An Efficient and Flexible Metadata Management Layer for Local File Systems

Yubo Liu[1,2], Hongbo Li[1], Yutong Lu[1], Zhiguang Chen[*,1], Ming Zhao[*,2]
[1]*Sun Yat-sen University,* [2]*Arizona State University*
{*yliu789, mingzhao*}*@asu.edu,* {*hongbo.li, yutong.lu, zhiguang.chen*}*@nscc-gz.cn*

***Abstract*—The efficiency of metadata processing affects the file system performance significantly. There are two bottlenecks in metadata management in existing local file systems: 1) Path lookup is costly because it causes a lot of disk I/Os, which makes metadata operations inefficient. 2) Existing file systems have deep I/O stack in metadata management, resulting in additional processing overhead. To solve these two bottlenecks, we decoupled data and metadata management and proposed a metadata management layer for local file systems. First, we separated the metadata based on their locations in the namespace tree and aggregated the metadata into fixed-size metadata buckets (MDBs). This design fully utilizes the metadata locality and improves the efficiency of disk I/O in the path lookup. Second, we customized an efficient MDB storage system on the raw storage device. This design simplifies the file system I/O stack in the metadata management and allows metadata lookup to be completed with constant time complexity. Finally, this metadata management layer gives users the flexibility to choose metadata storage devices. We implemented a prototype called Otter. Our evaluation demonstrated that Otter outperforms native EXT4, XFS, Btrfs, BetrFS and TableFS in many metadata operations. For instance, Otter has 1.2 times to 9.6 times performance improvement over other tested file systems in file opening.**

***Index Terms*—file system, metadata, path lookup, namespace**

I. Introduction

Metadata processing incurs a lot of overhead in real-world workloads [1]–[3]. To demonstrate this overhead, we used four workloads (Fileserver, Webserver, Webproxy, Varmail) of Filebench [4] in EXT4 on HDD for 60 seconds, with 50 GB data set on a machine with 64 GB RAM. We observed that metadata operations accounted for more than 39.4% of total overhead. According to our analysis, there are two bottlenecks in the metadata processing.

The first bottleneck is the high cost of path lookup. Path lookup is a frequent operation in the file system. Although caching directories can speed up path lookup, but it is unrealistic to cache all directories in memory [5]–[7], so many path lookups need to be performed on disk. However, on-disk path lookup is slow in traditional file systems (e.g. EXT4 [8], XFS [9] and Btrfs [10]). These file systems organize directories as special files. This design makes path lookup inefficient because it brings a lot of disk I/O when resolving directory files. In addition, directories need to be recursively resolved in the path lookup, which means that the metadata of the next level directory is difficult to be perceived, so traditional file systems are difficult to fully utilize the metadata locality.

The path lookup bottleneck also exists in KV-based file systems, such as TableFS [11] and BetrFS [12], which store metadata in write-optimized KV databases. The reasons for the inefficiency of path lookup in these systems are: 1) Queries are slow in the write-optimized databases in some cases. Most write-optimized databases optimize disk I/O by caching write operations, but additional overhead (e.g. multi-level table query and value update) is required for queries. In addition, directory/file entry lookups require logarithmic time complexity to complete in these write-optimized index (like LSM-Tree and B^{ε}-Tree). 2) These KV-based file systems are difficult to fully utilize the metadata locality in a large namespace. Although they can indirectly reflect metadata locality by sorting paths in the tables, the table compaction (in TableFS) and hierarchical sorting (in BetrFS) will destroy the locality when the namespace size increases.

The second bottleneck is the deep I/O stack. Traditional file systems manage metadata on memory and disk separately, which causes data structure conversion and additional memory copy. In the KV-based file systems, KV databases will bring some additional overhead, such as the KV (de)serialization. At the same time, the KV databases still need to run on top of the underlying file system.

We designed and implemented Otter, an independent metadata management layer to address the above two bottlenecks. Otter has two key designs. First, it improves the efficiency of disk I/O in path lookup by reducing the number of I/Os and cost of indexing on disk; Second, it decouples the metadata and data management in the local file systems and simplifies the I/O stack in metadata processing. We discarded stacking metadata management on file management or KV database to reduce the overhead of metadata structure conversion and additional memory copy.

In summary, this paper makes the following contributions: **1)** We designed a locality-aware metadata structure to accelerate the path lookup. The namespace tree is separated into some fixed-size metadata buckets (MDBs). The MDB design can improve the disk I/O efficiency because the path lookup process can benefit from the metadata locality of the MDBs. **2)** We designed an efficient MDB storage system based on the fix-sized characteristic of MDB, which directly manages the raw storage device. It simplifies the I/O stack in metadata management and allows the directory/file entry lookup to be

[*] Corresponding Author

978-1-5386-6648-7/19/$31.00 ©2019 IEEE
DOI 10.1109/ICCD46524.2019.00034

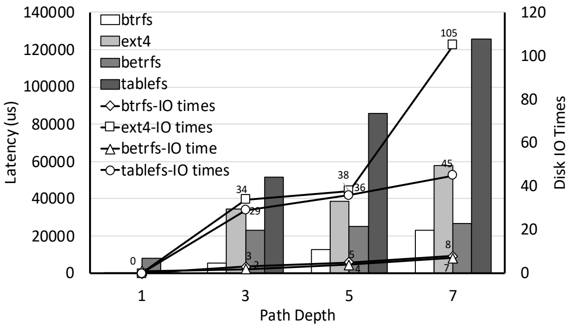


Fig. 1: Latency and Disk I/O Times of Opening File.

TABLE I: Overhead Breakdown

Path Depth		1	3	5	7
EXT4	**Lookup**	0%	24.9%	44.9%	60.2%
	Other	100%	75.1%	55.1%	39.8%
Btrfs	**Lookup**	0%	52.1%	68.5%	72.5%
	Other	100%	47.9%	31.5%	27.5%

done in constant time. **3)** We prototyped our solution based on FUSE, called Otter. Our evaluations show that Otter achieves substantial speedup over many well-known file systems in many metadata operations. For example, Otter has more than 1.2 times to 9.6 times performance improvement than the tested file systems in file opening.

The rest of the paper is organized as follows: Section II describes the background and motivations. Section III presents the design of Otter. Section IV presents the evaluation results. Section V introduces the related works. Section VI concludes the paper.

II. Background and Motivations

This section introduces and analyzes two common bottlenecks in the metadata management. First bottleneck is the slow path lookup. Second bottleneck is the deep I/O stacks in metadata processing.

A. Path Lookup

Path lookup is a frequent operation in file systems. Although directory caching can improve the efficiency of path lookup to some extent, path lookup still takes a lot of overhead in some scenarios (e.g. random workload, large namespace). We evaluated the cost of path lookup in EXT4, Btrfs, TableFS and BetrFS through file open operation on the cold cache. Among them, EXT4 and Btrfs are traditional file systems, whereas TableFS and BetrFS are KV-based file systems. We used *iostat* to detect the disk I/O times in file opening and used Perf [13] to break down the latency into path lookup and other overhead. The namespace trees used in these experiments consisted of ten layers. It includes approximately one million directories and files. Figure 1 and Table I illustrate that the latency and the path lookup overhead increases rapidly as the path depth increases. According to our analysis, there are two aspects that lead to inefficient path lookup:

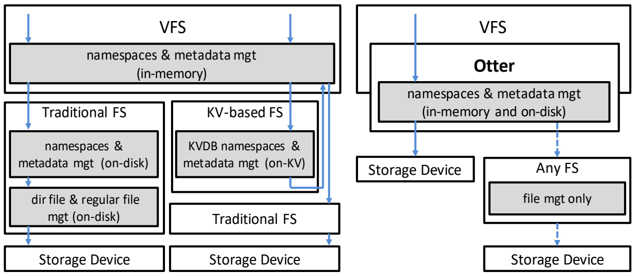
1) Metadata Organization Limitation: **a)** ***Traditional file systems***: These file systems organize namespace as special files. During path lookup, for each level of the target path, traditional file systems have to search the inode of the target directory and resolve the directory content to get the inode number of the next level directory. Figure 1 shows that EXT4 generated a lot of disk I/Os during path lookup. In particular, EXT4 has many I/Os not only because of the directory resolving but also due to its inode prefetch mechanism. However, this mechanism is almost ineffective in this experiment because EXT4 is difficult to guarantee the metadata locality on the disk (see the discussion in point 2). Btrfs is better than EXT4, but at least one disk I/O is required for each level of path.

b) ***KV-based file systems***: These file systems store the metadata on write-optimized KV databases. They perform well in some write-intense scenarios, but the path lookup has no advantage in these systems. Many write-optimized indexes improve I/O efficiency by caching the write operations on the tree root. These structures add some overhead to the query operation (e.g. path lookup). In TableFS, path lookup may need to retrieve multiple table files in the LSM tree. In BetrFS, each query operation has to apply all messages in the buffer on the parent nodes to the leaf before returning the value. Figure 1 shows that the open latency in KV-based file systems is still high.

2) Metadata Locality Limitation: The ability to take advantage of the metadata locality also affects path lookup efficiency. **a)** ***Traditional file systems***: These file systems are difficult to accelerate through metadata locality in the path lookup. First, path lookup needs to recursively resolve the directory files for each level. Second, metadata are stored as regular files and file storage cannot perceive the structure of the namespace, so traditional file systems cannot perform efficient metadata prefetching.

b) ***KV-based file systems***: In some cases, metadata locality can be reflected by sorting the paths in KV indexes, but this method does not work well if the namespace is large. TableFS sorts the metadata by their parent IDs and entry names on the tables. However, when the namespace is large, metadata locality is difficult to be reflected because they will be split into many SSTables and the table compaction process will destroy the metadata locality. For BetrFS, it is difficult to reflect the inter-layer locality in the large namespace. BetrFS sorts all metadata by their full-paths in a B$^{\varepsilon}$-Tree. They sort the paths first by the number of slashes, then lexicographically. However, when the namespace is large and deep, many blocks will be filled with metadata at the same layer in the directory tree. Therefore, the inter-layer locality cannot be reflected.

Motivation 1. To improve the path lookup efficiency, we separated the namespace tree with a subtree partition algorithm and aggregated the subtree segments into some fixed-size metadata buckets (MDBs). This method can fully utilize the metadata locality in any workload and namespace.



(a) Transitional & KV-based FS (b) Otter

Fig. 2: Architecture Comparison. The solid lines are metadata I/Os, the dot lines are (part of) data I/Os.

B. IO Stack in Metadata Management

The overhead of I/O stack also needs to be considered in the metadata management. In traditional file systems, Linux kernel uses VFS to manage in-memory metadata and provides a global namespace, whereas the underlying file systems are responsible for managing their own metadata and namespaces on disk (Figure 2(a)). So, traditional file systems use different metadata structures and management methods between memory and disk. This design increases the complexity of the file system I/O stack and takes additional overhead, such as memory copy and metadata structure conversion. The overhead of the I/O stack is more obvious on fast storage devices [14].

The I/O stack in the KV-based file systems is also complex. As shown in Figure 2(a), the metadata I/O needs to pass both the KV database and the underlying file system. In addition, KV databases will incur extra overhead. For example, metadata are stored as KV pairs in the KV database, and when metadata is read/written, the whole value needs to be read/rewritten even if only a small portion of the metadata is used. It results in unnecessary KV (de)serialization overhead [15].

Motivation 2. To simplify the I/O stack in metadata management, we designed an independent metadata management layer on the raw storage device for local file systems. We implemented an efficient storage system based on the structural characteristics of MDBs and used the same metadata structure on the memory and disk. These designs can provide fast on-disk metadata indexing and reduce the overhead of unnecessary data copy and structure conversion on the I/O stack.

III. Otter Design

This section first introduces the design and the main operations of Otter, and then introduces the metadata storage and show some advantages of metadata bucket design. Finally, it introduces the file management and consistency guarantee.

A. Overview

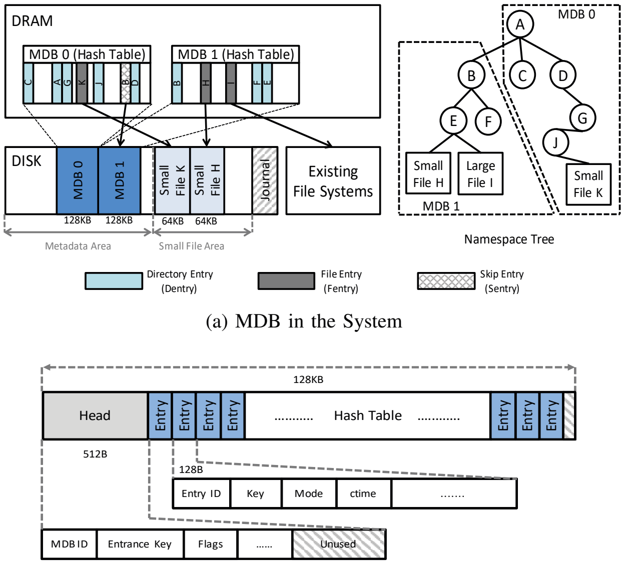
Otter is an independent metadata management layer (as shown in Figure 2(b)). Unlike existing architectures, we separated the metadata from data management in file systems. Otter is responsible for managing metadata, which means that all namespace operations will be done on Otter. Underlying file systems are only responsible for managing file data. We organized the metadata by a locality-aware method and used direct I/O to directly manage raw block device. Otter provides fast path lookup and a simple metadata I/O stack for local file systems, which accelerates many metadata operations. Furthermore, metadata are usually small and are accessed frequently so they are ideal for storage on devices with high speed but limited capacity devices (such as flash and non-volatile memory). Otter gives users the flexibility to choose metadata storage devices separately from the data storage devices.

B. Locality-Aware Metadata Organization

Instead of organizing metadata using files or KV pairs, Otter uses metadata buckets (MDBs) to store the metadata. An MDB is a fixed-size (default is 128 KB) segment and is the smallest I/O unit between the DRAM and disk. Figure 3 shows the metadata organization in Otter. The namespace tree is separated by a subtree partition algorithm, and the directories/files in the same partition are aggregated into the same MDB. An MDB includes an entry area and a head. The entry area includes many preallocated entries for storing metadata, and they are indexed by a hash table. For a given directory or file, we used the full path of the directory or file as a hash key and the offset in the hash table as the hash value. There are three entry types in an MDB: directory entries, file entries and skip entries. Since the namespace tree is divided into many subtrees, we used a skip entry to denote the split point. The MDB head contains some descriptive information (Figure 3(b)). Specially, entrance key records the root directory of the MDB.

Path Lookup. The main process of path lookup is to search the entry of each parent of the target path. Path lookup starts from the root directory and MDB 0 (the root directory must be in MDB 0). For each parent directory, Otter uses its full path as the key to search the entry in the current MDB. There are three different possibilities. In the first case, the key hits an entry in the MDB hash table that is not a skip entry, which means that the target metadata exists, so Otter can look up the next level. The second case is an entry miss, which means the parent does not exist and Otter needs to return the failure type to the lookup function caller. In the third case, the key hits a skip entry in the MDB hash table. This outcome means the target entry is in the other MDB. Otter fetches the MDB using the MDB ID recorded in the skip entry, then searches for the entry in the new MDB and increases the current MDB ID.

Creating Directory. The first step of directory creation is to look up the parent directory of the given path. Otter tries to create a new directory entry (dentry) in the MDB that contains the parent directory. If this MDB is not full then we created the new dentry in it and complete the create operation. If the MDB is full then Otter launches the MDB split process and



(a) MDB in the System

(b) MDB Layout

Fig. 3: Metadata Organization.

retry the directory creation. The MDB split process will be introduce in Section III.C.

Creating File. File creation is similar to directory creation. Otter first needs to look up the parent directory in which the file will be created and create a new file entry in the MDB that contains its parent directory. Otter allows users to directly store the small files on the disk. In this case, the data of the small file are stored in the small file area (as shown in Figure 3(a)). We will introduced the file management in Section III.F.

Link. To link the source file/directory to the target file/directory, Otter first looks up the entries of the source and the target files/directories, and then records the MDB ID and the in-MDB location (offset in the MDB) of the target file/directory into the source file/directory entry. The subsequent accesses on the source file/directory can be redirected to the target file/directory via the MDB ID and the in-MDB location.

Removal. Removing files and directories is fast in Otter. For file removing, Otter only needs to look up the target file entry and invalidate it. For directory removing, Otter does not need to remove all subdirectories and files under the target directory immediately. Instead, it records the removed directory entry to a list of invalidated entries so that all the directories and files under the removed subtree become unsearchable. The entries in the removed subtree still exists in the MDBs. Otter delays the cleaning process, and it can be merged to other processes such as MDB splitting and unmount.

Rename. File renaming is simple. Otter finds out the entry of the target file and rehash it in the MDB (use the new path as the key). For directory renaming, Otter needs to rehash the entire target directory. This process can be approximated by invaliding all entries under the target path and then creating an identical directory tree under the new path. Renaming directory will be costly if the target directory is large. In fact, it is a common challenge for file systems that use full path indexing [5], [12]. Fortunately, it is generally not a frequent operation in real-world workloads [15], [16].

C. Namespace Partition

Otter needs to split MDBs to adjust the growth of namespace. The ideal partition algorithm should reduce the frequency of MDB splitting while making the MDB as full as possible. However, it is a challenge in many cases. The best practice is to design the partition algorithm based on the workload characteristics, such as the structure of namespace tree, the frequency of various metadata operations, etc. In our prototype, we designed a simple algorithm for the case where the namespace tree is roughly balanced.

In Otter, an MDB will be split if its size exceeds a certain threshold. The MDB split process is as follows: 1) Find a split point in the MDB. The result of the partition is mainly affected by the choice of split point. In our algorithm, we randomly chose a subtree on the second layer of the entrance of the MDB as the split point. This lightweight algorithm can choose the split point with low overhead. 2) Move the entries (directories and files) under the selected subtree to the new MDB. 3) Add the skip entry to the old MDB. The skip entry will redirect requests to the new MDB if the path of the split point is accessed.

Our experimental results show that this algorithm is effective when the namespace is relatively balanced. We recursively created a namespace, and the experimental results show that storing a ten-layer namespace containing approximately one million directories and files requires approximately 3400 MDBs. This means that more than 70% of the space of the MDBs can be utilized. However, designing an efficient partition algorithm for the workloads with skew and frequently changing namespaces is challenging. We will solve this problem in our future work.

D. Metadata Storage

The efficiency of metadata storage and indexing on disk can also significantly affect the metadata processing performance. Traditional file systems store metadata as files, so they index the metadata on disk just like regular file data. Most of them use multilevel index tables or B-Trees (and variants of B-trees) to index the metadata on disk. KV-based file systems store the metadata in KV databases, but the metadata will eventually be stored in the table files of the database in the underlying file system. So they use the similar indexing methods to index metadata on disk with as traditional file systems. In Otter, we designed a more efficient indexing method than traditional and KV-base file systems.

By considering MDBs as fixed-size structures, we developed a simple method to index MDBs on the disk. Otter divides the disk into a number of fixed-size parts that are the same size as the MDBs (*mdb_size*). We did not use any tree index structure in Otter, but only maintained a maximum MDB ID and an on-disk FIFO queue of free MDB IDs. The offset of each MDB on the disk can be calculated as *mdb_ id* × *mdb_ size*. In the MDB allocation, the allocator returns the MDB ID on the front of the free MDB queue (if the FIFO queue is not empty). Otherwise, it returns current maximum MDB ID and then increases the maximum MDB ID. In MDB reclamation, allocator only needs to add the MDB ID into the free MDB queue. Thanks to this design, Otter can quickly locate the target metadata on disk. At the same time, Otter only needs to pay a very small overhead of index maintenance. On the other hand, entries are indexed by hash. so we also can quickly locate the target entry when the MDB is loaded into memory. In addition, Otter uses the same metadata structure (MDB) between memory and disk. It means that we can avoid extra overhead in the I/O stack, such like metadata structure conversion and (de)serialization, which is beneficial for high speed storage devices.

E. Advantages of MDB Design

Local file systems are beneficial from Otter: **Benefit 1.** Otter accelerates the path lookup, which improves the performance of many metadata operations. The locality-aware metadata organization (MDB) improves disk I/O efficiency in path lookup. Our evaluations show that Otter only needs about 3 disk I/Os, on average, to randomly lookup a ten-level path (namespace contains one million directories and files), but existing file systems require approximately one disk I/O per level. In addition, Otter can quickly index the MDB and the entry in MDB, which it can finish in constant time complexity. At the same time, Otter does not need to pay a lot of overhead on index maintaining.

Benefit 2. Otter reduces the depth of the I/O stack in metadata management. In traditional file systems, the metadata management is stacked on the file data management logic and it will cause a lot of memory operations and disk I/Os. In KV-based file systems, KV stores will bring some extra cost and the KV databases also need to be deployed upon traditional file systems. Unlike traditional and KV-based file systems, Otter stores metadata on raw device and uses the same structure to organize metadata on disk and memory. It can reduce unnecessary memory copy and structure conversion in metadata processing. In addition, separating metadata and data give users the flexibility to store metadata on high-speed storage devices.

F. File Management

Otter provides two methods to manage files: 1) store the data in the underlying file systems or 2) store the data directly in Otter. For the first method, users can mount Otter on the suitable file system based on the workload. For instance, EXT4 may be more suitable for large file read and write and BetrFS may perform well under the workload with a large amount of random small writes. Our FUSE-based prototype uses underlying file system as an object storage. Otter also allows users store the files on Otter directly, which is suitable for small files. As shown in Figure 3(b), Otter reserves an area for small files on the disk. This area is divided into multiple fixed-size buckets (one file per bucket). The bucket ID is recorded in the file pointer field of the small file entry. We used the similar method as for the MDBs to index buckets in small file areas.

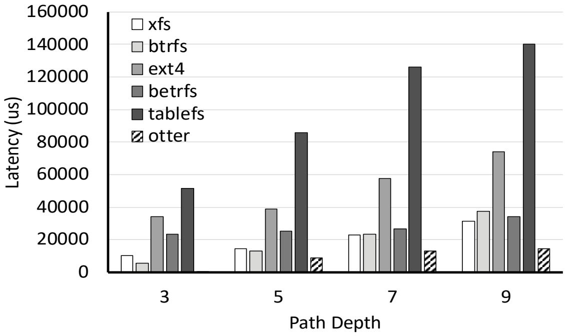


Fig. 4: Open Files.

G. Consistency

Otter uses journaling to guarantee file system consistency. We provided metadata consistency guarantee in Otter. It is similar to the default consistency mode (ordered data mode) that is provided in some mainstream journal file systems (e.g., EXT4, XFS). To guarantee the metadata consistency, dirty metadata will be logged in the journal after dirty data are written onto the disk. Therefore, when the system crashes, the file system can recovery metadata to a consistent state by redoing the metadata in the journal.

As shown in Figure 3(a), we set aside an area in the disk for journaling. For small files, we used the journaling mechanism (metadata are written to the journal after the data are written to disk) when synchronizing dirty metadata and dirty small files to the disk. The journaling process is slightly different for large files case because they are stored in the underlying file system. To guarantee the correct flushing ordering between the metadata and data of large file, we call fsync to flush the dirty file data to the underlying file system before the dirty metadata are written to the Otter's journal.

IV. Evaluation

In this section, we first evaluated Otter's performance in common metadata and data operations, and then we used applications to show Otter's performance in the real-world workloads. Finally, we evaluated the impact of MDB size on Otter's performance.

We compared Otter with EXT4, XFS, Btrfs, BetrFS (0.4 version), and TableFS. Among them, EXT4, Btrfs and XFS are widely used kernel file systems. BetrFS is a kernel KV-based (based on TokuDB) file system. TableFS is a KV-based (based on LevelDB) file system and it is also implemented on FUSE. All the results were collected on a server with two Intel E5 2456 CPUs, 64 GB RAM and 2 TB SAS HDD. Linux 3.11.10 (BetrFS can only be run on this version) was used for our evaluation. The evaluations were run on cold cache when we do not specify. We cleaned the memory and remounted the file system after each experiment to make the cache cold.

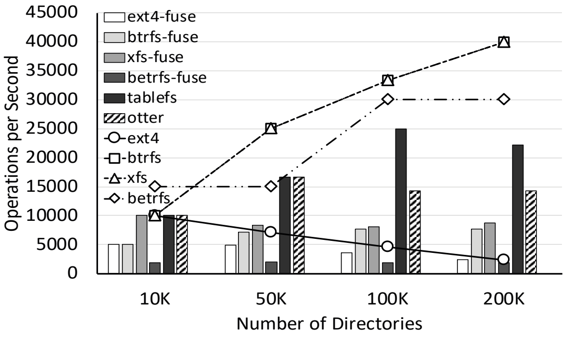


Fig. 5: Create Directories.

A. Metadata Operations

To evaluate the metadata performance of Otter, we selected four frequent metadata operations: open files, create directories, link and remove files/directories.

1) Open Files: We evaluated the performance of the file opening operation with different path depths. The namespace tree used in this experiment had ten layers and contained approximately one million directories and files. We randomly opened a file in the namespace and recorded the duration.

Figure 4 shows the performance comparison in file opening. Otter's performance is more than 1.2 times higher than the traditional file systems (EXT4, Btrfs, XFS), and it is more than 2.1 times higher than the KV-based file system (TableFS, BetrFS). The performance improvement of Otter primarily comes from the efficient disk I/O (most operations require less than 3 disk I/Os) and the shallow I/O stack. We can see that the KV-based file systems (BetrFS and TableFS) have no advantage in this experiment. One of the main reasons is that they fail to utilize the metadata locality in the path lookup. For BetrFS, the locality of depth is difficult to reflect in large namespaces. For TableFS, metadata will be stored on many SSTables when the namespace is large, thus destroying the locality. In addition, the KV databases also bring some extra overhead.

2) Create Directories: Filebench [4] was used to evaluate the performance of directory creation. We used single thread to create some directories in the file system and recorded the duration. In this experiment, we compared to kernel and FUSE versions of EXT4, XFS, Btrfs and BetrFS. Figure 5 shows the performance of directory creation.

Compared to traditional file systems, Otter exhibits a performance increase of approximately 3.5 to 13.3 times of kernel EXT4 but it is slower than kernel versions of XFS and Btrfs. Otter significantly outperforms the FUSE versions of EXT4, XFS and Btrfs. From this experiment, we also can see that FUSE will significantly degrade the performance because it will cause frequent context switching and additional data copying.

Compared to KV-based file systems, TableFS performs well in this experiment (compared to the FUSE version of other systems). One of the reasons is because LevelDB is very friendly for the insert-intensive workload. Otter is the second best system in the FUSE camp. Otter's main overhead comes from the MDB splitting. However, TableFS (LevelDB) puts some expensive tasks in the background, such as table compaction. In the future work, we will implement asynchronous MDB splitting to improve the performance of the create operations. On the other hand, it is difficult to use the metadata locality in this scenario because the directory creation process is recursive.

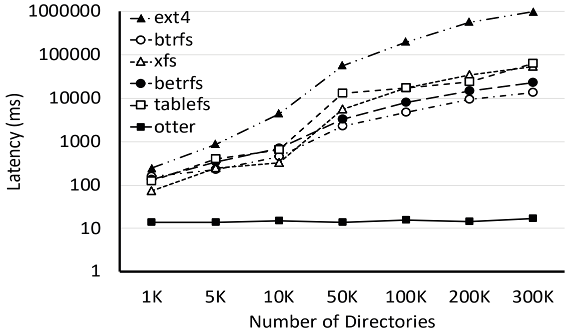


Fig. 6: Remove Directories.

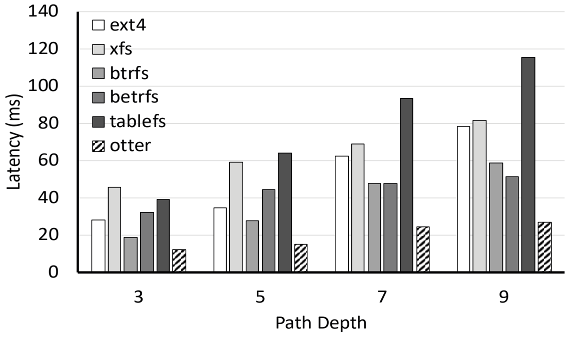


Fig. 7: Remove Files.

Another KV-based file system BetrFS does not show an advantage in this experiment. One of the reasons is that TokuDB (B^{ε}-Tree) does not perform well in the case of intensive read-after-write. Write operations in the B^{ε}-Tree are considered as messages and simply buffered in the inner node, but read operation needs to flush all messages in the inner nodes on the root-to-leaf path to the leaf node, and then returns the values. In directory creation, the values of parents directories can be accessed shortly after creation, so the caches of the inner nodes do not work much in this case. Otter performs better than these KV-based file systems because it does not need to pay for the index maintenance overhead.

3) Remove Files/Directories: We used the namespaces created in the directory creation experiment, and removed them to evaluate the duration of the directory removal operation. Figure 6 shows that Otter can significantly outperform the other tested file systems, and the latency of removal in Otter does not increase significantly as the directory size increases.

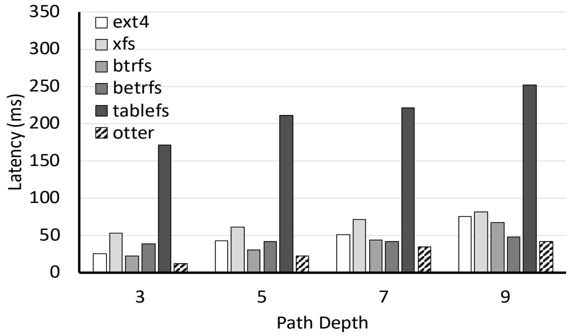


Fig. 8: Link Directories.

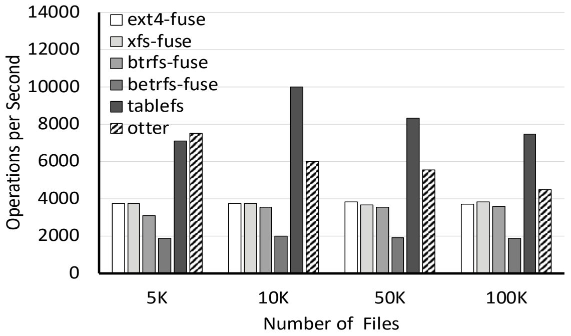


Fig. 9: Performance of File Operations.

The main reason for this performance improvement is that Otter does not need to recursively remove the subdirectories and files under the target removal point. Instead, Otter simply marks the entry of the removal point and adds it to the invalid list. The cleaning process then reclaims the invalidated entries in the background. The overhead of cleaning process will not be large because we can combine it into other processes, such as MDB splitting and unmount.

The file removal evaluation was run on a ten layer namespace that contains approximately one million files and directories. We removed files with different path depths from the namespace and recorded the durations. Figure 7 shows that Otter outperforms the other tested file systems by approximately 1.6 to 4.3 times. The main reason for the performance improvement is that Otter accelerates path lookup and reduces metadata processing overhead.

4) Linking: We evaluated the linking performance in a ten layer namespace tree used above. We linked directories in non-bottom layers (the x-axis in Figure 8) to directories in the bottom layer and recorded the execution times. The source directory and target directory were selected randomly. Figure 8 shows the result of link operation. Otter outperforms the other tested file systems by approximately 1.9 to 13.6 times. Otter's performance is more stable as the path depth of source directory grows. The main reason for the performance improvement is similar to the previous experiments. These experiments show that many metadata operations can benefit from our design.

B. File Operations

We used Filebench [4] to evaluate the file operations in single thread. There are three phases for each file in this workload: 1) create an empty file, 2) write 64 KB of data into the new file and 3) close the file. We evaluated the performance of creating different numbers of files. We compared to other kernel file systems on FUSE. Because FUSE enables the file system processing logic to be implemented in the user space, it results in considerable data movement between the kernel space and the user space [17] in the data operations, which brings additional overhead.

TABLE II: Applications Performance

Applications	find (/s)	tar (/s)	diff (/s)	stat - 5 (/us)	stat - 9 (/us)
EXT4-FUSE	734.1	1795.4	219.5	55688.2	89659.1
XFS-FUSE	118.2	283.9	26.8	49797.3	84525.2
Btrfs-FUSE	127.4	199.8	30.8	34388.6	66202.2
BetrFS-FUSE	266.8	499.2	67.6	71813.3	79911.7
TableFS	116.5	194.3	21.6	107277.5	180311.6
Otter	58.6	205.5	17.8	24674.2	32465.6

Figure 9 shows the results. We can see Otter's performance is higher than EXT4, XFS, Btrfs and BetrFS, but it is lower than TableFS. The reason for why Otter's performance is not as good as TableFS is that file data operation takes up most of the overhead in this experiment. In TableFS, the log-structure and compression designs of LevelDB can improve the efficiency of data I/O. In contrast, Otter only focuses on the optimization of metadata operations.

C. Applications

To show how Otter performs in real-world workloads, we evaluated Otter on commonly used applications. Because our prototype is implemented on FUSE, for fairness, we compared other kernel file systems on FUSE. We chose several command line applications for our experiment, including *find*, *tar*, *diff* and *stat*. We used the namespace from the previous experiments (ten layers, with one million files and directories). Table II shows the comparison results.

find. We created five target files on five different directories in the namespace tree and used the *find* command to search them. The results show that Otter significantly outperforms the other file systems. Because *find* contains many directory traversal operations, Otter can take full advantage of the metadata locality.

tar. We used *tar –czf* to package the entire namespace. Otter's performance is slightly slower than Btrfs and TableFS but better than EXT4, XFS and BetrFS. One reason for why Otter's advantage in this application is not obvious is that *tar* contains many data operations and they take up a lot of overhead, but Otter is mainly to speed up metadata operations.

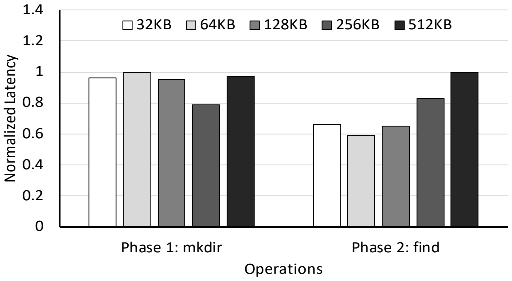


Fig. 10: Effects of Different MDB Sizes.

diff. We used *diff* to compare the two subtrees under the root of the namespace tree. Otter is faster than other compared systems. Similar to the *find*, the metadata locality is strong in this workload, so Otter has a good performance.

stat. We used *stat* to obtain the attributes of a file. In Table II, *stat-5* indicates that the object file is in the 5th layer of the namespace tree; *stat-9* is similar. Otter performs the best in all compared file systems. The performance gap increases as the depth of the target file increases. In particular, we can see that KV-based file systems have no advantage in this case. Because *stat* is a random access workload, KV-based file systems are difficult to benefit from the metadata locality.

D. MDB Size

The MDB size is an important parameter in Otter because it affects the overhead of MDB splitting and the efficiency of path lookup. We designed an experiment to show the impact of MDB size. The experiment includes two phases. We created approximately 20 thousand directories in phase 1 and then used the find command to search for a file that does not exist in phase 2 (in the cold cache). Figure 10 shows that the MDB size has a slight effect on the performance of directory creation; an appropriate MDB size is beneficial for lookup operations. We chose 128 KB in our experiments. In practice, the MDB size needs to be selected according to the workload. For example, if the locality of the application is strong, it may be more suitable for a large MDB size, and vice versa for a small MDB size.

V. Related Work

Metadata Organization. Traditional file systems mange namespace as files but this method is inefficient in creation and lookup. Therefore, some studies used KV store to manage metadata. For example, TableFS [11] and KVFS [18] are FUSE file systems based on LevelDB. BetrFS [12], [19], [20] is a kernel file system based on kernel TokuDB [21]–[24]. BetrFS 0.1 [19] uses full-path as the key, but it is not friendly for rename and large sequence write. BetrfS 0.2/0.3 [20] uses later-binding journaling and relative-path to handle the performance bottleneck in the BetrFS 0.1. BetrFS 0.4 [12] reduces the overhead of maintaining the relative-path in a KV index by using the full-path as the key and tree surgery technique. Compared to the file-based and KV-based method, Otter better utilizes the metadata locality to speed up the path lookup, thereby improving the efficiency of many metadata operations.

Path Lookup. Prior research also explored accelerating path lookup in local file systems. DLFS [5] optimized the path lookup by direct lookup, but it is difficult to extend to Linux security modules and difficult to take advantage of the metadata locality by treating the disk as a single hash table. Unlike DLFS, Otter retains the path lookup to support the full POSIX interface. The MDB design of Otter makes it possible to better utilize the metadata locality compared to DLFS. C. C. Tsai et al. [25] adds a fast lookup table in VFS and uses a full-path hashing method such as DLFS to accelerate path lookup in the memory. Otter accelerates the lookup on the disk, so they are complementary

Others. Otter simplifies I/O stack in metadata processing by directly managing disk. Deep I/O stack has significant impact on the speed of storage devices, especially, when they are fast. J. Condit et al. [26] and S. R. Dulloor et al. [27] pointed out that the deep I/O stack will become a bottleneck in the NVM-based architecture. In addition, B. K. R. Vangoor et al. [17] introduced the principle of FUSE and analyzed its overhead in great detail. EXTFUSE [28] proposed a low-cost file system framework in user space. FUSE overhead can be eliminated by porting Otter into the kernel.

VI. Conclusions

This paper considers the inefficient path lookup and complex I/O stack in metadata processing. We designed an efficient metadata management layer for local file systems. First, we reorganized the metadata using a locality-aware approach to improve the disk I/O efficiency in path lookup. Then, we designed a constant time complexity method to index metadata on the disk and memory. Finally, we stored metadata on raw device and used the same metadata structure between memory and disk to simplify the I/O stack in metadata processing. Our evaluations show that Otter can speed up many metadata operations. Our future work will involve two parts as follows: 1) Port Otter to the kernel to improve performance. 2) Explore methods based on machine learning to accommodate different workloads to improve the efficiency of the MDB splitting algorithm. 3) Eliminate the limit of fixed-size MDB on directory size by jointing multiple MDBs

Acknowledgments

We appreciate the insightful comments from anonymous reviewers. This work was supported by 2016YFB1000302 (National Key R&D Program of China), U1611261 (NSFC), 61872392 (NSFC), 2016ZT06D211 (Program for Guangdong Introducing Innovative and Entrepreneurial Teams), 201906010008 (Pearl River S&T Nova Program of Guangzhou), U.S. National Science Foundation CAREER award CNS-1619653 and awards CNS-1562837, CNS-1629888, IIS-1633381, CMMI-1610282 and Sun Yat-sen University International Program for Ph.D. Candidates.

References

[1] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller, "Measurement and analysis of large-scale network file system workloads." in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2008, pp. 5–2.

[2] C. Ruemmler and J. Wilkes, "UNIX disk access patterns," *Usenix Winter*, 1993.

[3] Q. Zhang, D. Feng, and F. Wang, "Metadata performance optimization in distributed file system," in *Proceedings of IEEE/ACIS International Conference on Computer and Information Science*, 2012, pp. 476–481.

[4] filebench, "Filebench," https://github.com/filebench/filebench, Apr. 2018.

[5] P. H. Lensing and T. Cortes, "Direct lookup and hash-based metadata placement for local file systems," in *Proceedings of International Systems and Storage Conference (SYSTOR)*, 2013, pp. 1–11.

[6] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Xu, "SANE: Semantic-aware namespacein ultra-large-scale file systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1328–1338, 2014.

[7] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems." in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2009, pp. 153–166.

[8] ext4.org, "Ext4," https://ext4.wiki.kernel.org, Sep. 2016.

[9] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system," in *Proceedings of USENIX Technical Conference (ATC)*, 1996, pp. 1–1.

[10] btrfs.org, "Btrfs," https://btrfs.wiki.kernel.org, Aug. 2018.

[11] K. Ren and G. Gibson, "TABLEFS: enhancing metadata efficiency in the local file system," in *Proceedings of USENIX Technical Conference (ATC)*, 2013, pp. 145–156.

[12] Y. Zhan, A. Conway, Y. Jiao, E. Knorr, M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, D. E. Porter, and J. Yuan, "The full path to full-path indexing," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2018, pp. 123–138.

[13] kernel.org, "Perf," https://perf.wiki.kernel.org, Sep. 2015.

[14] S. Swanson and A. M. Caulfield, "Refactor, reduce, recycle: Restructuring the I/O stack for the future of storage," *Computer*, vol. 46, no. 8, pp. 52–59, 2013.

[15] S. Li, Y. Lu, J. Shu, Y. Hu, and T. Li, "LocoFS: a loosely-coupled metadata service for distributed file systems," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017, pp. 1–12.

[16] P. H. Lensing, T. Cortes, J. Hughes, and A. Brinkmann, "File system scalability with highly decentralized metadata on independent storage devices," in *Proceedings of IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 366–375.

[17] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: performance of user-space file systems," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2017, pp. 59–72.

[18] P. Shetty, R. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with vt-trees," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2013, pp. 17–30.

[19] W. Jannen, J. Yuan, Y. Zhan, J. Esmet, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, L. Walsh, and L. Walsh, "BetrFS: a right-optimized write-optimized file system," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2015, pp. 301–315.

[20] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, L. Walsh, L. Walsh, L. Walsh, and M. A. Bender, "Optimizing every operation in a write-optimized file system," in *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 1–14.

[21] M. A. Bender, G. S. Brodal, R. Fagerberg, and D. Ge, "The cost of cache-oblivious searching," in *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, 2003, p. 271.

[22] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson, "Cache-oblivious streaming b-trees," in *Proceedings of ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2007, pp. 81–92.

[23] G. S. Brodal and R. Fagerberg, "Lower bounds for external memory dictionaries," in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2003, pp. 546–554.

[24] G. S. slting, E. D. Demaine, J. T. Fineman, J. Iacono, S. Langerman, and J. I. Munro, "Cache-oblivious dynamic dictionaries with update/query tradeoffs," in *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010, pp. 1448–1456.

[25] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter, "How to get more value from your file system directory cache," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015, pp. 441–456.

[26] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 133–146.

[27] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of European Conference on Computer Systems (EuroSys)*, 2014, pp. 1–15.

[28] A. Bijlani and U. Ramachandran, "Extension framework for file systems in user space," in *Proceedings of USENIX Annual Technical Conference (ATC)*, 2019, pp. 121–134.