

pFSCK: Accelerating Filesystem Checking and Repair for Modern Storage

David Domingo, Sudarsun Kannan

Rutgers University Department of Computer Science



RUTGERS

UNIVERSITY | NEW BRUNSWICK

Storage Issues: A Situation



```
root@ecmint:~# e2fsck -fn /dev/sdb1
e2fsck 1.42.12 (29-Aug-2014)
Warning! /dev/sdb1 is mounted.
Warning: skipping journal recovery because doing a read-only filesystem check.
Pass 1: Checking inodes, blocks, and sizes
Deleted inode 405212 has zero dtime. Fix? no

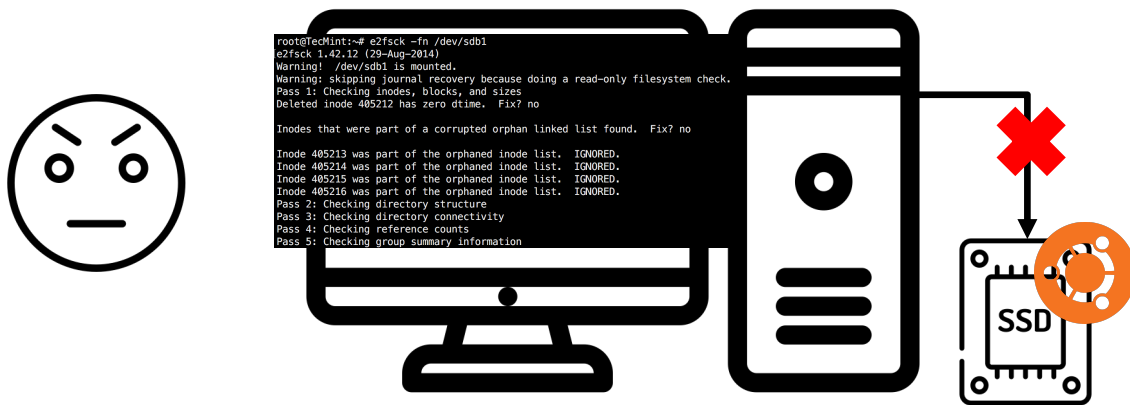
Inodes that were part of a corrupted orphan linked list found. Fix? no

Inode 405213 was part of the orphaned inode list. IGNORED.
Inode 405214 was part of the orphaned inode list. IGNORED.
Inode 405215 was part of the orphaned inode list. IGNORED.
Inode 405216 was part of the orphaned inode list. IGNORED.
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
```



- You frequently use your personal computer, home server, or production server
- System won't boot after a reboot due to update, restart, or crash
- Forced to run file system checker (ex. e2fsck for EXT4)
- Takes a long time to complete resulting in large downtime and decreased productivity

Storage Issues: Faster Storage



- Utilize faster modern storage to reduce runtime and make things easier
- Flash wear increases over time and errors inevitably come up
- Still a hassle and takes long (why?)
- Should run the file system checker proactively to find errors at the cost of availability

Storage Issues: Not a Coincidence

- Frustrating usability is not a rare occurrence
- File system checking has shown to be notoriously slow

The image displays three overlapping screenshots from different online forums, all discussing the issue of slow file system checks (fsck).

- Top Screenshot (askubuntu.com):** Shows a question titled "e2fsck taking a long time". The question was asked 4 years, 11 months ago, is active, and has been viewed 946 times. The user's text includes: "I have a HP Z420 desktop with a USB 3.0 that's giving trouble and I am".
- Middle Screenshot (UNIX & LINUX):** Shows a question titled "Extremely long time for an...". It was asked 6 years, 8 months ago and is active. The user's text includes: "The problem that I am having is the extra... have thoroughly made searches on Google... would resolve the problem." and "The command that I am running is 'sudo'".
- Bottom Screenshot (Red Hat Customer Portal):** Shows a knowledgebase article titled "File system check (fsck) is slow and running for a very long time". The article is marked as "SOLUTION UNVERIFIED" and was updated on September 15, 2016, at 7:44 AM. The issue description states: "A file system check operation (fsck) takes very long to finish and/or it seems to be hanging or stuck".

Storage Solution: Faster checking with pFSCK

- pFSCK provides faster checking runtimes
 - Parallelizes file system checkers at a fine granularity (e.g. inodes)
 - Ensures correctness through logical reordering
 - Adapts to file system configurations with dynamic thread scheduling
- Shows up to **2.6x** improvement over vanilla e2fsck (EXT checker) and **1.8x – 8x** improvement over xfs_repair (XFS checker)

Outline

Introduction

Background







Motivation

pFSCK Design

Evaluation

Summary

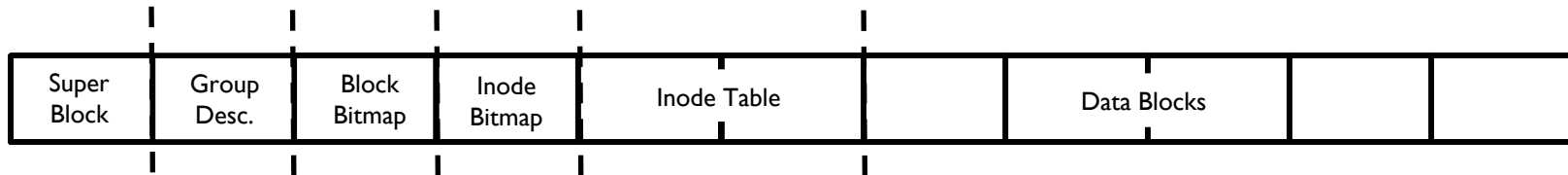
Background: Providing Storage Reliability

- File Systems Checkers
 - Typically used after kernel/security upgrades and crashes
 - Provides ultimate data reliability and recovery  unoptimized to exploit CPU/disk parallelism
- Utilize Modern Storage
 - Larger bandwidth and lower latencies 
 - Provides faster scanning and checking
- Utilize Modern Consistency Mechanisms
 - Journaling 
 - Copy on Write 
 - Erasure Coding  Reconstruction and re-sharding quite time consuming
 - Replication  Not always economically feasible

Traditional file system checking is still relevant!

Background: Disk Layout and Checking

- Linux EXT block group layout



- e2fsck** (EXT checker) scans through all file system metadata
- builds own view of file system in order to detect and fix inconsistencies
- Consists of 5 logical passes:
 - Inodes Pass**: Checks inodes (file and directory inodes)
 - Directories Pass**: Checks directories
 - Connectivity Pass**: Checks file reachability
 - Ref Counts Pass**: Verifies link counts for all files
 - Cylinders Pass**: Verifies cylinder group information

Background: Prior Works

- **e2fsck**: Parallelizes file system checking across disks/partitions
 - **xfs_repair** : Parallelizes file system checking across allocation groups
 - **FFsck (FAST '13)**:
 - Modifies file system and rearranges metadata blocks
 - Provides faster scanning by rearranges metadata blocks
 - **ChunkFS (HotDep '06)**:
 - Partitions file system into smaller isolated groups
 - Allows groups to be repaired in isolation
 - **SQCK (OSDI '08)**:
 - Uses declarative queries and databases for consistency checks
 - Allows for more expressive fixes with comparable run times
- parallelization limited to coarse granularity
- requires extensive modification to the file system
- requires complete overhaul of file system checker
-

Outline

Introduction

Background

Motivation

pFSCK Design

Evaluation

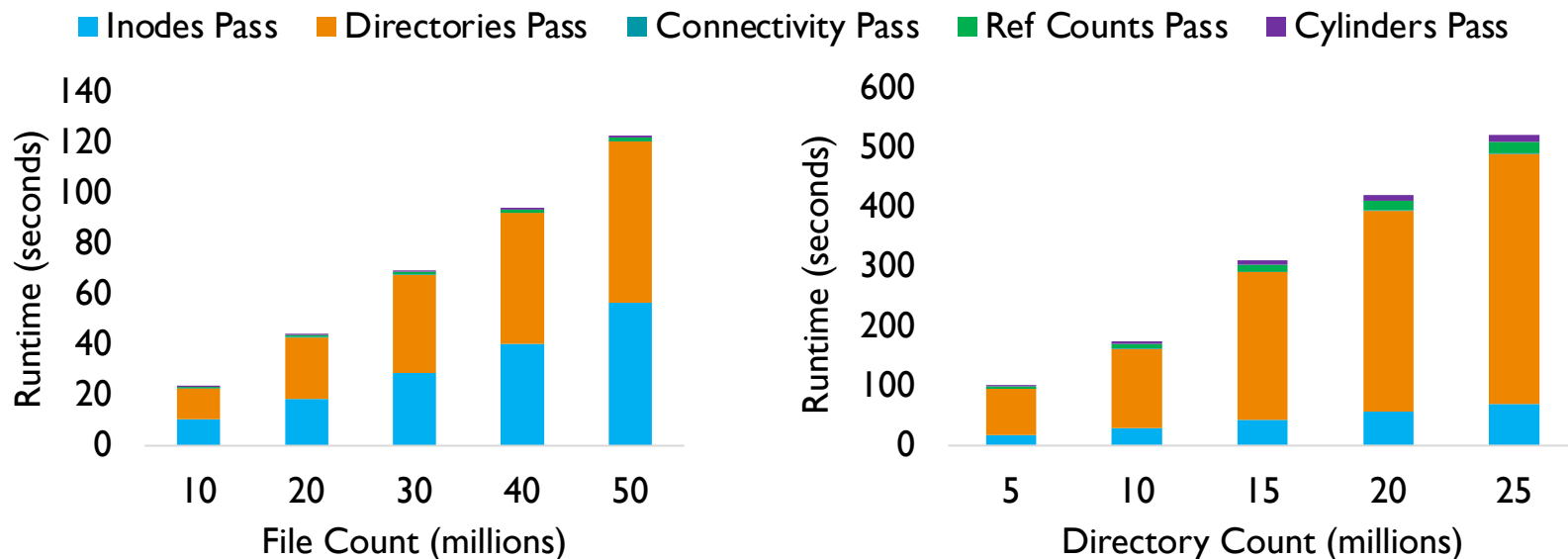
Summary

Evaluating Current e2fsck Performance

- System:
 - Dual Intel® Xeon® Gold 5218 @ 2.30GHz
 - 64 GB of memory
 - 1TB NVMe Flash Storage

- Methodology:
 - e2fsck against 840 GB file systems of varying configurations
 1. Varying file count (file size constant at 12kb, created across 5 directories)
 2. Varying directory count (1 file per directory, each file 24kb)

File System Sensitivity



- Majority of time is spent checking inodes and directory metadata
- Directory-intensive file systems take significantly longer than a file-intensive file system
- **Runtime scales linearly with file system utilization**

Research Questions

- How to speed up file system checking and repair without compromising correctness?
- How to adapt for different file system configurations?
 - ex. file-intensive vs directory-intensive

pFSCK Key Ideas

- Parallelize file system checking at finer granularity (ex. inodes, directories)
 1. Overlap as much independent logical checks within each pass
 2. Overlap as much logical checks across passes
 3. Reduce contention on shared data structures
 4. Efficient management of work for threads across passes

Outline

Introduction

Background

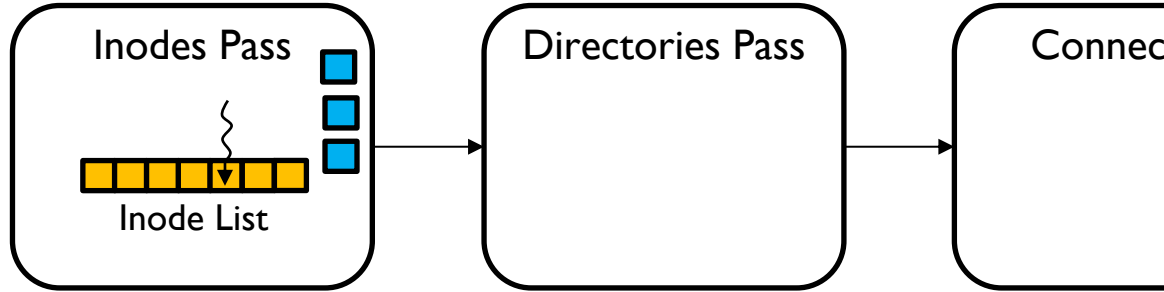
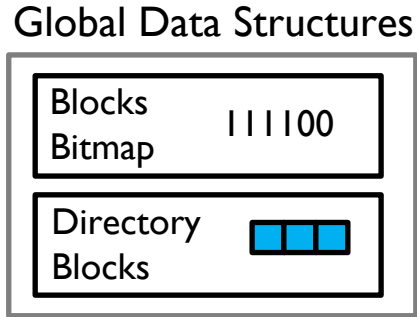
Motivation

pFSCK Design

Evaluation

Summary

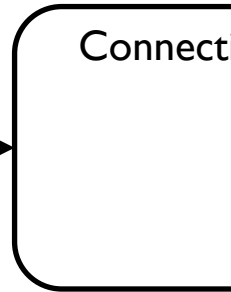
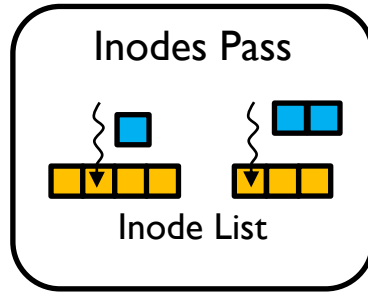
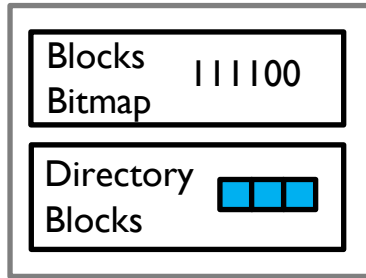
Serial Execution in e2fsck



- Serially checks file system metadata (ex. inodes)
- Updates global data structures to generate view of file system
- Generates work for the next pass (ex. list of directory block)

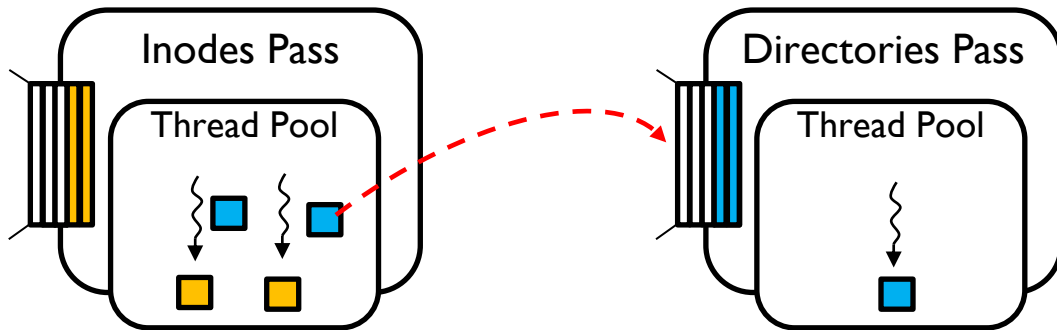
pFSCK Data Parallelism

Global Data Structures



- Split metadata within each pass into smaller groups
- Uses a pool of threads to check in parallel and generate intermediate lists
- Aggregate lists and repeat
- Critical, unisolated data structures limits potential concurrency (e.g. block bitmap)

pFSCK Pipeline Parallelism

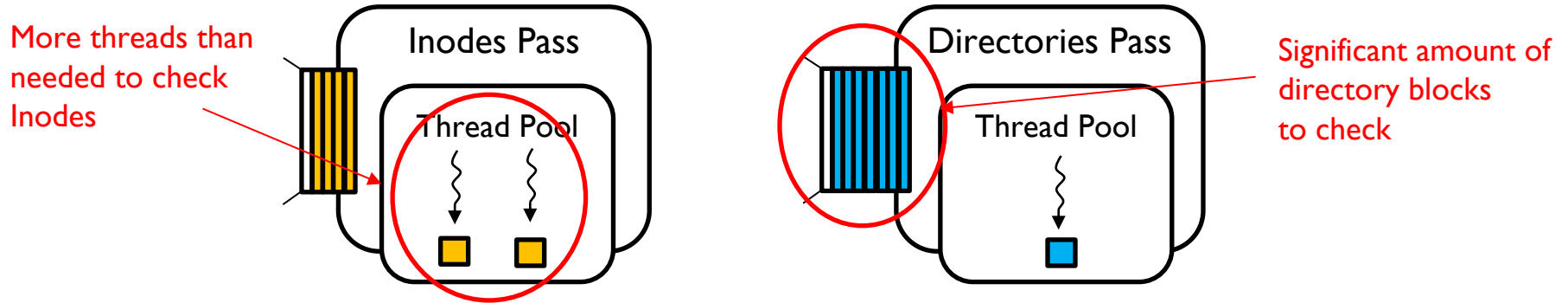


- Allow multiple passes to operate in parallel to hide synchronization bottlenecks
- Turn each pass into independent flows of execution
- Use per pass queues and thread pools
- Continuously feed subsequent passes with metadata
- Do not wait for previous pass to complete (**speculatively carry out future checks**)

Pipeline Parallelism: Work Imbalance

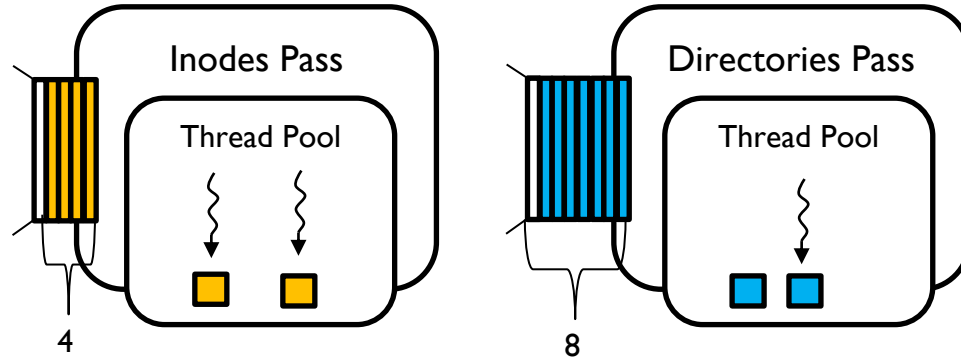
- Differing metadata densities causes uneven pass queues
- Not straight forward how many threads to assign to each pass
- Example:

Directory-intensive file system mainly will have more directory blocks to check



Solution: Dynamic Thread Scheduling

Scheduler Thread	
Total Threads =	3
Total Work =	12
Inodes Proportion =	4 / 12
Dirs Proportion =	8 / 12
Node Threads =	1
Dir Threads =	2



- Scheduler thread periodically samples the task queue lengths of each pass
- Calculates relative work among the passes and redistributes threads
- Allows pFSCCK to adapt to different file system configurations with differing metadata densities

More in Paper

- Resource-Aware Scheduling for Online Checking Support
- Error handling
- Delayed dependent checks
- Other optimizations

pFSCK: Accelerating File System Checking and Repair for Modern Storage
David Donaghy, Sudeep Kumar
Rutgers University

Abstract

We propose and design pFSCK, a parallel file system checking and recovery (CRR) tool designed to exploit compute and storage parallelism in modern storage devices. pFSCK achieves the greatest parallelism at the granularity of tracks and directory blocks without impacting the CRR's consistency. pFSCK has capacity data parallelism by simulating bandwidth operations at each step of the checking logic and fine-tuning dependent operations and shared data structures. However, full utilization of shared structures is infeasible and requires specialized apertures. To reduce maintenance headaches, pFSCK introduces pipeline parallelism, allowing multiple steps of CRR to run concurrently without impacting consistency. Further, pFSCK provides per-device I/O cache management, a custom thread placement across CRR stages, and a resource aware scheduler to reduce the impact of CRR on other applications during CPU and file system. Evaluation of pFSCK shows more than 2.6x gain over cfsck for file system CRR and more than 1.8x over XFS's CRR that provides great parallelism.

1 Introduction

Modern data tier storage devices such as SSDs, NVMe, and byte-addressable NVM storage technologies offer higher bandwidths and lower latency than hard-disk based backup, backup operations for exploiting CRR parallelism. While the I/O access performance has increased, storage hardware and software access have continued to grow coupled with server and enterprise high performance designs impacting file system reliability [15, 16, 32, 40]. For decades, the system checking and repair tools (referred to as CRR hereafter) have offered a novel role to reduce the reliability of software storage and availability of systems by identifying and correcting system inconsistencies [9].

A significant body of prior work has shown that, in the event of a system crash or failure in data centers, CRR are typically used to fix the immediate problem to system recovery. Prior work [11, 27] and directory based apertures [12, 28] on servers and IT cores of organizations show the CRR are not across system scenarios. This includes problems during recovery that are hardware or software errors [11, 27], periodic maintenance, or during secondary security upgrades [3]. When CRR are run as a check parallel to an offline backup, the operator's data is unavailable. Some CRR support online checking, but it is a manual task that does not parallel with other operations that run on the same devices. Thus, improving CRR performance and flexibility is critical for system availability and reducing performance impact on other applications.

File system CRR tools work by identifying and fixing the structural inconsistencies of the system scenarios. The inconsistencies could be in locale, data and inode linkage, links, or directory entries, corruption, MFT/bitmap and widely used tools such as cfsck [38] system checker for Ext4 [2] derive CRR error message logs (generally referred to as errors), with each pair representing the checking file system structure (e.g., directory, file, inode). However, CRR are known to be extremely slow, showing a linear increase in CRR time with an increase in file and directory count [10, 19–21], at times taking hours [17] or even weeks [11]. Addressing such task and NVMe based gaps provide from latency and bandwidth, current CRR tools fail to exploit such hardware capabilities to exploit CPU parallelism. While modern CRRs have attempted to increase parallelism, they adopt coarse-grained approaches, such as parallelizing CRR across logical volumes or logical drives, which are inefficient to accelerate CRR on file systems with data imbalance across logical drives [20, 30, 31].

To overcome such limitations, we propose pFSCK, a parallel CRR that exploits CPU parallelism and modern storage's high bandwidth to accelerate file system CRR in off-line and on-line form, thereby reducing system downtime and improving data and system reliability and availability [10, 20, 31, 39]. While pFSCK leverages ideas from prior work parallelism research [5, 45], it must solve several challenges specific to CRR, which include balancing consistency in the presence of complex file system layout and design of the system structure (e.g., minimal linkage) without impacting correctness, adapting to other applications of pFSCK, introducing the agent parallelism, i.e., parallelism at the granularity of tracks and directory blocks, resulting in a significantly faster execution than traditional CRR. pFSCK leverages data parallelism by breaking up the work done at each step, subdividing data structure to be available, and allowing multiple threads to perform checks in parallel. Although data parallelism accelerates checks, logic, and global data structures (linkages) within each step are designed to match the file system's topology [6]. Much thought in Ext4 file system and must be implemented to ensure checking correctness. As a result, with accompanying hardware, the overall implementation and installation are quickly enough to the performance gains. Hence, pFSCK introduces pipeline parallelism to parallelize CRR along with the highest available parallelism.

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

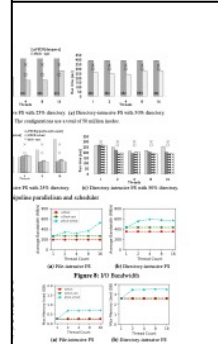


Figure 10: Memory Usage. Here, we compare single-threaded cfsck, cfsck-ctl, and multi-threaded pFSCK-ctrl.

4.4.4 Storage Throughput

Figure 11 and 13 show the storage bandwidth utilization for the metadata and directory structure operations on the file. First, our optimized cfsck-ctl check up to when the metadata structure checks was made impacting bandwidth utilization by 1.3x over cfsck. In contrast, pFSCK increases I/O throughput for the file operation and operation by 1.9x and 2.3x for 4 and 16 tracks, respectively, over cfsck-ctl throughput utilization for directory across the various operations by 1.7x, showing the benefits of pFSCK in utilizing available disk bandwidth effectively.

The response to I/O bandwidth utilization across from

6.4 Storage Throughput and Memory Usage

We analyze the effective storage bandwidth use and its impact to memory capacity with pFSCK on the file system and compare directory intensive I/Os directed towards

Outline

Introduction

Background

Motivation

pFSCK Design

Evaluation

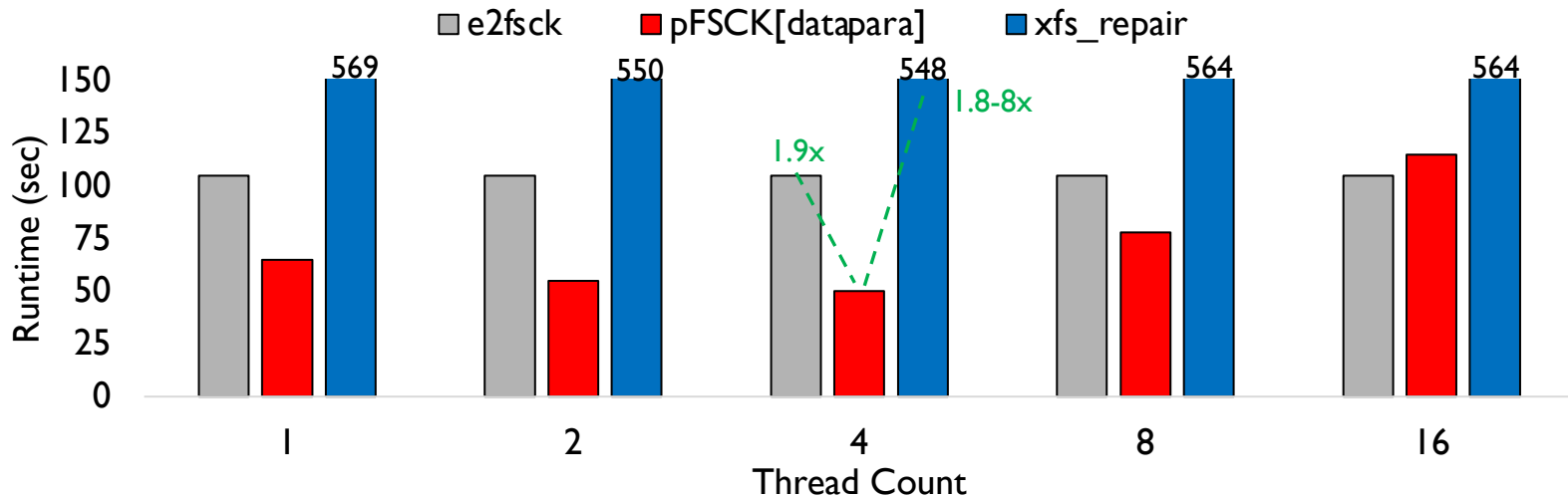
Summary

Methodology

- System:
 - Dual Intel® Xeon® Gold 5218 @ 2.30GHz, 64GB of DDR memory, 1TB NVMe Flash Storage
- Tools:
 - e2fsck (e2fsprogs release v1.44.4)
 - xfs_repair (xfsprogs release 4.9.0)
 - pFSCK (our system)
- File System Configurations:
 - File-Intensive FS (99% files to 1% directories)
 - Directory-Intensive FS (50% files to 50% directories) (see paper)

Evaluation: Data Parallelism

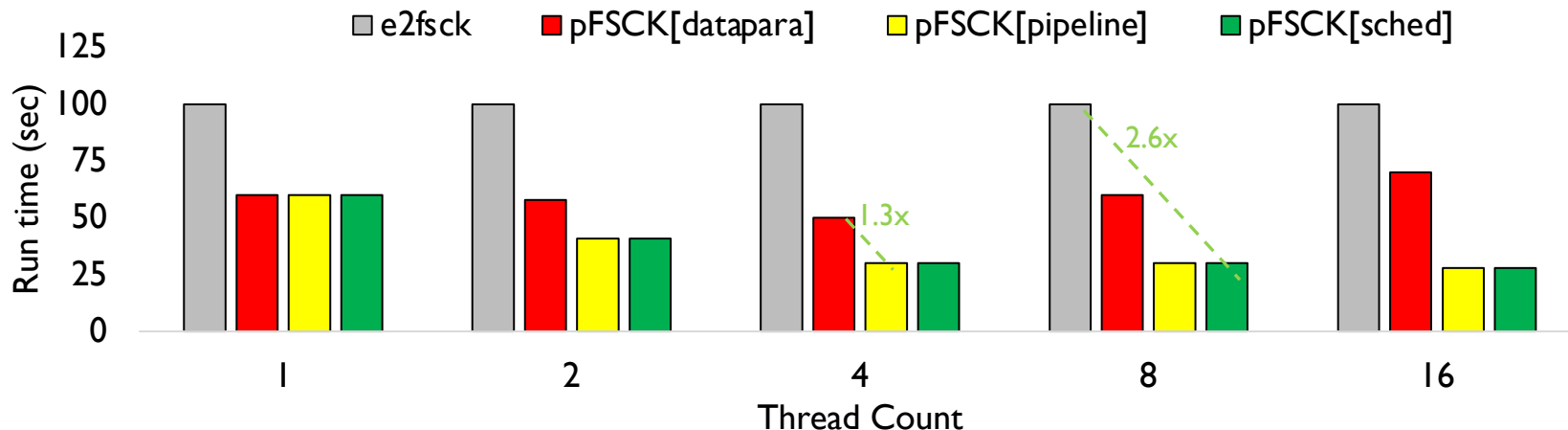
- pFSCK's data parallelism compared to vanilla e2fsck and xfs_repair



- Improves performance over vanilla e2fsck by up to **1.9x** (4 Threads)
- Improves performance over xfs_repair by up to **1.8x – 8x** with same thread count
- Contention on shared structures limits data parallelism scaling

Evaluation: Pipeline Parallelism and Dynamic Thread Scheduler

- Pipeline parallelism and dynamic thread scheduling compared to just data parallelism



- Pipeline parallelism increases performance by **1.3x** over just data parallelism
- Dynamic Thread Scheduler automatically identifies optimal thread assignment
- pFSCK[sched] automatically improves performance by up to **2.6x** over vanilla e2fsck

More in Paper

- More File System Configuration Analysis
- Memory Usage and I/O Throughput Analysis
- Resource-Aware Scheduling Performance
- Performance on SSD
- Error Fixing Performance

pFSCK: Accelerating File System Checking and Repair for Modern Storage
David Donaghy, Sudeep Kumar
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 achieves the greatest parallelism at the granularity of tracks and directory blocks without impacting the C/R consistency. pFSCK has capacity data parallelism by mounting redundant operations at each step of the checking logic and fine-tuning, sequential operations, and shared data structures. However, full utilization of shared resources is infeasible and requires specialized schemes. To reduce maintenance overheads, pFSCK introduces pipeline parallelism, allowing multiple steps of C/R to run concurrently without impacting consistency. Further, pFSCK provides per device I/O cache management, a custom thread placement across C/R steps, and a resource aware scheduler to reduce the impact of C/R on other applications during C/R and the system. Evaluation of pFSCK shows more than 1.6x gain over cfsck for file system C/R and more than 1.8x over XFS_C/R that provides coarse grained parallelism.

1 Introduction

Modern data center storage devices such as HDDs, NVMe, and byte-addressable NVM storage technologies offer higher bandwidths, capacities and lower latency than hard-disk providing better opportunities for exploring C/R parallelism. While the I/O access performance has increased, storage hardware and software access have continued to grow coupled with server and enterprise high performance designs impacting file system reliability [15, 12, 13, 45]. For decades, file system checking and repair tools (referred to as C/R) have been developed around a need to reduce the reliability of software storage and availability of systems by identifying and correcting the system inconsistencies [6].

A significant body of prior work has shown that, in the event of a system crash or failure in data centers, C/R are typically used in the hot standby relative to system recovery. Prior work [11, 27] and observations show that C/R are run across system reboots. This includes problems during reboot that are hardware or software errors [11, 27], periodic maintenance, or during mandatory security updates [3]. When C/R are run as a check parallelism is an efficient feature, the parallelism is infeasible. Some C/R support online checking, but it is unclear that they do not conflict with other applications that use the same devices. 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 the system metadata. The inconsistencies could be in locale, data and locale linkage, links, or directory entries, corruption, MFT/blocks and widely used tools such as cfsck [38] or system checker for Ext4 [2] divide C/R across multiple steps (generally referred to as phases), with each pass responsible for checking a file system structure (e.g., directories, files, links). However, C/R are known to be extremely slow, showing a linear increase in C/R time with an increase in file and directory count [16, 19–21], at times taking hours [17] or even weeks [11]. Addressing this task and NVMe based pipes provide better latency and bandwidth, current C/R tools fail to exploit such hardware capabilities to exploit 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 drives, which are ineffective to accelerate C/R on file systems with data redundancy across logical drives [20, 16, 45].

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 off-line and on-line forms, thereby reducing system downtime and improving data and system reliability and availability [10, 20, 11, 29]. While pFSCK leverages ideas from prior work parallelism research [15, 45], it must solve several challenges specific to C/R, which include maintaining consistency in the presence of complex file system layout and structures (e.g., mirrored links) without impacting operations, and reducing C/R impact on other applications. pFSCK introduces the greatest parallelism, i.e., parallelism at the granularity of tracks and directory blocks, resulting in a significantly faster execution than traditional C/R. pFSCK has multiple data parallelism by breaking up the work done at each pass, sub-dividing data checks into a pipeline, and allowing multiple threads to perform checks in parallel. Although data parallelism accelerates checks, high throughput and pFSCK data structures (linkages) within each pass are designed to match the file system's topology (e.g., MFT, block pointers, and Ext4 file pointers) and must be implemented to ensure checking correctness. As a result, with accompanying hardware, the overall implementation and installation can be quickly enough to performance gains. Hence, pFSCK introduces pipeline parallelism to parallelize C/R along with the highest level of data access parallelism.

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

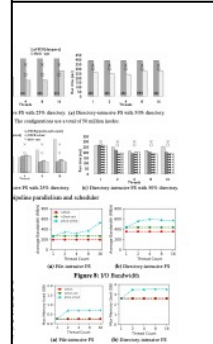


Figure 10: Storage Throughput
Here, we compare single-threaded cfsck, cfsck-opt, and multi-threaded pFSCK on a 4.4.4 Storage Throughput

Figure 10 and 11 show the storage bandwidth utilization for the metadata and directory structure operations on an MBfs. For an optimized cfsck check-opt when the workload is heavy, cfsck-opt can reach a throughput bandwidth utilization by 1.3x over cfsck. In contrast, pFSCK increases I/O throughput for the file consistency check by 1.9x and 2.1x for 4 and 16 tracks, respectively, over cfsck. I/O throughput utilization for directory access for the storage improves by 1.7x, showing the benefits of pFSCK in utilizing available disk bandwidth efficiently.

The response to I/O bandwidth utilization comes from

6.4 Storage Throughput and Memory Usage

We analyze the effective storage bandwidth use and its impact on memory capacity with pFSCK on the file metadata and content directory intensive IOPS directed workload.

Outline

Introduction

Background

Motivation

pFSCK Design

Evaluation

Summary

Summary

- pFSCK provides fine grained parallelism for file system checking
- Data parallelism allows more metadata to be checked at a time
- Pipeline parallelism enables parallelism across passes
- Dynamic thread scheduler adapts to file system configuration
- pFSCK is provides up to **2.6x** performance over vanilla e2fsck

Thank You!

David Domingo
djd240@cs.rutgers.edu

Sudarsun Kannan
sudarsun.kannan@cs.rutgers.edu