Efficient Metadata Indexing for HPC Storage Systems

Arnab K. Paul*, Brian Wang†, Nathan Rutman†, Cory Spitz†, Ali R. Butt*
*Virginia Tech, †Cray Inc.
{akpaul, butta}@vt.edu, {bwang, nrutman, spitzcor}@cray.com

***Abstract*—The increase in data generation rate along with the scale of today's high performance computing (HPC) storage systems make finding and managing files extremely difficult. Efficient file system metadata indexing and querying tools are needed to ease file system management. Current metadata indexing techniques either use spatial trees or an external database to index metadata. Both approaches have their drawbacks which reduce the performance of indexing and querying the metadata on large scale file systems.**

In this paper, we have developed BRINDEXER, a metadata indexing and search tool specifically designed for large-scale HPC storage systems. BRINDEXER is mainly designed for system administrators to help them manage the file system effectively. It uses a leveled partitioning approach to partition the file system namespace, and has an in-tree design to reduce resource utilization from an external database. BRINDEXER uses RDBMS for efficient querying of the metadata index database, also uses a changelog-based approach to effectively handle real-time metadata changes and re-index the metadata at regular intervals. We implement and evaluate BRINDEXER on a 4.8 TB Lustre store and show that it improves the indexing and querying performance by 69% and 91% when compared to state-of-the-art metadata indexing tools.

***Keywords*-Hierarchical Partitioning, High Performance Computing, Metadata Changelog, Leveled Partitioning, Lustre File System**

I. INTRODUCTION

From life sciences and financial services to manufacturing and telecommunications, organizations are finding that they need not just more storage, but high-performance storage to meet the demands of their data-intensive workloads. This has resulted in a massive amount of data generation (order of petabytes), creation of billions of files, and thousands of users acting on HPC storage systems. According to a recent report from National Energy Research Scientific Computing Center (NERSC) [5], over the past 10 years, the total volume of data stored at NERSC has grown at an annual rate of 30 percent. This ever-increasing rate of data generation combined with the scale of HPC storage systems make efficiently organizing, finding, and managing files extremely difficult.

HPC users and system administrators need to query the properties of stored files to efficiently manage the storage system. This data management issue can be addressed by an efficient search of the file metadata in a storage system [16]. Metadata search is particularly helpful because it not only helps users locate files but also provides database-like analytic queries over important attributes. Metadata search involves indexing file metadata such as inode fields (for example, size, owner, and timestamps) and extended attributes (for example, document title, retention policy, and provenance), represented as <*attribute, value*> pairs [15]. Therefore, metadata search can help answer questions like *"Which application's files consume the most space in the file system?"* or *"Which files can be moved to second tier storage?"*.

Metadata indexing on large scale HPC storage systems presents a number of challenges. *First, scaling metadata indexing technology from local file systems to HPC storage systems is very difficult.* In local file systems, the metadata index has to index only a million files, and thus can be kept in-memory. However, in HPC systems, the index is too large to reside in-memory. *Second, the metadata indexing tool should be able to gather the metadata quickly.* The typical speed for file system crawlers is in the range of 600 to 1,500 files/sec [8]. This translates to 18 to 36 hours of crawling for a 100 million file data set. A large scale HPC storage system can often contain a billion files, which implies crawl time in the order of weeks [8]. *Third, the resource requirements should be low.* Existing HPC storage system metadata indexing tools such as LazyBase [9] and Grand Unified File-Index (GUFI) [3] require dedicated CPU, memory, and disk hardware, making them expensive and difficult to integrate into the storage system. *Fourth, metadata changes must be quickly re-indexed to prevent a search from returning inaccurate results.* It is difficult to keep the metadata index consistent because collecting metadata changes is often slow [26] and therefore, search applications are often inefficient to update.

Current state-of-the-art metadata indexing techniques on HPC storage systems include Spyglass [16], SmartStore [12], Security Aware Partitioning [19], and GIGA+ [20]. All of these techniques use a spatial tree, such as k-d tree [30], or R-tree [11] to index metadata. However, both these trees have poor performance in handling high dimensional data sets [7], they handle missing values inefficiently, and do not perform well for data which have multiple values for one field [27]. These drawbacks reduce their ability to index metadata efficiently. Other metadata indexing techniques, like, GUFI [3], Robinhood Policy Engine [14], and BorgFS [1], use a popular approach for metadata indexing where an external database is maintained for indexing outside the HPC storage system. This approach involves a major issue of maintaining consistency because the metadata is managed outside the file system which is being indexed.

To address these issues in HPC storage system metadata indexing, we present an efficient and scalable metadata indexing and search system, BRINDEXER. BRINDEXER enables a fast and scalable indexing technique by using a *leveled partitioning approach* to the file system. Leveled partitioning is different and more effective than the *hierarchical partitioning approach* used in state-of-the-art indexing techniques discussed above. BRINDEXER uses an in-tree indexing design and thus mitigates the issue of maintaining metadata consistency outside the file system. BRINDEXER also uses RDBMS to store the index which makes querying easier and more effective. To overcome the drawback of slow re-indexing process, BRINDEXER uses a changelog-based approach to keep track of metadata changes in the file system.

We present BRINDEXER and the scalable metadata changelog monitor that helps track the metadata changes in HPC storage system. The HPC storage system that we choose for our implementation is Lustre. According to the latest Top 500 list [6], Lustre powers $\sim$ 60% of the top 100 supercomputers in the world. While the implementation and evaluation for BRINDEXER is shown in the paper as applied to Lustre storage system, its design makes it applicable to other HPC storage systems, such as IBM's Spectrum Scale, GlusterFS and BeeGFS. We compare indexing and querying performance of BRINDEXER with a hardware-normalized version of the state-of-the-art GUFI indexing tool and show that the indexing performance of BRINDEXER is better by 69%, querying performance is better by 91%. Resource utilization by BRINDEXER is lower than that of GUFI by 46% during indexing and 58% during querying 22 million files on a 4.8 TB Lustre store.

II. BACKGROUND & MOTIVATION

In this section, we describe the different partitioning approaches for indexing a file system, the metadata attributes motivated by some examples of file system search queries, the architecture of HPC storage system with an emphasis on Lustre file system, and finally we explain the different approaches to collecting metadata changes along with a motivation for BRINDEXER to use changelog-based approach.

A. Partitioning Techniques

To exploit metadata locality and improve scalability, HPC storage system's indexing tools partition the file system namespace into a collection of separate, smaller indexes. There are two main approaches to partitioning.

1) Hierarchical Partitioning: This is one of the most common approaches used in state-of-the-art metadata indexing tools. Hierarchical partitioning is based on the storage system's namespace and encapsulates separate parts of the namespace into separate partitions, thus allowing more flexible, finer grained control of the index. An example of hierarchical partitioning is shown in Figure 1a. As seen in the figure, the namespace is broken into partitions that represent disjoint subtrees. However, hierarchical partitioning faces an important challenge when the disjoint sub-trees are skewed, that is, some trees have more files than others.

2) Leveled Partitioning: This approach creates index nodes at a particular level in the storage system tree. An example of leveled partitioning is shown in Figure 1b. In the figure, leveled partitioning is done at level 2. Therefore, the file system namespace is divided into disjoint sub-trees from level 2, with index nodes at the root of each sub-tree. This mitigates the issue of hierarchical partitioning where some trees may be skewed which affects indexing performance. In the leveled approach, all directories up to the next index level are indexed at the root of the current level. Another major issue of hierarchical partitioning is that a file system crawler should be used before indexing to partition the file system namespace into uniformly-sized disjoint sub-trees. This requires extra resource consumption which can be overcome by leveled partitioning where no such crawler is needed before indexing. BRINDEXER uses the leveled partitioning approach to partition the file system namespace into smaller indexes.

B. Metadata Attributes

File metadata can be of two types.

- *Inode Fields:* They are generated by the storage system itself for every file, and are shown in Table I.
- *Extended Attributes:* These are typically generated by the users and applications. These may include *mime type* attribute, which defines the file extensions, and *permission* attribute specifying the read, write and execute permissions set by the application.

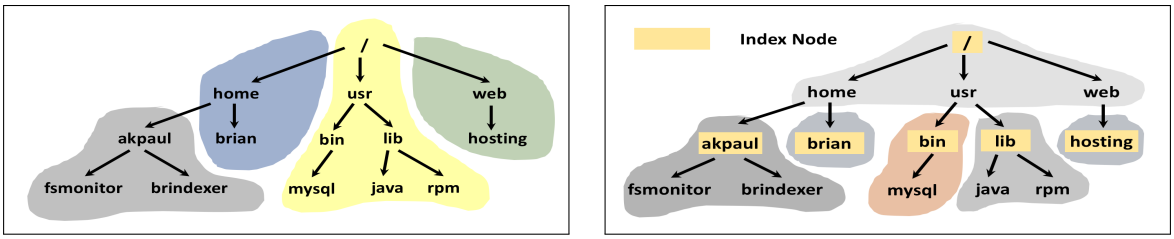
All attributes are typically represented in $<$*attribute, value*$>$ pairs that describe the properties of a file. For each POSIX file there will be at least 10 attributes, and for a large scale HPC storage system with a billion files, there will be a minimum of 10^{10} attribute pairs. The ability to search this massive dataset of metadata attributes pairs effectively gives rise to metadata indexing.

Attribute	Description		Attribute	Description
ino	inode number		*size*	file size
mode	access permissions		*blocks*	blocks allocated
nlink	number of hard links		*atime*	access time
uid	owner of file		*mtime*	modification time
gid	group owner of file		*ctime*	status change time

TABLE I: Metadata Attributes.

Some common metadata attributes used are shown in Table I. The *atime* attribute is affected when a file is handled by *execve*, *mknod*, *pipe*, *utime*, and *read* (of more than zero bytes) system calls. *mtime* is affected by the *truncate* and *write* calls. *ctime* is changed by writing or by setting inode information.

Some sample file management questions and the queries used to search the metadata attributes are shown in Table II. These show the importance of fast and scalable metadata indexing and querying that can help HPC storage system administrators.



(a) Hierarchical Partitioning **(b)** Leveled Partitioning

Fig. 1: Comparison between hierarchical partitioning and leveled partitioning approaches using the same file structure.

Storage System Administrator Question	Metadata Search Query
Which files should be migrated to secondary storage?	*size* >100 GB, *atime* >1 year
Which files have expired their legal compliances?	*mode* = file, *mtime* >10 years
How much storage do each user consume?	Sum *size* where *mode* = file, group by *uid*
Which files grew the most in the past one week?	Sort difference (*size* [today] - *size* [1 week before]) in descending order, group by *uid*

TABLE II: Some sample file management questions and the metadata search queries used.

C. HPC Storage System

HPC storage systems are designed to distribute file data across multiple servers so that multiple clients can access file system data in parallel. Typically, they consist of *clients* that read or write data to the file system, *data servers* where data is stored, *metadata servers* that manage the metadata and placement of data on the data servers, and networks to connect these components. Data may be distributed (divided into stripes) across multiple data servers to enable parallel reads and writes. This level of parallelism is transparent to the clients, for whom it seems as though they are accessing a local file system. Therefore, important functions of a distributed file system include avoiding potential conflicts among multiple clients and ensuring data integrity and system redundancy. The most common HPC file systems include Lustre, GlusterFS, BeeGFS, and IBM Spectrum Scale. In this paper, we have built BRINDEXER on the Lustre file system.

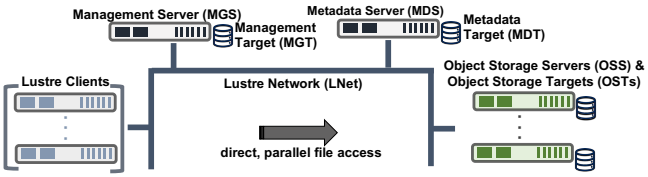


Fig. 2: An overview of Lustre architecture.

1) Lustre File System: The architecture of the Lustre file system is shown in Figure 2 [22], [21], [29]. Lustre has a client-server network architecture and is designed for high performance and scalability. The *Management Server (MGS)* is responsible for storing the configuration information for the entire Lustre file system. This persistent information is stored on the *Management Target (MGT)*. The *Metadata Server (MDS)* manages all the namespace operations for the file system. The namespace metadata, such as directories, file names, file layout, and access permissions are stored in a *Metadata Target (MDT)*. Every Lustre file system must have a minimum of one MDT. *Object Storage Servers (OSSs)* provide the storage for the file contents in a Lustre file system. Each file is stored on one or more *Object Storage Target (OST)*s mounted on the OSS. Applications access the file system data via *Lustre clients* which interact with OSSs directly for parallel file accesses. The internal high-speed data networking protocol for the Lustre file system is abstracted and is managed by the *Lustre Network (LNet)* layer.

D. Collecting Metadata Changes

After metadata indexing is done, regular re-indexing needs to be performed so that metadata search queries do not return out-of-date results. Re-indexing of the metadata can be performed by running the indexing tool at regular intervals to index the entire file system afresh. This is an approach that most state-of-the-art indexing techniques (GUFI [3], and BorgFS [1]) use which maintain the index in an external database outside the file system. However this is a very expensive approach for large filesystems as the size of the external database must scale with the size of the indexed filesystem. Another approach is to keep track of metadata changes and re-index based on the changes. There are two ways to collect metadata changes: *Snapshot-based approach* and *Changelog-based approach*.

- *Snapshot-Based Approach:* In this approach periodic snapshots are taken of the file system metadata. Snapshots are created by making a copy-on-write (CoW) clone of the inode file. Given two snapshots at time instant T_n and T_{n+1}, this approach will calculate the difference between these two snapshots and identify the files that have changed during the time interval between the two snapshots. The metadata index crawler can only crawl over the changed files to re-index them. This is much faster than periodic walks of the entire file system. However, this approach depends on a filesystem design incorporating CoW metadata updates.
- *Changelog-Based Approach:* This approach logs the metadata changes as the changes occur on the file system.

Event ID	Type	Timestamp	Datestamp	Flags	Target FID	Parent FID	Target Name
11332885	01CREAT	22:27:47.308560896	2019.11.28	0x0	t=[0x300005716:0x626c:0x0]	p=[0x300005716:0xe7:0x0]	hello.txt
11332886	17MTIME	22:27:47.327910351	2019.11.28	0x7	t=[0x300005716:0x626c:0x0]		hello.txt
11332887	08RENME	22:27:47.416587265	2019.11.28	0x1	t=[0x300005716:0x17a:0x0]	p=[0x300005716:0xe7:0x0] s=[0x300005716:0x626b:0x0] sp=[0x300005716:0x626c:0x0]	hello.txt hi.txt
11332888	02MKDIR	22:27:47.421587284	2019.11.28	0x0	t=[0x300005716:0x626d:0x0]	p=[0x300005716:0xe7:0x0]	okdir
11332889	06UNLNK	22:27:47.438587347	2019.11.28	0x0	t=[0x300005716:0x626b:0x0]	p=[0x300005716:0xe7:0x0]	hi.txt

TABLE III: A sample Changelog record showing *Create File*, *Modify*, *Rename*, *Create Directory*, and *Delete File* events.

This is done by recording the modifying events that occur on the file system. Every HPC storage system maintains an event changelog (used for auditing purposes) [23], example *mmaudit* in IBM Spectrum Scale, and *Lustre Changelog* in Lustre file system. Thus, building a scalable monitor that monitors the changelog could be a very efficient solution for collecting metadata changes. Only the files on which any modification event occurs need re-indexing. In BRINDEXER, we use a variant of the changelog-based approach to track modified directories, allowing us to reduce the tracking load by 90%.

Next, we explain the Lustre changelog which is used to keep track of file system events on Lustre file system.

1) Lustre Changelog: Table III shows sample records in Lustre's Changelog. We ran a simple script to see the events recorded in the Changelog. The script first creates a file, *hello.txt*, then the file is modified. The file is then renamed to *hi.txt*. A directory named *okdir* is then created. Finally, we delete the file.

Each tuple in Table III represents a file system event. Every row in the Changelog has an *EventID* – the record number of the Changelog; *Type* – the type of file system event that occurred; *Timestamp, Datestamp* – the date time of the event occurrence; *Flags* – masking for the event; *Target FID* – file identifier of the target file/directory on which the event occurred; *Parent FID* – file identifier of the parent directory of the target file/directory; and the *Target Name* – the file/directory name which triggered the event. It is evident that the Parent and Target FIDs need to be resolved to their original names before they can be processed by BRINDEXER. The following events are recorded in the Changelog:

- **CREAT**: Creation of a regular file.
- **MKDIR**: Creation of a directory.
- **HLINK**: Hard link.
- **SLINK**: Soft link.
- **MKNOD**: Creation of a device file.
- **MTIME**: Modification of a regular file.
- **UNLNK**: Deletion of a regular file.
- **RMDIR**: Deletion of a directory.
- **RENME**: Rename a file or directory.
- **IOCTL**: Input-output control on a file or directory.
- **TRUNC**: Truncate a regular file.
- **SATTR**: Attribute change.
- **XATTR**: Extended attribute change.

Note in Table III that Target FIDs are enclosed within $t = []$, and parent FIDs within $p = []$. *MTIME* event does not have a parent FID. *RENME* event has additional FIDs, $s = []$ denoting a new file identifier to which the file has been renamed, and $sp = []$ gives the file identifier for the original file. These features are important when resolving FIDs.

2) Motivation for using Lustre Changelog: We analyze a 24-hour Lustre Changelog obtained from a production system's petascale Lustre file system in Los Alamos National Laboratory (LANL).

Some observations from the analysis are:

- There are more than 34 million file system events which occur per day in a large-scale production-level HPC storage system.
- The number of unique files that get affected in 24 hours is $\sim$ 10.5 million.
- The number of unique directories on which metadata events occur is $\sim$ 110,000.
- The number of events for each individual event is shown in Table IV.

Event Type	# Events		Event Type	# Events
CREAT	1,322,010		MKDIR	67,791
HLINK	8,841		SLINK	94,711
MTIME	10,098,485		UNLNK	750,480
RMDIR	59,841		RENME	97,227
SATTR	3,432,589		XATTR	164

TABLE IV: Number of file system events for each metadata event in a 24-hour Lustre Changelog.

The analysis shows that performing a snapshot-based approach for keeping track of metadata changes in the file system may be very expensive for large, active filesystems. Also, to determine the directories for the affected 10.5 million files is time-consuming. The Lustre changelog, however, already reports the parent directories, which are also the directories which need to be re-indexed. This will improve the performance of BRINDEXER immensely because it does not need to keep track of all the 10.5 million files for re-indexing, but only the 110,000 unique directories. The challenge is to design an efficient and scalable changelog processing engine to get the parent FIDs (directories) of more than 10 million *MTIME* and more than 3 million *SATTR* files which are not already recorded in the changelogs. This is discussed in Section III-B.

III. SYSTEM DESIGN

The overall architecture of BRINDEXER is shown in Figure 3. BRINDEXER runs on the file system clients. It consists of the indexer, crawler and the metadata query interface. The

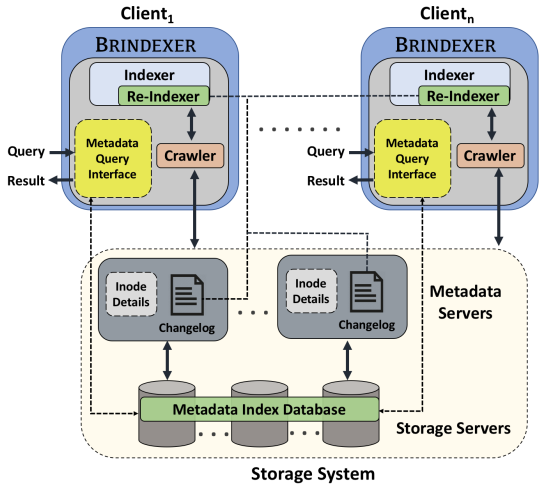
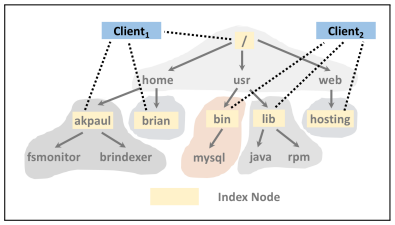


Fig. 3: Overall architecture of BRINDEXER.

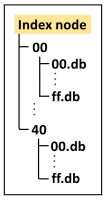
indexer and crawler are responsible for crawling the entire file system, collecting the inode details from the metadata servers and indexing the file system metadata. The re-indexer is part of the indexer in BRINDEXER and interacts with the file system changelog to keep track of the metadata changes. Users and applications interact with the the metadata query interface provided by BRINDEXER to query the metadata index database on the storage servers. Next, we describe each component of BRINDEXER in more details.

A. Indexer

The overview of the indexing process of BRINDEXER is shown in Algorithm 1. BRINDEXER uses a leveled partitioning technique to partition the file system namespace. This is described earlier in Section II. BRINDEXER performs the leveled partitioning approach in parallel, where the indexing task can be distributed on multiple client indexers for fast and scalable indexing. Each client node can be assigned a set of sub-trees and independently manages the file system indices under those sub-trees. This is shown in Figure 4a and this paralleled approach improves the performance of BRINDEXER. *Crawler* is responsible for doing the directory walk of the file system namespace.



(a) Parallelism in leveled partitioning of BRINDEXER.



(b) 2-level database sharding in BRINDEXER.

Fig. 4: Optimization strategies used in BRINDEXER.

The input to *Indexer* is the indexing level where all the directories at that level need to be indexed. The root directory

```
1  Function Indexing
       Input: Indexing Level: indexLevel
       Output: Metadata Index Database: indexdb
2      for Directory dir in directoryWalk do
3          if dir in Level indexLevel then
4              Setup database in Index Directory
5              processIndexDir(dir)
6          end
7      end
8      processRootDir(level)
9      return (indexdb)
10 Function processIndexDir
       Input: Index Directory: dir
11     for Directory subdir in recursive read of ll_readdir(dir) do
12         hash = Calculate hash of subdir
13         for File file in stat(subdir) do
14             new_lstat(file)
15             Place inode information of file in the database shard
                 with the hash value as hash
16         end
17     end
18 Function processRootDir
       Input: Indexing Level: indexLevel
19     Setup database in Root Directory
20     for Directory dir in directoryWalk do
21         if dir < Level indexLevel then
22             for Directory subdir in recursive read of
                 ll_readdir(dir) do
23                 hash = Calculate hash of subdir
24                 for File file in stat(subdir) do
25                     new_lstat(file)
26                     Place inode information of file in the
                         database shard with the hash value as hash
27                 end
28             end
29         end
30     end
```

Algorithm 1: Indexing function in BRINDEXER.

is responsible to index all directories above the indexing level. For each indexed directory, a recursive *readdir()* is performed to find all sub-directories. For every sub-directory, a *stat()* call is made to get the files in that directory, and to get the inode information for every file, new_lstat call is performed on the file.

Each individual index directory is set to a 2-level database sharding approach to keep the database shards to a reasonable size. This is done to maximize the database performance by querying an optimum number of files per database. This is shown in Figure 4b. The number of databases per index node is limited to 64 (0x40). This number is based on experiments to measure the time to index and query BRINDEXER for 1 billion files. 64 gives the optimum performance by having the optimal resource utilization. Within each database in the index node, there are database shards. Each database shard holds metadata information of one or more sub-directories of the index directory. To find the placement of the metadata information for a file, first MD5 hashing is done on the parent directory of the file to get the database shard. Next, MD5 hash is done on the index directory to find the database within which the database shard is placed. This 2-level sharding is done by BRINDEXER to maximize the query performance.

B. Re-Indexer

The architecture of *re-indexer* is shown in Figure 5.

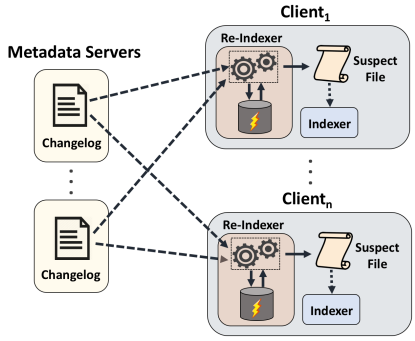


Fig. 5: Design of Re-Indexer in BRINDEXER.

BRINDEXER's *re-indexer* is a multi-threaded set of processes running on filesystem clients. One thread is responsible for processing the file system changelogs gathered from the metadata servers, which are processed in parallel on the clients. A fast and efficient caching mechanism is used to store the mappings of FIDs to paths to improve performance of processing of the changelogs. Another thread maintains a suspect file which has a collection of all suspect directories (directories which have been modified and need to be re-indexed) for a particular time period. This suspect file is given as an input to the indexer which then does a *stat()* for only the suspect directories.

1) Processing Changelogs: The *re-indexer* collects events from changelog in batches. Every event that is collected needs to be processed to collect the directory name in order to be placed in the *suspect file*. In particular, FIDs are not necessarily interpretable by BRINDEXER, and thus must be processed and resolved to absolute path names [24]. In Lustre file system, to process the FIDs, Lustre *fid2path* tool is provided which resolves FIDs to absolute path names. However, the *fid2path* tool is slow and can delay the reporting of events. For example, in Section IV-F1 we show that this delay can cause a decrease of 31.7% in the event reporting rate compared to the events generated in the file system. To minimize this overhead, *re-indexer* implements a Least Recently Used (LRU) Cache to store mappings of FIDs to source paths.

Algorithm 2 shows the processing steps for *re-indexer*. Changelog events are processed in batches. A LRU cache is used to resolve parent FIDs (directories) to absolute paths. Whenever an entry is not found in the cache, we invoke the *fid2path* tool to resolve the FID and then store the mapping *(FID – path)* into the LRU cache. `MTIME` and `SATTR` events do not have a parent FID and thus they are processed in the catch block, where the target FIDs are processed. The file name from the absolute path is removed to get the directory name and then the path is added to the cache, so that *fid2path* tool is not called on the file again. It should be noted that the cache only needs to track modified parent directories, so only 110,000 entries are present in a 24-hour suspect file rather than 10.5 million files. All of the resolved directory paths are added to the *suspect file* (not adding duplicates). After processing a batch of file system events from the Changelog, *re-indexer* will purge the Changelogs. A pointer is maintained

```
   Input: Lustre path lpath, Cache cache, MDT ID mdt
   Output: SuspectFile
1  while true do
2      events = read events from mdt Changelog
3      for event e in events do
4          resolvedPath = processEvent(e)
5          SuspectFile.add(resolvedPath)
6      end
7      Clear Changelog in mdt
8      return (SuspectFile)
9  end
10 Function processEvent
       Input: Event e
       Output: resolvedPath
11     Extract event_type, time, date from e
12     try:
13         path = cache.get(parentFID)
14         if parentFID not found in cache then
15             path = fid2path(parentFID)
16             cache.set(parentFID, path)
17         end
18     catch fid2pathError:
19         path = cache.get(targetFID)
20         if targetFID not found in cache then
21             path = fid2path(targetFID)
22             Remove file name from path
23             cache.set(targetFID, path)
24         end
25     end
26     return (path)
```

Algorithm 2: Processing Changelog events in Lustre file system.

to the most recently processed event tuple and all previous events are cleared from the Changelog. This helps reduce the overburdening of the Changelog with stale events.

Indexer periodically reads the suspect file and re-indexes the file system based on the suspect directories. Once *indexer* acts on a suspect file, a timestamp is given to the *re-indexer* and a new suspect file is written to add suspect directories from that time stamp.

C. Metadata Query Interface

Metadata query interface in BRINDEXER interacts with the *metadata index database* which is stored on the storage servers in the file system. The *metadata index database* uses RDBMS to store the index information of large-scale HPC storage systems. There are few reasons for selecting RDBMS for our implementation. First, we are not concerned with scalability of a single database because our design of *indexer* and *re-indexer* limits the database size. We use parallel leveled partition approach for speed and 2-level database sharding in the index level directory for scalability and optimal query performance. RDBMS therefore serves its purpose of providing a nice API for the users to query the database. Second, RDBMS is very efficient in handling bulk writes and appends which is needed during the re-indexing process. Also, doing bulk reads on RDBMS is efficient. Third, the limitation of RDBMS is when it has to handle continuous stream of inputs. In BRINDEXER, the *metadata index database* only has periodic input stream and RDBMS works efficiently in this case. Fourth, RDBMS also lowers performance when it has to handle contended writes as it has to deal with multiple locking issues. The 2-level

sharding and the namespace partition to handle disjoint subtrees in BRINDEXER does not involve *metadata index database* to handle contended writes.

IV. EVALUATION

We evaluate the performance of BRINDEXER by analyzing each component in detail. In this section, we describe the experimental setup for the evaluation, workloads that were used for analyzing the performance, and evaluate *indexer*, *re-indexer*, and *metadata query interface*.

A. Experimental Setup

To evaluate BRINDEXER, we use a Lustre file system cluster of 9 nodes with 4 MDSs, 3 OSSs and 2 clients. All nodes run CentOS 7 atop a machine with an AMD 64-core 2.7 GHz processor, 128 GB of RAM, and a 2.5 TB SSD. All nodes are interconnected with 10 Gbps bandwidth ethernet. Each MDS has a 128GB MDT associated with it. Furthermore, each OSS has 3 OSTs, with each OSS supporting 1.6 TB attached storage on OSTs. Therefore, our analysis is done on a 4.8 TB Lustre store.

B. Workloads

We use 2 kinds of workloads to test the performance of BRINDEXER as shown in Tables V and VI.

#Files	#Directories	Avg #Files per Dir	Total Size (MB)
400,000 (400k)	1,000	400	1.08
1,200,000 (1.2M)	1,000	1200	43.05
5,700,000 (5.7M)	5,700	1000	90.07
8,500,000 (8.5M)	8,500	1000	140.78
22,000,000 (22M)	222,000	1000	373.9

TABLE V: Workload 1: Flat directory structure: Smaller number of directories with higher average number of files per directory.

#Files	#Directories	Avg #Files per Dir	Total Size (GB)
400,254 (400k)	43,189	9.26	4.4
1,200,762 (1.2M)	129,565	9.27	13.3
5,736,974 (5.7M)	619,029	9.27	63.4
8,530,405 (8.5M)	815,753	10.46	95.1
22,013,970 (22M)	2,375,341	9.27	243.2

TABLE VI: Workload 2: Hierarchical directory structure: Large number of directories with lower average number of files per directory.

Both workloads have 5 different numbers of files (400k, 1.2M, 5.7M, 8.5M, and 22M). However, the workloads differ in the number of directories. Workload 1 has a flat directory structure with just one level consisting of lower number of directories with higher number of files per directory. Workload 2 has a hierarchical structure (with maximum directory depth of 17) consisting of higher number of directories with lower number of file per directory. Therefore, workload 1 represents ideal case while workload 2 takes care of the real-world use case.

We compare BRINDEXER with state-of-the-art indexing tool GUFI [3] for indexing. For workloads 1 and 2, BRINDEXER sets an indexing level of 1 and 3 respectively. This is because, workload 1 has only one level, and for workload 2, it turns out that number of directories at level 3 equals the average number of directories per level.

To evaluate querying performance of BRINDEXER, besides GUFI, we also compare BRINDEXER with Lustre's default *lfs find* tool [4] and *Robinhood policy engine* [14].

For evaluation of *re-indexer*, we evaluate the performance of Lustre changelog processing and compare it with *Robinhood* [14], as these are the only tools which use changelog-based approach to keep track of metadata changes.

All experiments are run five times and the evaluation shows the average of these runs. All caches are cleared between runs.

Next we give a brief description of GUFI and Robinhood.

GUFI stands for Grand Unified File Index. It uses breadth first traversal to traverse the entire file system tree in a parallel manner. GUFI uses one index database per directory to have the same security permission as that of the directory. This entire database tree is done outside the file system. Although GUFI is meant for indexing into an external database, we modify GUFI to run in-tree and perform metadata indexing inside Lustre file system itself. This allows us to compare the two techniques - GUFI and BRINDEXER, without any difference in hardware. BRINDEXER is not concerned with directory permissions because it is meant for system administrators.

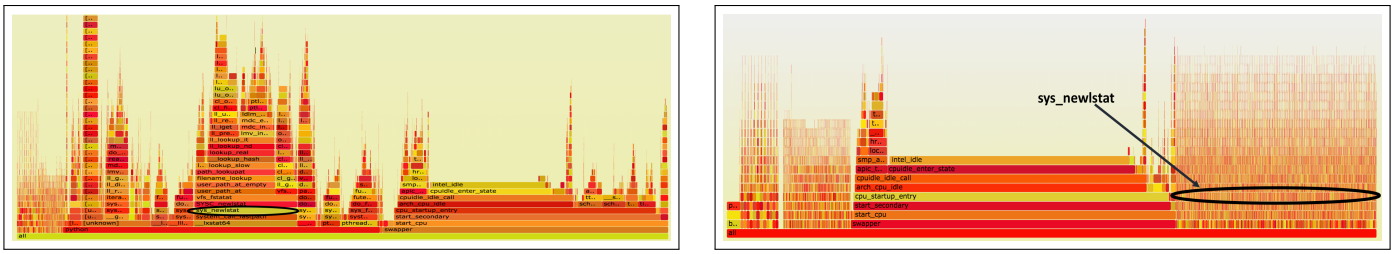
Robinhood collects information from the filesystem it monitors and inserts this information into an external database. It makes use of the Lustre changelog to monitor and keep track of file system events. For multiple MDSs, Robinhood uses a round-robin approach to keep track of changelogs.

C. Comparison of System Calls

Both BRINDEXER and Gufi were used to index 1.2 million files in hierarchical directory structure and the system calls were traced in a CPU flame graph [2]. This is shown in Figure 6. In a flame graph, each box represents a function in the stack. On the y-axis, the depth of the stack is shown and x-axis spans the sample population. The width of each box shows the amount of time a system call spends on CPU. The major observation from the flame graphs is that *sys_newlstat()* which is used for getting a file's inode information is represented as one block in BRINDEXER (Figure 6a) and multiple individual stack calls in GUFI (Figure 6b). Also, cumulative width of the boxes for *sys_newlstat()* in GUFI exceeds the width in BRINDEXER, which means that GUFI spends more time in CPU for retrieving file information. This shows that BRINDEXER is more effective in using the system call to retrieve file information than GUFI.

D. Evaluation of Indexer

1) Time Taken to Index: The time taken to index both workloads by BRINDEXER and GUFI is shown in Figure 7. As seen in the figure, GUFI performs better than BRINDEXER for workload 1 where there is a flat directory structure. This is because the design of GUFI enables optimization for a directory level of just 1 where each directory has a large number of files.



(a) Flame Graph for indexing with BRINDEXER.

(b) Flame Graph for indexing with GUFI.

Fig. 6: Comparison of system call stack for indexing 1.2M files in hierarchical directory structure by BRINDEXER and GUFI.

However, for the real world case in workload 2, BRINDEXER outperforms GUFI. As the number of files increase, the time to index in GUFI increases exponentially, with the maximum difference between BRINDEXER and GUFI of 69% in the time taken to index seen for 22M files.

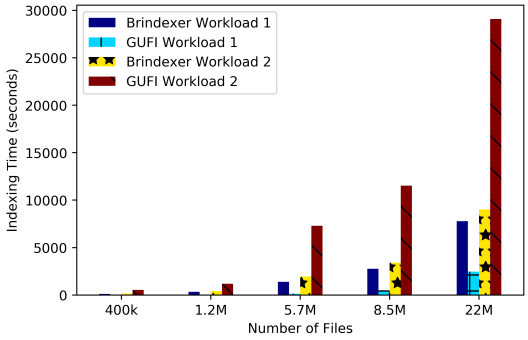


Fig. 7: Time taken to index by BRINDEXER and GUFI.

2) Resource Utilization: Figure 8 shows the resource utilization of BRINDEXER and GUFI when indexing the file system for both workloads. The legend used in Figure 8a is consistent for all the graphs in Figure 8. We only show the resource utilization of Lustre client and MDS. The behavior shown is similar on the OSSs. The CPU utilization of BRINDEXER during indexing is lesser than GUFI by 46.6% on clients and 86.04% on the MDSs. It can be further seen that GUFI is much more CPU intensive than BRINDEXER on MDS. This is because of the multiple individual stat calls that GUFI makes to the MDS as seen in Figure 6b. Even for workload 1, where GUFI takes less time to index than BRINDEXER, the CPU utilization is much more than BRINDEXER. Similar behavior as CPU is shown on other resources like network and disk. However, in case of memory, both BRINDEXER and GUFI have a similar memory utilization on clients and MDS as seen in Figures 8c and 8d.

E. Evaluation of Metadata Query Interface

To evaluate the *metadata query interface* of BRINDEXER, we run a query to find all files whose size is greater than 10 MB. We compare the query performance of BRINDEXER with GUFI, Lustre's *lfs find* tool, and *Robinhood policy engine*.

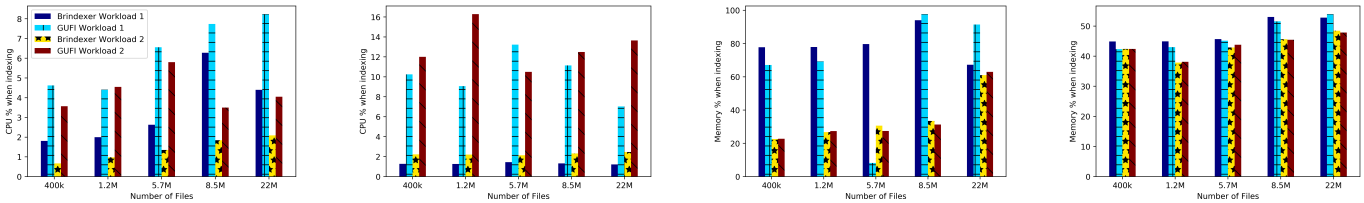
1) Time Taken to Query: Figure 9 shows the time taken to run the query and get the results back from the index database from BRINDEXER, GUFI, lfs find, and Robinhood. GUFI performs worse than BRINDEXER for both the workloads. This is because BRINDEXER makes use of parallel search on all the index nodes. The 2-level database sharding in BRINDEXER helps optimize queries further. We compare BRINDEXER with lfs find and Robinhood using queries on only workload 2. *lfs find* traverses the entire file system to get the results without using indexing and performs the worst. The difference in query performance between BRINDEXER and Lustre's default lfs find tool is proportional to the number of index nodes at the indexing level. The query performance of BRINDEXER and Robinhood is similar, though Robinhood uses an external database to index the file system. Therefore, BRINDEXER reaches an ideal query performance in the file system itself and improves upon the hardware-normalized version of the state-of-the-art GUFI by 91%.

2) Resource Utilization: Figure 10 shows the resource utilization of BRINDEXER and GUFI when querying the file system for both workloads. The legend used in Figure 10a is consistent for all the graphs in Figure 10. The resource utilization during querying is shown on OSSs instead of MDS because metadata index database resides in the OSSs of the file system. It is seen that GUFI's query task is much more CPU intensive than that of BRINDEXER. The memory utilization is similar for both. Therefore, BRINDEXER helps reduce CPU utilization during queries by 91.8% on clients and 57.8% on OSSs compared to GUFI.

F. Evaluation of Re-Indexer

The analysis of 24-hour Lustre changelog that was described in Section II-D2 shows that on a large scale production level Lustre store, more than 34 million events occur per day which corresponds to ∼400 events per second. We write a script that operates on the 22 million file dataset in workload 2 and generates 766 random events (create, modify and remove) per second per MDS. We then evaluate the performance of re-indexer in reporting these events.

1) Event Reporting Analysis: The event reporting rates (rate at which the suspect file is created) of BRINDEXER and Robinhood are shown in Table VII. Lustre's *fid2path* tool is resource intensive and slow. Therefore, there is a 28.7% improvement in event reporting rate when LRU cache



(a) CPU% of Client. **(b)** CPU% of MDS. **(c)** Memory% of Client. **(d)** Memory% of MDS.

Fig. 8: Resource utilization during indexing by BRINDEXER and GUFI.

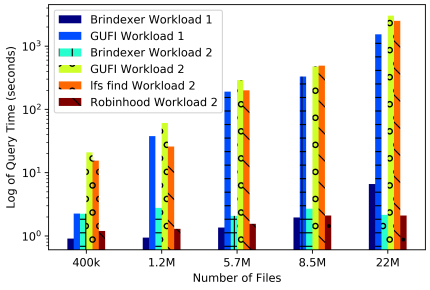


Fig. 9: Time taken to query by BRINDEXER, GUFI, *lfs find*, and Robinhood.

is used in BRINDEXER's re-indexer to store parent FID and path mappings. BRINDEXER performs better than Robinhood when it come to event reporting comparison because of Robinhood's round-robin approach to processing changelogs from the MDSs. BRINDEXER uses a parallel and scalable approach which improves the event reporting rate.

	#Events per second
Events generated	766
Events reported by BRINDEXER *without cache*	523
Events reported by BRINDEXER *with cache*	734
Events reported by Robinhood	710

TABLE VII: Event Reporting Rates by BRINDEXER and Robinhood.

2) Resource Utilization: Table VIII shows the effect of varying LRU cache size in re-indexer. The best event reporting rate with an optimal resource utilization is achieved when cache size is set to 5000. Therefore, re-indexer does not utilize a lot of cpu (2.94%) and memory (62.4 MB) and can run continuously to keep track of the metadata events in real-time.

Cache Size (#fid2path)	**CPU% on client**	**Memory (MB) on client**	**Events/sec reported by BRINDEXER**
200	4.8	88.7	578
500	3.5	84.3	624
1000	2.98	75.6	659
2000	2.95	61.3	698
5000	2.94	62.4	734
7500	2.92	60.7	720

TABLE VIII: BRINDEXER performance and resource utilization vs. cache size.

V. RELATED WORK

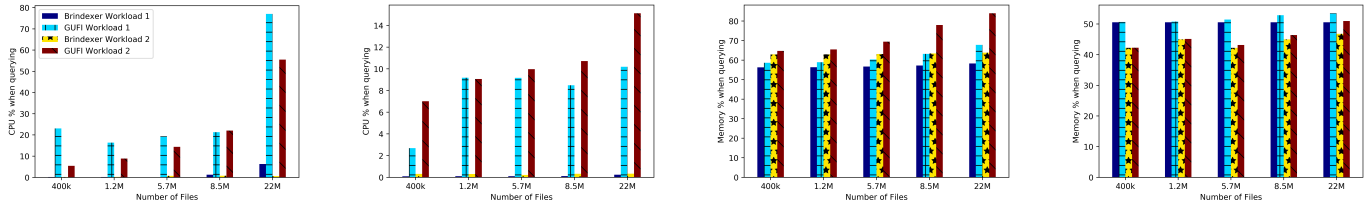
Inversion [18] is one of the first systems to propose integrating indexes into the file system. It uses a general-purpose DBMS as the core file system structure, rather than traditional file system inode and data layouts. BRINDEXER uses file system inode information to build the metadata index database. BeFS [10] uses B+tree to index file system metadata. However, it suffers from scalability issues.

Recent metadata indexing techniques include, Spyglass [16], SmartStore [12], Security Aware Partitioning [19], and GIGA+ [20] which use a spatial tree, such as k-d tree [30], or R-tree [11] to index metadata. BRINDEXER instead uses RDBMS with 2-level database sharding to efficiently store metadata index information. Other recent metadata indexing tools, GUFI [3], Robinhood Policy Engine [14], and BorgFS [1], use an external database for indexing. The metadata snapshot is taken to an external node where the indexing is performed. BRINDEXER uses an in-tree design so that no external resources are used that compromises scalability of the indexing approach. PROMES [17] is another recent approach which uses provenance to efficiently improve metadata searching performance in storage systems. However, provenance depends on building relationship graph which is infeasible on large-scale HPC storage systems. Therefore, this technique serves well for single node file systems.

Dindex [31] is a distributed indexing technique which comprises hierarchical index layers, each of which is distributed across all nodes. It builds upon distributed hashing, hierarchical aggregation, and composite identification. BRINDEXER uses leveled partitioning technique so that every disjoint subtree can be indexed in parallel. TagIt [25], Someta [28], and EMPRESS [13] are metadata management systems that enable "tag and searching". The metadata can be enriched by using custom tags for filtering, pre-processing or automatics metadata extraction. However, this is inbuilt into the storage system. Client nodes have more power and faster interconnect, therefore the indexing tool can leverage that power like BRINDEXER to index metadata from the file system clients.

VI. CONCLUSION

In this paper, we have presented BRINDEXER, a metadata indexing tool for large-scale HPC storage systems. BRINDEXER has an in-tree design where it uses a parallel leveled partitioning approach to partition the file system namespace into disjoint sub-trees. BRINDEXER maintains an



(a) CPU% of Client. **(b)** CPU% of OSS. **(c)** Memory% of Client. **(d)** Memory% of OSS.

Fig. 10: Resource utilization during querying by BRINDEXER and GUFI.

internal metadata index database which uses a 2-level database sharding technique to increase indexing and querying performance. BRINDEXER also uses a changelog-based approach to keep track of the metadata changes and re-index the file system. BRINDEXER is evaluated on a 4.8 TB Lustre storage system and is compared with state-of-the-art GUFI and Robinhood engines. BRINDEXER improves the indexing performance by 69% and the querying performance by 91% with optimal resource utilization.

In future, we plan to implement the re-indexer within the indexer of BRINDEXER so that there is no overhead from reading and writing entries to suspect files. We also plan to implement BRINDEXER for other HPC storage systems like BeeGFS and IBM Spectrum Scale.

ACKNOWLEDGMENTS

We thank Brad Settlemyer and Scott White at LANL for providing us the knowledge about GUFI and giving us a snapshot of a 24-hour Lustre changelog.

This work is sponsored in part by the National Science Foundation under grants CCF-1919113, CNS-1405697, CNS-1615411, CNS-1565314/1838271 and OAC-1835890.

REFERENCES

[1] BorgFS. https://www.snia.org/educational-library/borgfs-file-system-metadata-index-search-2014. Accessed: December 7 2019.

[2] Flame Graph. http://www.brendangregg.com/flamegraphs.html. Accessed: December 7 2019.

[3] GUFI. https://github.com/mar-file-system/GUFI. Accessed: November 30 2019.

[4] LFS Find. http://manpages.ubuntu.com/manpages/precise/man1/lfs.1.html. Accessed: December 10 2019.

[5] NERSC Report. https://www.nersc.gov/news-publications/nersc-news/nersc-center-news/2017/new-storage-2020-report-outlines-future-hpc-storage-vision/. Accessed: November 30 2019.

[6] Top 500 List. https://www.top500.org/lists/2019/11/. Accessed: November 30 2019.

[7] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '97, pages 78–86, New York, NY, USA, 1997. ACM.

[8] T. Bisson, Y. Patel, and S. Pasupathy. Designing a fast file system crawler with incremental differencing. *ACM SIGOPS Operating Systems Review*, 46(3):11–19, 2012.

[9] J. Cipar, G. Ganger, K. Keeton, C. B. Morrey, III, C. A. Soules, and A. Veitch. Lazybase: Trading freshness for performance in a scalable database. In *EuroSys*, pages 169–182, New York, NY, USA, 2012. ACM.

[10] D. Giampaolo. *Practical file system design with the Be file system*. Morgan Kaufmann Publishers Inc., 1998.

[11] M. Hadjieleftheriou, Y. Manolopoulos, Y. Theodoridis, and V. J. Tsotras. R-trees: A dynamic index structure for spatial searching. *Encyclopedia of GIS*, pages 1805–1817, 2017.

[12] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian. Smartstore: a new metadata organization paradigm with semantic-awareness for next-generation file systems. In *SC*, pages 1–12, Nov 2009.

[13] M. Lawson and J. Lofstead. Using a robust metadata management system to accelerate scientific discovery at extreme scales. In *2018 IEEE/ACM PDSW-DISCS*, pages 13–23. IEEE, 2018.

[14] T. Leibovici. Taking back control of hpc file systems with robinhood policy engine. *arXiv preprint arXiv:1505.01448*, 2015.

[15] A. Leung, I. Adams, and E. L. Miller. Magellan: A searchable metadata architecture for large-scale file systems. *University of California, Santa Cruz, Tech. Rep. UCSC-SSRC-09-07*, 2009.

[16] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller. Spyglass: Fast, scalable metadata search for large-scale storage systems. In *FAST*, volume 9, pages 153–166, 2009.

[17] J. Liu, D. Feng, Y. Hua, B. Peng, and Z. Nie. Using provenance to efficiently improve metadata searching performance in storage systems. *Future Generation Computer Systems*, 50:99–110, 2015.

[18] M. A. Olson et al. The design and implementation of the inversion file system. In *USENIX Winter*, pages 205–218, 1993.

[19] A. Parker-Wood, C. Strong, E. L. Miller, and D. D. Long. Security aware partitioning for efficient file system search. In *2010 IEEE 26th MSST*, pages 1–14. IEEE, 2010.

[20] S. Patil and G. A. Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *FAST*, pages 13–13, 2011.

[21] A. K. Paul, R. Chard, K. Chard, S. Tuecke, A. R. Butt, and I. Foster. Fsmonitor: Scalable file system monitoring for arbitrary storage systems. In *2019 CLUSTER*, pages 1–11. IEEE, 2019.

[22] A. K. Paul, O. Faaland, A. Moody, E. Gonsiorowski, K. Mohror, and A. R. Butt. Understanding hpc application i/o behavior using system level statistics. *SC*, 2019.

[23] A. K. Paul, A. Goyal, F. Wang, S. Oral, A. R. Butt, M. J. Brim, and S. B. Srinivasa. I/o load balancing for big data hpc applications. In *2017 IEEE Big Data*, pages 233–242. IEEE, 2017.

[24] A. K. Paul, S. Tuecke, R. Chard, A. R. Butt, K. Chard, and I. Foster. Toward scalable monitoring on large-scale storage for software defined cyberinfrastructure. In *2nd PDSW-DISCS in SC*, pages 49–54, 2017.

[25] H. Sim, Y. Kim, S. S. Vazhkudai, G. R. Vallée, S.-H. Lim, and A. R. Butt. Tagit: an integrated indexing and search service for file systems. In *SC*, page 5. ACM, 2017.

[26] C. A. Soules, K. Keeton, and C. B. Morrey, III. Scan-lite: Enterprise-wide analysis on the cheap. In *EuroSys*, New York, USA, 2009. ACM.

[27] D. A. Talbert and D. Fisher. An empirical analysis of techniques for constructing and searching k-dimensional trees. In *Proceedings of the sixth ACM SIGKDD*, pages 26–33. ACM, 2000.

[28] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol. Someta: Scalable object-centric metadata management for high performance computing. In *CLUSTER*, pages 359–369. IEEE, 2017.

[29] B. Wadhwa, A. K. Paul, S. Neuwirth, F. Wang, S. Oral, A. R. Butt, J. Bernard, and K. Cameron. iez: Resource contention aware load balancing for large-scale parallel file systems. In *IPDPS*, 2019.

[30] X. Yang, Q. Liu, B. Yin, Q. Zhang, D. Zhou, and X. Wei. k-d tree construction designed for motion blur. In *Proceedings of the Eurographics Symposium on Rendering: Experimental Ideas & Implementations*, pages 113–119. Eurographics Association, 2017.

[31] D. Zhao, K. Qiao, Z. Zhou, T. Li, Z. Lu, and X. Xu. Toward efficient and flexible metadata indexing of big data systems. *IEEE Transactions on Big Data*, 3(1):107–117, 2017.