained inode-level data and pipeline parallelism and efficient scheduling can be applied to other C/R tools, such as xfs_repair and fsck.f2fs that implement multiple passes to check on files, directories, links, and others. We will explore designing a generic C/R in our future work.

Name	Description
e2fsck	original FSCK for EXT file systems
e2fsck-opt	optimized e2fsck
xfs_repair	XFS file system checker
pFSCK	proposed file system checker

Table 2: C/R systems evaluated.

Name	Description
datapara	Only data parallelism enabled
datapara+pipeline-split-equal	Pipeline + data parallelism equally distributing threads across passes
datapara+pipeline-split-optimal	Same as above but manually selects optimal thread assignment
sched	Pipeline + data parallelism with dynamic thread assignment
rsched	Sched configuration with system-level resource-awareness

Table 3: pFSCK incremental system design

6 Evaluation

We evaluate pFSCK to answer the following questions:

- Does pFSCK's data parallelism reduce C/R runtime by increasing CPU parallelism?
- How effective is pFSCK's pipeline parallelism in achieving concurrent execution of C/R passes?
- How effective is pFSCK's dynamic thread placement for different file system configurations?
- Can pFSCK's resource-aware scheduler effectively minimize the performance impact on other applications?
- How does pFSCK perform for online C/R?
- How does pFSCK perform in light of file system errors?

6.1 Experimental Setup

We use a machine equipped with a 64-core Dual Intel® Xeon Gold 5218 running at 2.30GHz, 64GB of DDR memory, a 1TB NVMe, and a 2TB Micron 5200 SATA SSD running Ubuntu 18.04.1. We run pFSCK on various file system configurations with varying thread counts. As seen in Table 2, we compare against vanilla e2fsck, e2fsck-opt (optimized e2fsck with reduced language localization overheads and Intel hardware-accelerated checksumming), and xfs_repair. Table 3 shows pFSCK's incremental design approaches.

File system configurations. The number of files and directories in a file system can be variable and dependent on applications, and there is no publicly available data. Our analysis of workloads like RocksDB (a key-value store), video server, web server, and mail server (using filebench), and two shared servers in our organization show that the file count dominate (99% files to 1% directories). To understand the impact of pFSCK's design on file and directory-intensive configurations, we use a 1TB NVMe with an 840GB file system utilizing 50 million inodes and a 2TB file system on SATA SSD utilizing 100 million inodes. We evaluate pFSCK's impact on a file-intensive (99% files), a medium directory-intensive (25%

directories), and an extreme directory-intensive (50% directories) configuration.

6.2 Data Parallelism

To understand the performance improvements and implications of pFSCK's fine-grained inode-level data parallelism that partitions inodes across threads in each pass but running the passes serially, we evaluate a file-intensive configuration (in Figure 5a), a directory-intensive configuration with 25% directories (in Figure 5b), and an extreme directory-intensive configuration with 50% directories (in Figure 5c). The x-axis shows four C/R approaches: the vanilla e2fsck, our optimized e2fsck (e2fsck-opt), xfs_repair with coarse-grained parallelism, and finally, our proposed pFSCK with data parallelism (pFSCK[datapara]). For xfs_repair and pFSCK[datapara], we also vary the thread counts from 2 to 16 threads.

File-intensive configuration. First, as shown in Figure 5a, our optimized e2fsck-opt outperforms the vanilla e2fsck by optimizing the CRC mechanism and avoiding language localization overheads. Next, xfs_repair parallelizes C/R in the granularity of coarse-grained allocation groups, which is ineffective. There are 16 allocation groups for our XFS filesystem configuration (by default). While xfs_repair checks the sanity of allocation groups in parallel, the directory metadata within the allocation groups are not checked in parallel. Specifically, when files are small, xfs_repair cannot check directory entries and link counts in parallel, with a substantial increase in C/R time. Besides, varying inode counts across allocations groups further impact performance (increasing allocation groups did not improve performance). Both e2fsck and pFSCK outperform xfs_repair for all cases. Finally, pFSCK[datapara] with fine-grained inode-level parallelism, reduces the runtime of the first pass (inode checking) by 2.1x and directory checking pass (Pass-2) by 1.8x, resulting in an overall C/R speedup of 1.9x for four threads over the vanilla e2fsck and 1.52x over e2fsck-opt. Beyond four threads, pFSCK's data parallelism scaling is hindered by high serialization and lock contention to update shared structures such as the used/free block bitmap.

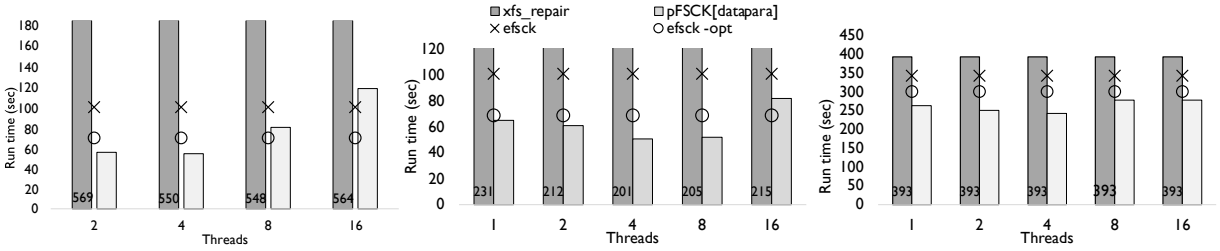
Directory-intensive Configurations. As shown in Figure 5b and 5c, the trends are similar for the directory-intensive file system. Even with 50% directories, pFSCK's Pass-1 and Pass-2 runtime reduces by 1.8x and 1.3x, respectively. pFSCK achieves an overall C/R speedup of 1.4x, 1.24x, and 1.8x over e2fsck, e2fsck-opt, and xfs_repair that does not parallelize directory metadata checking. For the 25% directory-intensive configuration, the gains are 2x over e2fsck. However, the synchronization overheads of global structures prevents pFSCK's data parallelism from scaling beyond 4 cores.

6.3 Pipeline Parallelism and Scheduling.

Next, we evaluate the benefits of combining data and pipeline parallelism and the need for a dynamic thread placement for a file-intensive configuration in Figure 6a and two directory-intensive configurations in Figures 6b and 6c. With pipeline parallelism, C/R passes run concurrently, and the threads of each pass add work for the next pass in a producer-consumer fashion. The x-axis shows the increase in the number of threads used for the C/R. We compare four cases: (1) *pFSCK[datapara]*, which only uses data parallelism running one pass at a time; (2) *pFSCK[pipeline-split-equal]*, which statically divides an equal number of threads for each of the simultaneously executing passes (e.g., two threads are assigned to the inode checking pass (Pass-1) and two threads to directory checking pass (Pass-2) in a 4-thread configuration); (3) *pFSCK[pipeline-split-optimal]*, which represents the best manually selected thread configuration; and (4) *pFSCK-sched*, which employs pFSCK's dynamic thread scheduler to dynamically assign threads based on the amount of outstanding work done within each pass. Single-threaded e2fsck and e2fsck-opt are marked as a baselines. Because e2fsck and pFSCK outperform xfs_repair in all cases, we do not show xfs_repair.

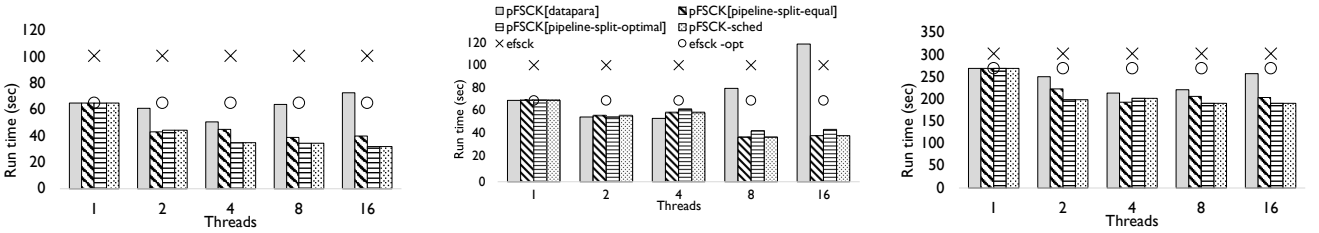
File-intensive Configuration. First, unlike the data parallelism-only approach, the pipeline parallelism approach improves performance when increasing thread count. Next, *pipeline-split-equal* approach splits threads equally across passes. For low thread counts (2 and 4 threads), the performance gains over data parallelism approach is minimal. This is because, for a file-intensive configuration, most work is done in the inode checking pass (Pass-1), static and equal division of threads across passes under-utilizes threads assigned in the directory checking pass. Increasing the thread count (along the x-axis) only marginally improves performance by increasing parallelism in the inode checking pass. In contrast, when employing a manually selected thread configuration using pFSCK's *pipeline-split-optimal* and assigning three-fourth of the threads to the inode checking pass, performance increases by up to 1.3x compared to data parallelism only. The concurrent work across passes also reduces the synchronization cost of data parallelism scaling beyond four threads. Finally, pFSCK's scheduler (*pFSCK-sched*) avoids the tedious manual process of optimal thread placement for different file system configurations by automatically migrating threads based on the relative amount of outstanding work to be completed across each pass. In fact, the dynamic thread placement improves performance by 1.1x compared to *pipeline-split-optimal*, resulting in an overall speedup of 2.6x compared to vanilla e2fsck.

Directory-intensive Configurations. Unlike the file-intensive configuration, for the extreme directory-intensive configuration (50% directories), both inode and directory checking passes demand substantial work, resulting in the need to frequently coordinate across the passes. We observe that automatic thread placement with *pFSCK-sched* controls the number of threads across passes, reducing contention within each pass while thread migration helps in accelerating inode checking (Pass-1) without accumulating a substantial number of directory inodes to be checked (in Pass-2). pFSCK employs delayed certification of directories (directory with subdirectory) as discussed in 5.2.2 which limits scalability.



(a) File-intensive FS. **(b)** Directory-intensive FS with 25% directory. **(c)** Directory-intensive FS with 50% directory.

Figure 5: Data Parallelism impact. The configurations use a total of 50 million inodes.



(a) File-intensive FS. **(b)** Directory-intensive FS with 25% directory. **(c)** Directory-intensive FS with 50% directory.

Figure 6: Comparison of pipeline parallelism and scheduler

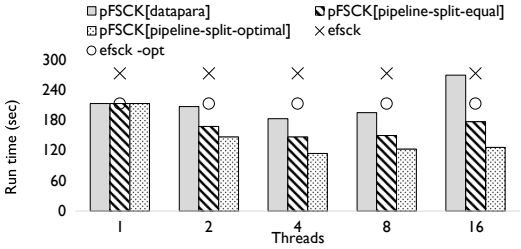
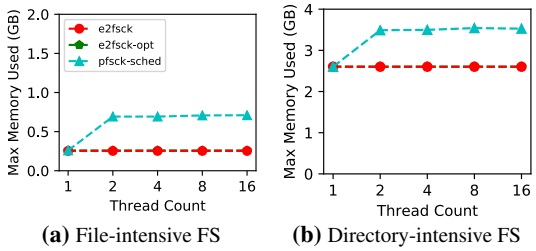
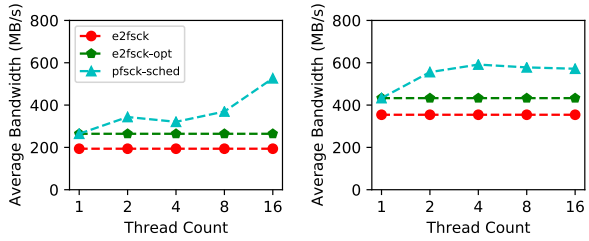


Figure 7: C/R on 2TB SSD with File-intensive FS.



(a) File-intensive FS **(b)** Directory-intensive FS

Figure 9: Memory Usage.



(a) File-intensive FS **(b)** Directory-intensive FS

Figure 8: I/O Bandwidth

Overall, the performance improves by up to 2.7x and 1.6x over e2fsck for 25% and 50% directory-intensive configurations.

Low-bandwidth SSD. Lastly, to understand gains on slower SATA-based SSDs (350MB/s sequential bandwidth) for a large 2TB configuration, Figure 7 shows pFSCK performance on a file-intensive configuration. pFSCK shows speedups of up to 2.1x and 1.73x over vanilla e2fsck and e2fsck-opt despite the lower bandwidth of SSDs compared to NVMe. *In summary, pFSCK's pipeline parallelism reduces serialization bottlenecks of data parallelism, and the dynamic thread placement reduces work imbalance, leading to significant performance gains in fast NVMe and SSD devices.*

6.4 Storage Throughput and Memory Usage

We analyze the effective storage bandwidth use and increase in memory capacity with pFSCK for the file-intensive and extreme directory-intensive (50% directories) configurations. For brevity, we compare single-threaded e2fsck, e2fsck-opt, and multi-threaded pFSCK-sched.

6.4.1 Storage Throughput

Figures 8a and 8b show the storage bandwidth utilization for file-intensive and directory-intensive configurations in MB/s. First, our optimized e2fsck (e2fsck-opt) reduces the overhead between synchronous reads improving bandwidth utilization by 1.3x over e2fsck. In contrast, pFSCK increases I/O throughput for the file-intensive configuration by 1.9x and 2.7x for 8 and 16 threads, respectively, over e2fsck. I/O throughput utilization for directory-intensive file system improves by 1.7x, showing the benefits of pFSCK to utilize available disk bandwidth effectively.

The improvement in I/O bandwidth utilization comes from a combination of an pFSCK's threading, ability to prefetch directory blocks, better caching, and scheduling, which can dynamically migrate threads across passes based on the pending work. For the file-intensive configuration in Figure 8a, the scheduler allows threads to read inode blocks in Pass-1 and migrates extra threads to Pass-2 to read directory blocks in

parallel, bumping up the bandwidth utilization. In Figure 8b, the I/O bandwidth utilization is better with most threads operating in the directory checking pass (Pass-2). However, due to serializing access to global shared structures (e.g., `db_list`) listed in Table 1, I/O bandwidth increase does not translate to higher performance with increasing thread count.

6.4.2 Memory Usage

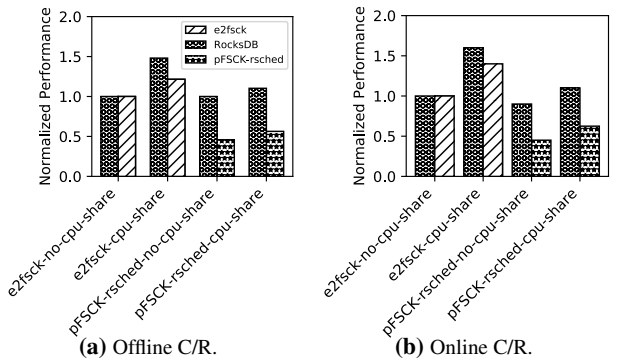
In Figure 9, we compare the memory (DRAM) capacity use. First, for file-intensive configuration in Figure 9a, the overall memory utilization is below 1GB for all approaches. Both e2fsck and e2fsck-opt show the same memory usage. Regarding e2fsck and e2fsck-opt memory utilization, which also applies to pFSCK, the memory use stems from data structures used for tracking directory information such as a `db_list` which hold a list of all directory data blocks, `dirinfo_list`, which tracks relationships between directories, `dx_dirinfo` list which keeps track of all directory HTREE blocks, as well as a dictionary structure used to verify consistency among the dirents within a directory. For *pFSCK-sched*, the memory usage increases by a nominal 300MB (2.1x). pFSCK's memory increase is mainly due to maintaining task queues. Apart from inode checking tasks for Pass-1, the threads discover directories and create directory block tasks for Pass-2 threads to process. Note that each task structure (in the queue entry) represents a fixed-size range of blocks to process. The queues are currently dynamically allocated and unrestricted but can be restricted to reduce memory increase. The range of blocks each task is assigned can also be increased to reduce the number of tasks being generated. Consequently, increasing the thread count does not or marginally increases memory use.

Next, for the directory-intensive configuration in Figure 9b, e2fsck and e2fsck-opt uses 2.6GB of memory. Similar to the file-intensive configuration, the main source of memory consumption is from data structures used for tracking directory information. With an extreme increase in directory count, the memory consumption of these data structures is significantly amplified. pFSCK's memory usage is comparable, only using 3.5GB, resulting in a 1.3x increase. Similar to the file-intensive configuration, the increase in memory usage is due to task structures being generated and added to task queues. With an extreme increase in directory count, the memory overheads of task structures also increase.

In general, memory usage is a function of file system utilization/configuration and not thread count. We argue that pFSCK's performance gains in today's system outweigh the nominal memory increase in today's systems with large memory capacity. Further, we believe memory use can be reduced through pFSCK's code optimizations.

6.5 System Resource-Aware Scheduler

File system C/Rs could run concurrently with other applications, where the C/R and applications can either operate on the same or separate file systems while sharing the same CPUs. To understand the effectiveness of pFSCK's resource-aware scheduler (pFSCK-rsched) in reducing the impact on other applications, we pick a popular multi-threaded and persistent I/O-intensive key-value store, RocksDB [4], which is used as a backend for several real-world applications [22, 43]. We evaluate pFSCK-rsched in an offline setting, where C/R is performed on a file system separate from the file system RocksDB is using, and an online setting, where online (live) C/R is performed on a file system that is concurrently being updated by RocksDB. For both offline and online settings, we evaluate the performance of the following cases: (1) *e2fsck-no-cpu-sharing*, where RocksDB and the vanilla e2fsck do not share CPU cores; (2) *e2fsck-cpu-sharing*, where CPUs are shared between RocksDB and the vanilla e2fsck; (3) *pFSCK-rsched-no-cpu-sharing*, which employs pFSCK-rsched without sharing CPUs with RocksDB; and finally, (4) *pFSCK-rsched-cpu-sharing* which employs pFSCK-rsched sharing CPUs with RocksDB. We run RocksDB with 12 threads and facilitate CPU sharing by running pFSCK-rsched with 12 threads and restricting the affinity of all threads to 16 cores, resulting in the overlapping of 8 cores. Similarly, for e2fsck, we restrict the affinity of all threads to 12 cores, resulting in an overlap of 1 core. Due to space constraints, we show only the results for checking a file-intensive file system configuration.



(a) Offline C/R. **(b)** Online C/R.

Figure 10: Impact of resource-aware pFSCK for offline and online C/R. Results shown for file-intensive configuration.

6.5.1 Offline C/R with CPU Sharing.

Figure 10a shows the offline approach performance using separate file systems for each C/R and RocksDB. The x-axis shows e2fsck and pFSCK-rsched approaches without and with CPU sharing. In the y-axis, the results are normalized to the performance of e2fsck running with RocksDB without sharing CPUs (*e2fsck-no-cpu-sharing*).

First, when sharing CPUs, the runtime of vanilla e2fsck and RocksDB is significantly impacted (shown as *e2fsck-cpu-share*) compared to *e2fsck-no-cpu-share* due to frequent context switches which take away effective CPU time from RockDB; e2fsck's performance degrades by 1.2x and RocksDB's performance degrades by 1.5x compared to the no-sharing approach. In contrast, with pFSCK's resource-aware scheduler, CPU sharing between the pFSCK-rsched and RocksDB (*pFSCK-rsched-cpu-sharing*) has minimal impact for both pFSCK and RocksDB. The resource-awareness

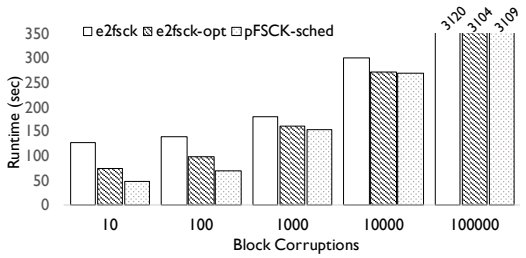


Figure 11: Repair runtime for varying corruption count.

capability adaptively downscales the number of threads being utilized to carry out C/R, reducing CPU context switches away from RocksDB and minimizing related overheads. Athough pFSCK-rsched and RocksDB intially overlap 8 out of the 16 cores, pFSCK-rsched is able to downscale threading to around 4-6 threads, allowing RocksDB to consistently utilize 12 out of the total 16 cores. As a result, pFSCK-rsched and RocksDB show minimal performance degradation of 1.07x and 1.05x, respectively, compared to the no CPU sharing case.

6.5.2 Online C/R with CPU Sharing.

Figure 10b shows the results when each C/R and RocksDB share the CPU as well as the file system. As discussed earlier in 5.4.2, pFSCK utilizes the LVM-based snapshots to capture file system changes and perform C/R on a stable version of the file system represented by the snapshot. Similar to offline C/R evaluation, we normalize the results to the performance of e2fsck running with RocksDB without sharing CPUs.

First, when overlapping e2fsck and RocksDB, performance significantly degrades by 1.4x and 1.6x, respectively. The degradation is mainly due to frequent CPU context switching between e2fsck and RocksDB. However, this main source of performance degradation increases the time the snapshot must remain active, resulting in further performance degradation due to LVM snapshot overheads. Next, with pFSCK-rsched, the performance degradation when co-running pFSCK-rsched with RocksDB is minimal. This is due to pFSCK-rsched's resource-aware thread assignment (similar to offline setting) which mitigates performance impact and context switching overheads by scaling the number of threads pFSCK uses. Because performance impact from context switching is minimized, the amount of time the snapshot must active is minimized, mitigating any further performance degradation due to LVM snapshot overheads. The performance degradation compared to the baseline (no CPU or file system sharing) for both pFSCK-rsched and RocksDB is 1.2x. Although higher than the offline approach, most of it is due to disk sharing and the resulting LVM snapshotting overheads.

Summary. pFSCK-rsched's resource awareness effectively adapts to the number of available CPU cores (and threads) for C/R and maximizes their utilization for better performance in both an offline and online setting. The performance impact on co-running application is also minimized.

6.6 Performance with Errors

To evaluate pFSCK's performance with file system errors, we use e2fsprogs's fuzzing tool, *e2fuzz*, to introduce random block corruptions to a file-intensive configuration. In Figure 11, we introduce up to 100K corruptions in the x-axis and compare e2fsck, e2fsck-opt, and pFSCK-sched that uses eight threads. Note that prior studies real-world systems show that the scale of corruptions can be just a few bits or bytes, and the hardware and software corruptions could significantly vary, ranging from silent bit corruptions to FTL metadata corruptions and shorn or incomplete writes [12, 18, 27].

First, even for 100 corruptions, pFSCK-sched speeds up C/R by up to 2.7x and 1.6x over e2fsck and e2fsck-opt, respectively. However, for 10K corruptions, pFSCK-sched performs similarly to e2fsck-opt and speeds up C/R by only 1.1x compared to e2fsck. pFSCK's speedup reduces with increasing corruption counts because long and serially executed error-fixing operations start to dominate the overall runtime for all C/Rs including pFSCK. Further, for pFSCK-sched, we observe that if the corruption count is greater than 10K, synchronization overheads start to further diminish performance gains from parallelism. To mitigate diminishing returns from thread synchronization, pFSCK tracks the number of errors encountered and reverts to serial checking after discovering a 10K errors. This allows pFSCK to perform similarly to e2fsck-opt for higher error counts, experiencing only slight performance deterioration due to initial thread synchronization. Our future work will focus on exploring ways to parallelize fixes to accelerate C/R for highly corrupted file systems.

7 Conclusion & Future Work

With a goal of accelerating file system checking and repair tools, we propose pFSCK, a parallel C/R tool that exploits CPU parallelism and the high bandwidth of modern storage devices to accelerate C/R time without compromising correctness. pFSCK explores fine-grained parallelism by assigning threads to inodes, blocks, or directories and efficiently performing C/R using data parallelism within each pass and pipeline parallelism across multiple passes. In addition, pFSCK enables efficient thread management techniques to adapt to varying file system configurations as well as minimize performance impact on other applications. As a result, pFSCK shows more than 2.6x gains over e2fsck and 1.8x over xfs_repair that provides coarse-grained parallelism. In light of pFSCK's limitations, future work will explore accelerating pFSCK for hard disks while mitigating costly seeks due to random accesses, reducing memory overheads through more efficient data structures and rate-limiting, and finally accelerating fixes for disks with higher corruption counts.

Acknowledgements

We thank the anonymous reviewers and Dean Hildebrand (our shepherd) for their insightful comments and feedback. We thank the members of Rutgers Systems Lab for their valuable input. This material was partially supported by funding from NSF grant CNS-1910593. We also thank Rutgers Panic Lab for helping with the storage infrastructure.

References

[1] Disk check takes too long to check. linuxquestions.org. https://www.linuxquestions.org/questions/linux-hardware-18/disk-check-takes-too-long-to-check-510584/.

[2] e2fsck: fsck for ext4. https://linux.die.net/man/8/e2fsck.

[3] e2scrub: online fsck for ext4. https://lwn.net/Articles/749106/.

[4] Facebook RocksDB. http://rocksdb.org/.

[5] Intel-Micron Memory 3D XPoint. http://intel.ly/1eICR0a.

[6] Linux sched() man page. http://man7.org/linux/man-pages/man7/sched.7.html.

[7] StackExchange - Extremely long time for an ext4 fsck. https://unix.stackexchange.com/questions/78785/extremely-long-time-for-an-ext4-fsck, Mar 2013.

[8] File system check (fsck) is slow and running for a very long time. https://access.redhat.com/solutions/2210281, Sep 2016.

[9] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 353–369, 2019.

[10] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated crash vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 151–167, Berkeley, CA, USA, 2016. USENIX Association.

[11] William (Bill) E. Allcock. Parallel File Systems at HPC Centers: Usage,Experiences, and Recommendations. https://www.nersc.gov/assets/Uploads/W01-DataIntensiveComputingPanel.pdf.

[12] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *ACM Trans. Storage*, 4(3), November 2008.

[13] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *ACM Transactions on Storage (TOS)*, 4(3):8, 2008.

[14] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS 07*, 2007.

[15] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243. ACM, 2013.

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

[17] Om Rameshwar Gatla, Muhammad Hameed, Mai Zheng, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojević, Cyril Guyot, and Robert Mateescu. Towards robust file system checkers. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 105–122, Oakland, CA, February 2018. USENIX Association.

[18] John Goerzen. Silent data corruption is real. https://changelog.complete.org/archives/9769-silent-data-corruption-is-real/.

[19] Haryadi S Gunawi, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Sqck: A declarative file system checker.

[20] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sqck: A declarative file system checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 131–146, Berkeley, CA, USA, 2008. USENIX Association.

[21] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 1–14, Oakland, CA, 2018. USENIX Association.

[22] Ethan Hamilton. Rocksdb is eating the database world. https://rockset.com/blog/rocksdb-is-eating-the-database-world/.

[23] Michael Hasenstein. The logical volume manager (lvm). *White paper*, 2001.

[24] Val Henson, Zach Brown, and Arjan van de Ven. Reducing fsck time for ext2 file systems. 04 2019.

[25] Val Henson, Amit Gud, Arjan van de Ven, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Proceedings of the Second Conference on Hot Topics in System Dependability*, HotDep'06, pages 7–7, Berkeley, CA, USA, 2006. USENIX Association.

[26] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *HotDep*, 2006.

[27] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 783–798, Renton, WA, July 2019. USENIX Association.

[28] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 783–798, 2019.

[29] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 783–798, Renton, WA, July 2019. USENIX Association.

[30] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[31] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redesigning lsms for nonvolatile memory with novelsm. In Haryadi S. Gunawi and Benjamin Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 993–1005. USENIX Association, 2018.

[32] Ram Kesavan, Harendra Kumar, and Sushrut Bhowmik. WAFL iron: Repairing live enterprise file systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 33–48, Oakland, CA, February 2018. USENIX Association.

[33] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[34] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, Santa Clara, CA, 2015.

[35] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. OOPSLA '09, page 227–242, New York, NY, USA, 2009. Association for Computing Machinery.

[36] W. Li, Y. Yang, J. Chen, and D. Yuan. A cost-effective mechanism for cloud data reliability management based on proactive replica checking. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 564–571, 2012.

[37] HPC-Users Mailing List. Outages in HPC Systems. https://maillists.uci.edu/pipermail/hpc-users/2019-December/000095.html.

[38] M. Lu, T. Chiueh, and S. Lin. An incremental file system consistency checker for block-level cdp systems. In *2008 Symposium on Reliable Distributed Systems*, pages 157–162, Oct 2008.

[39] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Marshall Kirk Mckusick. Ffsck: The fast file-system checker. *Trans. Storage*, 10(1):2:1–2:28, January 2014.

[40] Marshall K. McKusick. Improving the performance of fsck in freebsd. *;login:*, 38(2), 2013.

[41] Marshall Kirk McKusick, Willian N Joy, Samuel J Leffler, and Robert S Fabry. Fsck- the unix† file system check program. *Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version*, 1986.

[42] Mtanski. mtanski/xfsprogs github.com/mtanski/xfsprogs/preadv2/repair. https://github.com/mtanski/xfsprogs/tree/preadv2/repair, Feb 2015.

[43] Arjun Narayan and Peter Mattis. Why we built cockroachdb on top of rocksdb. https://www.cockroachlabs.com/blog/cockroachdb-on-rocksd/.

[44] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[45] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 145–160, Berkeley, CA, USA, 2018. USENIX Association.

[46] Omar Sandoval. A survey of bugs in the Btrfs filesystem. https://courses.cs.washington.edu/courses/cse551/15sp/projects/osandov.pdf.

[47] Ric Wheeler. fs_mark. https://sourceforge.net/projects/fsmark/.

[48] Matthew Wilcox and Ross Zwisler. Linux DAX. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[49] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, Santa Clara, CA, 2016.

[50] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, 2016.

[51] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. Reliability analysis of ssds under power fault. *ACM Trans. Comput. Syst.*, 34(4):10:1–10:28, November 2016.