FaultyRank: A Graph-based Parallel File System Checker

Saisha Kamat[1], Abdullah Al Raqibul Islam[1], Mai Zheng[2], and Dong Dai[1]

[1]Computer Science Department, University of North Carolina at Charlotte, {skamat1, aislam6, ddai}@uncc.edu
[2]Department of Electrical and Computer Engineering, Iowa State University, {mai@iastate.edu}

***Abstract*—Similar to local file system checkers such as e2fsck for Ext4, a parallel file system (PFS) checker ensures the file system's correctness. The basic idea of file system checkers is straightforward: important metadata are stored redundantly in separate places for cross-checking; inconsistent metadata will be repaired or overwritten by its 'more correct' counterpart, which is defined by the developers. Unfortunately, implementing the idea for PFSes is non-trivial due to the system complexity. Although many popular parallel file systems already contain dedicated checkers (e.g., LFSCK for Lustre, BeeGFS-FSCK for BeeGFS, mmfsck for GPFS), the existing checkers often cannot detect or repair inconsistencies accurately due to one fundamental limitation: they rely on a fixed set of consistency rules predefined by developers, which cannot cover the various failure scenarios that may occur in practice.**

In this study, we propose a new graph-based method to build PFS checkers. Specifically, we model important PFS metadata into graphs, then generalize the logic of cross-checking and repairing into graph analytic tasks. We design a new graph algorithm, FaultyRank, to quantitatively calculate the correctness of each metadata object. By leveraging the calculated correctness, we are able to recommend the most promising repairs to users. Based on the idea, we implement a prototype of FaultyRank on Lustre, one of the most widely used parallel file systems, and compare it with Lustre's default file system checker LFSCK. Our experiments show that FaultyRank can achieve the same checking and repairing logic as LFSCK. Moreover, it is capable of detecting and repairing complicated PFS consistency issues that LFSCK can not handle. We also show the performance advantage of FaultyRank compared with LFSCK. Through this study, we believe FaultyRank opens a new opportunity for building PFS checkers effectively and efficiently.

I. Introduction

To store and manage the massive data, HPC platforms heavily rely upon parallel file systems (PFSes), such as Lustre [1], GPFS [2], and PVFS [3], to serve data access requests from scientific applications. Therefore, the reliability of parallel file systems is critically important. However, as the scale and complexity of HPC systems rapidly increase, even the carefully-designed and well-maintained parallel file systems may fail and run into inconsistent states due to various reasons including hardware faults, software bugs, configuration errors, human mistakes, etc [4].

When a file system is in an inconsistent state, a checking and repairing program called *file system checker* is needed to bring the file system back to a consistent state. Essentially, file system checkers rely on the redundant metadata stored in different places of the file systems to work. The checkers can be either invoked explicitly or triggered implicitly to scan the file systems and cross-check whether the redundant metadata match with each other. If not, the checker may report and attempt to repair the inconsistencies. File system checkers have been widely used in local file systems, such as `e2fsck` [5] for Ext2/3/4 and `xfs_repair` for XFS [6]. Similarly, parallel file system checkers are also critically important for ensuring the integrity of these PFSes. For instance, LFSCK is responsible for checking and repairing Lustre file system [7], BeeGFS-FSCK is used for BeeGFS [8], and mmfsck is designed for GPFS [9]. Although the importance of the parallel file system checker has been well agreed upon, designing and implementing an effective checker that can identify and repair complicated inconsistencies under various failure scenarios is still challenging based on recent studies [10, 11].

The main challenge comes from the vast amount of possible failure scenarios that the PFSes may experience in practice. These distinct scenarios lead to a variety of different inconsistency issues in the end and require different logic to detect and repair them. Designing the complete set of logic for all possible inconsistencies is notoriously complicated for the developers. They often have to settle using a set of limited and fixed rules. For example, in most cases, Lustre's checker LFSCK simply checks whether the metadata stored in data object servers (OSS) matches its counterpart in the metadata servers (MDS). If not, LFSCK will directly use MDS metadata to overwrite the OSS metadata regardless of the root cause of the inconsistency. Hence, it is not surprising that these rule-based checkers may fail at repairing complicated PFS inconsistencies as reported in previous studies [10, 11].

In this study, we take a different approach to building parallel file system checkers. Instead of relying on manual efforts to specify fixed rules to check and repair inconsistencies, we model the metadata of parallel file systems into a metadata graph and leverage their point-to and point-back relationships as graph edges to understand the correctness of metadata [12]. Based on the metadata graph model, we design an iterative algorithm called FaultyRank to quantitatively calculate the *credibility score* of each metadata field. The calculated scores are then used to identify the root cause of the inconsistency and determine the corresponding repair strategy.

We implement a prototype of FaultyRank[1] in an offline manner (detailed in Section IV) on the widely used Lustre parallel file system and compare it with Lustre's default

[1]https://github.com/DIR-LAB/FaultyRank

checker (i.e., LFSCK). Our experiments show that FaultyRank can achieve the functionality of LFSCK elegantly. Moreover, we find that FaultyRank outperforms LFSCK in effectiveness and efficiency: it can detect and repair various complicated inconsistencies that LFSCK cannot handle, and it outperforms LFSCK in terms of speed by up to 10x due to its holistic design. In summary, the main contributions of this study include the following:

- Designing a graph model to describe complicated PFS checking-relevant metadata structures in a unified way. To the best of our knowledge, this is the first graph model to abstract PFSes metadata for checking.
- Developing an iterative algorithm FaultyRank to quantitatively calculate the correctness of different PFS metadata to help detect the root causes of inconsistencies and identify optimal repair strategies.
- Building a prototype of FaultyRank for the widely used Lustre file system and demonstrating the improvement over its own state-of-the-art checker in terms of both effectiveness and efficiency.

The rest of this paper is organized as follows. In Section II, we introduce background knowledge on the state-of-the-art parallel file system checkers. We present the core design of FaultyRank in Section III and its Lustre-based prototype implementation in Section IV. We then evaluate FaultyRank in Section V, and further discuss its generality and limitations in Section VI. We discuss related work in Section VII and lay out the future work in Section VIII.

II. Background and Motivation

In this section, we first introduce the internal architecture of parallel file systems and how the corresponding checkers work. We then analyze the fundamental issues of existing PFS checkers, which motivate our ideas. To make the discussion concrete, we use Lustre and its checker LFSCK as one specific example. Other parallel file systems may have different internal data structures and checker implementations but share similar design principles and underlying limitations.

A. Lustre Architecture and Metadata

Fig. 1 shows a typical Lustre cluster, which includes three types of servers: management server (MGS), metadata server (MDS), and object storage server (OSS). The management server (MGS) and metadata servers (MDSes) are often combined to store the configuration information and the namespace metadata of the file system. The object storage servers (OSSes) store the actual data of the file system. Files are stripped into fixed-size chunks and stored as data objects on OSSes. On both MDS and OSS servers, Lustre leverages the local file system, such as Ext4-based ldiskfs [13] and ZFS [14], to store the data and metadata.

There are essentially two categories of metadata in Lustre: the *namespace* and *data layout* metadata. The namespace metadata maintains the directory tree of the Lustre file system, such as the directory and its files. The data layout metadata maintains the relationships between the Lustre file and its

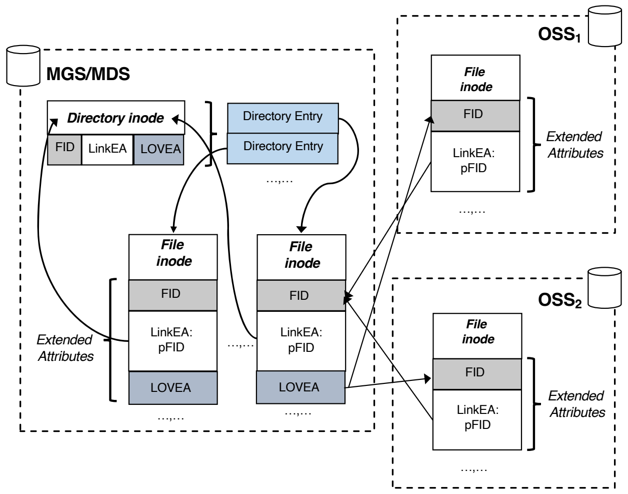


Fig. 1: The architecture of a Lustre cluster and the key metadata structures on both MDS and OSS.

stripes stored on different object storage servers. Lustre stores both metadata on the local file system.

Specifically, each Lustre file, directory, or data object corresponds to a file in the local file system. Their metadata are embedded into the Extended Attribute (EA) field of the inode of the corresponding local file. For instance, on MDS, each local file inode's extended attributes store the FID of the corresponding Lustre file, the LinkEA field pointing to its parent directory, and the LOVEA field pointing to all its stripe objects stored on OSS servers. On OSS, each local file's inode's extended attributes store the FID of the corresponding Lustre stripe object and the LinkEA field pointing to the file that the stripe belongs to, as shown in Fig. 1. To maintain the Lustre namespace metadata, the directory entry still exists and is extended to point to the child files or directories by storing both their local inode id and Lustre FIDs. The FID will point to the child file/directory in that directory. The child file or directory will use its LinkEA to point back to its parent directory by storing the parent's FID.

As we can see from Fig. 1, each metadata stored in Lustre has a redundant counterpart for cross-checking. For instance, if a directory entry of a directory points to a sub-file, then that sub-file will have its LinkEA field point back to the directory. If the bi-directional mapping is violated, it implies that Lustre is in an inconsistent state.

B. Parallel File System Checker and LFSCK

The goal of a parallel file system checker (e.g., LFSCK for Lustre) is to ensure that the metadata of the parallel file system is correct, i.e., the redundant metadata is consistent.

Fig. 2 shows a simplified example of metadata consistency. It contains two metadata objects a and b, where a has a property pointing to b; while b has a property pointing back to a. In Lustre, such a model can be mapped to many checking cases. For instance, a can be the MDS file object and b can be

Types of Inconsistency	Potential Root Causes	LFSCK Behaviors
a's property can not locate b (**Dangling Reference**)	a's property is wrong; b's id is correct or wrong. a's property is correct; b's id is wrong.	Ignore and can not identify/repair. Identify but do not repair b's id, put b in "lost+found".
No object refers to b. (**Unreferenced Object**)	b's id is correct, but all its neighbors' properties are wrong. b's id is wrong; a's property could be correct or wrong.	Ignore and can not identify/repair. Identify but do not repair b's id, put b in "lost+found".
More than one objects refer to b (**Double Reference**)	a's property duplicates c; both point to b. b's id duplicates c; a points to both b and c.	Ignore and can not identify/repair. Identify but do not repair b's id, put b in "lost+found".
Mismatch between a and b (**Mismatch**)	a's property and b's id are both correct; b's property is wrong. a's property and b's id are both correct; a's id is wrong.	Identify and correctly use a's id to repair b's property. Ignore and can not identify/repair.

TABLE I: Four categories of inconsistency, the potential root causes, and the corresponding behaviors of LFSCK.

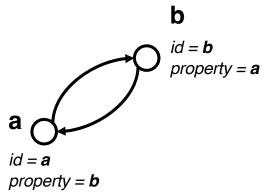


Fig. 2: A simplified example of metadata consistency.

its OSS stripe object, where a's property is LOVEA, pointing to an OSS stripe object, while b's property is LinkEA, pointing back to the MDS object.

Fig. 2 represents a consistent state where the two metadata objects point to each other correctly. In practice, however, a parallel file system may run into various inconsistent states. Table I lists four major types of metadata inconsistencies, their potential root causes, and the corresponding behaviors of LFSCK, based on the LFSCK design documents [15]. For clarity, we use Fig. 2 as a simplified scenario to illustrate the four different inconsistency cases below.

Specifically, the first type is *Dangling Reference*, which means a's metadata property should refer to b, but could not. There are two potential root causes for this inconsistency: 1) a's property is wrong, or 2) b's id is wrong. For instance, the MDS object uses the LOVEA property to refer to its OSS objects. So each LOVEA of an MDS object should contain a valid FID to locate an OSS object. But the LOVEA property could be wrong and refer to non-exist OSS objects, or the LOVEA is correct, but the OSS object's ID is wrongly assigned. Both of them may lead to the dangling reference issue. To handle such a case, LFSCK simply assumes whatever is stored in MDS or parent directory to be correct and should overwrite the counterpart. As a result, it cannot identify the potential root cause 1 and only puts b into 'lost+found' based on the assumption of root cause 2.

The *Unreferenced Object* case means an object b exists, but no other objects could reference it. For instance, an OSS stripe object exists, but there is no MDS file claiming the OSS stripe as part of it. Similarly, there are two possible root causes: 1) b's id is correct, then all its neighbors' properties must be wrong so that they can not refer it; 2) b's id is wrong, then a can not refer it. Similar to the *Dangling Reference* case, LFSCK does not identify a being wrong. It always assumes b is in trouble and fixes it by placing b in 'lost+found'.

The *Double Reference* case means more than one object claims the same relationship with b. It typically involves a third object c, whose property is replicated by a, hence both of them point to b; or whose id is replicated by b, hence a points to both objects b and c. The duplication case is difficult for LFSCK to handle as the sequential scanning in LFSCK does not identify duplication. Most of the time, LFSCK will simply treat such cases as *Dangling Reference* or *Inconsistency*.

The *Mismatch* case indicates scenarios where a can successfully refer b, but object b can not point back. This might be because b's property is wrong or a's id is wrong. Again, LFSCK will not consider a's id as being wrong and will simply overwrite b's property. So, it repairs the system based on the assumption of root cause 1, ignoring root cause 2.

Limitations of Existing PFS Checkers. Based on the discussions above, we can see that even state-of-the-art PFS checkers like LFSCK suffer from key limitations. First, they are designed and implemented using fixed rules predefined by developers, such as "metadata stored in MDS should overwrite its counterpart stored in OSS". Although these rules are designed based on the domain knowledge, real-world inconsistency scenarios could easily be more complicated, making these fixed rules inadequate and inaccurate. Additionally, due to the complexity of inconsistent scenarios, even PFS developers may not be able to design strategies optimally beforehand. They often have to play safe by placing files or their stripes into 'lost+found' and rely on users to manually fix it later, which is increasingly difficult and inconvenient for end users as the scale of HPC storage grows.

C. Our Observations & Key Idea

Despite the complexity of building PFS checkers, we observe that it is possible to identify the actual root cause for inconsistency by checking the file system more "intelligently." For example, if MDS object a can not refer to its OSS child b, we can check if a can refer to its other OSS children, as most files will contain multiple stripes. If a can not refer to any of them, then the a's property may be wrong instead of naively assuming b's id is wrong. Similarly, for namespace metadata, if the parent directory contains multiple sub-directories and files and all of their LinkEAs point to the same 'wrong' FID, then the parent directory's FID may be wrong instead of assuming all LinkEAs are wrong.

The key to achieving intelligence is to comprehensively check more relevant relationships to identify the root cause of inconsistency accurately. To this end, we observe that the relationships among PFS metadata are similar to those of web pages: a web page with more incoming links may be considered more authentic than others; web pages that are linked by important pages will also be more authentic. Inspired by how web pages are modeled and ranked in search engines [16], we propose to model the PFS metadata into a graph structure and quantitatively calculate the credibility score of each metadata field, such as a's property or b' id, in a global way, based on the point-to and point-back links between graph vertices. We then leverage the calculated credibility scores to determine the accurate root cause of inconsistency.

To be more specific, the point-to and point-back links between metadata in PFSes, such as the edges in Figure 2, work similarly as hyperlinks between web pages contributing credibility to each other. For example, if object a points to object b by correctly having its property refer to b's id, then the credibility of b's id is reinforced by such a link. Further, if there are lots of objects pointing to object b, then b's id is highly possible to be correct as it is unlikely that other objects' properties are all wrong but point to the same b. Additionally, such credibility is reversible. The credibility of b's id also contributes back to object a's property if a successfully pairs with b. Following such an idea, in this study, we propose FaultyRank, a PageRank-like algorithm, to calculate the credibility score of metadata for parallel file system checking. We describe the detailed algorithm design and implementation in the next section. Note that there are still important differences between FaultyRank's calculation of metadata credibility and PageRank's ranking of web pages. For example, in file system checking, we focus on the extremely low credibility scores for locating inconsistencies, not the rank values of different objects. The low-degree nodes in our system may have smaller credibility scores due to low connectivity, but they will still be considered correct as long as their links are consistent and receive enough credits from neighbors.

III. Design of FaultyRank

In this section, we first introduce the design of the proposed FaultyRank algorithm assuming the metadata graph has been built. We then introduce a prototype implementation of FaultyRank on the Lustre file system in the next section to explain how to build the graph and run FaultyRank in real-world settings.

A. Metadata Graph

To run FaultyRank, we assume the metadata graph has been built. The metadata graph is essentially a directed graph to represent the PFS-level metadata. For example, the graph vertices can represent the PFS directories, files, and stripe objects. The directed edges show the point-to and point-back relationships between these metadata objects. The left part of Figure 3 shows an example of the metadata graph built from the Lustre file system. It contains two Lustre files: `b` and `c` under the same Lustre directory `a`. As part of the namespace metadata, their metadata are stored on Lustre MDS server. The file b further contains multiple stripes, and object d is one of them, stored on one OSS server. The edges connect these vertices through corresponding properties of the vertices. For instance, a's DIRENT property identifies all the subdirectories and files; d's LinkEA property points to its file b. In the next section, we will discuss how the metadata graph was built.

Normally, there should always be paired edges between objects, for instance, directory a has its DIRENT property pointing to files c and b, each of which will have LinkEA property pointing back to a. In Fig. 3, we can see c's LinkEA property is missing, introducing an inconsistency. Similarly, b's LOVEA property is also missing to point to object d, whose LinkEA property points back to b correctly. We will show how FaultyRank can identify these inconsistencies later.

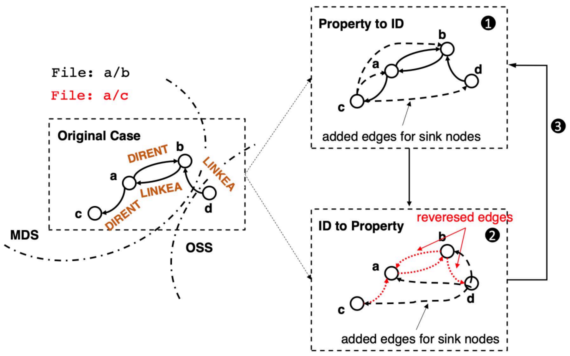


Fig. 3: FaultyRank iterative algorithm workflow.

B. ID Rank and Property Rank

In FaultyRank, we consider two ranks to calculate for each vertex as there are two major metadata fields for each object in the parallel file systems: its unique ID which is pointed back by other vertices, and its Properties which point to other vertices. Since these two fields are often updated in different file system operations, such as FID for file/directory creation and Properties for sub-directory creation, we consider their correctness to be independent and calculate them separately in FaultyRank.

Specifically, we define two credibility scores for each object: *ID Rank* (id_{rank}) and *Property Rank* ($prop_{rank}$), which correspond to the credibility score of ID and Properties, respectively. Note that we do not further differentiate the possible multiple properties of an object as we consider it unlikely that one of the extended attributes is wrong but others are correct. We plan to investigate how FaultyRank would work in this scenario in the future work.

C. Iterative Algorithm

The key idea of FaultyRank is to leverage the point-to and point-back edges between metadata objects to calculate their credibility. We summarize the core iterative algorithm in Alg. 1 and discuss it in detail below.

First, we assume each object has an initial ID Rank ($id^0_{rank} = 1$) and Property Rank ($prop^0_{rank} = 1$). The superscript 0 means iteration 0. Then, for each object, we can calculate its new ID Rank (id^1_{rank}) by aggregating the current Property Ranks ($prop^0_{rank}$) of all its neighbors who point to it. The intuition is very simple: the credibility of an object's ID Rank can be calculated by: 1) how many other neighboring objects point to it, and 2) how credible these neighbors' properties are. Such a calculation will be done for all metadata objects in the graph.

After obtaining the new ID Rank for each object (id^1_{rank}), the next step is to obtain the new Property Rank ($prop^1_{rank}$) for each object. As we discussed earlier, to update the credibility of the properties, we just need to check whether these properties correctly point to some credible IDs. The more correct IDs they point to, the higher the possibility is that these properties are correct. As we consider the direction of an edge means the credibility will be contributed from the source to the destination vertex, we can simply reverse (logically) all the edges from objects' Properties to IDs to do the calculation. With the reversed graph, we can calculate its new Property Rank ($prop^1_{rank}$) by aggregating the ID Ranks (id^1_{rank}) of all its neighbors, similar to the previous calculation.

We then repeat these two steps in multiple iterations until it converges: the *diff* value of id_{rank} in two consecutive iterations is smaller than ϵ. We use $\epsilon = 0.1$ in our experiments, which typically leads to less than 20 iterations. The final [id_{rank}, $prop_{rank}$] value for each object simply indicates the credibility of its ID and Property.

Algorithm 1 FaultyRank Iterative Algorithm

```
1: ◇ Metadata graph G and reversed graph $G_R$
2: ◇ Initial ranks: id_rank[G.v]=1 and prop_rank[G.v] = 1
3: while diff $> \epsilon$ do                          ▷ loop until converged
4:     for $v \in G$ do                                ▷ calculate ID rank
5:         sink nodes handling
6:         s = prop_rank[v]/outdegree(v)
7:         for $v_{out} \in v$.out-going-neighbors() do
8:             id_rank[$v_{out}$] += s
9:         end for
10:    end for
11:    for $v \in G_R$ do                        ▷ calculate Property rank
12:        sink nodes and weighted distribution handling
13:        s = id_rank[v]/outdegree(v)
14:        for $v_{out} \in v$.out-going-neighbors() do
15:            prop_rank[$v_{out}$] += s
16:        end for
17:    end for
18:    diff = calc_diff()
19: end while
```

D. Sink Nodes and Weighted Distribution Handling

In Alg. 1 lines 5 and 12, we introduce logic for processing sink nodes and handling weighted distribution.

First, handling *sink nodes* is a traditional task for PageRank-like algorithms. Here, sink nodes in a directed graph indicate those vertices that do not have any outgoing edges. During the iterative calculation, the rank values get lost due to these sink nodes. There are multiple ways to handle them in the PageRank algorithm [17]. In FaultyRank, we simply assume these sink nodes will point to all other vertices in the graph. Hence their rank values will be distributed to all other vertices.

The *weighted contribution*, however, is introduced in FaultyRank to particularly address a credibility distribution problem. As we described earlier, the fundamental insight of FaultyRank is if a node a points to node b by making $a.prop = b.id$, then we consider the credibility of $a.prop$ should contribute to $b.id$. In addition to that, FaultyRank also considers the reverse also works. Specifically, it leverages ID Ranks to update the Property Ranks. This is where the reversed graph is introduced. This makes sense because if $a.prop = b.id$ and $b.id$ is known to be correct, then $a.prop$ should be rewarded for pointing to b. However, such an intuition might be wrong in some cases because anyone can point to b to increase their own credibility. If an object falsely points to b, it still gets rewarded on its Property Ranks, which is not correct.

To address this issue, FaultyRank lowers the weights of unpaired edges in the reverse graph to penalize the objects which wishfully point to a high credibility object but do not receive an acknowledgment from it. Figure 4 shows an example. On the left, we show a normal graph that contains paired objects a and b, which will lead to highly credible ID Ranks and Property Ranks for both of them. At the same time, objects a and c do not have the paired edges.

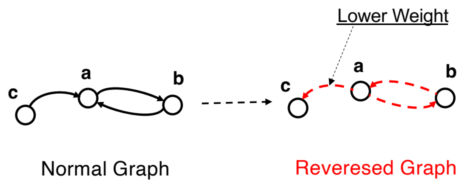


Fig. 4: An example of weighted distribution.

In the reversed graph, we can observe the unpaired edge from a to c. Based on the original FaultyRank, $a.id_{rank}$ should be equally distributed to $c.prop_{rank}$ and $b.prop_{rank}$. However, we should not treat b and c the same as b receives an acknowledgment from a but c does not (as shown in the normal graph). In FaultyRank, we empirically lower the weights of unpaired edges in the reversed graph to $\frac{1}{10}$ of normal edges. So, in this particular case, $b.prop_{rank}$ will receive $\frac{a.id_{rank}*10}{11}$ and $c.prop_{rank}$ will only receive $\frac{a.id_{rank}}{11}$. After several iterations, such a difference will lead to a very low $c.prop_{rank}$, which helps identify the root cause for such an unpaired inconsistency.

E. FaultyRank Running Example

The right part of Figure 3 shows the execution flow of FaultyRank on the example metadata graph, which has some inconsistencies. It first runs on the original graph to calculate the id_{rank} (❶), then runs on the reversed graph to calculate the $prop_{rank}$ (❷) until it converges.

We show the results of this FaultyRank calculation in Table II, which contains the final [id_{rank}, $prop_{rank}$] values of these four objects. From these results, we can observe that the Property Rank of object c and ID Rank of object d are extremely small (0.05) compared with other objects. Such a small value generally means the particular metadata field lacks support from other objects and hence is more likely to be wrong and causes the inconsistency. The results do match well with how we manually introduced the inconsistency in the original case, where we removed the LINKEA property of c and changed the ID of object d.

TABLE II: ID and Property Ranks of the example graph

Object Name	**ID Rank** (id_{rank})	**Property Rank** ($prop_{rank}$)
Object a	0.35	0.39
Object b	0.39	0.35
Object c	0.2	**0.05**
Object d	**0.05**	0.2

These results show a key advantage of FaultyRank: it can differentiate the root causes of the inconsistency. In Table I, we have discussed for each paired objects, the inconsistency may come from either side, such as the *dangling reference* may occur due to a's property or b's id being incorrect. Existing rule-based file system checkers can not differentiate them well and simply use what is stored on the metadata server to overwrite its counterparts. Using FaultyRank, we can clearly tell the inconsistency between a and c should come from c's property instead of a's id, simply because a's id has been correctly pointed to by another credible object b. Similarly, the inconsistency between b and d should be from d's id instead of b's property, because b's property has correctly pointed to a and should be correct.

F. Identifying and Fixing Inconsistency

After calculating the ranks ([id_{rank}, $prop_{rank}$]) for each graph node, the next step is to leverage these ranks to identify the reasons for inconsistencies and repair them.

To do that, we iterate graph vertices to examine whether the node has paired edges with all of its neighbors. The nodes with unpaired edges will be recorded in set S_{chk} for the next stage of checking. This step is typically done during FaultyRank iterations. Note that, although we do not record nodes with paired edges, we do not assume they will always be correct. In fact, for paired nodes a and b, it is possible that both of their properties $a.prop$ and $b.prop$ are wrong but successfully point to each other. If that happens, there will be another object c pointing to a but missing a proper point-back from a. In this case, the (a,c) paired will be recorded for further checking.

We then iterate through all records in S_{chk} and check their ID Ranks and Property Ranks. We use a threshold value of 0.1 to determine whether the corresponding field (ID or Property) might be incorrect. If the rank value is smaller than the threshold, then we consider the corresponding field as the reason for the inconsistency and repair it by overwriting its value based on its counterpart's value. In Figure 5, we show an example of how this procedure works.

Here, we show a **Mismatch** inconsistency case, where a and b *mismatch*, and its two possible reasons. From the users' perspective, the observation is the same: a points to b, but

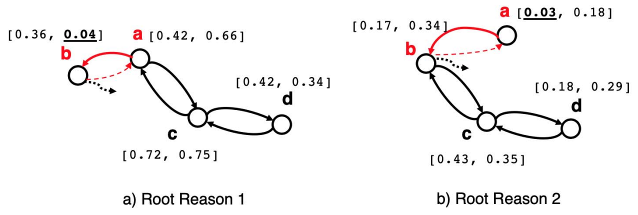


Fig. 5: The calculated ID Rank and Property Rank [id_{rank}, $prop_{rank}$] of two different cases, which have the same inconsistency observation.

b does not point back. As we have discussed in Section III, there are actually two possible reasons for such a case: 1) b's property is wrong so that it does not point to a, or 2) a's id is wrong so that b can not point to a. LFSCK does not differentiate them and simply uses the metadata stored on MDS or as the parent directory to overwrite its counterpart. In FaultyRank, however, we can accurately calculate the different rank values for both a and b's properties and ids. It essentially calculates a skewed distribution of correctness among nodes based on the existence of inconsistencies. The distribution then helps locate the errors. In the left part of Figure 5, we can observe $b.prop_{rank}$ is much smaller than 0.1 and therefore chosen to be the wrong one compared with $a.id$, which equals 0.42. In the right part, we can observe $a.id_{rank} = 0.03$ becomes extremely small while $b.prop_{rank} = 0.34$ is larger. These results directly tell us what is the root cause. Knowing that, delivering fixes becomes simple: if one node's property is wrong, we find out the corresponding unpaired node and use its id to overwrite the property; if one node's id is wrong, we find out the corresponding unpaired node and use its property to overwrite the id. The fixes are shown in Figure 5 as red dashed lines.

From this example, we can see the key for FaultyRank to differentiate the root causes of inconsistency is the extra edges connecting with other nodes. For example, the paired edges between a and c suggest both a's ID and Property are likely to be correct, while on the other hand, b does not have other supporters on its property. Together, they make us believe b's property is more likely to be the reason. That being said, if there are only two graph nodes a and b, then the root reason becomes a mystery and only the users may be able to know which part is wrong. Luckily, these extra edges commonly exist in real-world parallel file systems, such as a file not only connects to its parent but also connects to its stripe objects; or a directory connects with both its parent directory and child files or directories. FaultyRank leverages them to conduct the intelligent and accurate checking and repairing.

IV. FaultyRank Prototype on Lustre

To validate the idea of FaultyRank in real-world large-scale settings, we implement a prototype of FaultyRank on Lustre. The prototype consists of three essential components as Fig. 6 shows: 1) a *scanner* running on all MDS and OSS servers to extract metadata from the local server into partial graphs; 2) an *aggregator* running on MDS server to receive and combine all partial graphs into a unified graph; 3) executing the FaultyRank algorithm on the unified graph to identify and repair the inconsistencies. Since the third component has been discussed, we focus on the first two in the following subsections.

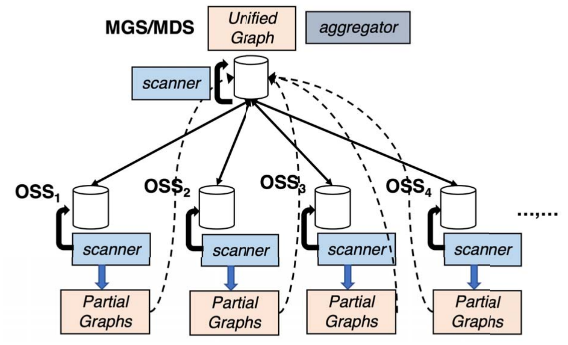


Fig. 6: The architecture of FaultyRank prototype on Lustre.

A. Extract Metadata into Graphs

Similar to many parallel file systems, Lustre relies on local file systems, such as ldiskfs or ZFS, to store its data and metadata. In this prototype, we focus on the ldiskfs case. Lustre metadata are stored in two places: 1) most of them are embedded as Extended Attributes (EA) of the local inodes, such as LinkEA or LOVEA; 2) the DIRENT metadata between the directory and its sub-directories or files are stored as the content of the directory. To extract Lustre metadata, we need to scan the extended attributes of inodes and the contents of directories in local ldiskfs completely.

We implemented a *scanner* which runs in parallel on all the Lustre MDS and OSS servers once called by users. Running the *scanner* is the first step of file system checking. To run it, users need to stop and unmount the Lustre file system, which allows the scanner to extract coherent metadata from disks. This means the current FaultyRank prototype is an offline checker. This is mostly for implementation simplicity and is not required by the FaultyRank algorithm. In the future, we plan to investigate how to implement FaultyRank online to further reduce file system offline time.

Once started, the *scanner* will scan the whole disk image from the superblock to each logic block group. Lustre's ldiskfs is essentially an extended version of Ext4, so we basically use the Ext4 disk layout to scan MDS and OSS servers. Most of the scanning is sequential and fast as it simply iterates all inodes and reads their Extended Attribute (EA) fields. The only exception is once it hits a directory, the *scanner* will move to the corresponding data blocks to read their DIRENT entries.

The output of the *scanner* is a partial graph that represents metadata stored on that server. The partial graph is a list of edges created during scanning. Each edge has a source vertex and destination vertex, each representing a Lustre directory, file, or stripe object. Since Lustre already assigns unique FIDs to these objects, we simply use those FIDs to uniquely identify these vertices in the partial graph. These global FIDs also help us match vertices generated from other storage servers.

B. Aggregate Partial Graphs into a Unified Graph

The next step is to aggregate the partial graphs collected from multiple concurrent *scanners* on MDS and OSS servers into a unified graph to execute the FaultyRank algorithm.

To generate the unified graph, we let the *scanners* on all OSS servers send their generated partial graphs to the MDS server *aggregator* once they finish the scanning. The MDS *aggregator* receives the partial graphs and simply aggregates them together with the local MDS partial graph. Since all the vertices have unique global FIDs, there will not be conflict during the aggregation. After the data transfer, a global graph is formed on the MDS server.

In addition to simply combining partial graphs on MDS, we do one more step to re-map the graph IDs before running the FaultyRank algorithm. In our current FaultyRank implementation, we used Compressed Sparse Row (CSR) data structure to store the graph in DRAM for extreme performance. For the best performance, we re-map the graph vertex IDs, which currently are 128-bit Lustre non-continuous FIDs, to vertex GIDs which continue from 0 to MAX_VERTEX_NUM-1. This processing happens in memory and takes minimal time, as we will show in later evaluation sections.

Note that the implementation details discussed in this section are based on Lustre. The FaultyRank algorithm and its idea is not limited to Lustre and can be implemented on other parallel file systems. The calculation phase will remain the same, but *scanner* and *graph building* components will need to be re-designed depending on the specific file systems.

V. Evaluations

In this section, we discuss the evaluations of FaultyRank on a realistic Lustre instance. We mainly examine two aspects of FaultyRank: 1) *functionality*: how well it can handle various inconsistency cases compared with the state-of-the-art Lustre checker LFSCK; 2) *performance*: how fast it runs and whether it will be a bottleneck in large-scale file systems.

A. Evaluation Testbed and Dataset

To conduct the evaluations, we built a local Lustre cluster with 1 MDS/MGS server and 8 OST servers as the testbed. The MDS/MGS server uses Intel(R) Xeon(R) Bronze 3204 CPU with 128GB DRAM and 256GB local SSD. The eight OSS servers use Intel(R) Xeon(R) CPU E5-2630 CPU with 32GB DRAM and 1TB hard disk (partially partitioned for Lustre). Based on the hardware, we installed Lustre version 2.12.8 (with the latest LFSCK implementation) and created a Lustre instance with a total of 2.4TB of storage space.

To create a realistic Lustre instance for evaluation, we leveraged the public data released by USRC (Ultrascale Systems Research Center) from LANL national lab [18]. Specifically, we used its *Archive and NFS Metadata* dataset. This dataset contains a file system walk of LANL's HPC systems with detailed information such as file sizes, creation time, modification time, UID/GID, anonymized file path, etc. The LANL dataset includes roughly 2PB of files.

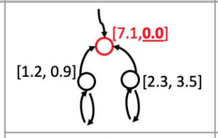
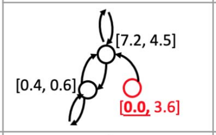
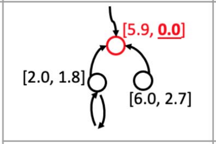
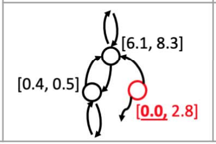
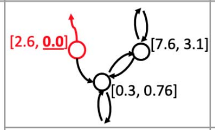
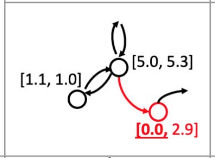
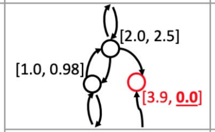
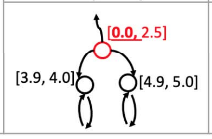
Type of Inconsistency	Example Plot	FaultyRank	LFSCK
a's property cannot locate *b* (**Dangling Reference)**		Property of the directory is Wrong , repaired using IDs of the parent/child files.	New children files created for the parent directory and put the original two subfiles in *lost+found*.
		ID of the directory is Wrong , repaired using parent directory's property.	
No object refers to *b* (**Unreferenced Object)**		Property of the parent directory is Wrong, repaired using child directories IDs.	The file will be added to *lost+found*.
		ID of the file is Wrong, repaired using parent directory's property.	
More than one object refers to *b* (**Double Reference)**		Property of the unreferenced parent is duplicate and should be repaired.	LFSCK do not identify or repair such case.
		ID of the non referencing child should be repaired.ID of the non referencing child should be repaired.	LFSCK will put one of the file object in *lost+found*
Mismatch between *a* and *b* (**Mismatch)**		Property of non referencing child file should be repaired using parent's ID.	Repair non-referencing child property using parent's ID.
		Property of the directory should be repaired using IDs of the children files.	Repair property of directory using ID of child files correctly.

Fig. 7: Comparison between FaultyRank and LFSCK on eight different types of inconsistency.

Although we have only 2.4TB storage space in our local testbed, we took several key measurements to generate metadata that can reflect the complete LANL trace. First, we used the actual file paths to re-create the same directory structures in our local testbed, which leads to the same Namespace metadata. Second, we tried to shrink the sizes of files in the 2PB file system without affecting the representativeness of generated Layout metadata. Specifically, we set the `stripe_size` of our Lustre directories to be extremely small (i.e., 64KB) and `stripe_count` to be -1. This allows us to generate extra stripes starting from files larger than 64KB. Since the testbed contains 8 OSTs and each stripe is 64KB, any file larger than 512KB (8*64KB) will create the same number of stripes regardless of its actual size. Hence, we can shrink files that are larger than 512KB to 512KB without affecting their layout metadata. All files smaller than 512KB will remain the same size and create stripes based on FILE_SIZE/64KB. It is arguable that real-world Lustre could have lots of OSTs, hence generate lots of stripes if having many large files, which our testbed can not represent. However, real-world Lustre's `stripe_size` is typically large (1MB by default) and most files in PFSes are actually very small (86% under 1MB, 95% under 2MB [19]). Together, these two factors limit the number of stripes in the real world. Our testbed, although smaller, leveraging the minimized `stripe_size`, will generate similar or even more stripes, leading to complex and representative Layout metadata for evaluations, since it is the size of metadata and not the data which determines the performance of file system checkers.

B. Functionality Evaluation on FaultyRank

We first evaluated how FaultyRank can check and repair different inconsistency scenarios compared with the state-of-art LFSCK. In this evaluation, we manually introduced eight inconsistent scenarios based on the failure cases discussed in LFSCK design documents [15], which represent what LFSCK handles in production systems. These inconsistencies match the four categories listed in Table I as well. To introduce the inconsistency for each case, we randomly selected one directory/file in the generated Lustre image and modified the Extended Attributes of corresponding ldiskfs inodes on MDS and OSS servers. We then ran both FaultyRank and LFSCK on these cases and recorded their behaviors. We compared whether they could identify the root reason for the inconsistencies and repair them. The results are summarized in Fig. 7. In the *Example Plot* column, we plot both how the faults were introduced and the $[id_{rank}, prop_{rank}]$ values calculated using FaultyRank for all of the vertices. Due to the space limits, we can not show the complete directory and file path for each vertex. But in most of the case, the vertices at the top indicate a directory, and the vertices at the bottom indicate a file or a stripe object. The red vertices indicate where the faults are actually injected. For example, in the first case of *Dangling Reference* inconsistency, we modified the properties of the parent directory, hence it has all other vertices pointing to it but it does not point to any other vertex. FaultyRank captures that by calculating its property rank value to be 0.0, which indicates the error is on its property, instead of assuming other neighbors' IDs are wrong. From these results, we can see across all the cases that FaultyRank is able to identify the root faults and fix them. By comparison, LFSCK is limited in many cases to identify the root cause or to repair the error.

C. Performance Evaluations on FaultyRank

To evaluate the performance of FaultyRank, we conducted two sets of experiments. First, we focused on the iterative algorithm itself and tested its performance on different graph datasets. This gives a basic idea of how long FaultyRank may take for extremely large file systems. Second, we conducted the full system evaluation on the local Lustre testbed. We ran both LFSCK and our FaultyRank prototype to compare their performance. Note that, we ran each experiment multiple times to ensure consistency and reliability and reported the averages.

1) Benchmark Iterative FaultyRank Algorithm: We implemented the iterative algorithm based on in-memory CSR structure. During execution, it first reads the edge-list file from local storage and builds the CSR format in DRAM. After that, the algorithm will run completely in DRAM. Using CSR leads to minimal memory usage and extremely high speed due to CSR's cache-friendly compact memory layout. Note that, the graph-building time is considered part of the FaultyRank execution time.

To benchmark the FaultyRank graph algorithm, we used well-known graphs with different sizes to evaluate the execution time and the memory footprints. The graphs we used include both real-world from SNAP [20] and synthetic graphs created using R-MAT library [21], listed in Table III. We generated the R-MAT graphs using probabilities $a = 0.57$, $b = 0.19$, $c = 0.19$ (recommended by Graph500 [22]) and set the average vertex degree to 8.

Datasets	Vertex Number	Edge Number
Amazon	403,393	4,886,816
Road-Net	1,971,281	5,533,214
RMAT-23	8,388,608	67,108,864
RMAT-24	16,777,216	134,217,728
RMAT-25	33,554,432	268,435,456
RMAT-26	67,108,864	536,870,912

TABLE III: Graph inputs and their key properties.

Datasets	Graph Building (s)	Iterations (s)	Memory Usage (GB)
Amazon	0.92	1.37	0.24
Road-Net	1.46	1.75	0.40
RMAT-23	31.99	21.64	3.31
RMAT-24	69.14	50.55	6.66
RMAT-25	148.80	116.92	13.3
RMAT-26	315.11	275.38	26.5

TABLE IV: FaultyRank performance and memory footprint.

The performance of FaultyRank algorithm on different graphs is listed in Table IV. From these results, we have two key observations. First, both the graph building and the iterative algorithm are fast. For a graph with more than 60 million vertices and 500 million edges (RMAT-26), we can finish the whole execution in roughly 590 seconds. If we consider each graph vertex represents 10MB of data, such a graph could represent a Lustre with more than 600TB of data. Finishing the checking within 10 minutes for such a scale is impressive. Second, we observe that running FaultyRank does not require extreme memory space. For example, the RMAT-26 graph with average degree of 8 takes 26.5 GB of memory, which can be stored and processed in a single MDS server.

In the previous evaluation, we scaled the number of vertices but fixed the average degree (i.e., 8) for all the RMAT graphs. It is also interesting to know how FaultyRank scales if the number of vertices is fixed but average degree changes. In this experiment, we further benchmarked FaultyRank on the same RMAT-26 graph but with varying average degrees (from 4 to 32). The results are listed in Table V. From these results, we can observe that FaultyRank still scales well in terms of execution time and memory usage. For instance, when the average degree reaches 32, the RMAT-26 graph will have more than 2 billion edges. For such a scale, FaultyRank can finish execution within 45 minutes and only use 90.4GB of memory.

Avg. Degree	Graph Building (s)	Iterations (s)	Memory Usage (GB)
4	165.05	180.46	15.9
8	315.11	275.38	26.5
16	727.06	623.37	48.7
32	1517.02	1168.86	90.4

TABLE V: FaultyRank performance and memory footprint on RMAT-26 graphs for varying average degree.

2) Full System Benchmark: We run LFSCK and FaultyRank and compare their performance. One of the main issues of Lustre LFSCK is its slow performance when running from scratch on a well-aged file system [7]. The slow performance mainly comes from its design, which consists of scalability bottleneck on the metadata server (MDS), relatively high fan-out ratio in network utilization, and unnecessary blocking among internal components. Specifically, LFSCK scans, checks, and repairs inodes individually via several closely coupled asynchronous kernel threads. It tangles disk scanning, network, and processing logic together. Hence, any delay in the pipeline may block others significantly.

Our graph-based parallel file system checker design addresses these issues as a nice side effect. First, the parallel *scanners* transfer the entire partial graphs in bulk only once after building the partial graphs. Such a bulk data transfer significantly reduces the network communication workloads. Second, there are no multiple dependencies among the internal components of FaultyRank. It simply conducts an iterative graph calculation, which could be done completely in DRAM in a single machine without complicated I/O operations. To show the performance advantages of the graph-based checkers, we further benchmarked the performance of FaultyRank and LFSCK on our testbed in this experiment.

It is worth noting that FaultyRank is currently implemented as an offline checker while LFSCK is an online checker. To conduct a fair comparison, each time, we re-mounted the Lustre file system and ran LFSCK to simulate a complete LFSCK run over the whole file system. We did not run other workloads nor limited the speed of LFSCK. For FaultyRank, we counted the end-to-end time, starting from un-mounting the file system until the FaultyRank algorithm converges. The reported execution time contains three parts: 1) metadata scanning time (T_{scan}); 2) graph transfer and processing time (T_{graph}); 3) FaultyRank algorithm execution time (T_{FR}).

Table VI shows the execution time of FaultyRank (including the detailed performance of each stage) and LFSCK towards

Lustre file system that was increasingly aged (with more inodes being used). Since the total number of available inodes in our Lustre testbed is around 4 million, we stopped the experiments at aging the file system with 4 million files. Also, since both LFSCK and FaultyRank work on metadata only, the taken storage space does not play a key role in performance, we do not list the actual data usage in the testbed. From these results, we can quickly notice that FaultyRank performs an order of magnitude faster than LFSCK in all cases. For instance, when there are around 2 million MDS inodes used, running a fresh LFSCK takes around 800 seconds. While for the same file system, the overall time taken by FaultyRank is only around 130 seconds. We can see the trend continues when more inodes are used in the file system.

MDS Inodes	LFSCK	FaultyRank	T_{scan}	T_{graph}	T_{FR}
651,553	207	12.40	3.8	3.2	5.40
1,099,717	364	37.69	16.74	11.74	9.21
1,555,351	525	57.62	25.09	19.39	13.14
2,007,043	667	83.28	43.78	22.46	17.04
2,231,988	803	130.47	79.22	32.22	19.03
3,335,597	1212	213.73	134.01	51.04	28.68
4,235,925	1612	292.83	185.92	70.79	36.12

TABLE VI: The execution time (in seconds) of FaultyRank and LFSCK on local Lustre testbed.

VI. Discussions on Generality and Limitations

So far, we have shown the efficiency and effectiveness of FaultyRank. In this section, we will further discuss its generality and limitations.

Generality. In this study, we implemented FaultyRank on the Lustre file system and mainly compared it with Lustre's LFSCK. But the core idea of FaultyRank, such as the graph-based metadata abstraction and the iterative credibility calculation, is generic to be applied to other parallel file systems. To implement it on a different PFS, the iterative calculation should remain the same once the metadata graph is built. But, the scanning and graph-building phases will be different and depend on the file system implementation. For instance, in the case of file systems like BeeGFS [8] that also use extended attributes (EAs) of the underlying local file systems to store metadata, the scanning and graph-building phases will be very similar. For parallel file systems which store metadata in a database (e.g., PVFS [3]), the scanning and graph-building may be implemented directly upon the existing database.

Limitations. Although FaultyRank performs well in both functionality and speed, it still has limitations. First, its current implementation is offline, which will require stopping and unmounting of the file system to work. This limitation is not fundamental and can be addressable by making *scanner* and *graph building* incremental. In this way, we can always run the FaultyRank algorithm on the latest snapshot of the metadata graph [23, 24]. We plan to investigate the online FaultyRank in the future. Second, some real-world inconsistent issues could be too complicated and beyond the capability of FaultyRank. For example, if multiple paired metadata are all wrong but pointing to each other coherently, FaultyRank cannot detect it. Existing tools would not work either. Users' inputs are likely necessary in such cases.

VII. Related Work

FaultyRank is mainly related to the existing efforts on file system checkers and parallel file systems. We elaborate on the two categories of related work in this section.

Improving File System Checkers. Given the importance of maintaining file system consistency, great efforts have been made to optimize file system checkers [25, 26, 27, 28, 29, 30, 31, 32]. For example, Carreira et al. [25] propose a tool called SWIFT to test file system checkers using a mix of symbolic and concrete execution to detect bugs in five popular checkers. Gunawi et al. [26] also find that file system checkers may create inconsistent or even insecure repairs and propose a more elegant design based on a declarative query language (i.e., SQCK). Gatla et al. [29] study the fault resilience of file system checkers and propose a transitional library (RFSCK) to enhance them. While these works are effective for their original goals, they only focus on local file system checkers. Mahmud et al. [32] analyze the configuration dependencies between file systems and the checkers. In terms of PFS checkers, Han et al. [33] study the defects in Lustre's LFSCK through fault injections, which partially motivates the design of FaultyRank. But they do not provide a solution for building more effective or efficient PFS checkers. Therefore, FaultyRank is complementary to the existing works.

Tool Support for Parallel File Systems. Besides file system checkers, many other tools have been proposed to improve parallel file systems, including instrumentation, profiling or tracing I/O activities, fault injections, and so on [34, 35, 36, 37, 38, 10]. For example, Sun et al. [38] propose to study the crash consistency of PFSes via replaying workload traces. Cao et al. [10] performs fault injections to PFSes and analyzes the failure handling mechanisms including the behaviors of PFS checkers. In addition, many of the existing tools originally designed for analyzing the performance of HPC systems may also help improve PFS reliability. For example, Darshan [34] is able to capture the I/O characteristics of various HPC applications. Since all I/O requests are served by the backend PFS, these captured I/O metrics could be used to help diagnose the root causes of reliability issues in PFSes. Overall, these existing efforts aim at improving PFSes from different perspectives and they do not directly enhance PFS checkers. Therefore, they are complementary to FaultyRank.

VIII. Conclusion and Future Work

In this study, we present FaultyRank, a new graph-based parallel file system checker. Different from the existing rule-based parallel file system checker design, FaultyRank leverages the graph model to represent PFS metadata and proposes a new iterative algorithm to quantitatively calculate the credibility of each metadata field for checking and repairing. We implemented a prototype of FaultyRank on Lustre and showed its advantages in both functionality and speed compared with

Lustre's file system checker, LFSCK. In the future, we plan to extend FaultyRank algorithm by investigating how separating multiple properties of an object will impact its design and correctness. Also, we plan to extend FaultyRank implementation from two aspects: 1) implement online FaultyRank on Lustre; 2) extend FaultyRank to other widely used parallel file systems and benchmark the performance.

Acknowledgments

We sincerely thank the anonymous reviewers for their valuable feedback. This work was supported in part by NSF grants OAC-1835892, CCF-1910727, CCF-1908843, CCF-1910747, and CNS-1943204.

References

[1] "Lustre File System," http://opensfs.org/lustre/.

[2] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*, 2002.

[3] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A Parallel File System for Linux Clusters," in *4th Annual Linux Showcase & Conference (ALS'00)*, 2000.

[4] "HPCC Power Outage Event at Texas Tech," http://www.ece.iastate.edu/~mai/docs/failures/2016-hpcc-lustre.pdf, 2016.

[5] "e2fsck(8)," https://linux.die.net/man/8/e2fsck, 2022.

[6] "xfs_repair(8) — Linux manual page," https://man7.org/linux/man-pages/man8/xfs_repair.8.html, Accessed: 01/2023.

[7] D. Dai, O. R. Gatla, and M. Zheng, "A Performance Study of Lustre File System Checker: Bottlenecks and Potentials," in *2019 35th Symposium on Mass Storage Systems and Technologies (MSST'19)*, 2019.

[8] J. Heichler, "An introduction to BeeGFS," 2014.

[9] "mmfsck command," Accessed: 01/2023. [Online]. Available: https://www.ibm.com/docs/en/spectrum-scale/5.0.5?topic=reference-mmfsck-command

[10] J. Cao, S. Wang, D. Dai, M. Zheng, and Y. Chen, "PFault: A General Framework for Analyzing the Reliability of High-Performance Parallel File Systems," in *Proceedings of the 32nd ACM International Conference on Supercomputing (ICS'18).*

[11] R. Han, O. R. Gatla, M. Zheng, J. Cao, D. Zhang, D. Dai, Y. Chen, and J. Cook, "A study of failure recovery and logging of high-performance parallel file systems," *ACM Transactions on Storage (TOS'21)*, 2021.

[12] D. Dai, Y. Chen, P. Carns, J. Jenkins, W. Zhang, and R. Ross, "Graphmeta: a graph-based engine for managing large-scale hpc rich metadata," in *2016 IEEE International Conference on Cluster Computing (CLUSTER'16).* IEEE, 2016, pp. 298–307.

[13] A. Blagodarenko, "Scaling LDISKFS for the future," 2016.

[14] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end Data Integrity for File Systems: A ZFS Case Study." in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'10)*, 2010.

[15] "Lustre LFSCK," https://wiki.lustre.org/Category:LFSCK, Accessed: 01/2023.

[16] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[17] M. Bianchini, M. Gori, and F. Scarselli, "Inside pagerank," *ACM Transactions on Internet Technology (TOIT'05)*, vol. 5, 2005.

[18] Los Alamos National Laboratory, "Ultrascale Systems Research Center (USRC) Data Sources," https://usrc.lanl.gov/data-sources.php.

[19] P. Carns, K. Harms, W. Allcock, C. Bacon, S. Lang, R. Latham, and R. Ross, "Understanding and Improving Computational Science Storage Access through Continuous Characterization," *ACM Transactions on Storage (TOS'11)*, vol. 7, 2011.

[20] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford Large Network Dataset Collection," http://snap.stanford.edu/data, 2014.

[21] F. Khorasani, "Multi-threaded Large-Scale RMAT Graph Generator." https://github.com/farkhor/PaRMAT, 2015.

[22] G. . S. Committee, "Graph 500 benchmark specification." https://graph500.org/?page_id=12, 2017.

[23] A. A. R. Islam, D. Dai, and D. Cheng, "VCSR: Mutable CSR Graph Format Using Vertex-Centric Packed Memory Array," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid'22)*, 2022.

[24] A. A. R. Islam, C. York, and D. Dai, "A performance study of optane persistent memory: from storage data structures' perspective," *CCF Transactions on High Performance Computing (THPC'22)*, vol. 4, 2022.

[25] J. C. M. Carreira, R. Rodrigues, G. Candea, and R. Majumdar, "Scalable Testing of File System Checkers," in *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, 2012.

[26] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-dusseau, and R. H. Arpaci-dusseau, "SQCK: A Declarative File System Checker," in *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)*, 2008.

[27] A. Ma, C. Dragga, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. K. Mckusick, "Ffsck: The fast file-system checker," *ACM Transactions on Storage (TOS'14)*, vol. 10.

[28] O. R. Gatla and M. Zheng, "Understanding the fault resilience of file system checkers," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'17)*, 2017.

[29] 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 16th USENIX Conference on File and Storage Technologies (FAST'18)*, 2018.

[30] D. Domingo and S. Kannan, "pFSCK:Accelerating File System Checking and Repair for Modern Storage," in *19th USENIX Conference on File and Storage Technologies (FAST'21)*, 2021.

[31] T. Mahmud, D. Zhang, O. R. Gatla, and M. Zheng, "Understanding configuration dependencies of file systems," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*, 2022.

[32] T. Mahmud, O. R. Gatla, D. Zhang, C. Love, R. Bumann, and M. Zheng, "ConfD: Analyzing Configuration Dependencies of File Systems for Fun and Profit." in *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'23)*, 2023.

[33] R. Han, D. Zhang, and M. Zheng, "Fingerprinting the checker policies of parallel file systems," in *2020 IEEE/ACM Fifth International Parallel Data Systems Workshop (PDSW'20)*, 2020.

[34] "Darshan:HPC I/O Characterization Tool," 2017. [Online]. Available: http://www.mcs.anl.gov/research/projects/darshan/

[35] J. S. Vetter and M. O. McCracken, "Statistical Scalability Analysis of Communication Operations in Distributed Applications," *ACM SIGPLAN Notices*, vol. 36, 2001.

[36] P. C. Roth, "Characterizing the I/O behavior of scientific applications on the Cray XT," in *Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing (PDSW'07)*, 2007.

[37] M. P. Mesnier, "TRACE: Parallel Trace Replay with Approximate Causal Events," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[38] J. Sun, J. Huang, and M. Snir, "Pinpointing crash-consistency bugs in the HPC I/O stack: a cross-layer approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)*, 2021.