MiF: Mitigating the intra-file Fragmentation in parallel file system

Letian Yi, Jiwu Shu, Youyou Lu, Wei Wang, Weimin Zheng
Department of Computer Science and Technology,
Tsinghua University, Beijing, China
The State Key Laboratory of High-end Server and Storage Technology,
Inspur, Jinan, China
lonat.front@gmail.com, shujw@tsinghua.edu.cn, luyouyou87@gmail.com, zwm-dcs@tsinghua.edu.cn

***Abstract*—parallel file systems have been broadly deployed in large scale data centers, supporting a wide range of applications across a variety of industries. Unfortunately, most parallel file systems suffer from the intra-file fragmentation which is the disk performance killer. This paper presents the design and implementation of *MiF*, which introduces two techniques: *on-demand preallocation* and *embedded directory*, to *Mitigate the intra-file Fragmentation* of data placement, improving the disk performance in parallel file system. The key insight of *on-demand preallocation* is that the preallocation of a file should be aware of concurrent process streams and recognizes the write characteristic. The background rationale of *embedded directory* is that, since modern parallel file systems aggregate many normal operation pairs, exploring the disk bandwidth for metadata access requires all metadata of sub-files in the same directory be placed adjacently. Measurements of our MiF implementation in a block-based parallel file system demonstrate that it can significantly improve I/O performance of parallel programs.**

***Keywords*-parallel file system; preallocation; embedded directory; fragmentation;**

I. Introduction

Improving throughput is the premier goal for all parallel file systems. However, if a file (directory) is fragmented inside, even sequential access to it is interleaved and the disk head has to move back and forth constantly among the different regions, resulting in poor overall throughput. This inside fragmenting in an individual file is termed *intra-file fragmentation*. Designers of parallel file systems have not been idle and attempted to reduce the fragmentation in IO servers by improving block allocation techniques of both normal files and directories.

To place a normal file contiguously on disk, one approach often used is to anticipate the files likely needing more blocks and perallocate them in advance [3][11][5][10][25]. Its core idea is that, for every file that is being extended, allocator[1] reserves a range of on-disk blocks near the last *non-hole* block of the file for it. Blocks needed by subsequent write (extend) operations for that inode are allocated from that range, instead of from the whole file system. Since no other inode is allowed to allocate blocks in the reservation range, it mitigates the *inter-file* fragmentation of the file systems. Also, if the subsequent workload is dominated by sequential write, this reservation ensures the contiguous placement.

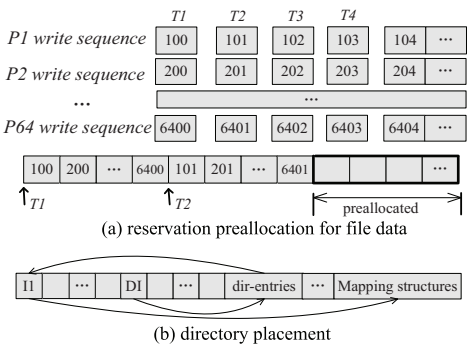


Figure 1. File and directory placement in modern parallel file systems. In (a), to simplify the presentation, we assume that the request size from each client is one block. *Tn* indicates the timestamps of write request from multiple streams (indicated by *Pn*, e.g., process on the cluster nodes) and the logic offset in the file to be written is sorted by process (write sequence). For example, at *T1*, *P1*, *P2* and *P64* writes file logic number of *100*, *200* and *6400* respectively. In (b), *DI* and *I* indicates the directory inode and file inode respectively.

However, this reservation can not reduce the fragmentation inside an individual file which is written by multiple processes concurrently. Figure 1(a) illustrates an example. In this figure, the file is concurrently accessed by 64 processes and the allocator reserves contiguous on-disk blocks for it. At *Tn*, all processes write one block into the uninitiated file region and these blocks are placed in the reserved space in the order of arrival time. As a result, the indirection from logic block to the physical block is fragmented and subsequent (even sequential) access to this file incurs a mass of disk head interference. Our experiments (in section 5), based on the trace analysis of scientific computing environment from previous study [16], show that these interference can reduce over 40% IO performance. To cope with this problem, recent efforts in file systems provide the *fallocate* syscall which persistently allocates all blocks for the file [9]. Nevertheless, it requires an application to have sufficient foreknowledge of how much space the file will need.

[1]We use the terms *allocator* to refer the component which manages the free blocks on disks. In some parallel file systems, allocator is located in their IO servers

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.46

On the side of metadata, modern parallel file systems co-locate related data objects (e.g., an inode and the data blocks) near each other on disk. However, since the blocks of directory entry are often separated from the file inode blocks [11][5][10][25], the directories are also fragmented inside intrinsically. As described in the example illustrated in figure 1(b), performing *fstat* operations should query the block of entry blocks inside its parent directory and then the inode block in order. In the applications with intensive metadata access, overall system performance can be limited by disk access times of metadata operation rather than disk bandwidth of metadata transport.

In this paper, we present the design and implementation of *MiF*, which introduces two techniques: *on-demand pre-allocation* and *embedded directory*, to *Mitigate the intra-file Fragmentation* and improve the disk performance in parallel file system. The key insight of on-demand preallocation is that the preallocation of a file should be aware of concurrent process streams and recognizes the write characteristic. With the on-demand preallocation, the file allocator initiates a allocation window for every write stream. The window contains some preallocated contiguous blocks which are persistent across reboots. It then predicts the future write size based on access history. Therefore, when the file is written by multiple processes concurrently, the blocks of every region of file can be kept contiguous, improving the file data placement.

The background rationale of embedded directory is that, since modern parallel file systems aggregate many normal operation pairs [1][5][3][4], such as *open-getlayout* and *readdir-stats*, exploring the disk bandwidth for metadata access requires all metadata of subfiles in the same directory be placed adjacently. Rather than storing the inode in a separated blocks, the inode is allocated from the blocks of directory content and the blocks storing directory entries are omitted from on-disk layout. Then, all layouts which map file logic offsets to the on-disk blocks are stuffed into the tail of file inode (or the block contiguous to the inode block if the mapping structure is too large). Therefore, the number of the disk positioning times can be decreased in most metadata access.

We have constructed an implementation that includes both of the techniques in our block-based parallel file system, named Redbud. Our experiments with representative benchmarks compare our implementation to two baseline systems: Lustre file system and our original Redbud version, which both use the traditional data placement algorithms. Measurements of our experiments show that, for shared file activity, on-demand preallocation algorithm mitigates the intra-file fragmentation in file system and thus incurs less metadata cost. This reduction translates into a performance increase of parallel programs: for example, the I/O throughput of *NPB BTIO* program can be increased by 19% as compared to that without using collective I/O. Embedded directory algorithm effectively reduces the number of disk positioning times, leading to 23%-170% improvement of metadata access throughput.

The rest of this paper is organized as follows: Section 2 provides background and related work. Section 3 and section 4 describes the design and algorithms of on-demand preallocation and embedded directory, respectively. In Section 5, we present the implementation of both techniques and evaluation results of our experiments. Finally, Section 5 summarizes our work.

II. Background and related work

In this section, we first provide some background and analysis on background that motivates our work. Then we examine the related work of mitigating fragmentation in parallel file systems.

A. Background

1) concurrent data access: Several studies on workload characterizations of realistic scientific environment have shown that strong file sharing among processes within a job is a common property [6][7][8][16]. For example, *Abaqus* application [26] for analysis of tectonic data when running on a cluster, requires all nodes to frequently read and write different regions of the same file which is suffixed with *.odb* (storing intermediate result). Analysis on trace data collected from the Cluster in Lawrence Livermore National Laboratory also shows that, in a typical physics simulation, a set of nodes frequently write collected data to a shared file, which will be used for further analysis [16].

If the space allocation algorithms of current file systems are not aware of extending operations on file sharing from multiple processes, the indirection from file logic address to physical disk block number is fragmented. Since the file data mapping would not be changed before file deletion in file systems, subsequent access to the shared file for computation or analysis incurs intra-file interference which is the disk performance killer. Also, increased metadata overhead of high fragmentation rate causes less efficient mapping. By running the same benchmark on different file models in the parallel file systems, Wang [16] found that the throughput of using an individual output file for each node exceeds that of using a shared file for all nodes by a factor of 5. Therefore, it is reasonable for allocation in parallel file systems to be well optimized for multiple concurrent streams.

2) aggregated metadata operation: Modern parallel file systems optimize most common metadata access scenarios by aggregating the operation pairs. These aggregations can be very successful at reducing communication overhead by decreasing the number of requests to metadata server (MDS). For example, since a *readdir* followed by a stat of each file (e. g., *ls* -l) is an common access pattern, a *readdirplus* [12] extension is proposed and supported by

most parallel file systems [5][3][4] to fetch the entire directory, including inode contents, in a single MDS request. By aggregating the *open-getlayout* operation, the pNFS protocol and the Lustre both allows their clients to acquire the file layout (*extent mapping* in pNFS in block-mode and *object id* in Lustre) on opening files.

Obviously, if all metadata to be accessed in an aggregated operation is placed contiguously on disk, there is opportunity to access disks in an efficient fashion, improving the disk performance of the metadata access. Unfortunately, most recent parallel file system failed to exploit it. For example, directories in Panasas file system are special files that store an array of directory entries and its file metadata (inode) is stored in object attributes on two of the N objects used to store the file's data. Therefore, performing a *readdirplus* operation involves at least three disk position time.

B. Related work of improving data placement

Most parallel file systems opt to utilize the local file system to manage their disk layout[31][5][10]. These local file systems try to mitigate the intra-file fragmentation by co-locating related data in the same cylinder group. pNFS in block-mode of Linux implementation [4] exports local file system (EXT4) to achieve parallel access based on standard NFSv4.1 protocol. CXFS [23] is a cluster version of XFS that allows multiple nodes to access data on shared disks in an XFS file system. Their preallocation algorithms are similar to the reservation approach as mentioned above, reserving blocks for the subsequent writes. Delayed allocation is also proposed in these file systems to postpone allocation to page flush time, rather than during the *write()* operation [23]. This method provides the opportunity to combine many block allocation requests into a single request, reducing possible fragmentation and saving CPU cycles. However, it assumes the data can be buffered in the memory for a long time, thus do not fit application with explicit sync requests well. Actually, since on-demand preallocation can improve data placement on concurrent access without any runtime assumption, it can be view as the complementarity of delayed allocation and *fallocate* system call which is used for the case of foreknowing the file size.

The idea of embedding inode in the directory content is not original: indeed, C-FFS [20] utilizes it to improve small file access performance and Ceph [3] uses it to allow the MDS to prefetch entire directories with a single OSD read request. This embedding works well in Ceph which has avoided any need for file mapping. However, by also stuffing the file mapping in the directory content, our work on embedded directory seeks a more general approach to target most parallel file systems which explicitly use file mapping, including both extent mapping in block-based file systems and object id in object-based file systems. For normal file blocks allocation, Ceph borrows the idea from LFS [19]. The object storage servers in Ceph file system aggressively perform copy-on-write: with the exception of superblock updates, data is always written to unallocated regions of disk. Assuming that free extents of disk blocks are always available, this approach works extremely well for write activity. Unfortunately, previous study have all indicated that the performance of read traffic can be compromised in many cases [21].

Recent researches also propose data replication to improve disk performance of systems, and some of them can mitigate the fragmentation. They reorganize data layout on a disk or replicate data , such as BORG [13] and FS2[14], within the same disk according to detected access patterns. Zhang [15] proposed to remove interference by replicating data in IO servers of parallel file systems. Since replication is not free at runtime, false predication of last IO timing still lead to the severe intra-file interference using these approaches.

III. On-demand preallocation

The goal of on-demand preallocation algorithm is to place file data as more as contiguously in parallel file systems. To achieve this, our on-demand preallocation should effectively support arbitrary number of concurrent write streams to write the same file. Also, the preallocation for each stream must be in on-demand fashion: more contiguous on-disk blocks for sequential workload and less for random scenario. In this section, we describe the algorithm details of this technique.

A. Core data structure

To track all the workloads on shared files from multiple nodes, on-demand preallocation first requires the file space allocator to identify different streams. It is straightforward for recent parallel file systems to achieve it. For example, in most file systems, the clients are identified by a unique ID for both maintaining global status and recovering system. Therefore, file allocator can distinguish the write streams using *stream ID*, which is constructed by combining the client ID and the thread PID on client.

The file allocator performs preallocation for every process stream which extends shared files. The core data structures for preallocation are *current window* and *sequential window*. Both windows have three components, a disk block number, a file logic block number and length. Like traditional reservation, *current window* contains the blocks which have been persistently preallocated to stream. The *sequential window*, on the other hand, is used to predict the future extending requests. Blocks in *sequential window* are temporarily reserved for streams, and other streams can not allocate any blocks from the occupied windows. Also, as the stream's workload changes, the window size is varied to make preallocation more adaptive (described as below). File allocator maintains both windows for each stream and any write workloads from different streams are thus not interleaved, enforcing the prediction of every individual stream to be isolated.

B. Allocation algorithm

To provide a concise framework which facilitates the implementation, On-demand preallocation employs two preallocation triggers, *layout_miss* and *pre_alloc_layout*:

- ***layout_miss:*** this trigger is hit if the current write blocks are not located in the *current window* or if the corresponding stream first performs extending operation on the file.
- ***pre_alloc_layout:*** this trigger is used for the reiterative preallocation. It is hit only if the current extending request locates in the *sequential window* and *layout_miss* is never hit. If hit, this stream is thought to perform sequential extending on the corresponding region of shared file and more blocks are thus preallocated for this stream for further extending.

```
extending-write:
    for each request
      if first written by this stream then   //initiation
          Setup sequential window;
          Preallocate with initial value;
      end if
      if the miss count exceeds threshold
          //other access pattern
          Set the fixed sequential window size;
          (Turn off on-demand preallocation of this stream)
      end if
      if hit layout_miss then
            // not in current window
            Increase the number of miss;
            decrease the sequential window size;
          end if
       else if hit pre_alloc_layout
          //in sequential window
          Ramp up and push forward window;
          Save last write position; //for next allocation
       end if
    end for
```

Figure 2. trigger hit algorithm

Figure 2 depicts the pseudo code of trigger hit algorithm. In the sequential workload, all write requests from the stream locate in the range indicated by current window. Reasonably, it is useful to preallocate more contiguous on-disk blocks for subsequent sequential write requests. After the *current window* becomes full, the *sequential window* moves forward to preallocate more blocks. The new *sequential window* temporarily reserves the contiguous blocks which are near to the last on-disk block of the shared file. The range presented by the new *current window* is replaced by the one indicated by original *sequential window*.

An example of on-demand preallocation performing on a shared file is shown in figure 3. At time *T1*, three extending requests (*100*, *200* and *300*) from different streams (*P1*, *P2* and *P3*) arrive in order. Since it is first time for them to write the shared file, the allocator first allocates one block for each request and initiates the sequential windows for each stream. Then it preallocates the space whose size is equal to the size of window and these blocks construct the sequential window. At *T2*, requests (*101* and *201*) of *P1* and *P2* arrive.

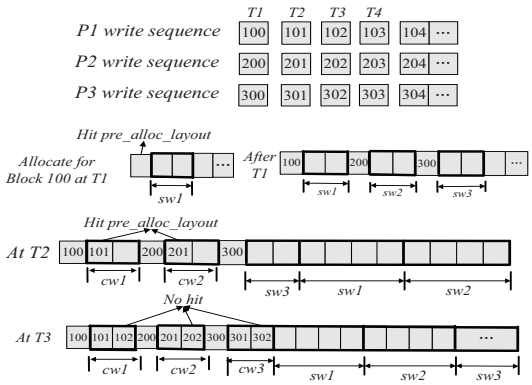


Figure 3. An example of on-demand preallocation. To simplify the presentation, we still assume that the request size from every client is one block. In the figure, the *cw* and the *sw* indicates the *current window* and the *sequential window*, respectively.

Since the request block locates in the sequential window, they hit *pre_alloc_layout*. Therefore, the allocator enlarges the sequential window size and push the window forward. The original sequential window become the current window. The subsequent requests (*102* and *202*) of *P1* and *P2* at *T3* neither hit the *layout_miss* (the block it wrote locates in the space of the previous preallocation) nor *pre_alloc_layout* (it do not reside in the current sequential window). Therefore, the allocator neither moves the sequential window ahead nor increases the miss number.

Instead of demanding a rigorous sequentiality, on-demand algorithm opts to do preallocation for extending operations that have high probability to be sequential. This is achieved by increasing the number of miss when hitting *layout_miss* tag. If the miss number arrives the threshold, we can recognize operations of this stream as workload other than a sequential one. As a result, in the face of random workload, the preallocation could be turned off immediately. In addition, since each write process is handled independently, on-demand algorithm ensures preallocation sequence of the sequential stream interposed by random streams is not interrupted.

C. Determining the preallocation size

Since preallocated blocks in the current window are persistent across system reboot, another concern arises from the preallocation size, which should be determined carefully based on both file size and access pattern. For example, due to a waste of free space, fewer persistent blocks should be allocated to small files; in our experiment on creating files (linux kernel code files), using static 256KB preallocation occupy 8GB space, 100 times more than static 16K preallocation. Contrarily, sequential large write on large files should allocate more blocks in each preallocation to keep more contiguous placement. Therefore, we adapt the size of sequential window with the varying workload. Assume the sequence of extending I/Os performed by a sequential

stream to be:

$$\{A_i = (start_i, size_i); i = 0, 1, \cdots, M\} \quad (1)$$

The size of sequential window can be determined as below:

1. if initiation: initialize window size using the write size:
$size = write\_size * scale,\ where\ scale\ is\ 2\ or\ 4;$

2. if subsequent preallocation: ramp up sizes exponentially:
$size = prev\_size * scale;$

We always enforce the maximum allowed preallocated size:
$size = min(size, max\_preallocation\_size);$
$where\ max\_preallocation\_size\ is\ tunable.$

IV. Organization of embedded directory

Although metadata servers in parallel file systems satisfy requests from their local cache as much as possible, they must access disk efficiently for some metadata access-intensive applications. In this section, we present how *embedded directory* algorithm mitigate the intra-directory fragmentation, exploring the disk bandwidth of metadata storage.

A. Basic organization

Embedded directory algorithm sequentially places all metadata of a file, including inode and layout mapping, in its parents directory contents. The embedded layout mapping expresses the mapping of file logic block number to disk physical location: it can be either the extents in block-based parallel file systems or the object id in the object-based file systems.

Conceptually, embedding these metadata in directories is straightforward. On creating a new directory, persistent preallocation is first performed in its contents for future subfiles creation. When directory enlarging, the number of preallocated blocks is scaled to support large directories, which is normal in large data centers. On creating a file, a new block is allocated from reserved directory blocks for the new inode. While extending a file, the mapping structure is first embedded into the tail of the inode.

As file system aging, a large number of mapping structures can be generated for an individual large file. In this case, extra blocks which hold them should be placed with the inode block contiguously. To identify whether the file system suffer from the fragmentation, a dedicated fragmentation degree is maintained in the directory inode structure. The degree value is simply calculated by dividing the number of layout mapping units (such as extent count in the block-based file system) to the number of files. If serious fragmentation is detected, an extra block is thus preallocated and used to stuff mapping structures to be generated. Note that although the mapping in some object-based file systems may be small enough (object id) to be stuffed in inode structure, we believe these extra blocks are very meaningful for some block-based parallel file systems which use extents or bitmap to express mapping. For example, *getlayout* operation in most block-based parallel file systems acquires all layout of the file, therefore all disk accesses can be combined in the same disk request, decreasing the disk positioning time than traditional approaches.

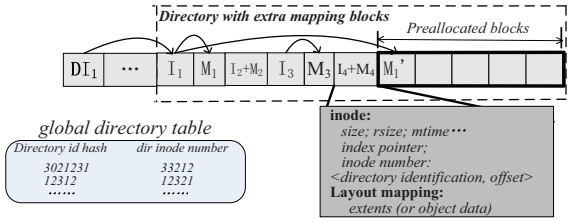


Figure 4. An example of embedded directory. This figure shows the on-disk locations of directory inode (marked *DI*), sub-file inode blocks (*I1-I4*), directory content block and the mapping structure for fies (*M1-M4*). Mapping of *I2* and *I4* is stuffed in its inode tail, and the mapping structures of fragmented *I1* and *I3* are stored in extra blocks.

Figure 4 illustrates an example of embedded directory. The directory entry is omitted from directory content and directory content is therefore placed more contiguously than traditional directory placement as shown in figure 1(b). In most cases, the file layout mapping is stuffed in the inode. For the unexpected fragmented large files, we may need more blocks to store the mapping structures, this scenario is illustrated by M_1 in the figure. In this case, two pointers in inode structure are reserved to indicate the address of extra blocks which are preallocated on creating file.

When reading the whole directory (e.g., *ls* operations), we opt to read all content in directory, including the extra mapping blocks (such as M_1, M_1' and M_3 in figure). Although this may enforce us to read more data than the traditional method, it is a rare case in a file system without suffering fragmentation. Moreover, since most I/O requests are performed sequentially on disk, extra cost would be quite smaller than huge latency caused by movement of modern disk head in traditional method. Deleting a file in directory do not release the blocks in directory content immediately. All freed files are batched and *lazy-free* is performed on freed blocks in the same directory.

B. Handling inode number

Some file management jobs in traditional parallel file systems rely on the constancy of the file ID (inode number) for a given file. Therefore, finding the location of an arbitrary inode given an inode number must still be both possible and efficient in embedded directory algorithm.

However, since embedded directory performs allocation on a set of inodes dynamically, direct translation between inode number and on-disk inode location is broken. To regain this relation, we introduce a dedicated *global directory table* in embedded directory algorithm. On creating a new directory, the new directory inode number is mapped to a unique *directory identification* and this mapping structure is stored into the *global directory table*. On creating a new subfile/subdir, its inode number is constructed by combining

its parent *directory identification* with offset in the directory. In our current implementation, the normal file inode number is expressed by a 64-bit number, and the *directory identification* and offset is sized at 32-bit. Although 64-bit design limits the file count in a directory and total directory count in file system, shifting to a 128-bit inode number with a 64-bit directory number and a 64-bit offset would overcome any realistic limitations.

Therefore, to locate the file inode with its inode number, we can use the directory identification portion of the inode number to index its parent directory's inode number using the directory table. Then we perform tracking back recursively until arriving at the root inode and getting the root inode blocks; Although this process may require extra disk IO to acquire the directory inode blocks, in most case, getting a file's inode number requires first looking up its parent directory which are cached in the first place.

Because embedded directory stores inodes inside the directory that contains them, moving a file from one directory to another (*rename* operation) involves moving the inode as well. When moving the file, because inode number encodes the inode's parent directory identification, the inode number must be changed. Unfortunately, changing the externally visible file ID can cause problems for system's online management using the file ID. Therefore, when renaming, the additional structure to correlate the old and new inodes is kept. If some applications intend to modify the new inode, the changes are also routed to the old one, and this correlation is maintained until the management routines exit.

C. Large directory support

Embedded directory algorithm is intended to optimize the *on-disk* metadata placement of a modern parallel file system like Ceph, PanFS etc. All these scalable file systems have fast indexing mechanism of in-memory directory entries in their individual metadata server, usually including Htree and Btree structure. Since these structures are constructed using the hash value of the subfile's name, they can be employed where needed to support medium sized (thousands of subfiles) directories indexing, without conflicting with the embedded directory organization.

On the other hand, although 99.99% of the directories have less than 8,000 entries as shown in previous studies, some applications do introduce extreme large directory (millions of subfiles) [30]. For example, ORNLs CrayXT5 cluster (with 18,688 nodes of twelve processors each) periodically write application state into a file per process, all stored in one directory. To support it, most parallel file systems build the metadata server cluster to balance load: subfiles in the extreme large directory are assigned to and managed by different servers. In this case, the cluster using embedded directory algorithm enforces the primary server (manage the parent directory content) to collect the hash value of the subfiles' name. Therefore, to lookup a specific file, the primary server find whether the hash value of the file name exists, avoiding to incur extra interactions with the subordinate servers which manage the subfile inodes' blocks.

D. Limitations

Our embedded directory algorithm essentially assumes that related metadata objects are often located in the same disk (or volume). Therefore, exploring the access locality by placing the metadata objects sequentially can improve the disk efficiency. When the metadata service is provided by the metadata server cluster, this assumption is still proper for the file systems which distribute metadata objects based on the directory subtree. In these file systems, all metadata in the subtree-based partition are delegated to an individual metadata server. Since on-disk metadata of a directory's subfiles is often accessed by the same metadata server, embedded directory algorithm can be integrated in the metadata storage seamlessly.

Unfortunately, this assumption can be broken by metadata servers which sacrifices locality for load distribution and index efficiency. For example, some metadata server clusters distribute the metadata objects by the hash value of the absolute pathname [5]. In this case, inode structures of the subfiles in the same directory are often managed by different servers in the cluster. The other example is the metadata server which distributes metadata in a flat database table to accelerate the metadata indexing [31]. Since the directory hierarchy is not explored for the data placement in this two scenarios, the embedded directory can not improve the disk performance.

V. Implementation and Evaluation

We have discussed how MiF to mitigate the fragmentation from the perspectives of both the normal file data preallocation and directory metadata management in previous sections. Here, we apply MiF in our block-based file system, Redbud, to verify the feasibility and effectiveness of our algorithm. We compare the MiF implementation to two baseline systems: the first is our original Redbud version which has been deployed in our laboratory for scientific simulation; the other is the Lustre file system, which is widely adopted in Top500 supercomputers.

A. Redbud overview

Redbud is a block-based parallel file system built in storage area network (SAN) environments. Figure 5 shows the main components of Redbud software architecture. Metadata server (*MDS*) collectively manages the storage of metadata, assisted by a dedicated metadata file system (*MFS*). The basic element of file layout is *extent*, which is identified by a tuple of *[file offset, group offset, length, flags]*. Client file system is optimized to reduce the interaction cost by congregating numbers of common operation pairs as mentioned

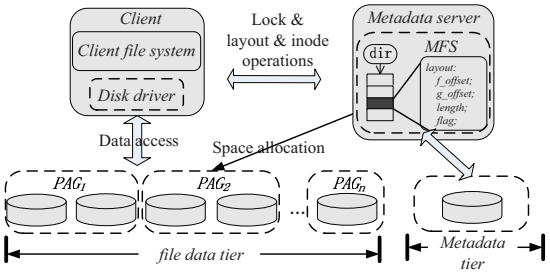
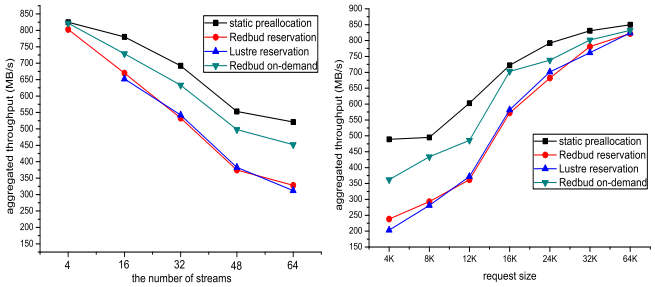


Figure 5. Redbud architectures



(a) stream count varies (b) impact of request size

Figure 6. Throughput using micro-benchmarks. In the case of *static* preallocation, we use the *fallocate* system call to preallocate the file space persistently for the shared file, and thus data of whole file is placed contiguously on disks, with the least fragmentation. Without involving the complex network interaction for consistency protocol, the performance of our original Redbud version is quite close to the Lustre.

in section 2. Shared disks are actual storage depositories for file data, and provide the block-based interface for clientsaccesses; these disks are divided into parallel allocation groups (*PAG*) for parallel management of free space.

In our experiments, we build the MFS using ext3 and then incorporate embedded directory into it. When creating a new file, we give up the usage of inode table in every block group and enforce allocating all inodes in the directory content. When creating a subdirectory, we retain original directory distribution algorithm, named *'rlov'*. As a result, while inode of subdirectory is created in its parent's directory content, the content of subdirectory is distributed between multiple groups.

B. Experimental setup

All our experiments are performed on client nodes with Intel Xeon processor (4 cores) running at 2.60GHz and 2048MB physical memory. Each machine is connected to the 32 ports Silk Worm fabric switcher by its own 400MB/s point to point link, using plugged Qlogic2432 card. Each node runs Linux 2.6.27 with default CFQ I/O scheduler and uses GNU libc 2.6. The version of the installed Lustre parallel file system is 1.6.6. In both file systems' configurations, communications between clients and MDS/OST all are GbE constructed by Catalyst 3750 Ethernet switches. Shared disks are fabric disks sitting in an individual JBOD array. Peak performance of an individual disk is about 170.2MB/s for sequential read and 171.3MB/s for sequential write.

C. The effectiveness of on-demand preallocation

1) ***Micro-benchmark evaluations****:* Our microbenchmark, based on the trace analysis of scientific computing environment [16], has two phases. The purpose of the first phase is to place file data on the disk using the method of all three preallocation strategies, including *reservation*, *static* and *on-demand* preallocation. The program started 4 threads on each client in the parallel file system, and all of them wrote different regions of a shared file concurrently. In the second phase, the shared file was split into 1024 segments and each one was sequentially read/written by a thread in cluster. In these experiments, we configured all data to be striped on five disks; Lustre utilizes the ext4 file system to builds its storage server (incorporating the reservation preallocation).

Figure 6 shows the throughput of the second phase in our program. The figure 6(a) plots that, the *on-demand preallocation* improves the throughput by about 17%, 27%, and 48% than reservation, for program runs with 32, 48, and 64 processes respectively. Since there are more different disk regions that are concurrently touched by the processes at a single disk using *reservation* preallocation, the improvement is mainly gained from the more contiguous data placement, especially on running a larger number of processes. Figure 6(b) shows the variance of throughput running 32 processes as the allocation size increases in the first phase. As expected, since the scheduler underlying file systems can not merge the fragmentary requests on disk, the preallocation with small size makes the subsequent file access suffering more from disk head interference. With *on-demand preallocation*, the interference is mitigated by more contiguous placement and the throughput is therefore improved. Coping with the interference among requests from the processes in runtime, however, may require storage system to predict the access pattern of request from multiple streams and dynamically reorganize the disk layout [15]. Therefore, *on-demand preallocation* does not help fix the problem. Compared with the *static preallocation*, while the decreased performance of on-demand preallocation ranges 2%-17% in the experiments, we believe that it can be used as the complementation for the static approach, especially in the environment without sufficient workload foreknowledge.

2) ***Macro-benchmark evaluations****:* We choose two widely used macro-benchmarks to evaluate the effectiveness of on-demand preallocation. The first benchmark is *IOR2* [28], which is configured at shared mode; basically it writes a large amount of data to one file and then reads them back to verify the correctness of the data; Each of the *m* MPI processes is responsible to read or write *1/m* of a file. The second benchmark is *BTIO*, which is an MPI program designed to solve the 3D compressible Navier-

Stokes equations using MPI-IO library for its on-disk data access[27]. We profile the executions of them using either non-collective I/O or collective I/O.

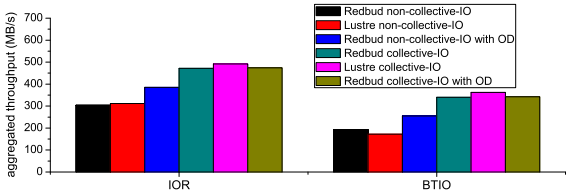


Figure 7. Throughput using macro-benchmarks. All of them are running on a 16-nodes cluster and 4 core of each node is used. all data are striped in eight disks.

Figure 7 shows the performance results of benchmarks with both the reservation and on-demand preallocation approaches. We observe the runs with on-demand preallocation maintaining higher throughput than the reservation mode by mitigating intra-file fragmentation. Compared with *BTIO*, the improvement for *IOR2* is smaller. This is because that, in *IOR2*, the request size is larger (32K-64K), and each process accesses contiguous data in its access scope. We can also see that the program's throughput with collective I/O performs is much better than its non-collective version. Through profiling we find that the size of collective-I/O requests is around 40MB, much larger than the size of requests with non-collective I/O. This may makes the effectiveness of on-demand preallocation be disappointed in this case.

Table I
NUMBER OF SEGMENTS AND AVERAGE CPU UTILIZATION OF MDS RESULTS IN RUNS.

Mode	Apps	Seg Counts	CPU utilization
Vanilla	IOR	2023	7%
	BTIO	1332	10%
Reservation	IOR	1242	6%
	BTIO	701	8%
On-demand	IOR	231	1.1%
	BTIO	106	1.0%

In the macro-benchmark runs, we also examine the metadata overhead using different preallocation algorithms. Table 1 shows the number of extents generated by the programs and average CPU utilization of MDS in the runs without using collective-IO. The *Vanilla* mode indicates no preallocation is used and the files are severely fragmented, suffering from more extents than others. We can also observe that on-demand approach has the potential to reduce the extents count (for both reads and writes) by a factor of 5-10 compared to the same file system with reservation preallocation. The less extents in the parallel file systems to be operated, such as merging and indexing, the less CPU load involved in MDS as revealed in the table. Since increased metadata overhead can cause less efficient mapping, we expect more benefits can be gained from on-demand preallocation in these programs when the system scales.

D. The effectiveness of embedded directory

*1) **Metarates evaluations**:* In these experiments, we tested the metadata performance of both Redbud (with/without incorporating embedded directory algorithm) and Lustre file systems with a single disk used at MDS end. MDS was configured to use synchronous writes for metadata integrity maintenance and a cluster of 10 clients concurrently accessed it. We used *Metarates application* [29], which was an MPI application that coordinated file system accesses from multiple clients.

Since the expected improvement introduced by embedded directory algorithm is mainly caused by the reduction of disk access count using embedded approach, we first examine the disk access count by intercepting the disk access in the general block layer in the kernel. The bar graph of figure 8 shows that the proportion of disk access count to the traditional mode (*Normal* directory) in four workloads varies greatly. In the *create* workload, *Metarates application* enforced each client to worke in its own directory; each single directory contained 5000 subfiles. In all compared file systems, to maintain the metadata integrity, journal was first sequentially done on the disk, the reduction of disk access counts mainly comes from the checkpoint operations. As the figure shows, first, the proportion to the traditional mode of *deletion* workload is much less than that of the others. The main reason is that, in deletion operation, the embedded mode only eliminates the disk access of the updates on the inode bitmap blocks. The disk access counts of the other operations, on the other hand, can be further decreased by avoiding to access the inode blocks. Second, it is interesting that, in the *readdir-stat* workload, the decreased disk access proportion increases as the directory size increases. This phenomenon is due to the design of the prefetching algorithm in the kernel: the size of prefetching window is gradually enlarged when it correctly predicts the blocks to be used. This optimization causes the system using embedded directory algorithm to essentially merge the individual *readdir-stat* operations to be some large read disk requests. Figure 8 also gives a performance comparison between the traditional directory and the embedded directory access. The graphs plot the throughput of typical metadata operations in different directory placement modes. As the figure shows, the performance increase introduced by embedded directory ranges from 23% to 170%.

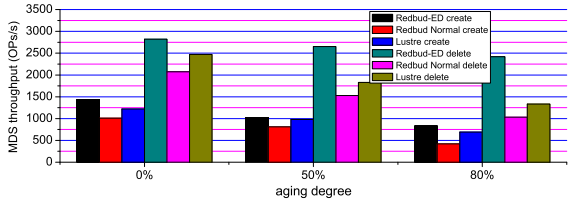
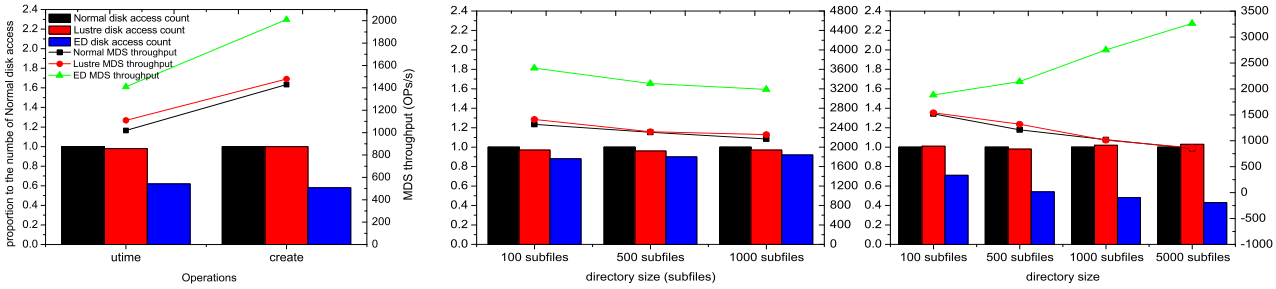


Figure 9. Impact of file system aging.



(a) *utime* and *create* workload results. (b) *delete* workload results (c) *readdir-stat* workload results

Figure 8. Performance comparison of file system with embedded-directory and normal placement. *Normal directory* indicates the access on the Redbud using traditional directory placement. The performance of the original Redbud version is quite close to that of the Lustre in all of the workloads. This is because the directory organization of ext4 in Lustre's MDS is quite similar to the ext3 employed in Redbud's MDS.

*2) **impacts of file system aging**:* We then examined the file system aging impact on the embedded directory. To handle the impact of file system fragmentation on the performance, we used an aging method similar to that described in the NetApp network file system report [17]. To achieve aging, our program created and deleted a large number of files. After reaching the desired file system utilization for the first time, our program executed a number of metadata access with the same distribution. Figure 9 shows performance for our micro-benchmark after the file system has been aged. Aging does have a significant negative impact on creation: at 80% capacity, the throughput for the creation using embedded directory decreases by 43%. Performance of deletion, on the other hand, is not severely compromised. This is because the most disk operations in deletion are to clear the bits in block bitmap of directory content. We also find that Lustre file system outperforms the Redbud using ext3. This makes sense because the ext4 used in the Lustre's MDS utilizes the Htree index to improve the performance of lookup operation which is involved in all metadata access operations. Even so, performance of operations on the embedded directory still outperforms both traditional approaches by over 26%.

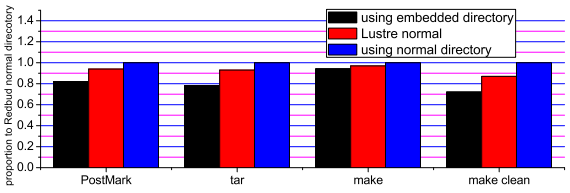


Figure 10. Executive time proportion of PostMark and applications. PostMark is configured by files-counts=100K, transaction-counts=500K and transaction-size is equal to file size; the three applications all use files (or tar.gz) of linux kernel code (v2.6.30).

*3) **PostMark and applications evaluations**:* Figure 10 shows the system performance of *PostMark* benchmark [2] and three different applications using two directory placement algorithms. In these experiments, 10 clients performed the same workloads presented in the figure in their own directories concurrently. Although these experiments are intended to approximate some of the activities common to small scale software development environments, we still observe 4%-13% reduction than Lustre file system in execution time for file-intensive programs, including *PostMark*, *tar* and *make-clean*. *Make* program, on the other hand, generates CPU-intensive workload in our environment. Therefore, we see a much smaller improvement of only 4%. However, we were actually quite glad at it because of the extremely untuned nature of certain aspects of our system (which has just barely reached the point, at the time of this writing, where such applications can be run at all).

VI. Conclusion

Fragmentation compromises the overall parallel file system performance. In this paper, we present design and implementation of MiF, which introduces two techniques: *on-demand preallocation* and *embedded directory*, to mitigate the intra-file fragmentation in parallel file systems, improving the disk performance in parallel file systems.

On-demand preallocation makes the file allocator be aware of multiple process streams and predicts the extending size at runtime. By reserving anticipated number of contiguous blocks for each stream, it improves the file data placement in the workload with shared file activity. Measurements of our on-demand preallocation implementation on a block-based parallel file system shows that it reduces the intra-file fragmentation effectively and achieved 17%-48% performance improvement in the workload dominated by the shared access.

Since modern parallel file systems opt to aggregate normal operation to reduce the interaction cost in their protocols, embedded directory is proposed to exploit the disk bandwidth of the metadata access. By contiguously placed all related metadata of a file in its parent directory content, embedded directory decrease the number of disk access of the normal metadata operations. This reduction is translated into both the metadata and small file access performance

improvement. Although our evaluation is preliminary, experiments with representative benchmarks show performance improvements ranging from 23%-170%.

ACKNOWLEDGMENT

This work was supported by the National High Technology Research and Development Program of China under Grant No. 2009AA01A403, the National Natural Science Foundation of China under Grant No. 60925006, and the State Key Laboratory of High-end Server & Storage Technology of China under Grant No. 2009HSSA01.

REFERENCES

[1] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan,M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. *The NFS version 4 protocol*. In Proc. of the 2nd International System Administration and Networking Conference(SANE2000), 2000, 94.

[2] J. Katcher, *PostMark: A New File System Benchmark*, Technical Report TR3022, Network Appliance Inc., 1997

[3] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C., *Ceph: A scalable, high-performance distributed file system*. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06).

[4] http://www.pnfs.com.

[5] P. J. Braam. *The Lustre storage architecture*. http://www.lustre.org/documentation.html, Cluster File Systems, Inc.

[6] D. F. Kotz and N. Nieuwejaar. *File-system workload on a scientific multiprocessor*. IEEE Parallel and Distributed Technology, 3(1):51C60, 1995.

[7] E. L. Miller and R. H. Katz. *Input/output behavior of supercomputing applications*. In Proceedings of Supercomputing 1991, pages 567C576, Nov. 1991.

[8] A. L. Narasimha Reddy and P. Banerjee. *A study of I/O behavior of perfect benchmarks on a multiprocessor*. In Proceedings of the 17th International Symposium on Computer Architecture, pages 312C321.IEEE, 1990.

[9] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, *The new ext4 filesystem: current status and future plans*. In Proceedings of the Linux Symposium, 2007.

[10] Welch, B., Unangst, M., Abbasi, Z., Gibson, G., AND Mueller, B. *Scalable performance of the panasas parallel file system*. In Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08), 2008, 17-33.

[11] F. Schmuck and R. Haskin. *GPFS: A shared-disk file system for large computing clusters*. In Proceedings of the 2002 Conference on File and Storage Technologies (FAST'02), 2002, 231-244.

[12] B. Welch. *POSIX IO extensions for HPC*. In Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST'05). 2005.

[13] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, *BORG:Block-reORGanization for Self-optimizing Storage Systems*, In Proceedings of the 7th USENIX Conference on File and Storage Technologies , San Fancisco, CA, 2009

[14] H. Huang, W. Hung, and K. Shin, *FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption*, In Proceedings of ACM Symposium on Operating Systems Principles, Brighton, UK, 2005.

[15] X.C Zhang and S Jiang, *InterferenceRemoval: Removing Interference of Disk Access for MPI Programs through Data Replication*. ICS10, June 2C4, 2010, Tsukuba, Ibaraki, Japan.

[16] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty. *File system workload analysis for large scale scientific computing applications*. In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST '04), 2004, 139-152.

[17] Andrew W. Leung , Shankar Pasupathy , Garth Goodson , Ethan L. Miller, *Measurement and analysis of large-scale network file system workloads*, USENIX Annual Technical Conference, 2008, 213-226.

[18] Tweedle, S. *EXT3, journaling file system*, July 2000.

[19] M. Rosenblum, J. Ousterhout, *The Design and Implementation of a Log-Structured File System*. ACM Transactions on Computer Systems,10(1),February 1992, pp. 25-52.

[20] G. R. Ganger and M. F. Kasshoek, *Embedded inodes and explicit grouping: exploiting disk bandwidth for small files*. In Proceedings of the annual conference on USENIX Annual Technical Conference, 1997.

[21] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, J.Ousterhout, *Measurements of a Distributed File System*. ACMSymposiumonOperating SystemsPrinciples, 1991, pp. 198C212.

[22] R. Latham, N. Miller, R. Ross, and P. Carns. *A next generation parallel file system for Linux clusters*. Linux-World, pages 56-59, Jan. 2004.

[23] *SGI CXS Clustered File System*, Datasheet, Silicon Graphics, Inc., 1600 Amphitheatre Pkwy. Mountain View, CA 94043.

[24] G. Ganger and M. F. Kaashoek. *Embedded Inodes and Explicit Groupings: Exploiting Disk Bandwidth for Small Files*. Proceedings of the USENIX Annual Technical Conference, 1997, 1-17.

[25] http://sourceware.org/cluster/gfs/

[26] http://www.simulia.com/products/unified_fea.html

[27] NAS Parallel Benchmarks, NASA AMES Research Center, http://www.nas.nasa.gov/Software/NPB/. Online-document, 2009.

[28] Interleaved or Random (IOR) benchmarks, http://www.cs.dartmouth.edu/pario/examples.html, Online-document, 2008.

[29] www.cisl.ucar.edu/css/software/metarates/

[30] Swapnil Patil and Garth Gibson, *Scale and Concurrency of GIGA+: File System Directories with Millions of Files*. In Proceedings of the 10th USENIX Conference on File and Storage Technologies , San Jose, CA, 2011.

[31] R. Latham, N. Miller, R. Ross, and P. Carns. *A next generation parallel file system for Linux clusters*. Linux-World, pages 56-59, Jan. 2004.