Understanding File System Checker Performance Analytics in Response to Simulated Aging and Metadata Corruption

BY

Chase Sterling White Gilbert

A final year report submitted to the Graduate School

in partial fulfillment of the requirements

for the degree

Master of Science in Computer Science

Submitted to:

Dr. Mai Zheng

Department of Computer Science

New Mexico State University

June 2018

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Mai Zheng, for all of his help and advice throughout my graduate school experience. I am immensely grateful for all he has done to kindle my passion for computer systems. I would also like to thank my colleagues in the NMSU Data Storage Lab for their support and encouragement.

VITA

2013-2017	B.S.C.S, New Mexico State University
2016	Intern, Honeywell Aerospace
2017	Graduate Intern, Honeywell Aerospace
2017	Teaching Assistant, New Mexico State University

FIELD OF STUDY

Major Field: Computer Science

ABSTRACT

Understanding File System Checker Performance Analytics in Response to Simulated Aging and Metadata Corruption

BY

Chase Sterling White Gilbert

Master of Science

New Mexico State University

Las Cruces, New Mexico, 2018

Dr. Mai Zheng, Chair

In this project, we discuss the necessity of file system checkers, and focus on the problems associated with the runtime of those checkers, which is often egregiously long. To this end, we propose an in-depth analysis of two popular file system checkers (e2fsck and xfs_repair), in response to corruption of specific metadata components. To do this, we create many different configurations of both XFS and EXT4 file systems. We also create a suite of tools to sufficiently age and inject

metadata corruption into the systems. We then monitor the performance of the checkers as they repair the corruptions in the different file system configurations, to discover trends of which metadata corruption has the greatest effect on the performance of each checker.

CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

1 INTRODUCTION

File system checkers are often thought of as our last line of defense for combatting corruption and disk failure. Although they are our "last resort", when preventative measures made by the file systems themselves do not suffice for protection, we are often required to make the decision to run the checker. This can pose a problem: the *runtime* of the checker.

In many cases, a file system checker's runtime is abhorrently long, which leads to two main issues. Firstly, long check times can halt workflow. In general, to run a checker on a file system, that file system must be unmounted first. Therefore, if the check time is many hours or days, the file system will be unusable for that full duration. This makes running a checker an extremely costly operation. Secondly, the longer a checker is running for, the more opportunity there is for it to crash, due to various faults or power outages. To make matters worse, a file system checker crash does not necessarily just require a checker restart. A checker crash can often cause irrevocable damage to the file system that the checker is attempting to repair (making the system unmountable, etc.) [3].

An example of this is with the Lustre file system used by the High Performance Computing Center in Texas [4]. In this case, a major power outage occurred, which resulted in the associated file system checker being run. This recovery process, as expected, was extremely slow. Two days into the recovery process, the center

experienced more power outages, resulting in major data loss, since it was not only a system crash, but one where the checker was in the midst of its operations.

The obvious question then, is if both of these issues scale according to the runtime of the checker, how can that time be reduced? To this end, an analysis of two popular file systems (EXT4 and XFS) and their respective checkers is proposed, to gain insight into what file system configurations and corruption types have the largest contribution to checker time and memory usage. This analysis consists of four major parts: Aging, Fault Injection, Memory Analysis and Time Analysis.

I. Aging - Running experiments on clean file systems may not produce accurate results, because in the real world, our in-use systems have been organically *aged* to some degree. Aging is the normal process of system performance degradation over a long period of time, due to fragmentation and other issues. Therefore, before we perform our analysis, we need to simulate some degree of aging as well (otherwise, our results could be drastically different than those found on a real-world system). To do this, we have looked at different methods of aging file systems, we have created two aging tools of our own, and we have compared their impact on different systems and system configurations. The aging tools created in this paper are also compared with another pre-existing aging method.

II. Fault Injection - To measure how file system checkers respond to different forms of corruption, a method is needed to induce that corruption. Both EXT4

and XFS have different layouts for their metadata, so a single corruption method is insufficient. In response to this, we have built a comprehensive, user-friendly fault injection tool that utilizes unique approaches to corrupting the metadata of both EXT4 *and* XFS file systems. For EXT4, an attribute of a user-selected block group is chosen, and the byte at the location containing that component is manipulated. For XFS, the user selected metadata component to corrupt is passed to debugging utility built for XFS, which subsequently corrupts a small number of matching attributes in the system.

III. Memory Analysis - To paint a clearer picture of the specific effects that our induced corruption has on the checkers, it is beneficial to monitor the virtual memory usage of the checkers before and after the fault injection. To do this in an effective manner, we use a sub-tool of Valgrind (called Massif) to isolate the memory used by the checking process, and to record the peak memory usage (this provides more accurate results then a standard ps aux call because Valgrind isolates the process, to prevent the memory peak from being affected by unrelated processes).

IV. Time Analysis - To see how our induced corruption has affected the checkers runtime for each file system configuration, we need to log the checker time before *and* after each corruption, to measure the the scale of the negative impact. To do this, we use the Linux time command in tandem with the checker call to monitor the check time in seconds. We then take the average of several runs of

the checker to account for variability in the results.

The rest of the paper is organized as follows: We first discuss related work (§2), and then give an in-depth description of our design and implementation (§3). We next provide an evaluation of our results (§4), and discuss future work (§5). Finally, we detail a summary of our conclusions (§6).

2 RELATED WORK

The desire to analyze and improve the performance time of file system checkers is not a new concept. There have been many different methods and attempts in this area. ffsck is a notable example. ffsck (the fast file system checker) is designed to work in tandem with a modified EXT3 file system, and according to the creators, this checker achieves significant speedup compared to e2fsck (the checker for the EXT family of systems), via the use of bitmap snapshots, as well as negating costly seek times for indirect blocks [1]. The downside is that ffsck *must* be paired with a special modified file system. Asking entities with large setups to switch over their entire infrastructure to this new system type is a fairly unrealistic request.

Another approach (the BetrFS file system) consists of building a file system from the ground up to be aging resistant, in an attempt to improve general system performance (which would, in turn, affect checker time in a positive way). According to the creators of BetrFS, in experiments where other popular file systems were heavily affected by aging, BetrFS exhibited almost no signs of aging at all [2]. However, current builds of BetrFS are fairly unstable, resulting in occasional crashes.

BetrFS is not the only file system built to combat aging. Modified systems like NoSwitch and SmartFrag show how small changes can improve the resiliency

of a system towards aging. Although NoSwitch and SmartFrag are outdated now (being based on FFS), we can still learn from their example [8].

There has also been a sizable amount of research done on the art of manually inducing aging into a system. Tools like fs_mark and filebench can age file systems in the style of real-world workloads, which can be very beneficial for benchmark purposes [5,6]. Other tools have a more focused take on file system aging, such as Geriatrix, which tries to make aging more user-friendly, and effective on solid state drives [7].

The reality of file system aging and its impact on checkers is ever-present, but there has been a fair amount of work to lessen that impact. In the next section, we will show how our efforts contribute to that goal.

3 DESIGN AND IMPLEMENTATION

In this section, the design and implementation of the utilities created to assist in this project are described. There are three main tools that have been built. Firstly, two separate utilities have been created for the express purpose of inducing aging in file systems: holePuncher and roundHouse. These tools use aging methods such as creation of dense folder hierarchies, alternating file operations, sparse file allocation, etc. The third utility (corruptFS) uses system-specific methods to induce corruption in user-specified metadata components. We will go into more detail for each tool in the following subsections.

3.1 Goals

We expect our project to display the following criteria:

I. Observable and competitive file system aging - Our aging methods should have not only a sufficient impact on the associated systems, but also should perform competitively in comparison to existent methods.

II. Effective and high-level corruption - Our corruption tool should have sufficient depth to allow the user to corrupt specific metadata. This should be easy to use and not overly complicated, but it should not sacrifice completeness for simplicity.

III. Data pool completeness - To form fair conclusions about our data sets, our

pool of information must be sufficiently large. There should be a wealth of test results for each system type, as well as results for different block, inode and file system sizes.

IV. Clear results of the effects of corruption - Throughout this project, we hope to come up with definitive answers to the question of which metadata components have the highest impact on checker performance.

3.2 Aging Tools

As mentioned previously, there have been two aging tools built for this project: roundHouse and holePuncher.

3.2.1 roundHouse

The first tool (roundHouse) has functions partially inspired by the aging method used by the creators of ffsck. In the paper on ffsck, the authors outline several components to their aging process (making many folders, randomly creating files, uniform file operations, etc.), but they do not list all of the details of their approach [1]. roundHouse takes these ideas and expands upon them. Instead of simply making many top-level folders, roundHouse creates three main layers of folders, in order to create a dense hierarchy. The tool then iterates through each folder in the hierarchy and calls a function called roundRobin once per folder, which manages the file operations. This process is repeated until the file system

runs out space.

The roundRobin function has four cases, one for each file operation (create, truncate, delete, and append). The case that is chosen depends on two counters: one inside and one outside of the roundRobin function. The inner counter makes sure that each roundRobin call uses the next operation in the sequence, and the outer counter offsets the starting case to utilize a round-robin style of operations.

The first case (create) is simple. Two files are created in the current folder using the linux fallocate command. This operation is wrapped in a try-catch statement, to terminate once the file system runs out of space.

The next case (truncate) selects a random file inside of the current folder and then truncates it to half of its current size via python's truncate command. (To protect against the possibility of empty folders, the initial folder hierarchy creation creates two files per folder).

The third case (delete) first selects a random file in the current folder. It then utilizes the OS to delete the selected file.

The last case (append) first selects a random file in the current folder. It then makes use of python's append command to append 256KiB to the selected file. This operation is also wrapped in a try-catch statement, to terminate once the file system runs out of space.

The pseudocode for roundHouse is shown in Algorithm 1.

Algorithm 1 roundHouse

```
1: procedure roundHouse(path)
2:     populate()                          ▷ Create the three-level folder hierarchy
3:     rCatch ← 0
4:     name ← 5
5:     while (1) do                        ▷ Run until the file system is full
6:         task ← rCatch
7:         for i in range (numOfTopLevelFolders) do
8:             roundRobin(task,i,-1,-1)
9:             for o in range (numOf2ndLevelFolders) do
10:                roundRobin(task,i,o,-1)
11:                for p in range (numOf3rdLevelFolders) do
12:                    roundRobin(task,i,o,p)
13:        rCatch ← rCatch + 1
14:        if rCatch = 4 then              ▷ Reset rCatch after each cycle
15:            rCatch ← 0
16:
17: procedure roundRobin(operation, top, sec, third)
18:    if operation = 0 then                                   ▷ Create case
19:        try
20:            fallocate(top,sec,third,name)                   ▷ Make new file
21:            name ← name + 1
22:            fallocate(top,sec,third,name)                   ▷ Make new file
23:            name ← name + 1
24:        catch(exception)                ▷ Terminate when file system is full
25:            exit()
26:    else if operation = 1 then                              ▷ Truncate case
27:        f ← open(randomFile(top, sec, third))
28:        f.truncate(f.size()/2)
29:    else if operation = 2 then                              ▷ Delete case
30:        f ← open(randomFile(top, sec, third))
31:        os.remove(f)
32:    else if operation = 3 then                              ▷ Append case
33:        try
34:            f ← open(randomFile(top, sec, third))
35:            f.append(131072, "z")           ▷ Append 256KiB to the file
36:        catch(exception)                ▷ Terminate when file system is full
37:            exit()
38:    task ← task + 1
39:    if task = 4 then            ▷ Reset task after each cycle of operations
40:        task ← 0
```

3.2.2 holePuncher

The other aging tool built for this project (holePuncher) takes an entirely different approach: punching holes in a file. Punching holes in a file makes portions of that file sparse, meaning that only metadata is stored, which reduces the space the file actually takes up. When a file is sparse, the file system checker still needs to scan over the file in the locations where the user data is. Therefore, if many sparse files are created, each with a large initial size, and then hole-punched, user data is still spread out over the file system, leading to many costly seeks for file system checkers and search utilities.

In light of this, holePuncher first creates a very large file (81% of the file system size). It then uses the –punch-hole feature of the Linux fallocate command to punch four large holes in the file. The first three holes each take up 20% of the file system. The last hole makes the remaining part of the file sparse. 2048 bytes of user data are left before each hole, so that the user data of the file is spread out over the length of the file system, but leaving the file itself to only take up 8192 bytes of real space. This sequence of operations is executed 1000000 times, but will terminate early if the file system is full via a try-catch block.

The pseudocode for holePuncher is shown in Algorithm 2.

Algorithm 2 holePuncher

```
1: procedure HOLEPUNCHER(path, fileSystemSize)
2:     chunk ←(fileSystemSize * 0.8) / 4.0
3:     size ←(fileSystemSize * 0.81)
4:
5:     offset1 ← 2049                          ▷ Set hole punch checkpoints
6:     offset2 ← offset1 + chunk + 2049
7:     offset3 ← offset2 + chunk + 2049
8:     offset4 ← offset3 + chunk + 2049
9:     last ← size − offset4
10:
11:    for i in range (1000000) do             ▷ Execute one million times
12:        try
13:            os.call("fallocate -length",size,i)          ▷ Make file of size size
14:
15:            //Punch a hole of size chunk starting at the specified offset
16:            os.call("fallocate –punch-hole –offset",offset1,"–length",chunk,i)
17:            os.call("fallocate –punch-hole –offset",offset2,"–length",chunk,i)
18:            os.call("fallocate –punch-hole –offset",offset3,"–length",chunk,i)
19:
20:            //Punch a hole of size last starting at offset4
21:            os.call("fallocate –punch-hole –offset",offset4,"–length",last,i)
22:
23:        catch(exception)                    ▷ Terminate when file system is full
24:            exit()
```

3.2.3 git pulls

In this project, we also consider another pre-existing aging technique: the git pull method used by the creators of BetrFS [2]. This method uses a suite of python scripts to perform a user-specified number of git pulls from a given github repository to a local repository located in the desired file system. Since each pull alters the contents in the local repository, fragmentation occurs, thus contributing to aging. We will discuss the results of comparison with this approach in the evaluation section (§4).

3.3 Corruption Tool

As mentioned previously, this project focuses on two file systems: EXT4 and XFS. The file system layout and metadata is fairly different between these systems, so our corruption tool (corruptFS) uses a unique approach for both systems.

Firstly, corruptFS decides via user input which file system procedure to call (ext4Attack for EXT4 and xfsAttack for XFS). The program terminates once the chosen procedure has returned.

3.3.1 ext4Attack

ext4Attack is a fairly complicated function. First, system dump information from the selected file system is saved to a file via the *dumpe2fs* linux command from the package *e2fsprogs*. The information gleaned from this command contains

detailed descriptions of the system and its metadata, including in-depth listings of the contents of each block group. Next, the dump file is opened for reading, and the total number of block groups is displayed for the user. This total is calculated via the getBlockGroupTotal procedure. (getBlockGroupTotal simply loops through the first 40 lines of the dump file to check the block count, block size, and blocks per group of the file system. It then puts them into a short equation to arrive at the total number of block groups, which it then returns).

After printing the total number of block groups, one specific block group is chosen via user selection. Next, the procedure loops through each line in the dump file, looking for the block group entry that matches the choice of the user. When the group is found, the contents are displayed for the user, and the byte location of each metadata component is stored in a corresponding variable. After these tasks, the loop is terminated with a break statement. Next, one of the previously displayed metadata components is selected via user input as the candidate for corruption. The location of the desired component is retrieved via a switch statement, and then passed to a procedure called byteAttack.

byteAttack takes care of the actual corruption. First, it opens the file system itself for both reading and writing. The procedure then seeks to the byte location of the user-selected metadata component and copies the sought byte into a buffer. The first bit of the byte is then stored in an integer. The integer is subsequently bitmasked to zero out the leading bits, and XORed with 128 to invert the copied

bit. The flipped bit is then written back to the buffer, and the corrupted buffer is written back to the file system, effectively corrupting the metadata compenent, and thus the file system itself.

3.3.2 xfsAttack

xfsAttack is much simpler than the previous procedure. There is no clear way (either from the associated tools or the XFS documentation) to discern the actual byte locations of specific metadata components in an XFS file system (other than the superblock). So, corruptFS uses an external tool: the xfs_db utility from the xfs_progs package. xfs_db is a debugging tool for XFS, and it contains a feature entitled *blocktrash* for metadata corruption. (*blocktrash* normally just fills random blocks in the system with garbage, but we use the *-t* argument to specify which metadata component to corrupt).

xfsAttack first displays the metadata components available for corruption to the user. One specific component is then selected via user input. This component type is subsequently passed to an OS call of xfs_db *blocktrash*, which "trashes" random metadata blocks of the specified type.

The pseudocode for corruptFS is shown in Algorithm 3.

Algorithm 3 corruptFS

```
1: procedure CORRUPTFS(path, fsType)
2:     if fsType = ext4 then
3:         ext4Attack()                                   ▷ ext4-specific function
4:     else if fsType = xfs then
5:         xfsAttack()                                    ▷ xfs-specific function
6:     exit()
7:
8: procedure EXT4ATTACK()
9:     os.call("dumpe2fs > out")          ▷ Save ext4 system dump info to a file
10:    file ← open("./out")                              ▷ Open dump file for reading
11:    print(getBlockGroupTotal(file)) ▷ Display total number of block groups
12:    scan(groupNum)                  ▷ Get desired block group from user input
13:
14:    for line in file do                               ▷ Loop through the dump file
15:        if line ="group "+groupNum then                          ▷ Group match
16:            lineHead ← line.split()[0]                          ▷ First substring
17:            while lineHead !="group" do
18:                print(line)          ▷ Display metadata items in current group
19:                switch lineHead do          ▷ Get byte locations for each item
20:                    case "Superblock":
21:                        superLoc ← line.split()[4]              ▷ Last substring
22:                        ⋮                          ▷ Check each metadata item
23:            break
24:
25:    scan(corruption)          ▷ Get metadata item to corrupt from user input
26:    switch corruption do                  ▷ Get byte location of corruption
27:        case "Superblock":
28:            byteNum ← superLoc            ▷ Assign correct value to byteNum
29:            ⋮                                  ▷ Find the correct metadata item
30:
31:    byteAttack(byteNum)        ▷ Call function to corrupt the metadata item
32:    return
33:
34: procedure XFSATTACK()
35:    print(metadataComponents)                         ▷ Display corruption options
36:    scan(corruption)          ▷ Get metadata item to corrupt from user input
37:    os.call("xfs_db blocktrash -t "+corruption)       ▷ xfs_db blocktrash call
38:    return
```

```
39: procedure byteAttack(byteNum)
40:     fp ← open(path)     ▷ Open the file system image for reading and writing
41:     seek(fp, byteNum)                          ▷ Set the file cursor to byteNum
42:     buff ←read(fp, 1)                  ▷ Get the byte pointed to by byteNum
43:     value ← buff[0]                ▷ Set value equal to the leading bit of buff
44:     value ← value & 255                                        ▷ Bitmask value
45:     value ← value ⊕ 128                              ▷ Use XOR to bitflip value
46:     buff[0] ← value                ▷ Replace the leading bit of buff with value
47:     write(buff, 1, fp)           ▷ Write the corrupted byte back to the system
48:     return
49:
50: procedure getBlockGroupTotal(file)
51:     lineCount ← 0
52:     for line in file do                                    ▷ Loop through the file
53:         if lineCount ≥ 40 then                  ▷ Only check the first 40 lines
54:             break
55:         //If the line contains an appropriate variable, record its value
56:         if line.split(':')[0] = "Block Count" then
57:             bc ← line.split(':')[1]
58:         else if line.split(':')[0] = "Block Size" then
59:             bs ← line.split(':')[1]
60:         else if line.split(':')[0] = "Blocks per group" then
61:             bpg ← line.split(':')[1]
62:         lineCount + +                              ▷ Increment the line counter
63:
64:     numOfGroups ← ((bc ∗ bs)/(bpg ∗ bs))    ▷ Calculate total group number
65:     return numOfGroups                              ▷ Return number of groups
```

These corruption and aging tools are high-level, straightforward and easy to use. Not only are they directly applicable to the project at hand, but they could be useful in many other contexts as well.

In the next section, the results of the direct application of these tools is shown as well as their performance contrasted with that of the pre-existing aging tool (the git pulls method).

4 EVALUATION

In this section, we will discuss the experiments that were performed, as well as the results gleaned from them. All tests were done using a 3.3GHz Intel i5 CPU, and a 2TB Dell HDD with Linux 4.40-97, e2fsprogs 1.44.1 and xfs_progs 4.15.1.

4.1 Initial Aging Analysis

To discern which aging method to use, several experiments were conducted. First, we ran all three aging methods on a 100GiB EXT4 file system as well as a 100GiB XFS file system, both with default configurations. Each aging tool ran for exactly 12 hours. To compare the effects of these different methods, we first used the Linux time command together with the associated checker for each system (e2fsck for EXT4 and xfs_repair for XFS). The results from these tests are displayed in Figure 1.

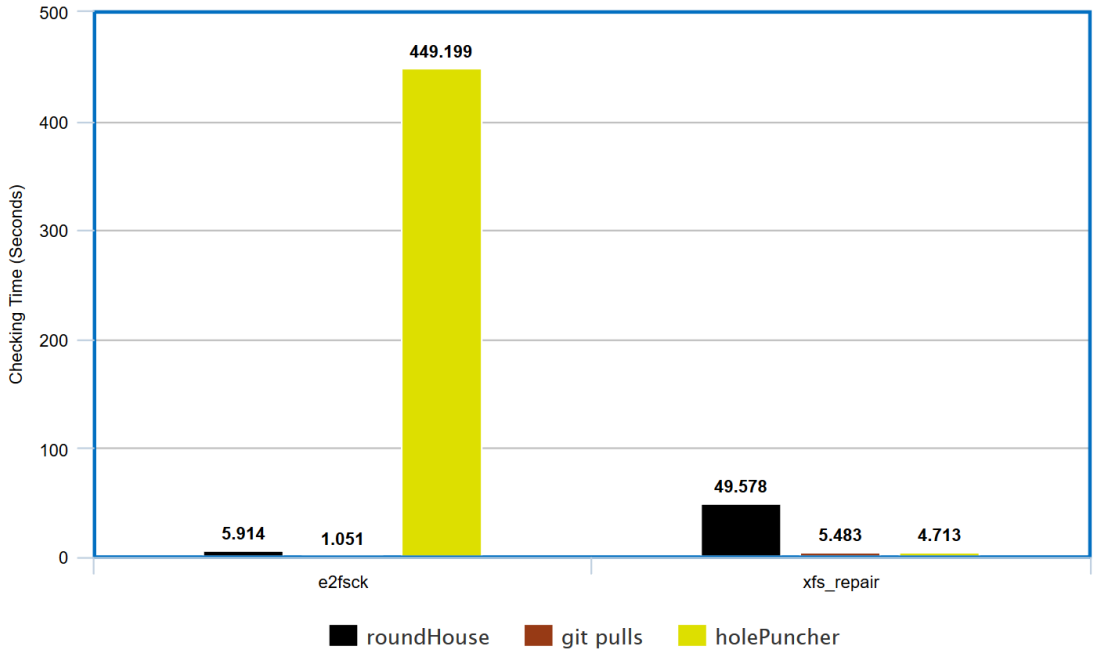

Figure 1: Runtime of e2fsck and xfs_repair in response to aging

As we can see, the runtime of e2fsck was most affected by the holePuncher utility and the runtime for xfs_repair was most affected by the roundHouse utility. The time is increased much more heavily for e2fsck, but applying the roundHouse tool to the XFS system still has a noticeable impact in comparison to the other two applied methods. The average time for either of these checkers to run on a clean system of 100GiB is 0.2 seconds, which the time achieved by the aging tools starkly contrasts.

Next, we monitored the peak virtual memory usage of both checkers for each aging method. To accomplish this, we used a sub-tool of Valgrind, called Massif. Massif is a memory-profiling utility that records memory usage of user-specified

programs, and records snapshots of different levels of memory use over the programs lifetime to determine the peak. It also isolates the process inside of Valgrind, so as to ignore any outside factors and only record the memory usage of the program itself. The results from these tests are displayed in Figure 2.

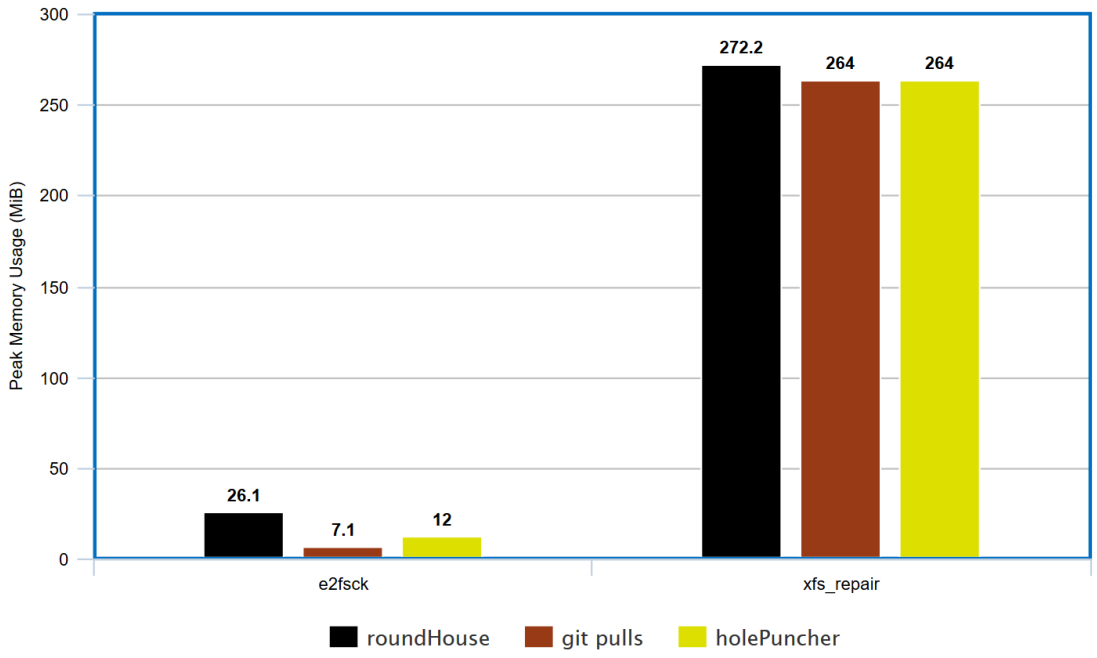

Figure 2: Memory peak of e2fsck and xfs_repair in response to aging

We can see that for both checkers, roundHouse has the strongest impact, but not by very much. On a clean 100GiB system with default settings, the average memory peak for e2fsck is 6.3 MiB, and 263 MiB for xfs_repair. Therefore, although the aging operations definitely affect the memory usage of the checkers, this effect is fairly minor.

Our last set of initial experiments consisted of a comparison of grep tests. Grep tests are a common approach to measuring the aging level of a particular system, that are comprised of using the Linux time command to record the duration of a simple recursive grep call. The results of our grep tests on both systems for each aging method are shown in Figure 3.

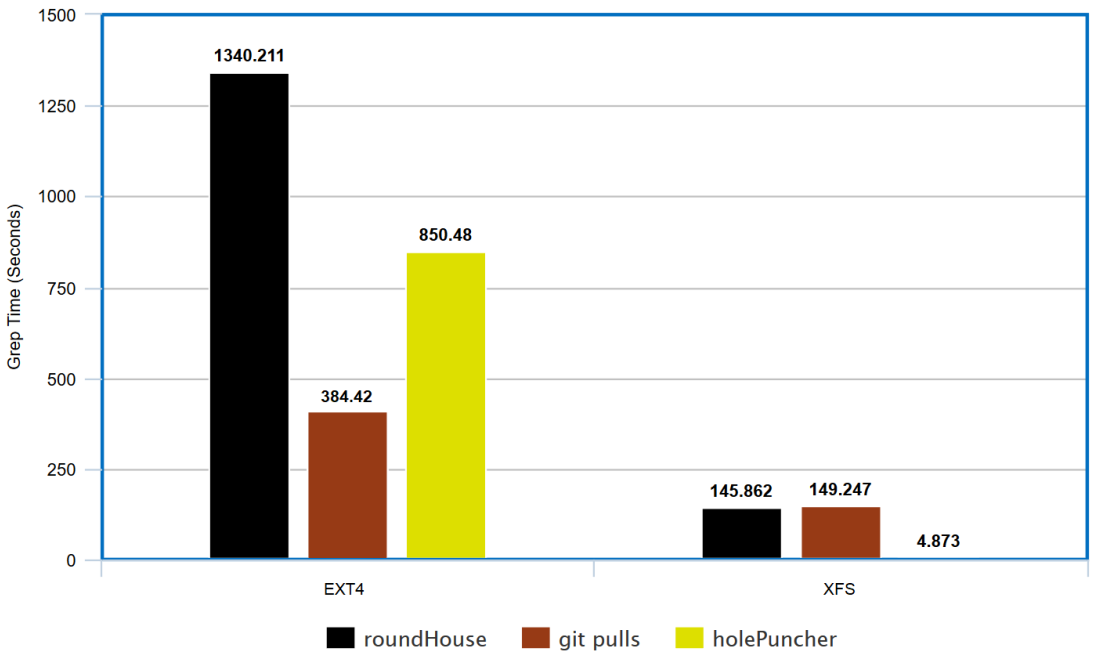

Figure 3: Grep test time of EXT4 and XFS in response to aging

As we can see, the grep time for EXT4 is the highest when the roundHouse tool is applied, with holePuncher also having a significant effect. The reason that roundHouse has a larger effect on the grep test is that the recursive search is bogged down when there are a large number of folders. Since the holePuncher tool

creates no new folders, its effect on the grep search is small compared to roundHouse. For XFS, the effect of holePuncher is negligible, but both roundHouse and the git pulls method have noticeable impact (as well as achieving similar results).

The reason that these tools have a different impact on the two file sytems is because of the differences in the file systems themselves. A standard EXT4 file system is made up of tens of thousands of block groups, while XFS generally has very few (<10) of its own groups (called *activity groups*). When holePuncher is executed on an EXT4 system, the initial large file is distributed over many of the block groups, meaning that to scan a file, each of the many containing block groups must be checked (which increases checking time). For an XFS system, holePuncher is ineffective, because there is only negligible slowdown associated with checking each of the few activity groups. So, the standard round-robin sequence of file operations works better. Both methods end up beating the git pulls method (on average) for their respective file systems. The git pulls method does have an impact on each system, but it is a substantially smaller one than the two chosen methods, in part because (due to the only slight variations in structure from each pull) it takes much more time compared to the other two methods for a large impact to be obtained.

Based on the information gathered from these tests, we choose holePuncher as the tool to age the EXT4 file systems, and roundHouse as the tool to age the XFS file systems.

4.2 Checker Response to Corruption

The configurations chosen for both file system types are as follows: For both EXT4 and XFS, we created file systems of size 100GiB, 300GiB, 500GiB, and 800GiB. For each of these four sizes, we created images that had unique block and inode sizes.

I. Chosen Block Sizes - The block sizes that were chosen for both EXT4 and XFS were 1024 bytes and 4096 bytes.

II. Chosen Inode Sizes - For EXT4, the chosen inode sizes were 128 bytes, 256 bytes, and 1024 bytes. For XFS, the chosen inode sizes were 256 bytes, 512 bytes, and 1024 bytes. (The reason that a size of 512 bytes is used for XFS instead of 128 bytes is that xfs_progs 4.15.1 does not allow an inode size of 128 bytes).

To get a clear picture of how the induced corruption affected the file system checkers in different configurations, the following tests were performed. First, each file system configuration was aged for approximately 4 hours with the appropriate aging tool. Next, the checker was run and monitored to determine the average runtime and memory usage peak *before* any induced corruption. Then, specific metadata components were corrupted, the checker was run again, and the differences in memory and runtime were recorded.

For EXT4, the metadata components that were corrupted were the Superblock, Group Descriptors, Reserved GDT Blocks, Block Bitmap, Inode Bitmap, and the

Inode Table. For XFS, the components that were corrupted were the Inodes, Superblock, B+ Tree Inodes, B+ Tree Block Number, and the B+ Tree Count.

First, the average amount that the runtime of xfs_repair increased for all XFS configurations is displayed in Figure 4.

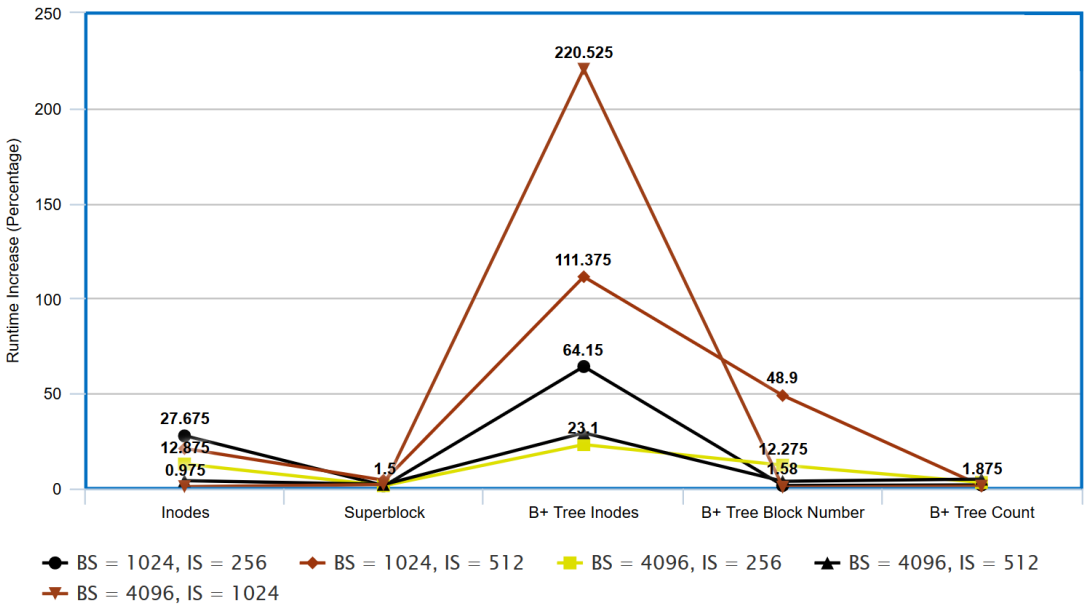

Figure 4: xfs_repair runtime growth after corruption

As is shown, the corruptions that caused the greatest slowdown were consistently those affecting the B+ tree inodes of each configuration. It can also be seen that corruption of the superblock and the B+ tree count caused the least amount of slowdown. Next, the average amount that the peak memory usage of xfs_repair increased for all XFS configurations is displayed in Figure 5.

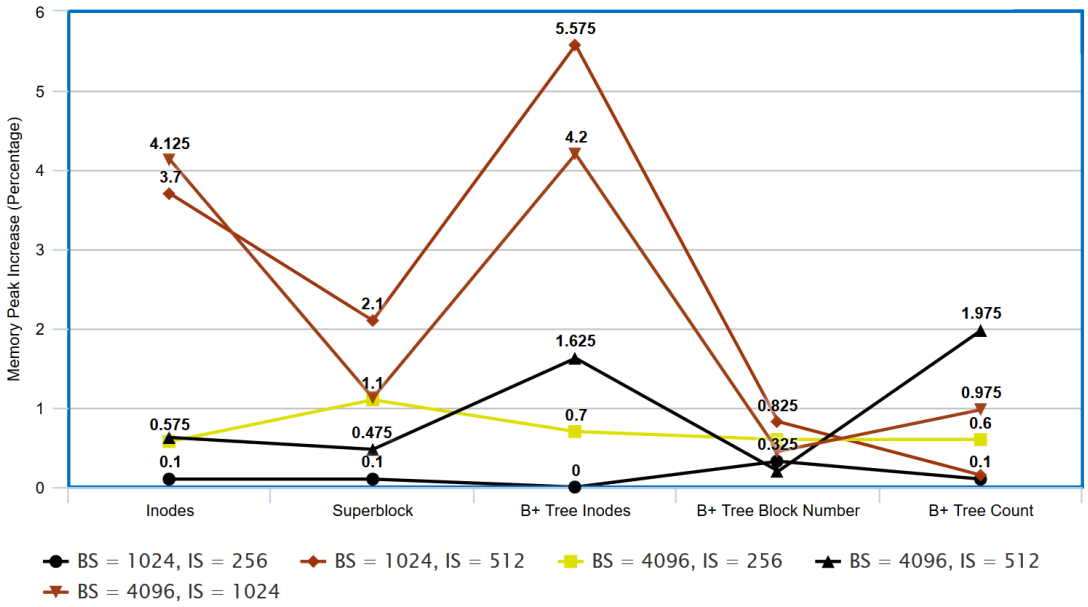

Figure 5: xfs_repair memory peak growth after corruption

As is shown, the corruptions that caused the greatest memory peak increase were consistently those affecting the B+ tree inodes of each configuration, followed by those affecting the superblock. It can also be seen that corruption of the B+ tree block number caused the least amount of memory increase. Next, the average amount that the runtime of e2fsck increased for all EXT4 configurations is displayed in Figure 6.

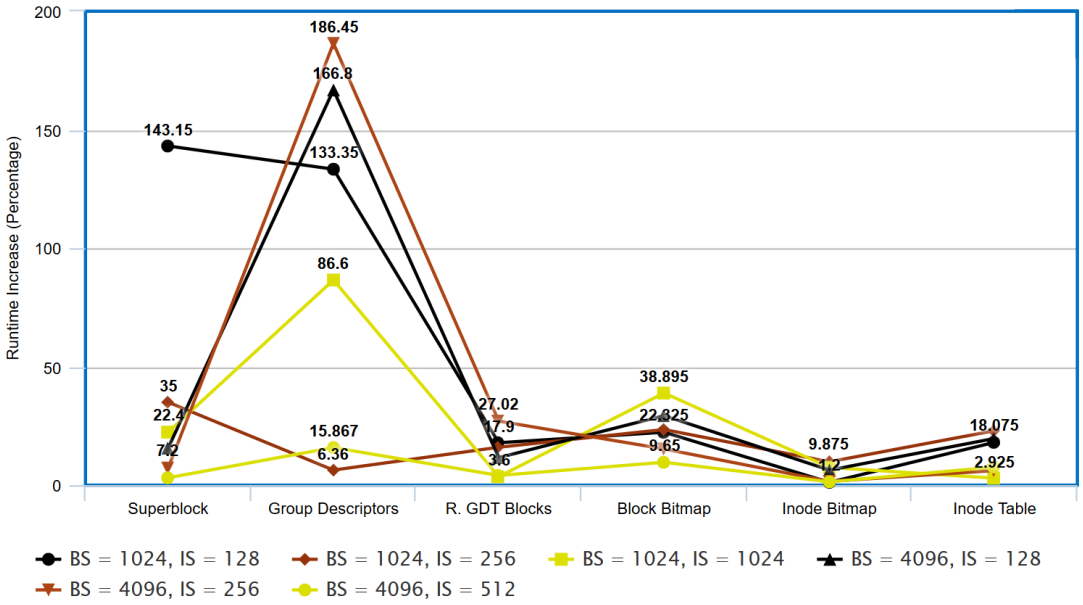

Figure 6: e2fsck runtime growth after corruption

As is shown, the corruptions that caused the greatest slowdown were consistently those affecting the group descriptors of each configuration. We can also see that corruption of the inode bitmap caused the least amount of slowdown. Next, the average amount that the peak memory usage of e2fsck increased for all EXT4 configurations is displayed in Figure 7.

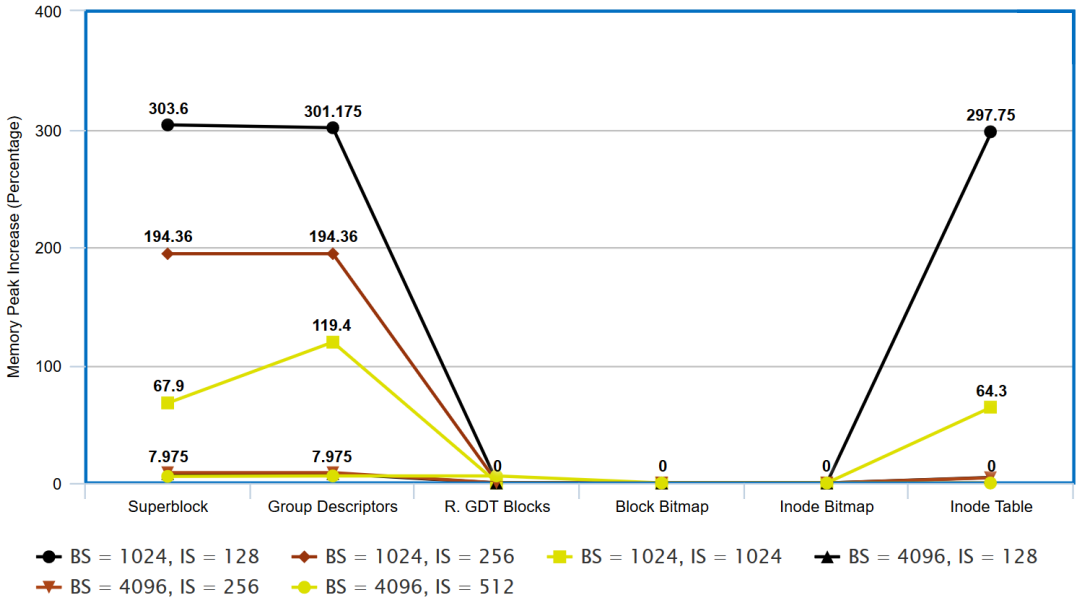

Figure 7: e2fsck memory peak growth after corruption

As is shown, the corruptions that caused the greatest memory peak increase were those affecting the superblock and group descriptors of each configuration. It can also be seen that corruption of the reserved GDT blocks, block bitmap, and inode bitmap caused the smallest memory increase.

Given these performance measurements, we can see that for XFS and EXT4, our proposed aging tools behave competitively in regard to grep tests, and are in fact preferable for scenarios that require checker slowdown. Secondly, our experimental results have shown clear corruption-effect patterns. For XFS, corruption of the B+ tree inodes has the greatest negative impact on xfs_repair, and for EXT4, corruption of the group descriptors impacts e2fsck markedly more than other corruption types.

5 FUTURE WORK

From the baseline given in this project, there are many possible avenues of topic expansion. The most obvious route of improvement would be to run the same tests and analysis on more file systems. XFS and EXT4 are only two of the many popular file system choices, so a logical next step would be to analyze other similarly popular systems such as btrfs, F2FS, NTFS, etc.

Another way to expand on the gathered results would be to run the same experiments with several different degrees of aging per configuration. Currently, each configuration is aged to the same extent, but more detailed results could be achieved if we compared a set of systems aged for 12 hours to a set aged for 36 or 48 hours, and so on.

Lastly, the information collected from this project could ideally be used to directly improve the checkers that were tested. Since through this project we have determined the leading metadata components responsible for degrading checker performance, this could prompt an excursion into the source code of xfs_repair and e2fsck, with the goal of determining the cause or causes of the negative effect of repairing the aforementioned metadata. Which in turn, could lead to an improvement of those checkers, specifically in response to weaknesses discovered through this project.

6 CONCLUSION

File systems checkers are something of a necessary evil. It is almost never desirable to halt workflow for the checker to slowly walk through the repair process, but as has been previously established, the inevitable scenario often arises where we are left with no other recourse. Thus, further analysis of file system checkers to discover the leading contributors of runtime degradation is a logical next step.

To that end, many test configurations of two popular file systems (EXT4 and XFS) are analyzed in this project across a broad range of sizes (100GiB-800GiB). Each configuration is sufficiently aged with appropriate tools to simulate a more realistic system state. After aging, runtime and memory use of each systems respective checkers is measured in response to induced metadata corruption to provide insight on performance trends over a large pool of system configurations.

The results of these analyses provide an unambiguous picture of which metadata corruptions have the largest impact on checker performance. Group descriptor corruption had the greatest effect on the performance of e2fsck, and B+ tree inode corruption had the greatest effect on the performance of xfs_repair.

Throughout this project, we have not only gathered important information on file system checker performance trends in regard to metadata corruption, but we also created accessible tools for effective file system aging and corruption, which have potential for future use in a variety of scenarios.

REFERENCES

[**1**] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. ffsck: The Fast File System Checker. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 13)*, San Jose, California, February 2013.

[**2**] A. Conway, A. Bakshi, Y. Jiao, W. Jannen, Y. Zhan, J. Yuan, M. A. Bender, R. Johnson, B. C. Kuszmaul, D. E. Porter, et al. File Systems Fated for Senescence? Nonsense, Says Science! In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 17)*, Santa Clara, California, February - March 2017.

[**3**] O. R. Gatla, M. Hameed, M. Zheng, V. Dubeyko, A. Manzanares, F. Blagojevic, C. Guyot, and R. Mateescu. Towards Robust File System Checkers. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST 18)*, Oakland, California, February 2018.

[**4**] High Performance Computing Center (HPCC) Power Outage Event. Email Announcement by HPCC, Monday, January 11, 2016 at 8:50:17AM CST. https://www.cs.nmsu.edu/mzheng/docs/failures/2016-hpcc-outage.pdf, 2016.

[**5**] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A Flexible Framework for File System Benchmarking. *USENIX ;login:* 41, 1 (2016).

[**6**] fs_mark: https://github.com/josefbacik/fs_mark

[**7**] S. Kadekodi, V. Nagarajan, and G. A. Gibson. Aging Gracefully with Geriatrix: A File System Aging Suite (2016).

[**8**] M. Seltzer, and K. Smith. File system aging: Increasing the relevance of file system benchmarks. In *Proceedings of the 1997 ACM SIGMETRICS*, ACM, New York, 1997.