Accelerating Filesystem Checking and Repair with pFSCK

David Domingo, Sudarsun Kannan, Kyle Stratton

Rutgers University Department of Computer Science

RUTGERS
UNIVERSITY | NEW BRUNSWICK

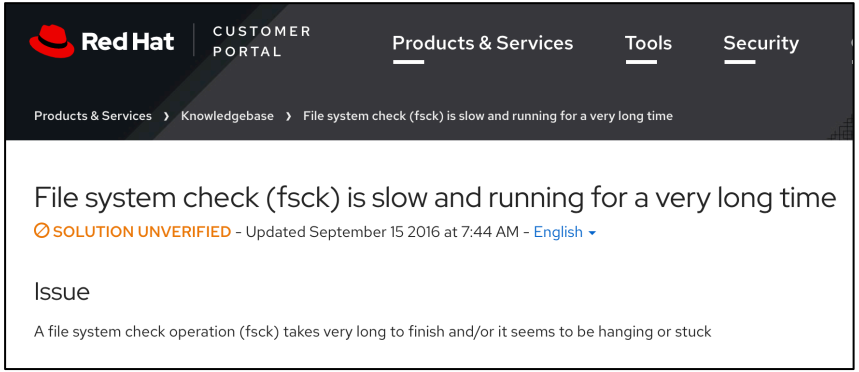
File System Checking

- Used to ensure a file system's metadata is consistent
    - Checks whether file system tree is connected
    - Checks whether file and directory metadata is not corrupt
    - Checks whether all data blocks are accounted for
- Used in the event where the file system is left in an incorrect state
    - System crash or buggy file system
    - Error in the storage hardware (ex. bit-flip)

-

File System Checking (cont.)

- FSCK
- Come in online and offline forms
  - Offline checking requires the storage partition/device to be unmounted and offline
  - Online checking can be done while the partition/device is mounted and in use
- Most common form is offline
- Historically notorious for long runtimes
  - Can potentially incur large downtimes
  - Hassle for system admins



Products & Services › Knowledgebase › File system check (fsck) is slow and running for a very long time

File system check (fsck) is slow and running for a very long time

SOLUTION UNVERIFIED - Updated September 15 2016 at 7:44 AM - English

Issue

A file system check operation (fsck) takes very long to finish and/or it seems to be hanging or stuck

Storage Evolution and Modern Consistency Mechanisms

- Modern storage technology has evolved since hard drives
  - Higher bandwidth (ex. Intel Optane NVMe 1-2GB/s)
  - Lower latencies (ex. Intel Optane NVMe 10-30 µs)
- Modern crash consistency and recovery mechanisms have been developed
  - Journaling
  - Copy on Write
  - Erasure Coding

Cannot detect silent bit corruption

Reconstruction and re-sharding quite time consuming

- Dense flash storage (MLC) is prone to cell wear and bit corruption
- Offline File System Checking is still relevant
- Lots of room for improvement in terms of file checking performance

pFSCK

- Parallelism for file system checkers at a fine granularity (e.g. inodes)
- Ensures correctness through logical reordering
    - Delay dependent checks until dependencies are resolved
- Adapts to file system configurations (varying file/directory count)
    - Dynamic Thread Scheduler balances work in an optimal manner
- Shows up to 2.7x improvement over vanilla FSCK and 1.8x – 8x improvement over XFS

Challenges on Parallelizing File System Checking

- Isolating global data structures and contexts among the file system checker
- Overcome dependencies across multiple passes
- Enabling effective CPU utilization and exploiting disk bandwidth

Outline

Introduction

Background

Motivation

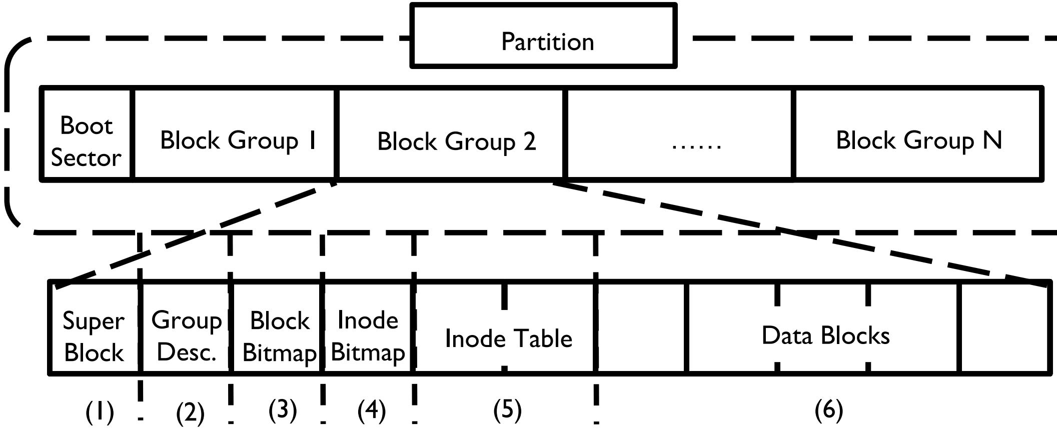
pFSCK Design

Evaluation

Summary

Conclusion

Traditional EXT Disk Layout

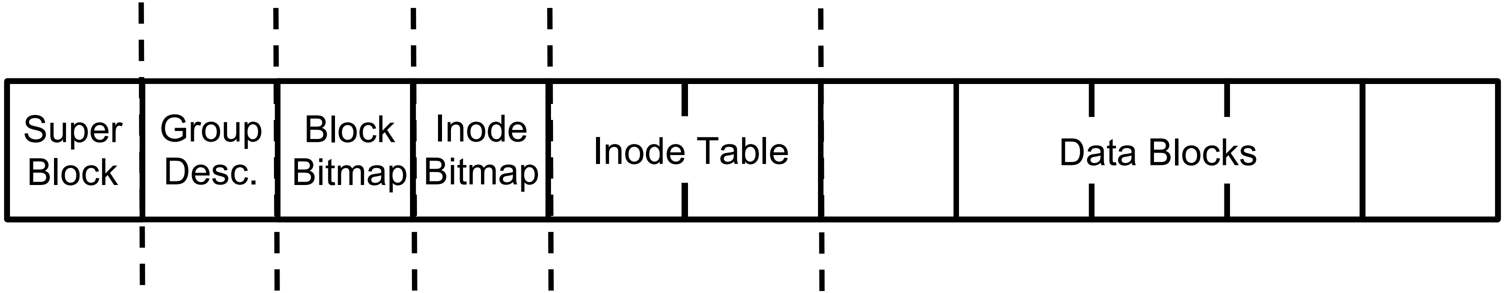


1. Super Block: Stores general information about the file system
2. Group Descriptor: Bitmap/table locations and file/directory count
3. Block Bitmap: Bitmap of used blocks
4. Inode Bitmap: Bitmap of used inodes
5. Inode Table: Table of inode structures
6. Data Blocks: Blocks that store file data including extended file/directory metadata

File System Checker and Repair

- Linux EXT file systems use FSCK
- Checks and fixes any inconsistencies
- Consists of 5 logical passes
- Scans through all file system metadata
- Checks the integrity of file system metadata objects
- Checks consistency across all metadata objects
- Builds own view of a consistent file system in order to repair the actual file system

File System Checker Passes



Pass 1. Check Inodes (file and directory inodes)

Pass 2. Check Directories

Pass 3. Check Connectivity

Pass 4. Check Reference Counts

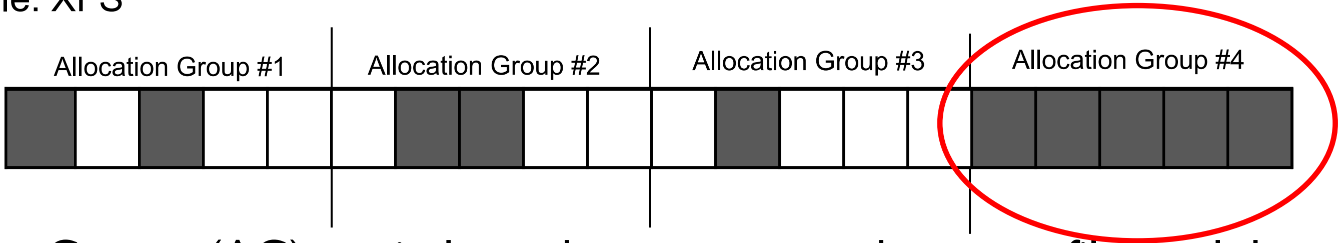
Pass 5. Check Cylinder Groups

Prior Works

parallelization limited to coarse granularity

- **FSCK**: Parallelizes file system checking across disks/partitions
- **XFS_Repair** : Parallelizes file system checking across allocation groups

Example: XFS



- Allocation Groups (AG) are independent structures that store files and data
- File system aging may cause utilization imbalance
- Checking run time will be limited to the most heavily utilized AG
- AG imbalance limits threading benefits

Prior Works (cont.)

- **FFsck (FAST '13)**:
  - Modifies file system and rearranges metadata blocks
  - Provides minimal seek times for faster scanning
- **ChunkFS (HotDep '06)**:
  - Partitions file system into smaller isolated groups
  - Allowed groups to be repaired in isolation

requires extensive modification to the file system

- **SQCK (OSDI '08)**:
  - Uses declarative queries and databases for consistency checks
  - Allows for more expressive fixes with comparable run times

requires complete overhaul of file system checker

Outline

Introduction

Background

Motivation

pFSCK Design

Evaluation

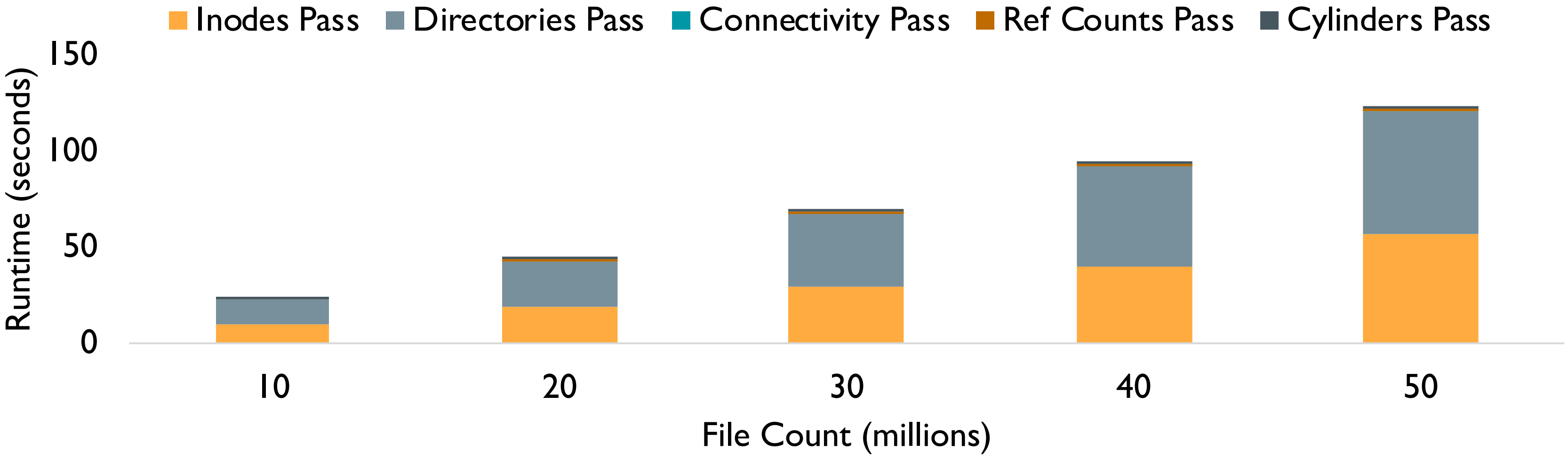
Summary

Conclusion

Evaluating Current FSCK Performance

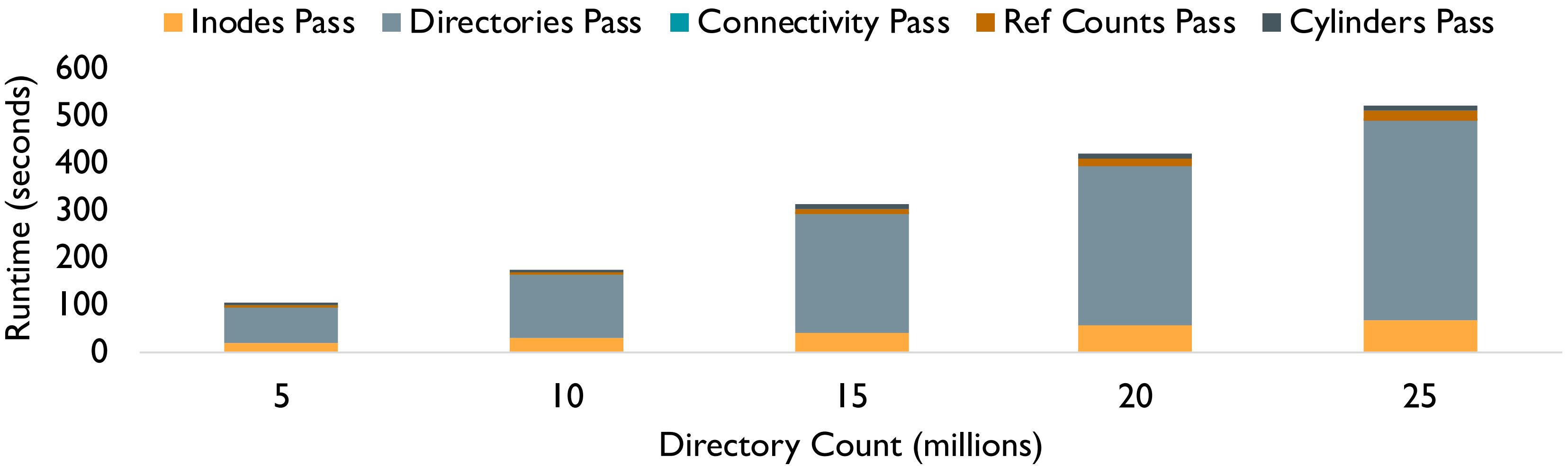
- System:
  - Dual Intel® Xeon® Gold 5218 @ 2.30GHz
  - 64 GB of memory
  - 1TB NVMe Flash Storage
- Methodology:
  - FSCK against 800 GB file systems of varying configurations
    1. Varying file count (file size constant at 12kb, created across 5 directories)
    2. Varying directory count (1 file per directory, each file 24kb)

File Count Sensitivity



- Runtime scales linearly with file count
- As file count increases, directory entry count increases, increasing directories pass runtime

Directory Count Sensitivity



- Runtime scales linearly with directory count
- Effective runtime significantly longer than a file-intensive file system
- Directories pass runtime dominates due to directory block checksumming

Take Aways and Research Questions

- Current file system checker runtime scales linearly with increase in file system utilization
- Current file system checkers do not exploit modern storage performance
  - Passes done sequentially
  - Fail to support fine-grained parallelism
- 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

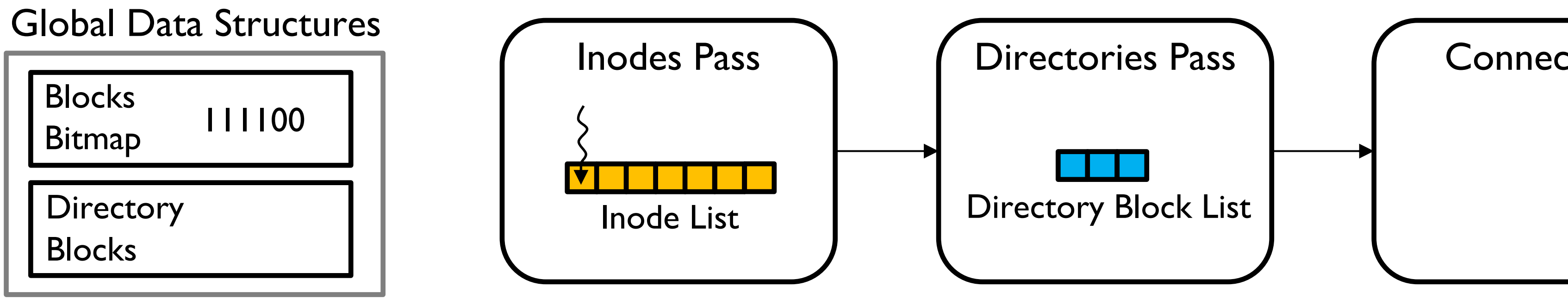
pFSCK Design

Evaluation

Summary

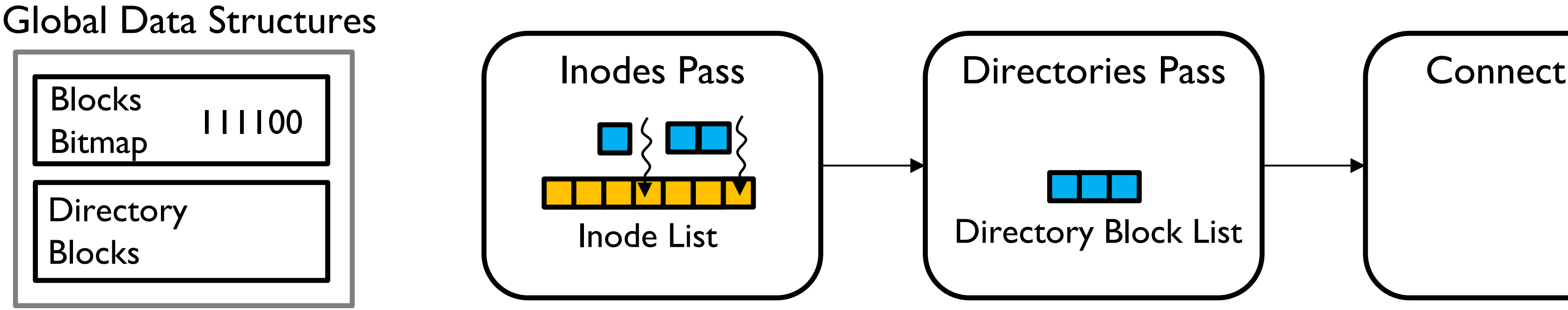
Conclusion

Serial Execution in FSCK



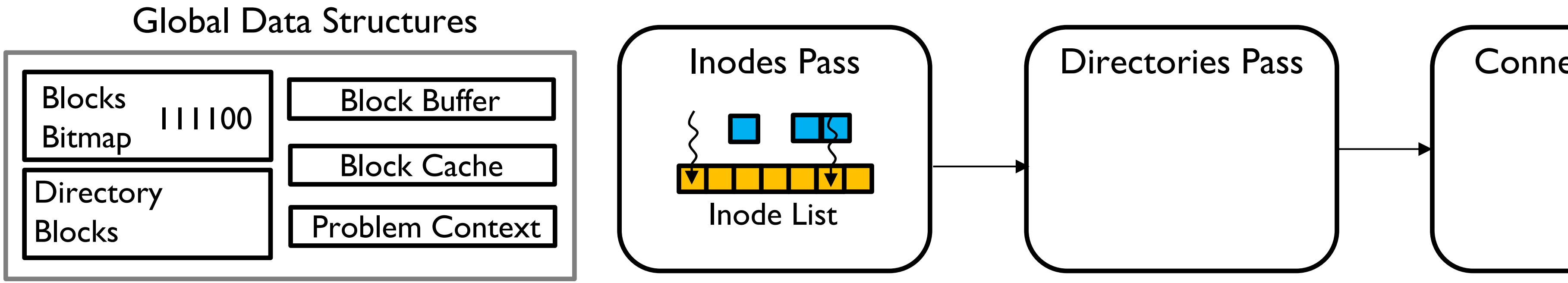
- Serially checks file system metadata (ex. inodes)
- Updates global data structures to generate view of file system based on traversed metadata
- Generates work for the next pass (ex. list of directory block)

pFSCK Data Parallelism



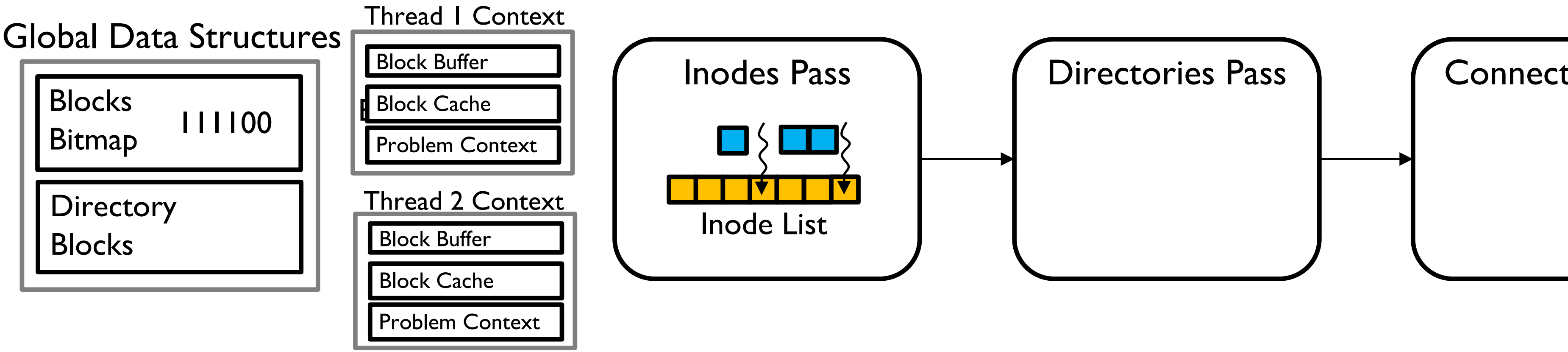
- Split metadata within each pass into smaller groups
- Uses a pool of threads to check in parallel and generate intermediate lists
- Aggregate lists and repeat

Challenge: Buffers and Intermediate Data Structures



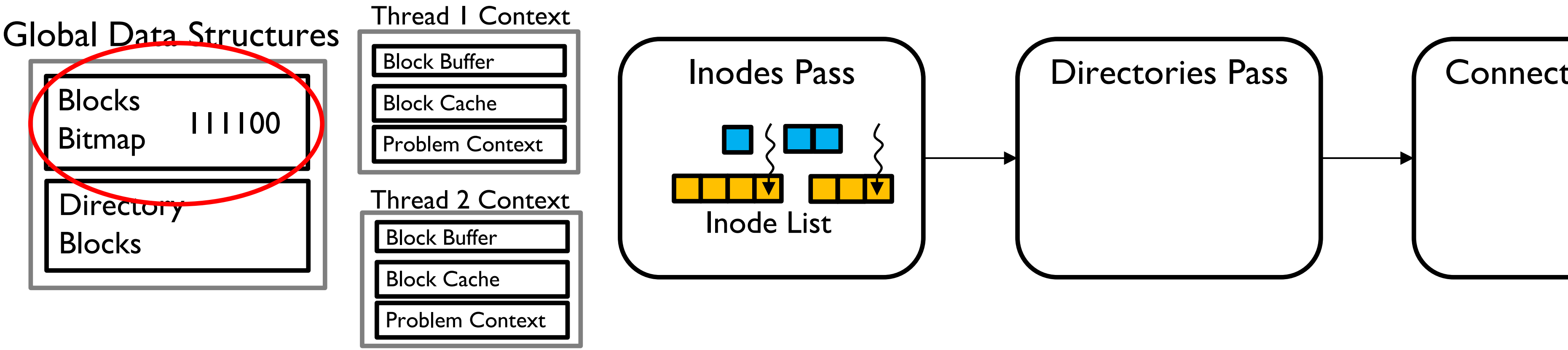
- All data and buffers needed for execution stored in FSCK
  - Block buffers are used to read in blocks from the disks
  - Block cache stores recently read blocks
  - Problem context stores information to give contextual details in the event of an issue
- Naïve solution: serialize accesses across threads using locks
- Reduces concurrency

pFSCK Approach: Per-thread Contexts



- Solution: Introduce per-thread contexts
- Each thread context has its own buffers, cache, and problem context
- Enables threads to operate in parallel

Limitations of pFSCK Data Parallelism

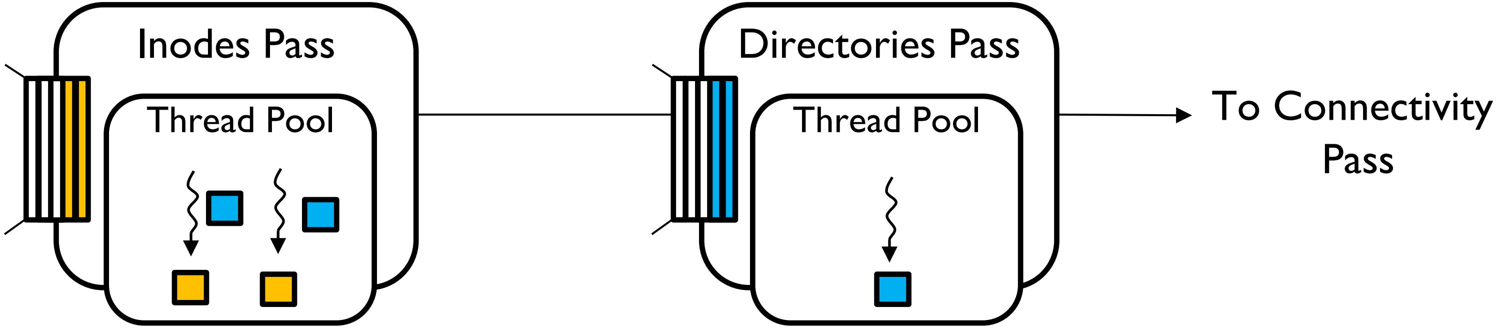


- Not all global data structures can be isolated
  - ex. block bitmap needed to check for consistency between inodes
- Synchronization unavoidable without complete redesign of FSCK
- Limits concurrency in each pass

pFSCK Pipeline Parallelism

- Idea: Hide data parallelism synchronization bottlenecks
- Solution: Allow multiple passes to operate in parallel
- Increase metadata being checked across passes instead of a single pass

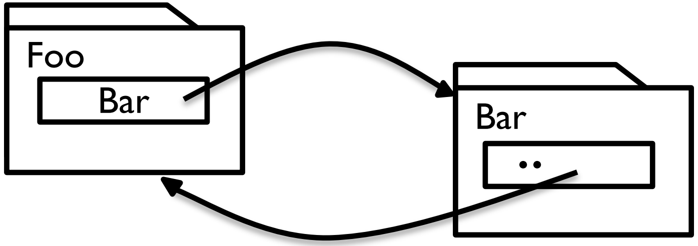
pFSCK Pipeline Parallelism



- Turn each pass into independent flows of execution
- Use per pass queues to store inputs from previous pass
- Have multiple thread pools for each pass
- Continuously feed subsequent passes with metadata
- Do not wait for previous pass to complete (**speculatively carry out future checks**)

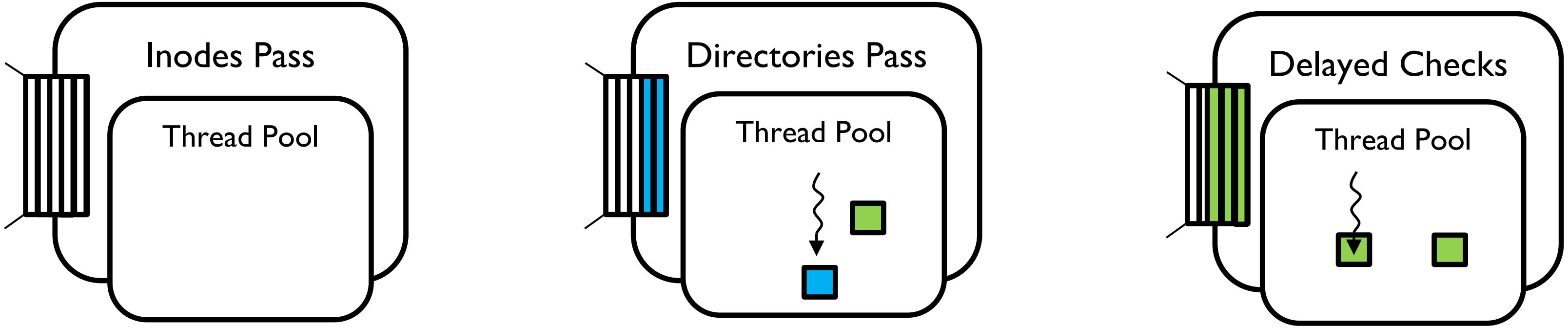
Pipeline Parallelism: Dependent Check Problem

- Some logical checks depend on information generated by the previous passes
- For Example: Directory Relationships in Directory Checking Pass (Pass 2)



  - Check: any subdirectory must have a reference to the current directory
  - Cannot be done until subdirectory's inode has been checked
- Naïve Solution: Wait for dependent information to be generated from prior pass
  - Stalls working thread indefinitely

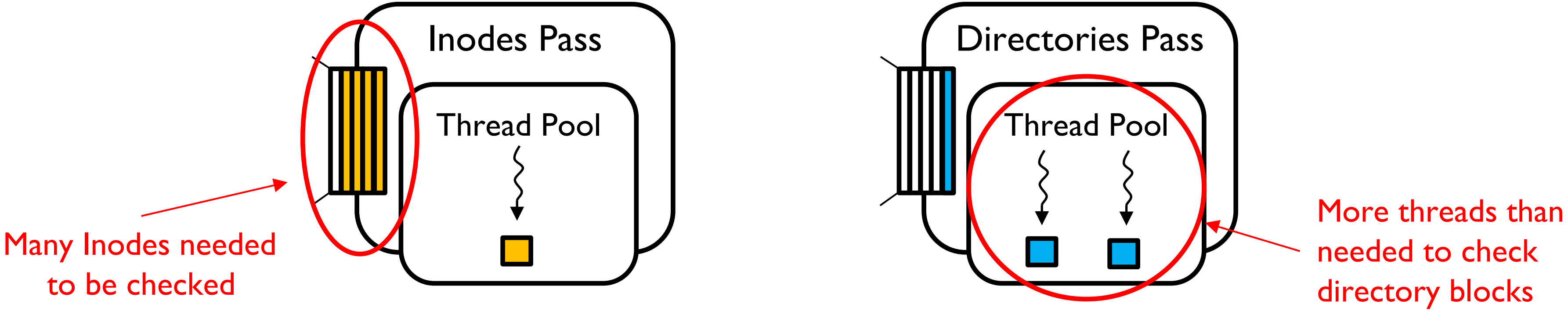
Delaying Dependent Checks



- Solution: Delay dependent checks
  - Mark intermediate tasks to be carried out later
  - Execute checks once previous pass had finished
- Ensures checks are eventually carried out for correctness
- Prevents threads from stalling
- Note: Still various edge cases in later passes that are currently investigated

Pipeline Parallelism: Work Imbalance Problem

- Issue: Work imbalance between the passes
- Not straight forward how many threads to assign to each pass
- Examples:
  1. File-intensive file systems mainly will have more inodes to check



Many Inodes needed to be checked

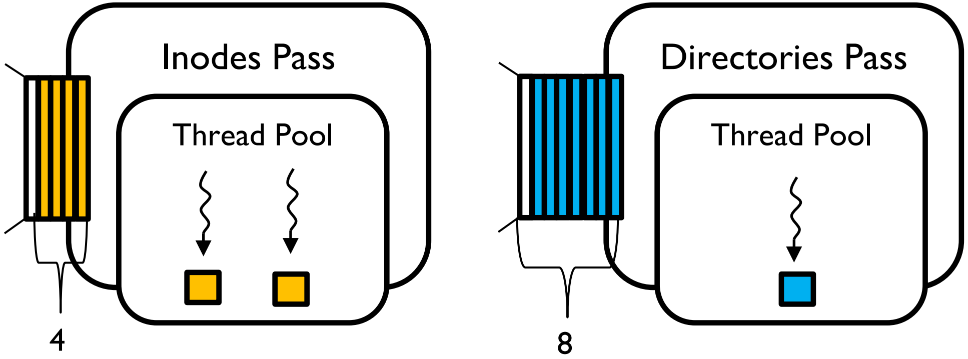
More threads than needed to check directory blocks

Solution: Dynamic Thread Scheduling

- Calculate proportions of work among the passes and redistribute threads

Scheduler Thread

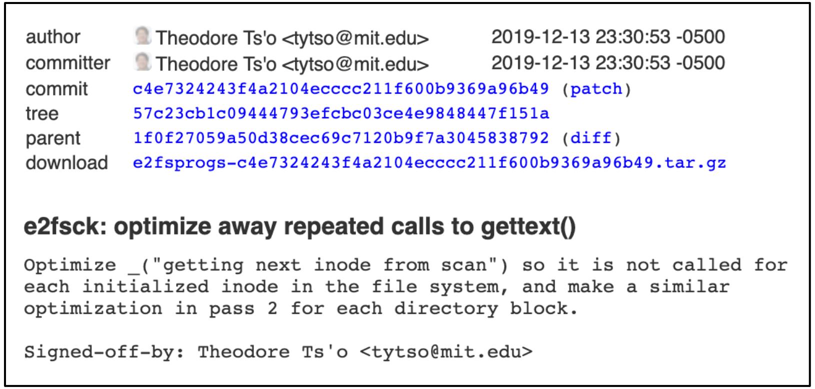
Total Threads = 3
Total Work = 12
Inodes Proportion = 0.66
Dirs Proportion = 0.33
Inode Threads = 1
Dir Threads = 2



- Scheduler thread periodically samples the task queue lengths of each pass
- Calculates total amount of work that needs to be done with normalized weights
- Calculates proportion of work for each pass
- Calculates thread distribution and redistributes threads
- Allows pFSCK to adapt to different file system configurations with differing metadata densities

FSCK-Opt

- Optimization for serial FSCK in addition to pFSCK
- Repeated string localization for every metadata object – high overheads
  - dcigettext() from the glibc library
  - To give contextual information in preferred language when an inconsistency is detected
- Solution: cache translations, instead of re-localizing for every metadata object
- We reported this and it has since been fixed within the current release of e2fsprogs



author	Theodore Ts'o <tytso@mit.edu>	2019-12-13 23:30:53 -0500
committer	Theodore Ts'o <tytso@mit.edu>	2019-12-13 23:30:53 -0500
commit	c4e7324243f4a2104ecccc211f600b9369a96b49 (patch)	
tree	57c23cb1c09444793efcbc03ce4e9848447f151a	
parent	1f0f27059a50d38cec69c7120b9f7a3045838792 (diff)	
download	e2fsprogs-c4e7324243f4a2104ecccc211f600b9369a96b49.tar.gz	

e2fsck: optimize away repeated calls to gettext()

```
Optimize _("getting next inode from scan") so it is not called for
each initialized inode in the file system, and make a similar
optimization in pass 2 for each directory block.

Signed-off-by: Theodore Ts'o <tytso@mit.edu>
```

pFSCK with file system errors

- When errors are detected threads serialize
- Thread that detected error, fixes error if possible.
- Once fixes are complete, all threads continue in parallel
- Even with errors, pFSCK outperforms FSCK
- **More exhaustive testing in progress**

Outline

Introduction

Background

Motivation

pFSCK Design

Evaluation

Summary

Conclusion

Evaluation Goals

- Does pFSCK's fine grained parallelism improve performance?
- Can pFSCK adapt to various file system configurations?
  - ex. file-intensive vs. directory-intensive file system

Methodology

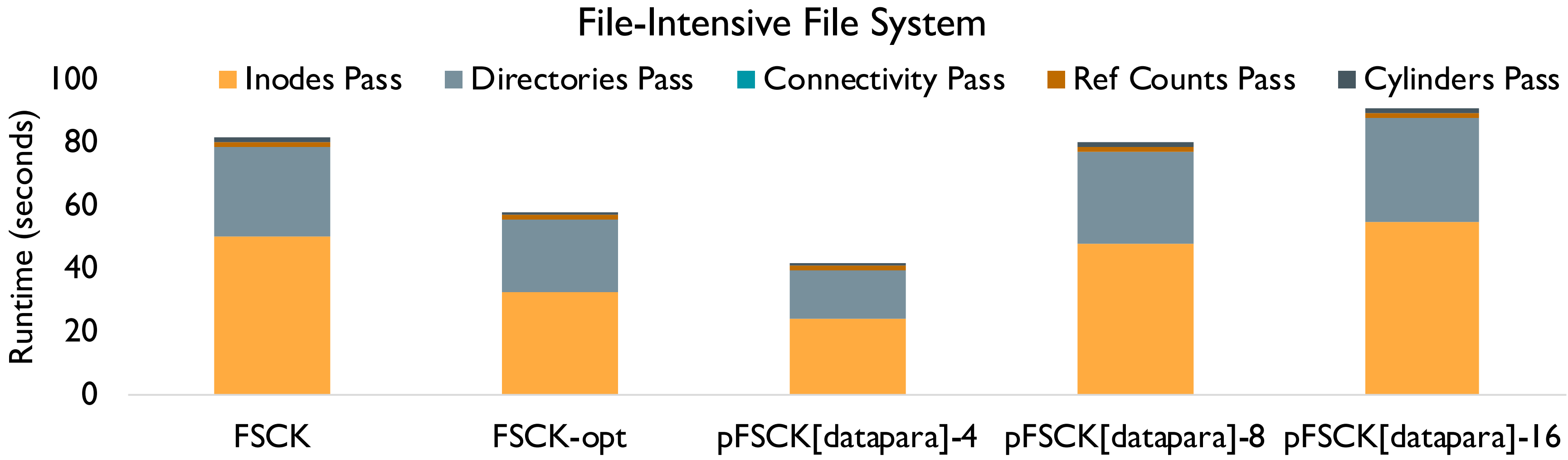
- System:
  - Dual Intel® Xeon® Gold 5218 @ 2.30GHz
  - 64GB of DDR memory
  - 1TB NVMe Flash Storage
- Tools:
  - FSCK (e2fsprogs release v1.44.4)
  - XFS_Repair (xfsprogs release 4.9.0)
  - pFSCK (proposed system)

Evaluation: Data Parallelism

- pFSCK's data parallelism compared to vanilla FSCK and XFS_repair

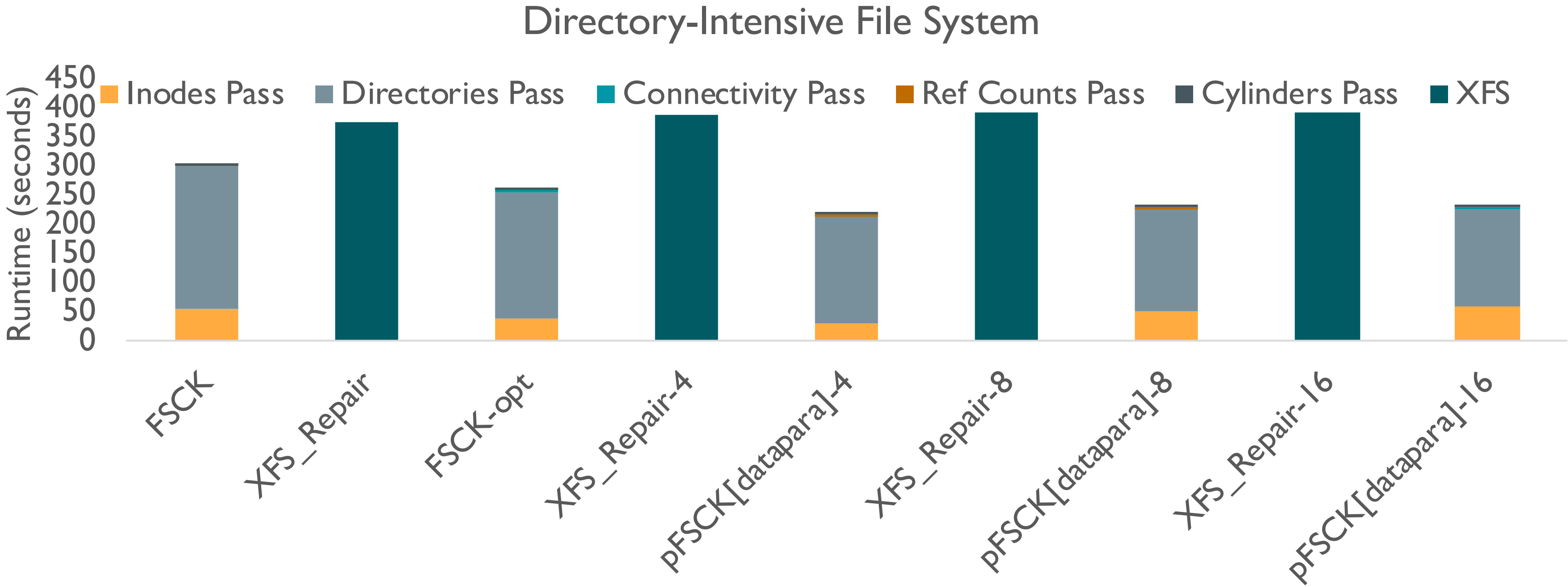
Name	Description
FSCK	Vanilla FSCK for EXT file systems
FSCK-opt	Serial, but optimized FSCK
pFSCK[datapara]	Proposed file system checker with various thread counts
XFS_Repair	XFS file system checker with various thread counts

Evaluation: Data Parallelism



- Data parallelism alone improves performance over vanilla FSCK by up to 2x
- Data parallelism improves performance over XFS_Repair by up to 1.8x – 8x with same thread count
  - Currently analyzing XFS performance variation on NVMe for different images
- pFSCK does not scale past 4 threads (contention on shared structures)

Evaluation: Data Parallelism



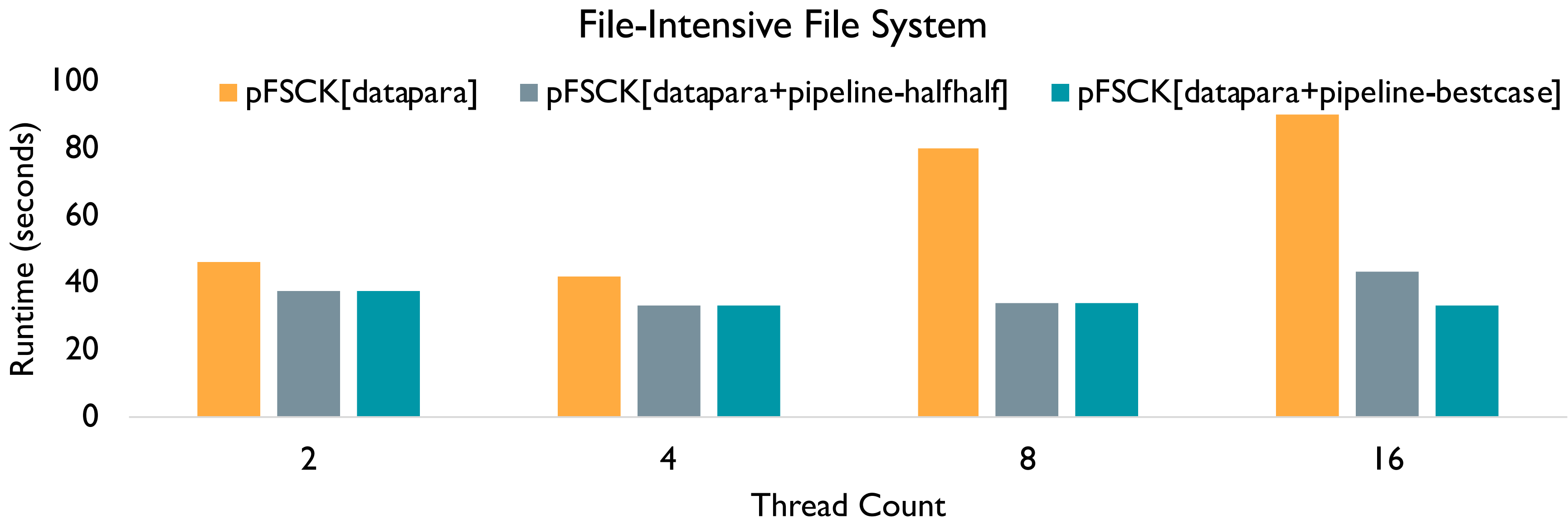
- Data parallelism and optimizations alone improves performance over vanilla FSCK by up to 1.4x
- Data parallelism alone improves performance over XFS by up to 1.7.x
- pFSCK does not scale past 4 threads (contention on shared structures)

Evaluation: Pipeline Parallelism

- How effective is pipeline parallelism compared to just data parallelism?
- Compare pFSCK with data parallelism vs pFSCK with data and pipeline parallelism

Name	Description
pFSCK[datapara]	only data parallelism enabled
pFSCK[datapara+pipeline-halfhalf]	data and pipeline parallelism with half the threads checking inodes and half the threads checking directories
pFSCK[datapara+pipeline-bestcase]	data and pipeline parallelism with static best case thread assignment

Evaluation: Pipeline Parallelism



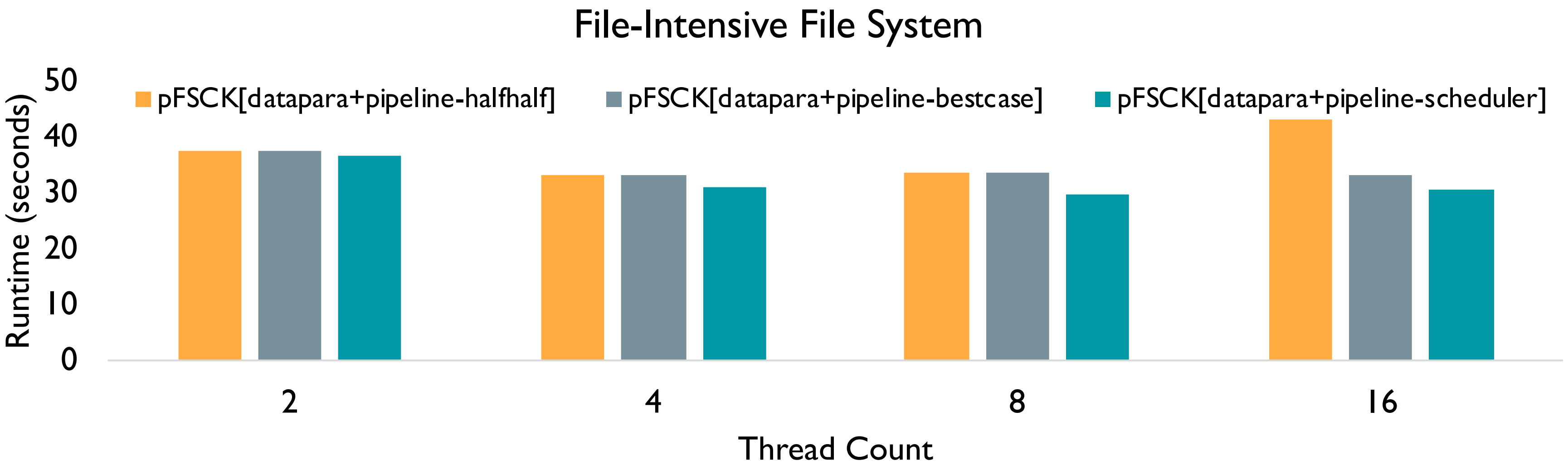
- Pipeline parallelism improves performance by up to 1.3x over data parallelism
- Improves performance by up to 2.5x over the vanilla FSCK

Evaluation: Dynamic Thread Scheduler

- How effective dynamic thread scheduler at optimizing thread assignments?
- Compare best manual pFSCK configuration to pFSCK with scheduler

Name	Description
pFSCK[datapara+pipeline-halfhalf]	data and pipeline parallelism with half the threads checking inodes and half the threads checking directories
pFSCK[datapara+pipeline-bestcase]	Best manual pFSCK thread configuration
pFSCK[datapara+pipeline+scheduler]	Automatic pFSCK thread configuration

Evaluation: Dynamic Thread Scheduler



- Dynamic thread scheduler was able to find an optimal thread assignment
- Scheduler has more impact as the number of threads increase
- Improves runtime by up to 1.1x over data and pipeline parallelism
- Improves runtime by up to 2.7x over vanilla FSCK

Outline

Introduction

Background

Motivation

pFSCK Design

Evaluation

Summary

Conclusion

Summary

- pFSCK provides fine grained parallelism for file system checking
- Data parallelism allows more metadata to be checked at a time
- Pipeline parallelism overcomes data parallelism bottlenecks
- Dynamic thread scheduler adapts to file system configuration
- pFSCK is provides 2.7x performance over vanilla FSCK

Outline

Introduction

Background

Motivation

pFSCK Design

Evaluation

Summary

Conclusion

Conclusion

- Fine grained parallelism is required for lower checking and repair runtimes
- Fined grained parallelism is needed to exploit modern storage capabilities
- Plans to adapt pFSCK for online checking
- Apply pFSCK approaches to other file system checkers