BFO: Batch-File Operations on Massive Files for Consistent Performance Improvement

Yang Yang*, Qiang Cao*✉, Hong Jiang†, Li Yang*, Jie Yao*, Yuanyuan Dong‡, Puyuan Yang‡
*Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System of Ministry of Education,
School of Computer Science and Technology, Huazhong University of Science and Technology
†Department of Computer Science and Engineering, University of Texas at Arlington
‡Alibaba Group
✉Corresponding Author: caoqiang@hust.edu.cn

Abstract—**Existing local file systems, designed to support a typical *single-file* access pattern only, can lead to poor performance when accessing a batch of files, especially small files. This single-file pattern 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. 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. 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, with 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 boosts the write performance by up to 111.4× and 2.9× with HDD and SSD respectively. BFO also demonstrates consistent performance advantages when applied to four representative applications, Linux cp, Tar, GridFTP, and Hadoop.**

I. 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 applications, such as storage device upgrade and replacement [1] [2], data aggregation [3] [4], big-data analytics [5] and cloud computing [6], inter-cloud synchronization [7] [8], heavily depend on batch-file accesses and operations.

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 long-standing but not well resolved problem. Existing file systems, such as ext4 [9] and Btrfs [10], only provide the standard file-access system calls based on a single-file access (called *single-file*) pattern. With this pattern, 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 two separate I/Os, one for metadata and one for data. Therefore, reading/writing a file causes a traditional file system to be extremely inefficient due to such non-sequential and dependent small I/Os with high overhead [11] [12]. More importantly, accessing a batch of files, especially small files, with the single-file pattern can make things much worse because of traversing and reading/writing all files or directories involved one by one. In this paper, we focus on optimizing the file operations 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 alleviate the bottlenecks. Multi-threading is user-space solution with limited, if not negative effect, due to potential I/O contention [13] [14]. Prefetching [15] [16] can indirectly and implicitly improve the read performance only if the prefetched file will be accessed next in the buffer cache. On the other hand, 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 pattern cannot make full use of these new hardware (shown in the Section II). Other solutions [17] [18] 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 pattern, losing opportunities to improve the performance when accessing massive numbers of files.

Therefore, in this paper, 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 pattern 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, and 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. On the other hand, 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, BFO can be integrated into current file systems without modifying the latters data structures and existing procedures.

The major contributions of this work are:

1) We analyze the I/O behaviors when accessing a batch of files under different layouts and access orders through extensive experiments. We observe that the single-file pattern of traditional file systems requires the accesses to files to jump back and forth between metadata and data areas from one file to another, and these files are also accessed in a random order, resulting in a large number of small, non-sequential, and often dependent I/Os in the underlying storage device, degrading the overall access 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 have implemented a BFO prototype on ext4, one of the most popular file systems. Our evaluation results show that BFOs 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 apply BFO to boost the overall performance of real-world applications.

The rest of the paper is organized as follows. In Section II, we present the background and motivation for the BFO research. The design and implementation of BFO are detailed in Sections III and Sections IV respectively. We evaluate BFO in Section V. The related works are reviewed in Section VI. Finally, we conclude our work in Section VII.

II. Background and Motivation

A. Batch-File Access

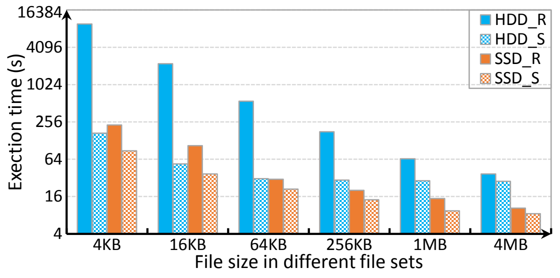
Emerging data-intensive applications in the big data era are rapidly transforming the data processing landscape. One of the prevalent trends is the increasing storage of and access to large-scale file sets. Enterprises require to backup considerable amounts of files from servers and desktops frequently. File-level data replication and data archiving also demand the copying and migration of massive numbers of files to ensure data availability [19] [20]. In big data analytics systems such as Hadoop and Spark, applications need to fetch a large number of files to process them in parallel, and create many files as either intermediate or final results to store [5] [6]. In IoT environments, a typical sensor system can collect a tremendous amount of sampling files from thousands of sensors with high sampling precisions and rates to edge-computing nodes. These files eventually swarm into clouded data centers [3] [4]. 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 [21] [22]. 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 pattern, called *single-file* access pattern, cannot fully utilize the potential capacity offered by the underlying storage systems, leading to subpar or even unpleasant user experiences when processing massive files, particularly small files [23] [24].

A large body of prior studies strive to overcome the inefficiency of batch-file processing, and can be roughly divided into four categories: application-level optimization, indirect system-level optimization, dedicated file-systems, and new hardware deployment. First, several application tools such as fastcopy [25] adopt multi-threading and larger buffer to accelerate batch-file copy while potentially leading to more random and contentious I/Os. Second, current Linux operating systems have provided several mechanisms to indirectly improve the performance of accessing batched files. The prefetching mechanisms [15] [16] use a large I/O to read consecutive data likely belonging to multiple files 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 space. The page cache mechanisms 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 potentially compromises file persistency [26] [27], and can be highly inefficient when the number of file writes is huge due to the limited capacity and passively flushing. And block-level I/O schedulers, such as CFQ [28] and Deadline [29], reorder and dispatch the I/O requests to specific devices by using scheduler queues. But they cannot change the serialized order of file-level accesses under the single-file access pattern. Third, several proposed solutions [17] [18] pack and store small files and metadata together to reduce I/O overhead for file accesses. However, these solutions have to redesign their own file systems with new data structures, disk layout, and software flow, which will not be easily portable to existing file systems. Finally, emerging solid-state drives with several orders of magnitude higher IOPS than traditional hard disks can significantly improve I/O performance. However, the batch-file accesses based on the single-file access pattern still cannot fully exploit the performance potentials of these new hardware.

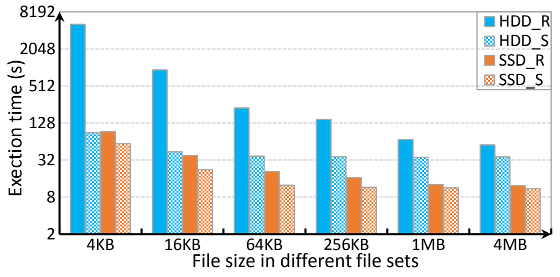
As a real-world actual example, a file set of the meteorological administration of Hubei Province of China, consists of 8,639,303 weather sampling files (about 1.5TB in total) collected from hundreds of locations in 5-years, and needs to be migrated from a source hard disk with NTFS to a target RAID 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.

B. Problem Analysis

The single-file access pattern, 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 pattern 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 access, this approach accumulates I/O overhead of each file, potentially leading to very low efficiency.



(a) Read



(b) Write

Fig. 1. The overall execution time of accessing different file sets on three storage devices with different access orders. The y-axis is in log scale.

1) Inefficiency: In order to experimentally explore the inefficiency of the single-file pattern 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 [30] to generate multiple file sets with the same total amount of data (i.e., 4GB) with different file sizes (i.e., from 4KB to 4MB) and file counts 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, users 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 for 4KB-sized files is up to 57.8× longer than the sequential under the same read case, 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 file set. On the other hand, we also observe in Figure 1(a) that the read performance of large-file set (i.e., 4MB-sized files) gradually reaches the peak performance of the storage devices, the performance of small-file set (i.e., below 1MB-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. 4KB) is significantly slower than the same case with large files (e.g. 4MB) by about 5×. Notice that they have the same consecutive file data layout, and it takes about extra 28 seconds to access inodes of 4KB-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 1MB) in a random manner, and can hardly make full use of the underlying devices.

2) Storage Behavior: In order to better understand the I/O behaviors under the single-file access pattern in typical file systems, we employ *blktrace* [31] to capture I/O footprints when accessing the Linux kernel source codes (ver 3.5.0) as a real file set.

Figure 2 and Figure 3 illustrate the read and write behaviors respectively during accessing the file set with three representative file systems, ext4 [9], Btrfs [10], and F2FS [32]. 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, 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 access metadata and data respectively. On the other hand, 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 [33], the write operation also invokes additional journaling

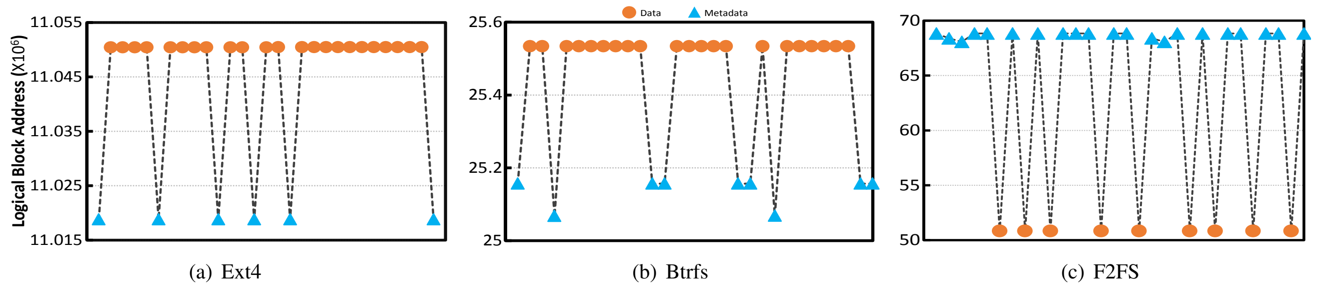


Fig. 2. File access behaviors for reading the Linux-kernel-source-code file set with three representative file systems in sequential manner. The metadata I/Os break contiguous I/Os into many non-sequential and random read I/Os.

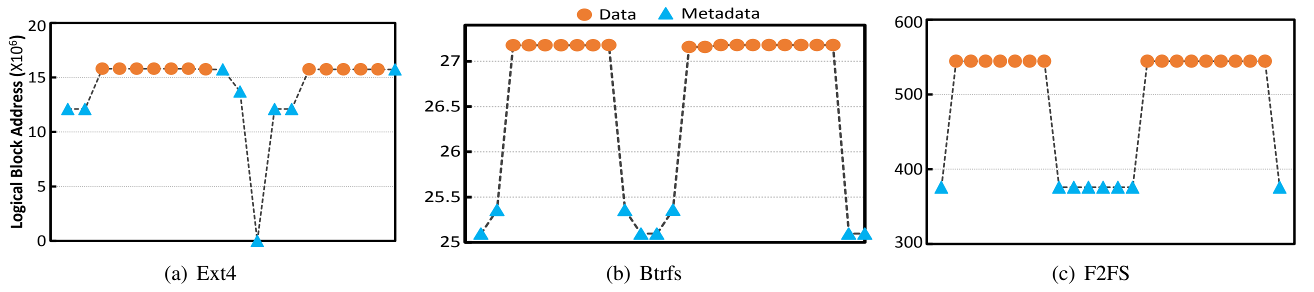


Fig. 3. File access behaviors when writing the Linux-kernel-source-code file set with three representative file systems in sequential manner. The metadata I/Os also break contiguous data write into multiple non-sequential I/Os.

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 leads to the back and forth seek operations 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.

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.**

C. Motivation

Current single-file access approaches are inefficient for accessing batched files, especially small files. Therefore, in order 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 when accessing batched files from the block devices.

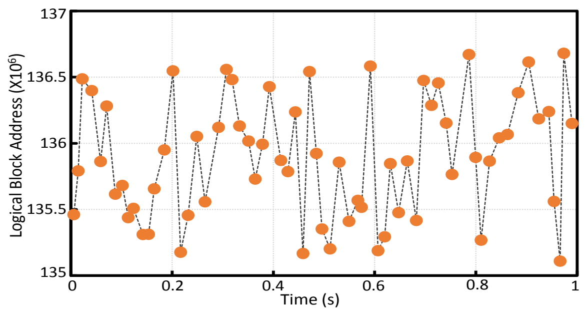


Fig. 4. Block trace of file data excluding the metadata with a 1s window when reading the Linux-kernel-source-code file set.

III. Design of BFO

To overcome the drawback of the single-file access pattern in existing file systems for efficiently processing batched files, we design the BFO approach, based on a set of effective Batch-File Operations. The key to BFO are two fundamental batch-file operations, the batch-file read operation BFOr and the batch-file write operation BFOw. 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 special cases of BFOw. Besides, BFOr and BFOw can be integrated together to accelerate the combined batch-file operations such as batch-file copy.

A. 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 pattern 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.

Batch-File Read: *Batchread(list<filename>, VolumeID)*: The *list<filename>* contains the file paths of all accessed files. 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 creates and writes all files and their corresponding inodes into to the target volume *VolumeID*.

B. BFOr

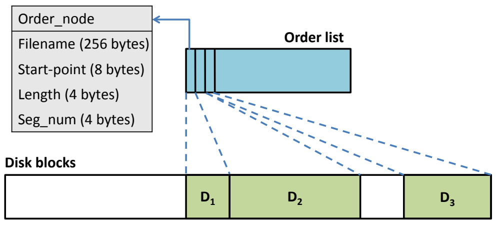


Fig. 5. The structure of the ORDER_LIST.

1) Two-Phase Read: For batch-file reads, in order 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 2MB space for inodes in a 128MB data group in ext4 [9]), 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.

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 penalty, as analyzed in Section II. 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.

To determine the data layout of all files, we design a data structure called ORDER_LIST as illustrated in Figure 5, 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 5 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 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 in order to fetch them with a single or a small number of I/O operations. 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. Besides, considering that the number of blocks the block device can handle at a time is limited, a long batch-file read process can potentially block the data-hungry applications, which wait for new on-disk data to analyze. As a result, BFO issues the block I/Os, either periodically (i.e., 100ms), or after batching a predetermined amount of read requests, and fetches these file data into memory for subsequent processing.

Algorithm 1 Batch-file read algorithm

```
Input: $fileset$: file names;
Output: $filedata$: in-memory file data of the file set;
1: Initialize inodes[], order_list, filedata;
2: for each batched files in the file set do
3:     i ← 0;
4:     clear order_list;
5:     for each file in a batch do
6:         inodes[i++] ← ReadInode(filename);
7:         order_node ← ParseInode(inode);
8:         insert order_node into order_list;
9:     end for
10:    SortAndMerge(order_list);
11:    filedata ← ReadData(order_list);
12: end for
13: return filedata;
```

C. BFOw

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 finally 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. In the second phase, BFOw updates the inode for each file in a batched manner 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 in order 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. 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 [34] 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 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.

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 128MB, we flush these data into an ext4 data group, which contains 128MB data blocks.

Algorithm 2 Batch-file write algorithm

```
Input: $filedata$: all files' data in a batch; $filename$: all files' paths;
1: Initialize address, steminode, inodes[];
2: steminode, order_list ← WriteFile(filedata);
3: address ← ParseInode(steminode);
4: for i=0; order_list[i]!=null; i++ do //update inodes
5:     inodes[i] ← ReadInode(order_list[i].filename);
6:     inodes[i].i_atime(i_mtime, i_ctime) ← steminode.i_atime(i_mtime, i_ctime);
7:     inodes[i].i_size ← order_list[i].length;
8:     inodes[i].address ← address;
9:     address ← address+order_list[i].length;
10: end for
```

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 [9] or copy-on-write (CoW) [10]. For example, in ordered journaling mode, ext4 journals metadata only, but all data is persisted prior to its metadata being committed to the journal. However, the phase of updating all inodes is vulnerable to file system corruption. Therefore, we design a light-weight consistency strategy 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.

IV. Implementation

In this section, we describe the implementation details of BFO on top of the ext4 file system. And the governing 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 6 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 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. Additionally, the Batch Module can also be implemented as a kernel module.

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, from the inode, by calling **filp_open, file_inode(), and ext_inode_hdr()**, and further create the address information using the starting addresses and lengths of these extents by calling **ext4_ext_pblock(extent), and ext4_ext_get_actual_len(extent)**, 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 all inodes of newly written files by invoking **filp_open(filename), filp_inode(), and ext4_raw_inode()**, and extracts all extent information (i.e., the starting addresses, lengths and offsets) from the stem inode into an *extent_item* as described above. Finally, we use this existing information to update these raw inodes as follows: **STEP 1:** update the file length of each file using **ext4_isize_set(raw_inode, order_list[i].size)**; **STEP 2:** update the time attributes of each inode by assigning the ones of the stem inode; **STEP 3:** update the *ext4_extent* structure of each file inode according to the *extent_item* and the ORDER_LIST. More specifically, we first update the extent structure of the VFS inode with *extent->ee_start = extent_item.startaddr, extent->ee_len = order_list[i].size, extent_item.startaddr = extent_item.startaddr + order_list[i].size*, and update the extent structure (i.e, i_block[]) of the *ext4_raw_inode* using the VFS extent information, and update the block count (i.e, i_blocks) of the file.

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. Notice that widely used existing file systems such as ext4 and Btrfs are indeed extremely complex and have evolved over a dozen years with high maturity. We do not alter their on-disk structure and software flow. BFO merely adds a set of new flows on top of these components, keeping their existing functionality and maturity.

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 design BFOcp 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.

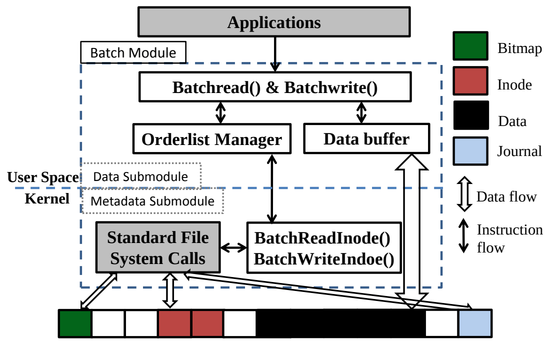


Fig. 6. Architecture of the batch module.

A. Applicability

When users need to access a large number of files explicitly, such as file backup and data synchronization, BFO can be invoked to process these files efficiently. Moreover, in some scenarios, applications are unaware of future access behaviors, for example Globus GridFTP [35] and Hadoop Wordcount application [36], and cannot directly use BFO interfaces when they have to access a batch of files within a certain period. Therefore, our proposed BFO idea can also be implemented as a file-level request scheduler in the kernel to implicitly optimize the batch-file access requests, without modifying the traditional POSIX interfaces in user applications. When users expect a higher access performance for a batch of files, rather than a single file, the BFO request scheduler can be used for them as a kernel module.

V. 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 @ 2.40GHz and 16GB RAM. The storage subsystem contains a RAID0 with 5 Western Digital 7200RPM 4TB SAS HDDs, a Western Digital 4TB SAS HDD, and a 480GB 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 4096 bytes and 256 bytes respectively. All experiments are conducted on a cold-cache basis.

We examine the performance of BFOr, BFOw separately in comparison with the single-file access approaches in ext4. We then explore the performance impact of BFO by varying the number of files for each batch-file operation and buffer size. Finally, we measure the performance of real-world applications using BFOr or BFOw.

A. Batch-file Access Performance

In this subsection, we first evaluate BFO in term of read and write performances compared with the single-file pattern under different conditions, including different file sizes (from 4KB to 4MB), 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., 4GB) but differ in file size (i.e., ranging from 4KB to 4MB). Then, we also analyze the I/O behaviors of BFO.

1) Read Performance: Figure 7 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 4KB 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 [37]. 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 4MB, BFO still has higher performance for random access. However, for sequential read, when the file size reaches 256KB, the execution time of traditional approach is close to that of BFOr, and can also 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.

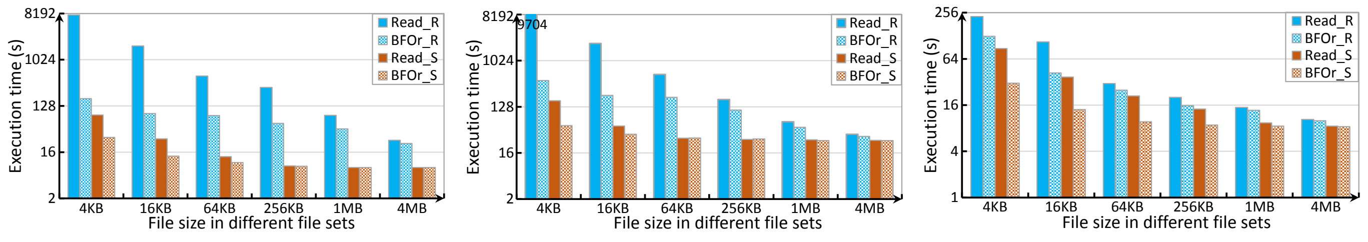
2) Write Performance: Figure 8 shows the execution times when using BFOw and the write() system call to write all buffered file data of different file sets to the three types of storage devices in different access orders. The sequential write is invoked when users write new files into unused disk areas, and the random write is used when users updates all files in-place. For these two 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 4KB 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.

3) I/O Behavior: In this subsection, we use a real file set to further observe the detailed I/O behaviors of BFO. To do so, we first duplicate the Linux source code (ver 3.5) six times as the real file set, called Linux-kernel-source-code file set, containing 233,988 files in 14,587 directories with 3.1GB in size.

Figure 9 shows the execution times for reading all files in the file set into memory. 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 pattern on ext4. 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, 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 results, we take a closer look at the distribution of the read I/O sizes when fetching the file set sequentially in Figure 11. As the figure shows, the number



(a) RAID (b) HDD (c) SSD

Fig. 7. The execution time of reading a batch of files as a function of file size on three types of the storage devices with different access orders. The y-axis is in log scale.

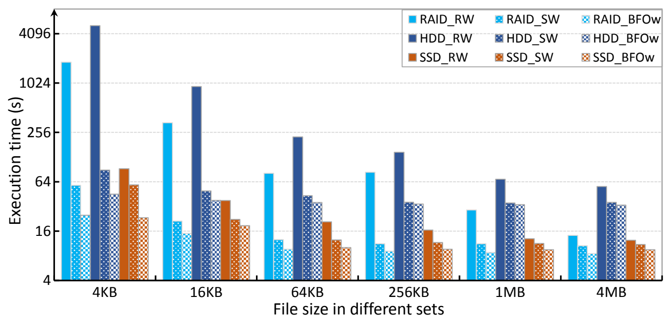


Fig. 8. The execution time of writing a batch of files as a function of file size on three types of the storage devices with different access orders. The bars of each different color represent the write performances of different approaches with the different storage devices. The y-axis is in log scale.

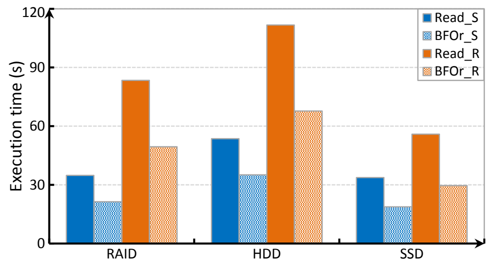


Fig. 9. The execution time of reading a file set from different storage devices with different access orders. Read_S(R) denotes the traditional read approach reading data in sequential (random) workload.

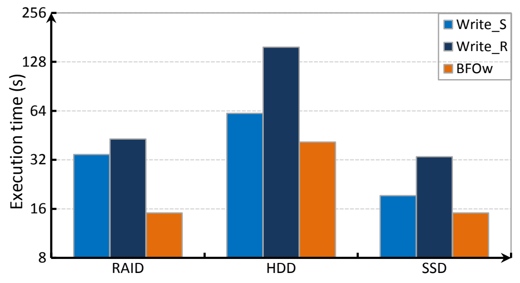


Fig. 10. The execution time of writing a file set into different storage devices with different access orders. Write_S(R) denotes the traditional write approach writing data in sequential (random) order. The y-axis is in log scale.

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

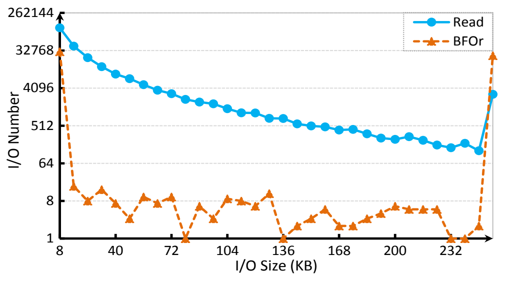


Fig. 11. The distribution of the read I/O sizes when using BFOr and the single-file read approach on ext4.

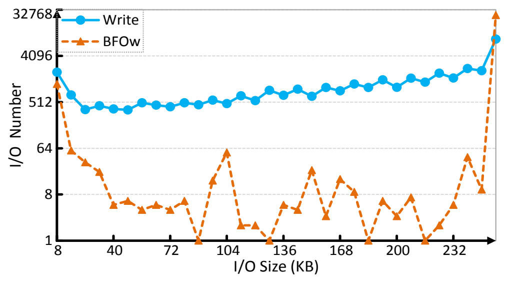


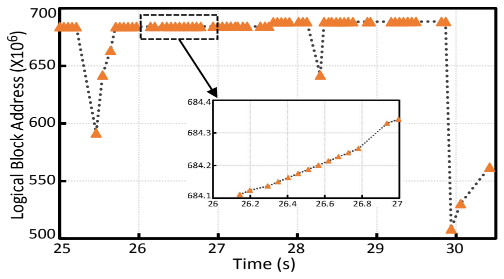
Fig. 12. The distribution of the write I/O sizes when using BFOw and the single-file write approach on ext4.

larger I/Os (256KB). Moreover, these files are mostly accessed sequentially as shown in Figure 13(a).

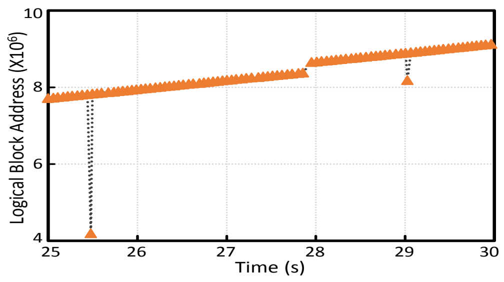
Furthermore, for the write process, Figure 10 shows the execution times when using BFOw and the write() system call to write the Linux-kernel-source-code file set. We observe that BFOw exhibits higher write performance in all cases, and decreases the execution time by up to 52% under the sequential write pattern, and 74% under the random write pattern, on all storage devices.

Figure 12 plots the distribution of the write I/O sizes. Similar to the read case, BFOw merges many small I/Os into fewer large ones, alleviating a performance penalty caused by the single-file pattern. 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.



(a) Read



(b) Write

Fig. 13. Block traces when using BFO to read/write the file set in sequential order with a 5s window.

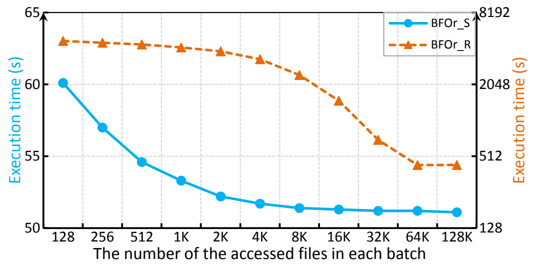


Fig. 14. The execution time of accessing a 4GB file set as a function of number of files accessed by each BFOr operation.

B. 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 131072. We use a 4GB file set with 1,000,000 4KB fixed-size files. The HDD is used as the underlying storage device.

The results of BFOr are shown in Figure 14. 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.

On the other hand, 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 4GB 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 256KB, we can achieve its peak performance as shown in Figure 8.

C. Impact of Buffer Size

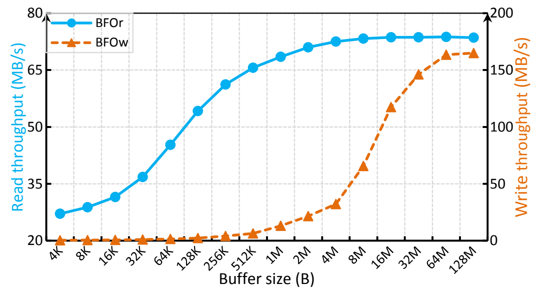


Fig. 15. The throughput of accessing a batch of files as a function of buffer size.

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 batch of small files, each of size 4KB. The HDD is used as the underlying storage device. Figure 15 shows the throughput as a function of the buffer size. The read throughput of BFOr with a 128KB buffer is two times higher than that with a 4KB buffer. Considering that all files are of size 4KB, a 4KB buffer means that BFOr cannot merge any file. And with 2MB buffer, BFOr exhibits a read throughput close to its maximum. With a buffer larger than 2MB, 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. In order 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 15 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, the write performance increases more notably with the buffer size. When the buffer size reaches 64MB, BFOw achieves the highest performance for writes. Therefore, even with a relatively small buffer (64MB), BFOw can achieve the maximum read performance and write performance.

D. Applicability

To further demonstrate that the real-world applications can benefit from BFO, we choose four representative applications

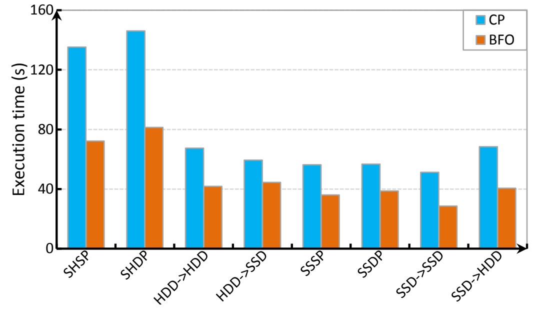


Fig. 16. The execution time of copying a file set with different storage devices. SHSP (SSSP) means that the files are migrated within the same partition of the same HDD (SSD), SHDP (SSDP) means that the files are migrated between the different partitions of the same HDD (SSD).

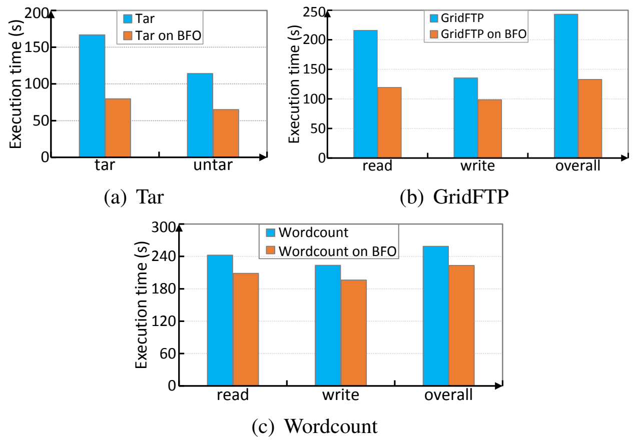


Fig. 17. Application performance with or without BFO. When testing the read performance, we write all output files to memory, in order to avoid the effect of the write process; similarly, when testing the write performance, we read all input files from memory.

to run with and without BFO. These four applications are Linux cp, which is frequently used to copy files locally, Linux Tar [38], which is a widely-used tool to compress and decompress files, Globus GridFTP [35], which is the most popular transfer tool in the BigSet scientific computing facility [39], and Hadoop Wordcount [36], which is a typical MapReduce application that processes the text files.

Batch-file Copy We re-implement the Linux cp (called BFOcp) based on BFO, and evaluate the batch-file copy performance with BFOcp and Linux cp by copying the Linux-kernel-source-code file set between the storage devices or the disk partitions of the same disk. BFOcp periodically launches BFOr and then BFOw to copy the file set. Figure 16 plots the execution times of BFOcp and Linux *cp* based on the single-file access pattern. Both BFOcp and cp access all files sequentially. Overall, BFOcp reduces the execution time by about 46.6% compared to the traditional *cp* tool. When copying within a storage device, our solution can not only improve the read and write performances respectively, but also reduce the I/O contention. For the file copy between different storage devices, we can efficiently read and write the data in parallel using BFOr and BFOw concurrently.

Based on the above results, we observe that Linux cp can benefit from BFO, regardless of the underlying storage devices. Next, we focus on the performance improvements of these applications brought by BFO, and only use the HDD as the storage device.

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.

GridFTP GripFTP is used to transfer the Linux-kernel-source-code file set. In order to avoid the impact of network transfer, the experiment reads the file set to memory as the client end, and writes all files to the destination disk as the server end. When measuring the performance impact of the read process, we transfer all on-disk files into memory to avoid the effect of the write process; similarly, when testing the impact of the write process, we transfer all input files from memory into the underlying HDD. The overall performance is tested when transferring the file set between two different HDDs locally in order to avoid the network overheads. The results are shown in Figure 17(b). GridFTP with BFO gains up to 81% performance improvement from the batch-file operations. When transferring the file set, GridFTP needs to take most of the execution time to read and write the file set. Therefore, BFO can support efficient file processing for GridFTP.

Wordcount The Wordcount application is used to analyze a text file set obtained from [40] containing 7,850 files with 195MB 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(c) 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.

VI. Related Work

A. 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 [15] [16] 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 [26] [27], 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 [28] and Deadline [29], 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.

B. Dedicated File System

To improve the inefficient access patterns 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 [18] [21] [41]. CFFS [18] 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 [21] [41].

Packing and storing files and metadata together is another effective way to improve I/O performance when writing a batch of small files. Btrfs [10] 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 [42] [43] 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 [17] 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 can not 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.

VII. Conclusion

In this paper, we experimentally investigate the root cause of the inefficiency of the traditional single-file access pattern 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. We implement BFO in the ext4 file system as a case study, and show that BFO can improve the access performance significantly, and remove a significant amount of random and non-sequential I/Os from the state-of-the-art techniques.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments. This work is supported in part by NSFC No. 61872156, Creative Research Group Project of NSFC No.61821003, the Fundamental Research Funds for the Central Universities No. 2018KFYXKJC037, the US NSF under Grant No.CCF-1704504 and No.CCF-1629625, and Alibaba Group through Alibaba Innovative Research (AIR) Program.

References

[1] J. Liang, Y. Xu, Y. Li, and Y. Pan, "ISM- an intra-stripe data migration approach for RAID-5 scaling," in *2017 International Conference on Networking, Architecture, and Storage, NAS 2017, Shenzhen, China, August 7-9, 2017*, 2017, pp. 1–10.

[2] W. Yan, J. Yao, Q. Cao, C. Xie, and H. Jiang, "ROS: A rack-based optical storage system with inline accessibility for long-term data preservation," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pp. 161–174.

[3] M. P. Andersen and D. E. Culler, "Btrdb: Optimizing storage system design for timeseries processing," in *Proceedings of the 2016 USENIX Conference on File and Storage Technologies*, pp. 39–52.

[4] Y. Liu, S. Hu, T. Rabl, W. Liu, H. Jacobsen, K. Wu, J. Chen, and J. Li, "Dgfindex for smart grid: Enhancing hive with a cost-effective multidimensional range index," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1496–1507, 2014.

[5] H. Zhang, B. Cho, E. Seyfe, A. Ching, and M. J. Freedman, "Riffle: optimized shuffle service for large-scale data analytics," in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, 2018, pp. 43:1–43:15.

[6] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: a framework for large scale data processing," in *ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012*, 2012, p. 4.

[7] *Skyvia*, 2018, https://skyvia.com/data-integration/synchronization.

[8] Netapp, *Cloud Sync*, 2018, https://cloud.netapp.com/cloud-sync.

[9] S. C. Tweedie, "Ext3, journaling filesystem," in *InOttowa Linux Symposium, Ottowa, ON, Canada, July 20, 2000.*

[10] O. Rodeh, J. Bacik, and C. Mason, "Btrfs:the linux b-tree filesystem," *Acm Transactions on Storage*, vol. 9, no. 3, pp. 1–32, 2013.

[11] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul, "The tokufs streaming file system," in *4th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage'12, Boston, MA, USA, June 13-14, 2012*, 2012.

[12] S. Fu, L. He, C. Huang, X. Liao, and K. Li, "Performance optimization for managing massive numbers of small files in distributed file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 12, pp. 3433–3448, 2015.

[13] Y. Kim, S. Atchley, G. Vallée, and G. M. Shipman, "LADS: optimizing data transfers using layout-aware data scheduling," in *Proceedings of the 2015 USENIX Conference on File and Storage Technologies*, pp. 67–80.

[14] T. Li, Y. Ren, D. Yu, and S. Jin, "RAMSYS: resource-aware asynchronous data transfer with multicore systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 5, pp. 1430–1444, 2017.

[15] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "Diskseen: Exploiting disk layout and access history to enhance I/O prefetch," in *Proceedings of the 2007 USENIX Annual Technical Conference, Santa Clara, CA, USA, June 17-22, 2007*.

[16] Y. Joo, S. Park, and H. Bahn, "Exploiting I/O reordering and I/O interleaving to improve application launch performance," *TOS*, vol. 13, no. 1, pp. 8:1–8:17, 2017.

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

[18] S. Zhang, H. Catanese, and A. A. Wang, "The composite-file file system: Decoupling the one-to-one mapping of files and metadata for better performance," in *Proceedings of the 2016 USENIX Conference on File and Storage Technologies*, pp. 15–22.

[19] B. W. Settlemyer, J. D. Dobson, S. W. Hodson, J. A. Kuehn, S. W. Poole, and T. Ruwart, "A technique for moving large data sets over high-performance long distance networks," in *MSST, 2011*, pp. 1–6.

[20] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "WAN optimized replication of backup datasets using stream-informed delta compression," in *Proceedings of the 2012 USENIX Conference on File and Storage Technologies*, p. 5.

[21] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage," in *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2010,*, pp. 47–60.

[22] Alibaba, 2018, http://code.taobao.org/p/tfs/src/,.

[23] Binfer, 2018, https://www.binfer.com/high-speed-file-transfer-software/,.

[24] Nexor, 2018, https://www.nexor.com/case-studies/files-transfer-secure-networks/.

[25] Shirouzu, *FastCopy*, 2018, https://ipmsg.org/tools/fastcopy.html.

[26] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering," in *Proceedings of the 2012 USENIX Conference on File and Storage Technologies*, p. 9.

[27] M. K. Mckusick and T. J. Kowalski, "Fsck - the unix file system check program," 2007.

[28] Kernel, *CFQ*, 2018, https://www.kernel.org/doc/Documentation/block/cfq-iosched.txt.

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

[30] *Filebench*, 2018, http://sourceforge.net/projects/filebench/.

[31] J. Axboe, *Blktrace*, 2018, https://git.kernel.org/pub/scm/linux/kernel/git/axboe.

[32] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proceedings of the 2015 USENIX Conference on File and Storage Technologies*, pp. 273–286.

[33] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system," in *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996*, 1996, pp. 1–14.

[34] T. S. Pillai, R. Alagappan, L. Lu, V. Chidambaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Application crash consistency and performance with CCFS," in *15th USENIX Conference on File and Storage Technologies, FAST 2017, Santa Clara, CA, USA, February 27 - March 2, 2017*, 2017, pp. 181–196.

[35] W. E. Allcock, J. Bresnahan, R. Kettimuthu, and M. Link, "The globus striped gridftp framework and server," in *SC, 2005*, p. 54.

[36] Apache, *Hadoop*, 2018, http://hadoop.apache.org/.

[37] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in ssd-conscious storage," in *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.*, 2016, pp. 133–148.

[38] GNU, *Linux tar*, 2018, http://www.gnu.org/software/coreutils/coreutils.html.

[39] Z. Liu, R. Kettimuthu, I. T. Foster, and Y. Liu, "A comprehensive study of wide area data movement at a scientific computing facility," in *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, 2018, pp. 1604–1611.

[40] textfiles.com, *TextFiles*, 2018, http://www.textfiles.com/bbs/.

[41] W. Yu, J. S. Vetter, S. Canon, and S. Jiang, "Exploiting lustre file joining for effective collective IO," in *7th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2007*, pp. 267–274.

[42] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter, "Betrfs: A right-optimized write-optimized file system," in *Proceedings of the 2015 USENIX Conference on File and Storage Technologies*, pp. 301–315.

[43] J. Yuan, Y. Zhan, W. Jannen, P. Pandey, A. Akshintala, K. Chandnani, P. Deo, Z. Kasheff, L. Walsh, M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, and D. E. Porter, "Optimizing every operation in a write-optimized file system," in *Proceedings of the 2016 USENIX Conference on File and Storage Technologies*, pp. 1–14.