Batch-file Operations to Optimize Massive Files Accessing: Analysis, Design, and Application

YANG YANG, QIANG CAO, and JIE YAO, Huazhong University of Science and Technology, China
HONG JIANG, The University of Texas at Arlington, USA
LI YANG, Huazhong University of Science and Technology, China

Existing local file systems, designed to support a typical *single-file access mode* only, can lead to poor performance when accessing a batch of files, especially small files. This single-file mode essentially serializes accesses to batched files one by one, resulting in a large number of non-sequential, random, and often dependent I/Os between file data and metadata at the storage ends. Such access mode can further worsen the efficiency and performance of applications accessing massive files, such as data migration. We first experimentally analyze the root cause of such inefficiency in batch-file accesses. Then, we propose a novel batch-file access approach, referred to as BFO for its set of optimized *Batch-File Operations*, by developing novel *BFOr* and *BFOw* operations for fundamental read and write processes, respectively, using a *two-phase access* for metadata and data jointly. The BFO offers dedicated interfaces for batch-file accesses and additional processes integrated into existing file systems without modifying their structures and procedures. In addition, based on BFOr and BFOw, we also propose the novel batch-file migration *BFOm* to accelerate the data migration for massive small files. We implement a BFO prototype on ext4, one of the most popular file systems. Our evaluation results show that the batch-file read and write performances of BFO are consistently higher than those of the traditional approaches regardless of access patterns, data layouts, and storage media, under synthetic and real-world file sets. BFO improves the read performance by up to 22.4× and 1.8× with HDD and SSD, respectively, and it boosts the write performance by up to 111.4× and 2.9× with HDD and SSD, respectively. BFO also demonstrates consistent performance advantages for data migration in both local and remote situations.

CCS Concepts: • **Software and its engineering → File systems management**; • **Information systems → Information storage technologies**;

Additional Key Words and Phrases: Batch-file operations, two-phase access, layout-aware scheduler

This work is supported in part by Creative Research Group Project of NSFC Grant No. 61821003, National key research and development program of China (Grants No. 2018YFA0701804 and No. 2018YFA0701805), NSFC Grant No. 61872156, the U.S. NSF under Grant No. CCF-1704504 and No. CCF-1629625, and Alibaba Group through Alibaba Innovative Research (AIR) Program.

Authors' addresses: Y. Yang, Q. Cao (corresponding author), and J. Yao, Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Ministry of Education of China and School of Computer Science and Technology, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, China; emails: {yangyang_hust, caoqiang, jackyao}@hust.edu.cn; H. Jiang, Department of Computer Science and Engineering, The University of Texas at Arlington, 500 UTA Blvd, Arlington, Texas, USA; email: hong.jiang@uta.edu; L. Yang, Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, Ministry of Education of China and School of Computer Science and Technology, Huazhong University of Science and Technology, Luoyu Road 1037, Wuhan, China; email: yl_for_work@foxmail.com.

1553-3077/2020/07-ART19 $15.00
https://doi.org/10.1145/3394286

ACM Reference format:
Yang Yang, Qiang Cao, Jie Yao, Hong Jiang, and Li Yang. 2020. Batch-file Operations to Optimize Massive Files Accessing: Analysis, Design, and Application. *ACM Trans. Storage* 16, 3, Article 19 (July 2020), 25 pages.
https://doi.org/10.1145/3394286

1 INTRODUCTION

In the Big Data era, batch-file accesses are observed to be prevalent in a variety of data processing platforms, ranging from mobile devices, personal computers, storage servers, to even data centers. Many routine application scenarios, such as storage device upgrade and replacement [25, 42], data aggregation [5, 28], big-data analytics [43, 46], and inter-cloud synchronization [2, 31], heavily depend on the performance of batch-file accesses.

Unfortunately, batch-file accesses fail to fully utilize the I/O capacity potentials offered by the underlying storage system. In particular, accessing a batch of small files has been a longstanding but not well-resolved problem. Existing file systems, such as ext4 [41] and Btrfs [35], only provide the standard file-access system calls based on a *single-file access mode*. With such mode, reading a file entails first reading the file metadata from the block device, which is then followed by fetching the file data using the addresses obtained by parsing the metadata. Similarly, when writing a file, its metadata and file data also are in turn written into different locations. In other words, the access of any file, regardless its size, requires at least two separate I/Os, one for metadata and one (or more) for data. Therefore, reading/writing a file causes a traditional file system to be extremely inefficient due to such non-sequential and dependent I/Os with high overhead [12, 13]. More importantly, accessing a batch of files, especially small files, with the single-file mode can aggregate the inefficiency of each file accessing, and make things much worse. In addition, when migrating files between different volumes or nodes with the standard system calls, the single-file access mode further lowers the migration efficiency and performance because of reading, transferring, and writing all files or directories involved one by one. In conclusion, current single-file access mode cannot sufficiently exploit performance potential of underlying hardware, such as storage and network, when processing a large number of files.

Prior works attempt to address this issue by leveraging techniques such as multi-threading, prefetching, page cache, emerging storage hardware, and specialized file systems to improve batch-file accesses. Multi-threading is user-space solution with limited, if not negative effect, due to potential I/O contention [22, 24]. Prefetching [11, 20] can indirectly and implicitly boost the read performance only if the prefetched file will be accessed next in the buffer cache. However, page cache can buffer file writes and absorb metadata updates, reducing the number of actual write I/Os. However, several limitations such as buffer capacity, persistency enforcement, and flushing overheads, actually weaken its effect. Emerging storage media such as solid-state drive can significantly improve the actual data access performance. However, the batch-file accesses based on the single-file mode cannot make full use of these new hardware (shown in the Section 2). Other solutions [34, 47] have to redesign a file system with new data layout and access procedures, and are not easily portable to existing file systems. More importantly, although these techniques can indirectly lessen the inefficiency in reading/writing files, they cannot fundamentally change the inherent serialized file-access mode, losing opportunities to improve the performance when processing massive numbers of files.

Therefore, in this article, we propose BFO, a novel *Batch-File* access mechanism to explicitly speed up processing file sets, particularly for batched small files. Complementing the single-file access mode using the standard file operations (i.e., *read()*, *write()*) in traditional file systems, BFO provides *Batch-File Read (BFOr)* and *Batch-File Write (BFOw)* operations to optimize batch-file accesses. The key idea behind BFO is to treat metadata differently from file data of files, process each type in a batch separately from the other, and further re-order and optimize the storage I/Os when accessing a batch of files. More specifically, BFOr scans the metadata of all directories and files to determine their data locations in the first phase, and then leverages a *layout-aware I/O scheduling policy* to read these data with a minimal number of large I/Os in the second phase. However, BFOw first stores all file data of multiple files into contiguous free blocks, and then updates their corresponding file metadata to be eventually flushed to disks with the fewest large I/O operations. These two fundamental batch-file operations can also be easily extended to other batch-file operations, such as *create*, *update*, and *append*. Moreover, combining these two batch-file operations, we propose a new *batch-file migration (BFOm)* to improve the migration performance for massive files.

The major contributions of this work are:

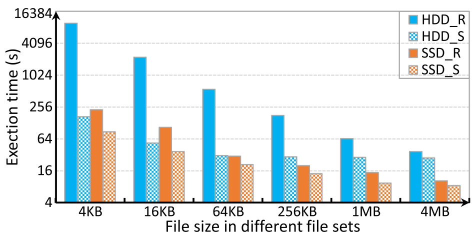
(1) We analyze the I/O behaviors when processing a batch of files under different layouts and access orders through extensive experiments. We observe that the single-file mode of traditional file systems passively causes a large number of small, non-sequential, and often dependent I/Os in the underlying storage device, degrading the overall access and migration performance.
(2) We propose BFO to optimize batch-file operations by developing two novel and fundamental batch-file access operations, BFOr and BFOw, for read and write, respectively. BFO separates operations on metadata from those on file data, and data of each type from batched files are operated together in a batched fashion.
(3) We also design a novel batch-file migration strategy BFOm by combining BFOr and BFOw with aggregate policy and appropriate order.
(4) We have implemented a BFO prototype on ext4, one of the most popular file systems. Our evaluation results show that BFO's read and write operations consistently outperform the traditional approaches regardless of access patterns, data layouts, and storage media under synthetic and real-world file sets. We also verify that BFOm boosts the overall performance of massive data migration.

The rest of the article is organized as follows. In Section 2, we present the background and motivation for the BFO research. The design and implementation of BFO are detailed in Sections 3 and 4, respectively. We evaluate BFO in Section 5. The related works are discussed in Section 6. Finally, we conclude our work in Section 7.

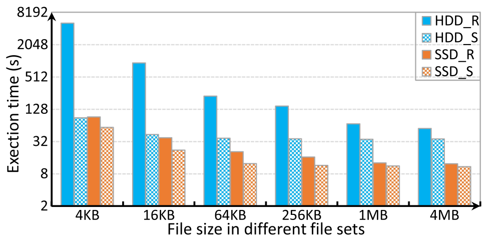
2 BACKGROUND AND MOTIVATION

2.1 Batch-file Access

Emerging data-intensive applications in the big data era are rapidly changing the data processing landscape. One of the prevalent trends for these applications is to access and process large-scale on-disk file sets. Enterprises require to backup considerable amounts of files from servers and desktops frequently. File-level data replication and data archiving also need to copy and migrate massive numbers of files to ensure data availability [36, 37]. In big data analytics systems such as Hadoop and Spark, the applications need to fetch a large number of files to process them in parallel, and create many files to store either intermediate or final results [43, 46]. A recent study estimates that billions of the users of social media and online shopping websites browse and upload trillions of photos and videos each day [3, 8]. Most of the data generated in the above scenarios are organized, stored, and accessed in sets, or batches of (often small) files. Unfortunately, the existing batch-file access operations are to invoke the standard system calls (e.g., open, read, and write) to access the files one by one. Such access mode, called *single-file access mode*, cannot fully utilize the



(a) Read



(b) Write

Fig. 1. The overall execution time of accessing different file sets with the traditional single-file access approach. These file sets contain the same total amount of file data (i.e., 4 GB) but differ in file size (i.e., ranging from 4 KB to 4 MB). We access these file sets on two types of the storage devices (i.e., HDD and SSD) with different access orders (R: Random, S: Sequential). The y-axis is in log scale.

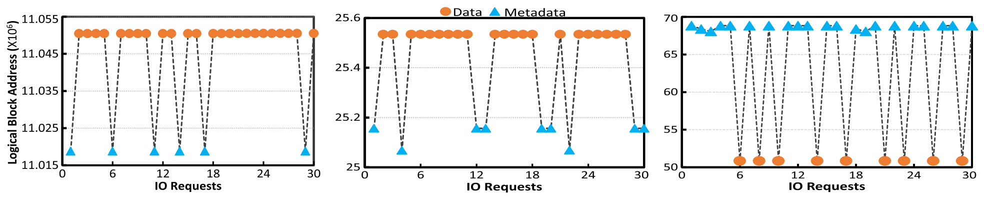
potential capacity offered by the underlying storage systems, leading to subpar or even unpleasant user experiences when processing massive files, particularly small files [9, 32].

As a real-world example, a file set of the meteorological administration of Hubei Province of China, consists of 8,639,303 weather sampling files (about 1.5 TB in total) collected from hundreds of locations in 5 years and needs to be migrated from a source hard disk with NTFS to a target RAID array with ext4. As a result, it takes about two days to duplicate all files via the USB3.0 interface. We also employed configurable system-level optimizations such as large buffer, prefetching, I/O scheduling, and hardware RAID with higher bandwidth, however, to little avail. This motivates us to explore the root cause of the inefficiency.

2.2 Problem Analysis

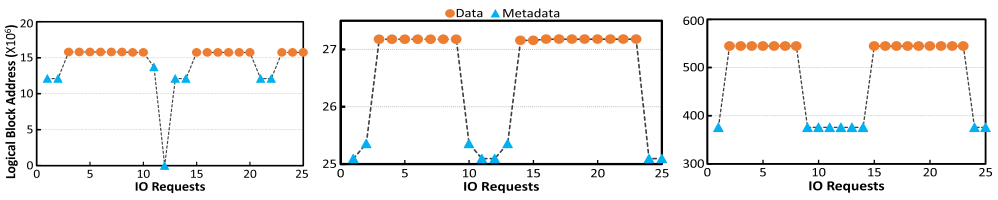
The single-file access mode, using the standard POSIX system calls, is universally applicable and effectively hides sophistical internal implementation of file systems from the applications. However, when accessing a batch of files, the mode needs to repeatedly pass through a full storage I/O stack, and frequently read/write metadata and data on different locations of the underlying storage device, resulting in many non-sequential, random and often dependent I/Os. Therefore, for batch-file accessing, this approach accumulates I/O overhead of each file, potentially leading to very low efficiency.

2.2.1 Inefficiency. To experimentally explore the inefficiency of the single-file mode in batch-file access situations, we design a set of experiments to investigate the impact of file size and access order on the overall performance. We use Filebench [1] to generate multiple file sets with the same total amount of data (i.e., 4 GB) with different file sizes (i.e., from 4 KB to 4 MB) and file counts (i.e., from 1M to 1K) on hard disk and SSD under default ext4 configuration. Every file set is consecutively stored in the storage devices, which is an ideal layout for sequential accesses. However, applications are unaware of the locations of all accessed files, and may access these files in any order. Therefore, to simulate two extreme access cases, we further read all files in each file set in totally sequential and random manners, and collect their execution times, shown in Figure 1(a). On the one hand, the execution time of the random read is noticeably longer than the sequential under the same read case, up to 57.8× for 4 KB-sized files when using hard disk as the underlying storage device. Even using SSD with higher performance, for random access, there still exists about 2.6× performance degradation compared to the sequential access case for the 4 KB-sized file set. On the other hand, we also observe in Figure 1(a) that the read performance of large-file set (i.e., 4 MB-sized files) gradually reaches the peak performance of the storage devices,



(a) Ext4 (b) Btrfs (c) F2FS

Fig. 2. File access behaviors for reading the Linux-kernel-source-code file set with three representative file systems in sequential manner. The metadata I/Os break contiguous I/Os into many non-sequential and random read I/Os.



(a) Ext4 (b) Btrfs (c) F2FS

Fig. 3. File access behaviors when writing the Linux-kernel-source-code file set with three representative file systems in sequential manner. The metadata I/Os also break contiguous data write into multiple non-sequential I/Os.

the performance of small-file set (i.e., below 1 MB-sized files), however, is much lower than that of large-file set in both access orders. For example, the sequential case with small files (e.g., 4 KB) is significantly slower than the same case with large files (e.g., 4 MB) by about 5×. Notice that they have the same consecutive file data layout, and it takes about extra 28 seconds to access inodes of 4 KB-sized file set. Therefore, the consecutive file data is not fetched sequentially. Likewise, the performances of updating (writing) a batch of files under different configurations are illustrated in Figure 1(b). The performance behaviors are still similar to the previous read case.

In summary, the traditional single-file access approach is very inefficient for batch-file operations, especially for small files (below 1 MB) in a random manner, and can hardly make full use of the underlying devices.

2.2.2 Storage Behavior. To better understand the I/O behaviors under the single-file access mode in typical file systems, we employ *blktrace* [7] to capture I/O footprints when accessing the Linux kernel source codes (version 3.5.0) as a real file set.

Figures 2 and 3 illustrate the read and write behaviors, respectively, during accessing the file set with three representative file systems, ext4 [41], Btrfs [35], and F2FS [23]. We randomly select a 100ms time window, and analyze the I/O types (i.e., metadata I/O, and data I/O) according the I/O traces, then present the access behaviors with the LBAs of these I/Os in the figures. The test file set is initially stored contiguously on the storage device in the read case, and is totally buffered in memory in the write case. Nevertheless, as shown in the figures, the expected large and sequential I/Os for the file data are actually broken into more, smaller, and potentially non-sequential read/write I/Os, due to the interweaving between metadata and file data I/Os.

For the read operation, the underlying file systems first access file metadata to determine the location of each file data, and then read the file data. Considering that the file data and metadata are always stored in different disk locations, each file read operation actually entails at least two I/Os to

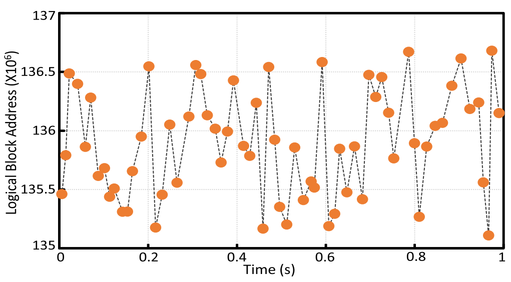


Fig. 4. Access behaviors of file data excluding the metadata with a 1-second window when reading the Linux-kernel-source-code file set on ext4.

access metadata and data, respectively. However, for these file systems, a file write operation first modifies the file inode, and then update the global metadata (e.g., bitmap) to confirm the allocated disk space, and finally writes the data. For the journaling file systems like ext4 and XFS [38], the write operation also invokes additional journaling procedure to ensure file consistency. As a result, the file systems generate several small and non-sequential I/Os for writing a file. Therefore, such a single-file access approach needs to seek back and forth between the metadata area and data area, resulting in many non-sequential I/Os.

We further analyze the I/O behaviors of the file data (excluding the metadata) during reading the file set, when the metadata have been buffered in memory. Ideally, the actual file data in contiguous locations should be accessed sequentially. However, as shown in Figure 4, the read I/Os are completely random when accessing file data. This is because the traditional single-file access approach is unaware of the underlying data layout, and may read these files/directories in any order, such as depth/breath-first way, but leading to random I/Os in the block-level.

Most file systems have employed various optimizations, such as prefetching [11, 20] and caching, to reduce the number of I/O operations to improve the access performance. Specifically, for read operations, we can prefetch several data blocks with a large I/O. The prefetching mechanism is more effective for metadata, since the metadata is usually stored in the relatively centralized storage areas. But the files' data are stored at wider and more discontinuous data areas, and are accessed at uncertain orders. The mechanism can hardly fetch the expected data that will be accessed in the near future. For write operations, we still can use the page cache to absorb metadata I/Os, but the page cache has limited impacts on data I/Os and journal I/Os, especially for update-in-place file systems. However, the traditional access approaches based on the single-file access mode cannot change the serialized execution of file-level accesses, and therefore still cannot fully exploit the performance potentials of the underlying disks.

In conclusion, the traditional access approaches unconsciously seek forth and back between different areas, and also access these file data in random order, thus resulting in many small, non-sequential, and often dependent I/Os, harming the access performance.

2.3 Batch-file Migration

Local and remote batch-file migrations are commonly used applications. The single-file access mode can further deteriorate the overall performance of such applications. Generally, copying files locally needs to read a file from the source location, and then write it to the destination before processing the next file, referred to as the *ping-pong* migration procedure. Migrating a batch of files has to repeatedly execute ping-pong procedure, which sequentially invokes the single-file access to read/write the metadata and data on different locations. The overhead of accessing each file in the file set is accumulated continually, significantly worsening the overall performance of file

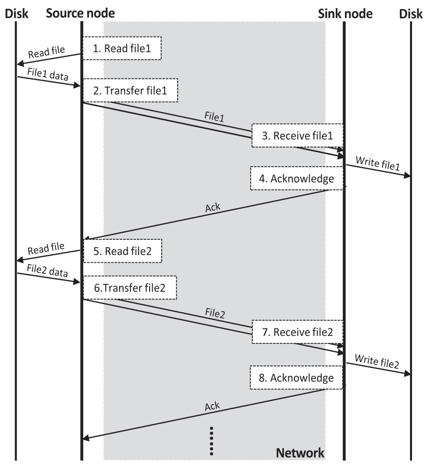


Fig. 5. Timing diagram illustrating the file migration procedure of traditional *ping-pong* approaches when migrating a batch of files.

migration. A remote file migration extends the overhead of ping-pong procedure with a network transfer stage, further increasing the overhead of each file migration.

Figure 5 illustrates the file migration process via network. When migrating a file in Linux, mainstream approaches, such as *bbcp* [18] and *scp* [14], issue a file read request to the Linux operating system. The system fetches the requested file into memory from the source storage device. Then, the requested file metadata and data are packed into one or more network packets, and sent to the network adapter. And the adapter delivers these packets to the remote destination node via network. After receiving these network packets, the destination node parses these packets to get the file metadata and file data, and then executes the local file write operation onto the storage device, and finally acknowledge the completion of the migration of the file. This procedure is repeated until the end of the migration for all files. Therefore, such ping-pong migrating procedure with single-file access mode also results in a significant transfer overhead when processing a large number of small files, as shown in Figure 5.

Many research works have proposed various techniques to optimize the migrating procedure, such as the batch processing [4, 40] to merge multiple files to reduce the network overhead, and thread-level pipelining [22, 24] to overlap the multiple processing stages. However, they rarely optimize the access (read and write) stages, and still use the single file access mode to read/write the migrated files one by one. Therefore, the overall migrating time increases linearly with the number of these files.

2.4 Motivation

In summary, current single-file access approaches are inefficient for accessing and migrating batched files, especially small files. Therefore, to fully unleash the power of the underlying storage devices, we are motivated to propose an explicit and fundamental batch-file access approach for existing local file systems to holistically optimize the overall performance. Based on the batch-file access approach, we can further optimize applications of accessing massive files.

3 DESIGN OF BFO

To overcome the drawback of the single-file access mode in existing file systems for efficiently processing batched files, we design the BFO approach with a set of effective Batch-File Operations.

The key of BFO is two fundamental batch-file operations: the batch-file read operation (*BFOr*) and the batch-file write operation (*BFOw*). BFOr and BFOw are proposed to efficiently read and write batched files. The core principles of these operations can also be easily extended to other batch-file operations such as batch-file *create*, which can be considered as a special case of BFOw that only revises the metadata. The batch-file *update* and *append* are also considered as special cases of BFOw. Besides, with BFOr and BFOw, we also propose the batch-file migration BFOm to accelerate the migration for massive small files in both local and remote situations.

3.1 Batch-file Operations

Different from the single-file access interfaces that only need the information about the target file, BFO is designed for the *Batch-file* access mode and needs to pre-determine the targeted file list and their storage volume. To this end, BFO provides two new batch-file access interfaces for the read and write operations, respectively. Users can invoke the access interfaces with appropriate parameters to read/write batched files efficiently.

Batch-File Read: *Batchread(list<filename>, list<buffer>, VolumeID)*: The *list<filename>* contains the file paths of all accessed files, and *list<buffer>* is used to keep the fetched data. The batch-file read operation eventually fetches these file data into memory from the source volume *VolumeID*.

Batch-File Write: *Batchwrite(list<pointer>, list<filename>, VolumeID)*: The *list<pointer>* and *list<filename>* contain the pointers of the buffered data and the corresponding file paths, respectively. The batch-file write operation writes all files' data into the target volume *VolumeID*, and updates their corresponding inodes to record the correct attributes and index their written data. It is easy to extend the BFOw interface with an offset array to support the batch-file append operation.

3.2 BFOr

3.2.1 Two-phase Read. For batch-file reads, to avoid frequently seeking back and forth between the metadata area and data area due to serially reading files, BFOr uses a two-phase read mechanism to separately read the metadata and file data of all accessed files in batches. In the first phase, BFOr scans the inodes of these files from the underlying storage devices according to file paths. In the second phase, BFOr directly issues disk I/Os to sequentially read data in all storage locations covered by these files without any file system interventions.

The metadata area takes up relatively very small disk space (for example, just 2 MB space for inodes in a 128 MB data group in ext4 [41]), and a data block contains multiple file inodes. Therefore, all inodes in a batch may be stored in a small and contiguous disk region. This batching technique can use existing prefetching mechanisms to enhance their performance of accessing all file metadata and the associated global metadata of the file system.

Once the metadata are read into the buffer, BFOr can obtain all raw inodes (i.e., ext4_raw_inode in ext4) recording the address information (file address, file length, etc.) of each file data. According to this information, we can further fetch all file data from disks.

3.2.2 Layout-aware Scheduling. Unlike the metadata, the data blocks of all files can be stored in more discontinuous locations. Random accesses to these data can lead to a large number of small random I/Os, and incur severe latency penalties. Fortunately, the address information of all files is made available from the first phase, which can help determine the information about file data layout on disks. Therefore, we propose a layout-aware read scheduler to access all file data efficiently.

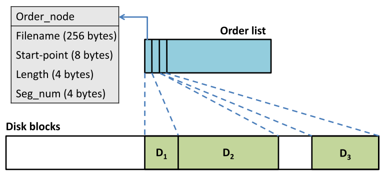


Fig. 6. The structure of the ORDER_LIST.

To determine the data layout of all files, we design a data structure called ORDER_LIST as illustrated in Figure 6, to record the address information of each file. More specifically, each ORDER_NODE in the ORDER_LIST contains the address information about the location of a contiguous data segment for a file. Figure 6 indicates that each NODE in the LIST holds four parameters for a file segment, including *filename* (256 bytes), *startpoint* (8 bytes), *length* (4 bytes), and *seg_num* (4 bytes). *Filename* records the file name of each file, while *startpoint* and *length*, used to locate this segment, represent the starting address and size of a contiguous segment of the file, respectively. Since each file may contain more than one contiguous segment on the storage device, we use one or more ORDER_NODES to locate these segments, and *seg_num* to record the order of these segments for each file.

The main purpose of the read scheduling policy is to access these data segments as sequentially as possible, and launch fewer but larger sequential read I/Os. Therefore, before issuing these data read requests, we need to sort and merge the data segments within a batch in increasing order of their starting addresses. To do so, when scanning all metadata in the first read phase, BFOr also extracts the address information from the inode for each file, and then creates one or more ORDER_NODES using this information, finally inserts these NODES to the ORDER_LIST in order of their *startpoint*s. Meanwhile, BFOr periodically traverses the ORDER_LIST to merge contiguous segments of different files into a data region to fetch it with a single I/O operation. In summary, the order of these segments stored on disks is approximately equal to the order in which they are kept in the ORDER_LIST. Therefore, when the number of accessed files reaches a threshold, BFOr sends block I/O requests to the storage devices, and can sequentially fetch all file data into memory according to the order of these ORDER_NODES in the list. The pseudo-code of BFOr is shown in Algorithm 1.

For large contiguous file segments, BFOr does not merge them to read together, because the traditional file systems already offer high sequential read performance for these segments. Therefore, BFOr directly sends the read requests into the storage devices for these segments whose size exceeds a pre-determined threshold, which eliminates any negative effect on reading these large segments. The threshold value will be further discussed in the evaluation section. In summary, with BFOr, we can access all data as sequentially as possible, and significantly reduce the number of I/O operations, which has a strong impact on the access performance for small files.

3.3 BFOw

3.3.1 Two-phase Write. Similar to BFOr, BFOw also involves a two-phase process to write a batch of files into a destination disk. In the first phase, BFOw creates a global file to store all file data once. The underlying file system (i.e., ext4, F2FS, or Btrfs) allocates as few contiguous disk spaces as possible to accommodate all data by default, and creates a specific inode (called stem inode) for this global file to maintain the file metadata information, including file address information, file length, and other attributes. Then, we need to modify each file's original inode to record the correct attributes, such as modify time (mtime), and index the corresponding written data. Therefore, in the second phase, BFOw updates the inodes for all files in a batched manner

ALGORITHM 1: Batch-file read algorithm

```
Input: filenames: file paths of all files in a fileset;
Output: filedata: in-memory file data of the fileset;
 1: Initialize inodes[] to keep the inode information of each file;
 2: Initialize order_list and order_node to record the data layout;
 3: Initialize filedata to keep the file data;
 4: divide the filenames into multiple batches;
 5: for each batch of the filenames do
 6:     i ← 0;
 7:     clear order_list;
 8:     for each filename in a batch do
 9:         inodes[i] ← ReadInode(filename);
10:         order_node ← ParseInode(inodes[i++]);
11:         insert order_node into order_list;
12:     end for
13:     SortAndMerge(order_list);
14:     filedata ← ReadData(order_list);
15: end for
16: return filedata;
```

using the stem inode and existing information, such as the system time, the size of newly written file data.

For these inodes, most of the metadata attributes (i.e., file permissions, owners, and groups) do not need to be changed at all for these files, except for the time attributes, the file sizes, and the index data of newly written data. For the time attributes (*atime*, *mtime*, and *ctime*) in the inodes, we can use the current system time or the time attributes from the stem inode to modify them. And the file size can be updated according to the size of newly written file data for each file. Therefore, the most important task in this phase is to obtain the logical block address of each file to update the index data (i.e., the *extent tree* in ext4) within the corresponding inode. Since the global file contains all file data sequentially, the index data within its inode records the data block addresses of all files. We can use the index data of the stem inode to restore the index data of all inodes. However, the stem index data only records the addresses of all data blocks of these files, and we cannot determine which file each block belongs to. To solve this problem, when writing the data of all files into the global file, we use an ORDER_LIST to record the order of the written files and the length of each file. Therefore, when updating the inodes for these files, we first extract the block addresses of all data from the stem inode, and then use the file order and file lengths in the ORDER_LIST to determine the range of the block addresses of each file, and finally update these inodes with the corresponding address ranges, and persist them to disks. The pseudo-code of BFOw is shown in Algorithm 2.

This two-phase write process is different from the traditional file write policy in two aspects. First, BFOw writes the data and metadata of all files in separate batches. Second, the conventional file write generally allocates and creates a file inode before the file data is written into the disk, while BFOw first writes the file data and then generates their own inodes. We refer to this approach as reordering-write-allocation. Compared to the traditional delayed allocation [33] in current file systems, which does not allocate blocks for the file until its data are written to disk, the reordering-write-allocation approach actively allocates and flushes a batch of file data buffered in memory into a large contiguous free disk space and then generates the relevant inodes for these files in

ALGORITHM 2: Batch-file write algorithm

```
Input: $filedata$: all files' data in a batch; $filenames$: all files' paths in a batch;
 1: Initialize address, steminode;
 2: Initialize order_list to record the lengths of the written files;
 3: Initialize *inodes[] to point the file inodes;
 4: steminode, order_list ← WriteFile(filedata);
 5: address ← ParseInode(steminode);
 6: for i=0; order_list[i]!=null; i++ do //update inodes
 7:     inodes[i] ← ReadInode(filenames[i]);
 8:     inodes[i].i_atime(i_mtime, i_ctime) ← steminode.i_atime(i_mtime, i_ctime);
 9:     inodes[i].i_size ← order_list[i].length;
10:     inodes[i].address ← address;
11:     address ← address+order_list[i].length;
12: end for
13: FlushInode(inodes);
```

batches, as well as updates global metadata once for the underlying file system. The two-phase write mechanism fundamentally unlocks the strict order constraint among the inter-file and intra-file write I/Os of metadata and data, and significantly reduces the number of write I/O operations.

BFOw not only needs to provide high write performance by using its two-phase write process but also needs to flush the data onto the underlying storage device in a timely manner to avoid memory overflows and data loss in crashes. Therefore, BFOw flushes these in-memory data periodically (e.g., 100ms), or when the data size exceeds a pre-determined threshold, whichever comes first. Taking ext4 as an example, when the size of these in-memory file data approaches 128 MB, we flush these data into an ext4 data group, which contains 128 MB data blocks.

3.3.2 Data Consistency. It is extremely important to preserve the data consistency in BFOw, since BFOw needs to store a considerable amount of data and update multiple inodes at a time. When storing all the file data, BFOw invokes the standard POSIX calls to create a global file and write all file data, the underlying file system can maintain the consistency actively by employing some mechanisms such as journaling [41] or copy-on-write (CoW) [35]. For example, under ext4 in ordered journaling mode, the file system logs only the updated metadata. However, it enforces an ordering constraint to guarantee file system consistency, in which the transaction-related data writes must be completed before the journal writes of the metadata. While the phase of updating all inodes is vulnerable to file system corruption. Upon crashes or power failures, we lose all in-memory data structures (e.g., ORDER_LIST) and cannot determine which files are in the batch-file write interface and which inodes have been updated. Therefore, we design a light-weight consistency mechanism to protect the inodes. Before writing all file data into the storage devices, BFOw first writes the ORDER_LIST into journal files as an atomic operation. Even in the absence of metadata consistency, we can still fetch the ORDER_LIST and the stem inode to continue creating all inodes. Finally, BFOw deletes and reclaims the stem inode and the on-disk ORDER_LIST when the batch-file write operation is completed.

3.4 BFOm

We further design a batch-file migration (BFOm) by combining BFOr and BFOw to optimize a representative batch-file application as data migration within a node or between storage nodes via network. When performing file migration within a node, BFOm first launches BFOr to fetch all files from the source volume, keeps all file data in memory, and then generates the corresponding ORDER_LIST. Afterwards, BFOm can invoke BFOw to write these buffered files to the destination volume efficiently. For a large number of files, we can group them into multiple batches, and execute the above migration process repeatedly for each batch of files.

When migrating batched files between two storage nodes, an important design decision is to determine the order of the migrated data, including file data, file metadata, and even directory data. A naive solution (used in RAMSYS [24]) is to first transfer the directory data, then send all file metadata, and finally transfer all file data. At the destination node, the migration tool rebuilds the whole directory structure using the directory data, and then creates the metadata of the regular files, finally writes all file data on the storage device. Unfortunately, such design choice leads to multiple updates of metadata. For example, after storing the file data, we need to update the file metadata again to record their corresponding data addresses. A potential alternative is to keep all metadata and construct the whole directory structure in memory. When these metadata are updated after the actual data are written, we flush the final-version metadata and directory data into the storage device. However, this approach requires a large memory space for metadata and directory, and yet risks data loss when system crashes.

BFOm takes a reverse data migration order compared to the above solution. This means that BFOm first access a batch of files with BFOr, and transfers these files' data to the destination node, and then sends their metadata after receiving the ack signal, and finally transfers the directory information. BFOm also needs to transfer the ORDER_LIST to the destination node upon finishing the metadata transfer. At the destination node, we invoke BFOw to efficiently store these files and directories on the storage device. More specifically, BFOm writes all received files' data into a newly created file, and obtains the logical block addresses of the file. Then with the received ORDER_LIST, BFOm computes the logical address for each file, amd creates the inodes with the metadata information. Finally, we create the directory files with the file names and their corresponding inode numbers in depth-first order.

Compared to traditional ping-pong migration approaches for batched small files, which require to perform the data migration process serially and strictly on per-file basis, BFOm can access all metadata or data of batched files at a time, and transfer them in batches, and finally flush all data together. The file migration process is illustrated in Figure 7. Therefore, BFOm can significantly reduce the number of I/O operations during migrating compared to the traditional migration approaches.

In addition, BFOr and BFOw can be used independently for applications. For example, Mapreduce applications process a large number of files at a time, we can use BFOr to fetch these files together to improve the read performance.

4 IMPLEMENTATION

In this section, we describe the implementation details of BFO on top of the ext4 file system as the most commonly used local file system. And the design principles in this implementation are equally applicable to other file systems.

To implement BFO, we develop a Batch Module that is designed as a file system metadata middleware layer on top of the existing ext4 file system. As Figure 8 shows, the Batch Module contains two sub-modules: the data sub-module, the metadata sub-module. The data sub-module implements the aforementioned two new batch-file interfaces, **Batchread()** and **Batchwrite()** in the user space for BFOr and BFOw operations, respectively. Therefore, applications can directly invoke these two interfaces in the library mode to access batched files efficiently. The data sub-module also maintains a file request queue to keep the file access requests, and an ORDER_LIST to record the mapping information for all files. For the BFOr operation, the sub-module sorts the ORDER_NODES based on

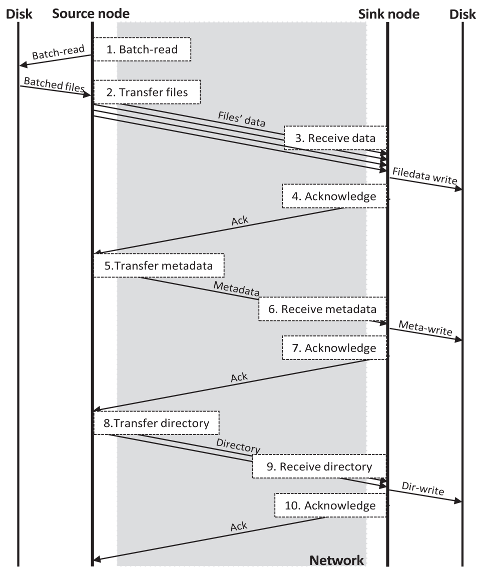


Fig. 7. Timing diagram illustrating the file migration procedure of Batch-file migration for a batch of files.

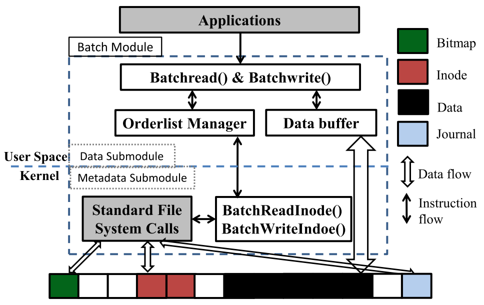


Fig. 8. Architecture of the batch module.

their *startpoint*s, and merges contiguous file segments to read with large I/Os; and for the BFOw operation, the sub-module creates a new file, and writes all file data into this file using the standard file system interfaces, then records the write order of all files and file lengths into the ORDER_LIST, and finally sends the LIST to the metadata sub-module for the metadata creation or update.

Because local file systems do not directly expose the file data layout to the user space, we design a metadata sub-module to obtain and modify the underlying storage layout in the kernel. We develop a function, called **BatchReadInode()**, to fetch the *ext4_raw_inode* structure in the kernel, then extracts the *ext4_extent* structure, which contains the starting addresses and lengths of all segments, and further create the address information using the starting addresses and lengths of these extents, and finally send the mapping information to the data sub-module. Moreover, for the BFOw operation, we also design a **BatchWriteInode()** function in the metadata sub-module to fetch the stem inode and the existing inodes of newly written files, and extracts all *ext4_extent* information (i.e., the starting addresses, lengths and offsets) from the stem inode into an *extent_item* structure. Finally, we use this existing information to update these existing inodes as follows: *(1)* update the file length of each file; *(2)* update the time attributes of each inode by assigning the ones of the stem inode; *(3)* update the *ext4_extent* structure of each file inode according to the *extent_item* and the ORDER_LIST.

The data sub-module is file-system-independent, while the metadata sub-module is file-system-dependent. Therefore, for other file systems, we only need to redesign the metadata sub-module according to the data structures of the corresponding file system. We do not alter the on-disk structure and the software flow of existing file systems. BFO merely adds a set of new flows on top of these components, keeping their existing functionality and maturity. The BFO operations only can be executed exclusively and atomically. Only after a BFO operation is completed, its involving files can be accessed by the other read/write operations, avoiding data inconsistency and I/O contention.

Moreover, for comparison with the traditional single-file access approaches using the standard POSIX interface for batched files, we also modify the Linux cp source code to implement the read, write, and copy operations using the standard POSIX interface for batched files. Based on BFOr and BFOw, we also implement BFOm by periodically launching BFOr and then BFOw. There are 500+ lines of code for the data sub-module based on Linux cp source code and 700+ lines of code for metadata sub-module.

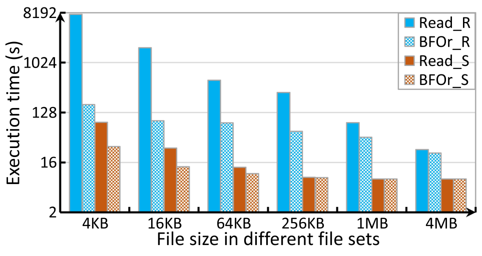
5 EVALUATION

To thoroughly and fairly evaluate the effectiveness of BFO, extensive experiments were conducted on a BFO prototype running on a server equipped with an Intel (R) Xeon (R) CPU E5620 @ 2.40 GHz and 16 GB RAM. The storage subsystem contains a RAID00 (which consists of 5 Western Digital 7,200 RPM 4 TB SAS HDDs), a Western Digital 7,200 RPM 4 TB SAS HDD, and a 480 GB SAMSUNG 750 EVO SSD. The operating system is Ubuntu 16.04 with kernel version 4.4.24. We use ext4 as the default file system. By default, dirty pages are asynchronously committed to the storage device every 5 seconds. The sizes of block and inode are set to be 4,096 and 256 bytes, respectively. The experiments are conducted on a cold-cache basis unless otherwise stated.

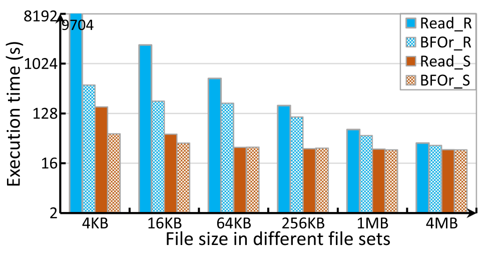
We examine the performance of BFOr, and BFOw separately in comparison with the *single-file* access approaches in ext4 using multiple synthetic file sets and a real-world file set. We then explore the performance impacts of BFO with a cold/warm cache and by varying the number of files for each batch-file operation and buffer size. We also measure the performance of real-world applications using BFOr and BFOw. Finally, we compare the migration performance of BFOm to the traditional approaches with the *single-file access mode.*

5.1 Batch-file Access Performance

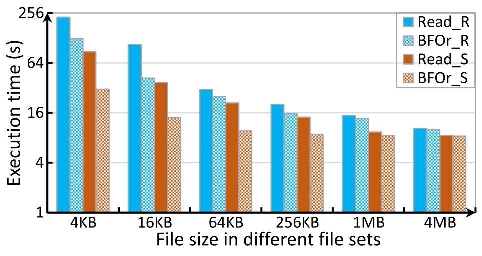
In this subsection, we first evaluate BFO in term of read and write performances compared with the single-file mode under different conditions, including different file sizes (from 4 KB to 4 MB), different storage devices (RAID, HDD, and SSD), and different access orders (sequential and random). The file sets used in the experiments contain the same total amount of file data (i.e., 4 GB) but differ in file size (i.e., ranging from 4 KB to 4 MB), and are stored contiguously on these devices. For read operations, we divide each file set into a set of batches with the same number of files, then read each batch in turn using the two access approaches. We record the total access time of each approach. More specifically, for sequential read, we divide the on-disk contiguous files into each batch; for random read, all files are randomly divided into each batch, which means that the files in a batch are actually dispersed in the disk space. Therefore, the performance improvement of BFO for accessing a whole file set can approximately represent the improvement for accessing a subset of files within the file set. Then, we also measure the migration performance of BFOm with a real file set. To do so, we duplicate the Linux source code (version 3.5) six times as the file set,



(a) RAID



(b) HDD



(c) SSD

Fig. 9. The execution time of reading the different file sets as a function of file size. These file sets contain the same total amount of file data (i.e., 4 GB) but differ in file size (i.e., ranging from 4 KB to 4 MB). We use BFOr and the single-file access approach to read these file sets from three types of the storage devices (i.e., RAID, HDD, and SSD) with different access orders (R: Random, S: Sequential). The y-axis is in log scale.

called Linux-kernel-source-code file set, containing 233,988 files in 14,587 directories with 3.1 GB in size.

5.1.1 Read Performance. Figure 9 shows the execution times of reading different file sets into memory using BFOr and Linux read() system call, respectively, in all cases. As we can see, for random read, BFOr outperforms the traditional read process by up to 42.1× and 22.4× with 4 KB files on RAID and HDD, respectively. This is because BFOr sorts and merges small I/Os, and can access all file data as sequentially as possible with fewer read I/Os, the storage devices do not need to seek back and forth for different files, especially for random accesses on hard disks. Even for SSD, BFOr achieves up to 81.4% performance improvement, since SSDs have higher sequential access performance compared to random one [29]. For sequential read, the performance improvement of BFOr is relatively small, due to prefetching and I/O scheduling mechanisms. Nevertheless, BFOr brings 1.6×, 2.0×, and 1.8× performance gains for RAID, HDD, and SSD, respectively. We also observe that the performance gap shrinks with the increase of file size. When the file size reaches 4 MB, BFO still has higher performance for random access. However, for sequential read, when the file size reaches 256 KB, the execution time of traditional approach is close to that of BFOr, and the traditional approach can sufficiently benefit from the sequential access of storage devices. In practical scenarios, users are unaware of the layout of all accessed files, especially for aged storage systems, therefore, it is almost impossible to access all files in a totally sequential way for these users.

5.1.2 Write Performance. Figure 10 shows the execution times when using BFOw and the write() system call to write all buffered file data of different file sets to the three types of storage devices in different access orders. The sequential write is invoked when users write new files into unused disk areas, and the random write is used when users updates all files in-place. For these two

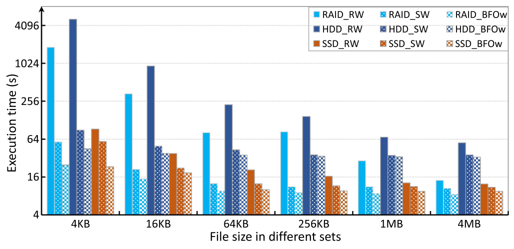


Fig. 10. The execution time of writing the different file sets as a function of file size. These file sets contain the same total amount of file data (i.e., 4 GB) but differ in file size (i.e., ranging from 4 KB to 4 MB). We use BFOw and the single-file access approach to write these file sets onto three types of the storage devices (i.e., RAID, HDD, and SSD) with different access orders (RW: Random Write, SW: Sequential Write). The bars of each different color represent the write performances of different approaches with the different storage devices. The y-axis is in log scale.

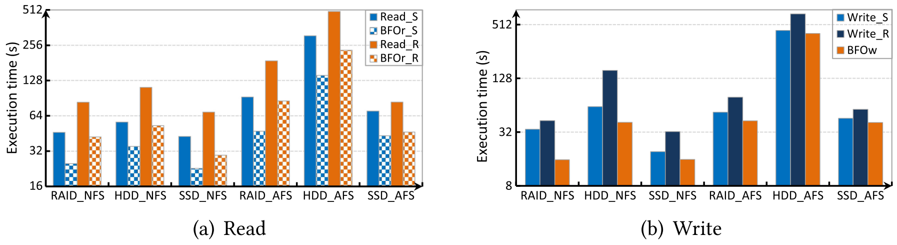


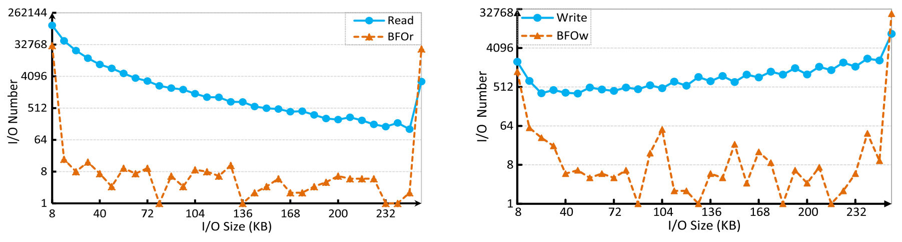
Fig. 11. The execution time of accessing a real-world file set from different storage devices with different access orders. Read/Write)_S(R) denotes the traditional read approach reading (or writing) data in sequential (or random) workload. BFOr_S(R) denotes BFOr accessing the data in sequential (or random) order. RAID(HDD/SSD)_NFS(AFS) denotes different storage devices (i.e., RAID, HDD, and SSD) with a New (or Age) File System.

write patterns, BFOw simply writes all file data into the newly allocated free space, and updates the metadata for all files. Therefore, for BFOw, we do not distinguish between the sequential and random orders in the experiments. BFOw also exhibits consistently higher write performance in all cases. When writing all 4 KB files randomly, BFOw outperforms the traditional write approach by up to 71.8×, 111.4× and 2.9× on RAID, HDD, and SSD, respectively. This is mainly because the traditional write approach needs to locate each file by retrieving its metadata first, and then update the file data and file metadata, which leads to a large number of random writes. BFOw can write all file data sequentially with a write request, and update all inodes at once. Even for the sequential write pattern, the performance improvement of BFOw is still significant (up to 1.32×). Furthermore, with the increase of file size, BFOw still outperforms the traditional write approach, even though the performance of traditional approach is close to that of BFOw.

Therefore, BFO demonstrates consistent performance gains, both batch read and batch write, over the traditional approaches, regardless of access patterns, data layouts, and storage media.

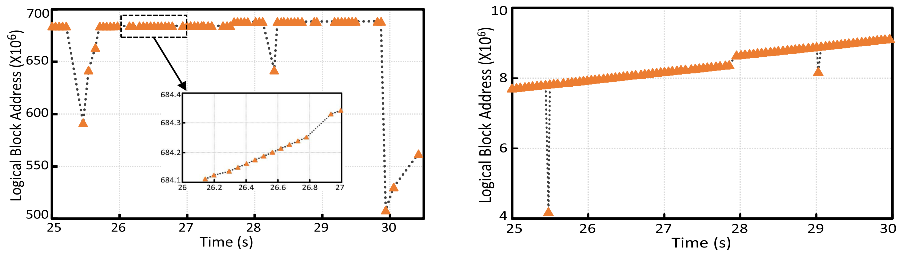
5.1.3 I/O Behavior. In this subsection, we further observe the detailed I/O behaviors of BFOr and BFOw with the real-world Linux-kernel-source-code file set to pinpoint the root causes of the performance improvements. We run the tests in a new ext4 file system, and an aged ext4 file system on different storage devices (i.e., RAID, HDD, and SSD).

Figure 11(a) shows the execution times for reading the real-world file set into memory. In the new file system, the file set is stored contiguously on the storage devices. The aged file system has



(a) Read (b) Write

Fig. 12. The distribution of the I/O sizes when using BFO and the single-file access approach in a new ext4 file system when accessing the Linux-kernel-source-code file set sequentially. We collect the blktraces during accessing the whole file set, and analyze the I/O sizes according to the blktraces. Both approaches access the same sized data, but with different distributions of the I/O sizes.



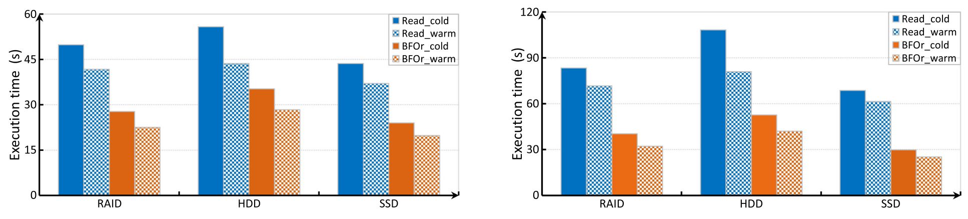
(a) Read (b) Write

Fig. 13. Access behaviors when using BFO to read/write the file set in sequential order with a 5-second window.

about 5 GB of free storage space to place the 3.1 GB-sized file set, and the free space contains many dispersed and small fragments throughout disks, therefore these files are laid out discontiguously in the aged file system. As can be seen, the results share similar trends to previous experiments. BFOr exhibits the higher read performance in all cases, and can reduce the execution time by up to 47.1% compared to the single-file access mode on ext4. For the new file system, BFOr can reap substantial performance improvements by sorting and merging small I/Os to access all data as sequentially as possible with fewer I/Os. For the aged file system, even though the discontiguous data layout hinders the merge operations, BFOr can still sort the small I/Os to access them sequentially. Even for SSD, the access policy can approximately double the overall performance in the best case. Moreover, for sequential read, although the traditional single-file access approach can leverage the prefetching mechanism and the I/O scheduler in the Linux kernel to fetch several accessed files each time, especially for the new file systems, the performance improvement contributed by these techniques is limited, and BFOr achieves up to 1.44× speedup when accessing files sequentially.

To better understand the above results, we take a closer look at the distribution of the read I/O sizes in Figure 12(a) when fetching the file set sequentially in the new file system. Both read approaches will access the whole file set, so the total sizes of the accessed data are the same. However, as the figure shows, the number of total I/Os and the number of small I/Os of BFOr are both significantly lower than that of the traditional read approach, and most of the accesses for small files are combined into larger I/Os (256 KB). Moreover, these files are mostly accessed sequentially as shown in Figure 13(a).

Furthermore, for the write process, Figure 11(b) shows the execution times when using BFOw and the write() system call to write the Linux-kernel-source-code file set. For the new file system,



(a) Sequential access (b) Random Access

Fig. 14. The execution time of reading the Linux-kernel-source-code file set in a new file system on three types of the storage devices (i.e., RAID, HDD, and SSD in x-axis). We use BFO and the single-file access approach (i.e., BFOr and Read) with cold and warm caches (i.e., _cold and _warm) in different access orders (i.e., sequential and random).

we write the file set into a contiguous storage space. In the aged file system, the files are written into the dispersed fragments of the corresponding disks in sequential or random order. We observe that BFOw exhibits higher write performance in all cases (especially for the new file systems running on the HDD-based devices (i.e., RAID, and HDD)), and decreases the execution time by up to 52% under the sequential write pattern, and 74% under the random write pattern. For the aged file system, BFOw can sequentially write the file set into the free fragments, just like the traditional sequential write approach (Write_S in the figure), therefore exhibits similar performance to Write_S, but BFOw can slightly reduce the execution time due to updating inodes in batches.

Figure 12(b) plots the distribution of the write I/O sizes when writing the whole file set into a contiguous space in the new file system. Similar to the read case, when accessing the same file set, the traditional write approach results in a large number of small write operations, while BFOw can merge many small I/Os into fewer large ones, alleviating a performance penalty caused by the single-file mode. In addition, these write I/Os are sequential as shown in Figure 13(b).

The experiments reveal that the effectiveness of BFO comes from its optimization of storage I/Os in the batch-file access situations, making full use of the underlying storage capability.

5.2 Impact of Cache

We also evaluate the performance of BFOr with cold and warm caches. For warm cache, we allocate 1.5 GB of memory for read cache to buffer the 3.1 GB-sized file set. The results are shown in Figure 14. The execution times with cold cache are similar to that in Figure 11. While with warm cache, both BFOr and the traditional read approach can reduce the overall execution times by about 17%, since the accessed data (especially the metadata) might have been cached in memory. In addition, BFOr still can reduce the execution time by up to 55% compared to the single-file access approach in all cases. This is mainly because for most uncached data, BFOr can reap performance benefits by accessing these on-disk data as sequentially as possible.

5.3 Impact of the Batch Size

We investigate the effect of the batch size, i.e., the number of accessed files for each batch-file operation on the performance of BFO. The number of files varies from 128 to 131,072. We use a 4 GB file set with 1,000,000 4 KB fixed-size files. The HDD is used as the underlying storage device.

The results of BFOr are shown in Figure 15. The execution time of BFOr drops as the number of accessed files increases. For sequential access, BFOr can achieve the peak performance when reading 2,048 files each time, and can take full advantage of the sequential disk accesses. Moreover, significant performance improvement can still be achieved when accessing 128 files at once.

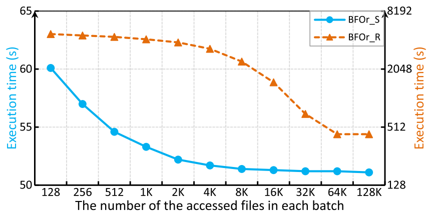


Fig. 15. The execution time of accessing a 4 GB file set with 4 KB fixed-size files as a function of number of files accessed by each BFOr operation. We set different numbers of files in the BFO interface and measure the overall execution time.

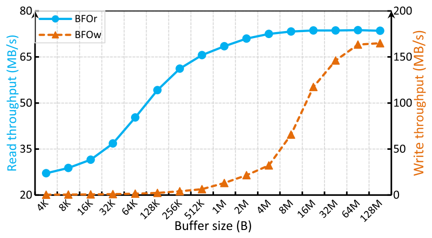


Fig. 16. The throughput of accessing files as a function of buffer size with BFO. We continually call the BFO operations to access these files, and measure the read and write throughputs.

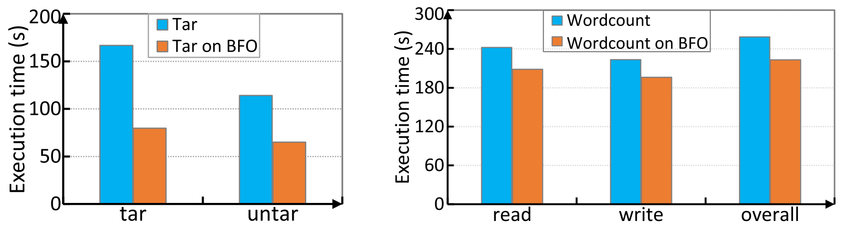
However, for random accesses, when the number reaches 65,536 (about 6.6% of total number of files), the highest performance is achieved for BFOr. This is because with the increasing number of files, BFOr can not only merge many contiguous I/Os but also reduce the number of the batch-file read operations. And for each BFOr operation, we need to access almost the entire 4 GB disk space for these files in a batch. Even though the BFOr operation is invoked to fetch 128 files each time, 105% performance improvement over the traditional read approach is achieved by BFOr.

For BFOw, when the total size of in-memory file data is above 256 KB, we can achieve its peak performance as shown in Figure 10.

5.4 Impact of Buffer Size

Next, we evaluate how much buffer space is appropriate for merging data, or for buffering the written data. The file set in this set of experiments is a series of small files, each of size 4 KB. The HDD is used as the underlying storage device. Figure 16 shows the throughput as a function of the buffer size. The read throughput of BFOr with a 128 KB buffer is two times higher than that with a 4 KB buffer. Considering that all files are of size 4 KB, a 4 KB buffer means that BFOr cannot merge any file. And with 2 MB buffer, BFOr exhibits a read throughput close to its maximum. With a buffer larger than 2 MB, the throughput of BFOr does not increase significantly, because the I/O size of the data has saturated the maximum I/O size of the storage device. However, the throughput cannot reach the peak I/O bandwidth of the device, because BFOr cannot merge all metadata I/Os. And some small but synchronized I/Os from other tasks have a side impact on the performance.

For write operations, because of the existence of the page cache, the file system itself absorbs and delays some small I/Os. To avoid the interference from the page cache, in this set of experiments, we employ the synchronized write operations. And when the buffer is full, we flush the buffered data onto the storage device forcefully. As Figure 16 shows, when the buffer size is below 256 KB, BFOw has very low write throughput, since a small quantity of (about 64) files are kept in the buffer, and can be merged to flush onto the storage device with a single I/O operation. After that,



(a) Tar (b) Wordcount

Fig. 17. Application performance with or without BFO. When testing the read performance, we write all output files to memory, to avoid the effect of the write process; similarly, when testing the write performance, we read all input files from memory.

the write performance increases more notably with the buffer size. When the buffer size reaches 64 MB, BFOw achieves the highest performance for writes. Therefore, even with a relatively small buffer (64 MB), BFOw can achieve the maximum read performance and write performance.

5.5 Applicability

To further demonstrate that the real-world applications can benefit from BFOr and BFOw, we choose two representative applications to run with and without BFO. These two applications are Linux Tar [15], which is a widely used tool to compress and decompress files, and Hadoop Wordcount [6], which is a typical MapReduce application that processes the text files.

Tar. The Linux Tar application is used to compress the on-disk Linux-kernel-source-code file set to a large in-memory file and then decompress the in-memory file to restore all files and write to the storage device. We measure the execution times of the above two phases with and without BFO as shown in Figure 17(a). Tar with BFO provides 109% and 75% higher performance than the one without BFO for the compression and decompression phases, respectively, because reading/writing all files when compressing/decompressing the file set is aggregated into large, sequential disk I/Os.

Wordcount. The Wordcount application is used to analyze a text file set obtained from Reference [39] containing 7,850 files with 195 MB in size at a single node, and it needs to fetch all files to analyze, and write the intermediate and final results into disk. We also write the output files into memory, or read the input files from memory when measuring the impact of the read or write process. Figure 17(b) shows the execution times for the Wordcount with or without BFO. For the application, the performance gain is relatively small (up to 16%) with BFO, this is because the Wordcount reads only a few megabytes of data into memory to process every second, and generates hundreds of files to store the intermediate results or final results. Such access patterns represent very light load for the Wordcount application, which explains why the application can benefit only marginally from BFO.

5.6 Batch-file Migration Performance

Finally, we evaluate the file migration performance by migrating the Linux-kernel-source-code file set between different storage volumes in both local and remote scenarios. Figure 18 depicts the results of BFOm, Linux *cp*, and *Rsync* [40] when migrating data locally with different storage volumes. Overall, *Rsync* and *cp* exhibit similar performance, since they do not optimize the read and write processes, and access all files with the single-file access mode, while BFOm can reduce the execution time by about 49.5% compared to the traditional tools. This is mainly because BFOm sorts and merges small I/Os, thereby reducing the number of I/Os when accessing files. When migrating within a storage device, our solution can also reduce the I/O contention. For the file

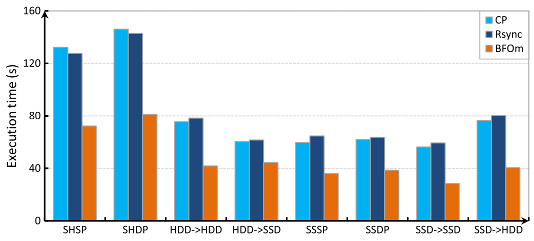


Fig. 18. The execution time of copying the Linux-kernel-source-code file set by different approaches (including cp, Rsync, and BFOm) with different storage devices. SHSP (SSSP) means that the files are migrated within the same partition of the same HDD (SSD), SHDP (SSDP) means that the files are migrated between the different partitions of the same HDD (SSD).

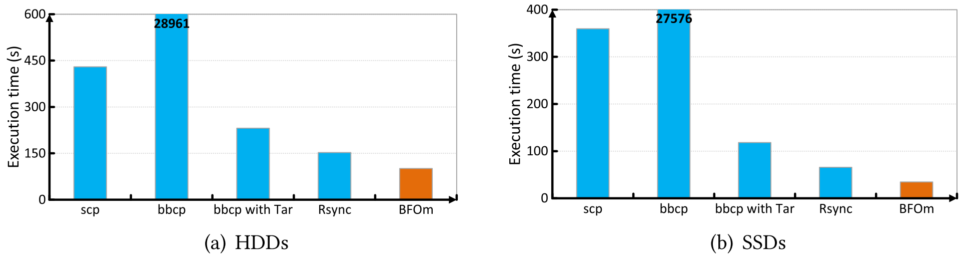


Fig. 19. The remote migration time between two nodes with different migration approaches. The Linux-kernel-source-code file set is migrated between two HDDs or two SSDs in different servers under a 1 Gbps LAN by scp, bbcp, bbcp with Tar, Rsync, and BFOm. bbcp with Tar denotes that we first archive the file set into a large file with Tar tool, then migrate the file by bbcp, and finally restore the file set at the destination node.

migration between different storage devices, we can efficiently read and write the data in parallel using BFOr and BFOw. For SSD devices, BFOm only exhibits a small performance improvement, because SSD offers fast random access without seek time, but BFOm still improves the file migration performance by reducing the number of I/Os.

Figure 19 shows the remote data migration performance. The file set is migrated between two HDDs or two SSDs in different servers under a 1 Gbps LAN. Compared to Linux *scp*, BFOm reduces the remote migration time by about 77% and 87% for HDDs and SSDs, respectively. *scp* uses the ping-pong migration approach, which merely accesses and moves one file at a time, are inefficient for small file migration. Even though SSD devices have higher access performance, the traditional migration approach still cannot exploit the great potential of the underlying devices, since the ping-pong migration procedure will hinder the access processes. *bbcp* is a very efficient tool to migrate large files. However, for the Linux-kernel-source-code file set, which contains a large amount of small files, *bbcp* takes the longest execution time (up to 8 hours) to migrate all files for both types of storage devices. This is because *bbcp* migrates a new file only when the source node receives an ack packet from the destination node. BFOm is highly optimized for small file migration. It collectively accesses and migrates the file set based on contiguous on-disk file segments, and finally stores all file data and metadata, respectively. Even though we use *tar* tool to pack the file set into a large file, then send the file by *bbcp*, and finally unpack the file set at the destination node, the total execution time is 1.31× and 2.42× higher than BFOm for HDDs and SSDs, respectively. The migration approach can reduce the network overhead by removing the ping-pong procedure, but it still uses the inefficient single-file mode to access the whole file set, and also incurs extra processing overhead to pack and unpack all files. Rsync provides high migration performance for small files, since it minimizes the network overhead by merging multiple files to transfer together. However, Rsync does not optimize the file accessing, so that the overall execution time is 60% higher for HDDs, and 89% higher for SSDs, compared to BFOm.

6 RELATED WORK

6.1 System-level Optimization

Current Linux operating systems have made some implicit efforts to improve the performance of accessing batched files. For the read operation, the prefetching mechanism [11, 20] uses a large I/O to read consecutive data likely belonging to multiple files at once. Nevertheless, the effectiveness of this approach heavily depends on the data layout and future access patterns. Incorrect prefetching can even harm the overall performance due to a waste of storage I/Os and memory cache. For the write operation, the page cache mechanism buffering new and updated data in memory can quickly acknowledge the file write requests, absorb multiple updates for the same pages, and periodically flush dirty pages. However, such implicit buffering can potentially compromise file persistency [10, 30], and it could be inefficient for a huge number of file writes due to limited capacity and passive flushing. And block-level I/O schedulers, such as CFQ [26] and Deadline [27], reorder and dispatch the requests to specific devices when accessing a batch of files, but they focus mainly on I/O priority and deadline, rather than the overall performance.

6.2 Dedicated File System

To improve the inefficient access modes for batched small files, emerging file systems leverage schemes that combine metadata of multiple files into a single unit of storage, thus reducing the number of metadata I/Os for small file accesses [8, 44, 47]. CFFS [47] introduces an internal physical representation to allow multiple small files to share a single inode, which enables users to access these file metadata with a single I/O. However, CFFS requires a new inode structure that is not easily extendable to other file systems. In parallel and distributed systems, the idea of consolidating metadata has also been explored, with solutions tailored for various homogeneous data types in different systems [8, 44].

Packing and storing files and metadata together is another effective way to improve I/O performance when writing a batch of small files. Btrfs [35] stores metadata of files and directories in copy-on-write B-trees. Small files are grouped into one or more fragments, which are then packed inside the B-trees. For small files, the fragments are indexed by object identifiers (analogous to inode numbers); the locality of a directory with multiple small files depends upon the proximity of the object identifiers. BetrFS [19, 45] stores metadata and data as key-value pairs in two Bε-trees. Nodes in a Bε-trees are large in size (2–4 MB), amortizing seek costs. Key-value pairs are packed within a node by sort-order, and these nodes are periodically rewritten, using copy-on-write, as changes are applied in batches. TableFS [34] uses LevelDB to store file-system metadata. It packs small files and metadata to a chunk in LevelDB. These approaches will provide a good write performance, but usually by sacrificing the read performance. More importantly, these solutions require the redesign of a file system with new data structures and layout to implement their access procedures, and cannot be ported or extended to existing popular file systems. In contrast, BFO is more portable and able to provide a good performance for small file accessed by reducing the number of disk I/Os without modifying the existing data layout, and thus without sacrificing any read performance.

6.3 Data Migration

Many prior studies have performed on the design and implementation of massive data migration frameworks [18, 22, 36]. *bbcp* [18] is an efficient data migration utility for moving large files securely. It uses a file-based I/O to access files, and its I/O bandwidth is limited by the stripe width of a file. XDD [36] optimizes the disk I/O performance, enabling file access with direct I/Os and multiple threads for parallelism to improve I/O access times. LADS [22] also uses multiple threads to fetch data from different disks, and proposes a congestion-aware I/O scheduling algorithm to increase the data processing rate per thread. These approaches are useful for moving large data faster and securely from source node to remote node over the network, but are not suitable for small files. And they do not fundamentally improve the inefficient ping-pong migration procedure.

Other works have focused on protocol-level optimizations. New protocols such as CUBIC TCP [17], Scalable TCP [21], and UDT [16] use a smaller multiplicative factor and/or a more aggressive additive factor to improve the data transfer rate. GridFTP [4] extends the standard File Transfer Protocol (FTP), and provides high speed, reliable, and secure data transfer. However, the main performance bottleneck of data migration for small files is the high cost of access operations at the storage ends.

7 CONCLUSION

In this article, we experimentally investigate the root cause of the inefficiency of the traditional single-file access mode for batched files. To solve the problem, we present a novel solution, BFO, for batch-file access, with optimized batch-file read (BFOr) and write (BFOw) operations for local file systems. BFO performs metadata and file data operations on all involved batched files separately in batches, eliminating unnecessary discontinuous I/O overhead. And based on BFOr and BFOw, we also propose the novel batch-file migration (BFOm) to accelerate the migration process for massive small files. We implement BFO in the ext4 file system as a case study, and show that BFO can improve the access and migration performances significantly, and remove a significant amount of random and non-sequential I/Os from the state-of-the-art techniques.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their valuable comments on the early draft of this article. Thanks to all the people who are fighting against COVID-19.

REFERENCES

[1] Vasily Tarasov and George Amvrosiadis. 2018. *Filebench*. Retrieved from http://sourceforge.net/projects/filebench/.
[2] Skyvia.com. 2018. *Skyvia*. Retrieved from https://skyvia.com/data-integration/synchronization.
[3] Alibaba. 2018. *TFS Project*. Retrieved from http://code.taobao.org/p/tfs/src/.
[4] William E. Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. 2005. The globus striped GridFTP framework and server. In *Proceedings of the ACM/IEEE Supercomputing Conference (SC'05)*. 54.
[5] Michael P. Andersen and David E. Culler. 2016. BTrDB: Optimizing storage system design for timeseries processing. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 39–52.
[6] Apache. 2018. *Hadoop*. Retrieved from http://hadoop.apache.org/.
[7] Jens Axboe. 2018. *Blktrace*. Retrieved from https://git.kernel.org/pub/scm/linux/kernel/git/axboe.
[8] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. 2010. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*. 47–60.
[9] Binfer. 2018. *High-speed File Transfer Software*. Retrieved from https://www.binfer.com/high-speed-file-transfer-software/.
[10] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 9.
[11] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. 2007. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *Proceedings of theUSENIX Annual Technical Conference*. USENIX, 261–274.

[12] John Esmet, Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. 2012. The TokuFS streaming file system. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'12).*

[13] Songling Fu, Ligang He, Chenlin Huang, Xiangke Liao, and Kenli Li. 2015. Performance optimization for managing massive numbers of small files in distributed file systems. *IEEE Trans. Parallel Distrib. Syst.* 26, 12 (2015), 3433–3448.

[14] GNU. 2018. *Linux scp*. Retrieved from http://www.gnu.org/software/coreutils/coreutils.html.

[15] GNU. 2018. *Linux tar*. Retrieved from http://www.gnu.org/software/coreutils/coreutils.html.

[16] Yunhong Gu and Robert L. Grossman. 2007. UDT: UDP-based data transfer for high-speed wide area networks. *Comput. Netw.* 51, 7 (2007), 1777–1799.

[17] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A new TCP-friendly high-speed TCP variant. *Operat. Syst. Rev.* 42, 5 (2008), 64–74.

[18] Andrew Hanushevsky. 2018. *BBCP*. Retrieved from http://www.slac.stanford.edu/ abh/bbcp/.

[19] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2015. BetrFS: A right-optimized write-optimized file system. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 301–315.

[20] Yongsoo Joo, Sangsoo Park, and Hyokyung Bahn. 2017. Exploiting I/O reordering and I/O interleaving to improve application launch performance. *Trans. Stor.* 13, 1 (2017), 8:1–8:17.

[21] Tom Kelly. 2003. Scalable TCP: Improving performance in highspeed wide area networks. *Comput. Commun. Rev.* 33, 2 (2003), 83–91.

[22] Youngjae Kim, Scott Atchley, Geoffroy Vallée, and Galen M. Shipman. 2015. LADS: Optimizing data transfers using layout-aware data scheduling. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 67–80.

[23] Changman Lee, Dongho Sim, Joo Young Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 273–286.

[24] Tan Li, Yufei Ren, Dantong Yu, and Shudong Jin. 2017. RAMSYS: Resource-aware asynchronous data transfer with multicore SYStems. *IEEE Trans. Parallel Distrib. Syst.* 28, 5 (2017), 1430–1444.

[25] Jie Liang, Yinlong Xu, Yongkun Li, and Yubiao Pan. 2017. ISM- An intra-stripe data migration approach for RAID-5 scaling. In *Proceedings of the International Conference on Networking, Architecture, and Storage (NAS'17)*. 1–10.

[26] LinuxKernel. 2018. *CFQ*. Retrieved from https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

[27] LinuxKernel. 2018. *Deadline*. Retrieved from https://www.kernel.org/doc/Documentation/block/deadline-iosched.txt.

[28] Yue Liu, Songlin Hu, Tilmann Rabl, Wantao Liu, Hans-Arno Jacobsen, Kaifeng Wu, Jian Chen, and Jintao Li. 2014. DGFIndex for smart grid: Enhancing hive with a cost-effective multidimensional range index. *Proc. VLDB Endow.* 7, 13 (2014), 1496–1507.

[29] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating keys from values in SSD-conscious storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 133–148.

[30] Marshall Kirk Mckusick and T. J. Kowalski. 2007. Fsck—The UNIX file system check program. Retrieved from https://dl.acm.org/doi/10.5555/107172.107210.

[31] Netapp. 2018. *Cloud Sync*. Retrieved from https://cloud.netapp.com/cloud-sync.

[32] Nexor. 2018. *Secure and Efficient Manual Release of Files Across Networks*. Retrieved from https://www.nexor.com/case-studies/files-transfer-secure-networks/.

[33] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application crash consistency and performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. 181–196.

[34] Kai Ren and Garth A. Gibson. 2013. TABLEFS: Enhancing metadata efficiency in the local file system. In *Proceedings of the USENIX Annual Technical Conference*. 145–156.

[35] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *ACM Trans. Storage* 9, 3 (2013), 1–32.

[36] Bradley W. Settlemyer, Jonathan D. Dobson, Stephen W. Hodson, Jeffery A. Kuehn, Stephen W. Poole, and Thomas Ruwart. 2011. A technique for moving large data sets over high-performance long distance networks. In *Proceedings of the IEEE Conference on Mass Storage Systems and Technologies (MSST'11)*. 1–6.

[37] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. 2012. WAN optimized replication of backup datasets using stream-informed delta compression. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 5.

[38] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference*. 1–14.

[39] Textfiles.com. 2018. *TextFiles*. Retrieved from http://www.textfiles.com/bbs/.

[40] Andrew Tridgell. 2018. *Rsync*. Retrieved from https://rsync.samba.org/.

[41] Stephen C. Tweedie. 2000. EXT3, journaling filesystem. In *Proceedings of the Ottowa Linux Symposium.*

[42] Wenrui Yan, Jie Yao, Qiang Cao, Changsheng Xie, and Hong Jiang. 2017. ROS: A rack-based optical storage system with inline accessibility for long-term data preservation. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17).* 161–174.

[43] Wangdong Yang, Kenli Li, and Keqin Li. 2019. A pipeline computing method of SpTV for three-order tensors on CPU and GPU. *Trans. Knowl. Discov. Data* 13, 6 (2019), 63:1–63:27.

[44] Weikuan Yu, Jeffrey S. Vetter, Shane Canon, and Song Jiang. 2007. Exploiting lustre file joining for effective collective IO. In *Proceedings of the 7th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID'07).* 267–274.

[45] Jun Yuan, Yang Zhan, William Jannen, Prashant Pandey, Amogh Akshintala, Kanchan Chandnani, Pooja Deo, Zardosht Kasheff, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. 2016. Optimizing every operation in a write-optimized file system. In *Proceedings of the USENIX Conference on File and Storage Technologies.* 1–14.

[46] Haoyu Zhang, Brian Cho, Ergin Seyfe, Avery Ching, and Michael J. Freedman. 2018. Riffle: Optimized shuffle service for large-scale data analytics. In *Proceedings of the 13th EuroSys Conference (EuroSys'18).* 43:1–43:15.

[47] Shuanglong Zhang, Helen Catanese, and An-I Andy Wang. 2016. The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance. In *Proceedings of the USENIX Conference on File and Storage Technologies.* 15–22.

Received September 2019; revised March 2020; accepted April 2020