Design and Implementation of an Asymmetric Block-Based Parallel File System

Letian Yi, *Student Member, IEEE* , Jiwu Shu, *Member, IEEE* , Ying Zhao, Yingjin Qian, Youyou Lu, and Weimin Zheng, *Senior Member, IEEE*

Abstract—Existing block-based parallel file systems, which are deployed in the storage area network (SAN), blend metadata with data in underlying disks. Unfortunately, such symmetric architecture is prone to system-level failures, as metadata on shared disks can be damaged by a malfunctioning client. In this paper, we present an asymmetric block-based parallel file system, Redbud, which isolates the metadata storage in the metadata server (MDS) access domain. Although centralized metadata management can effectively improve the reliability of the system, it faces some challenges in providing high performance and availability. Towards this end, we introduce an embedded directory mechanism to explore the disk bandwidth of the metadata storage; we also introduces adaptive layout operations to deliver high I/O throughput for various file access pattern. Besides, by taking the MDS's load into consideration, we propose an adaptive timeout algorithm to make the MDS failure detection adaptive to the evolving workloads, improving the system availability. Measurements of a wide range of workloads demonstrate the benefit of our design and that Redbud gains good scalability.

Index Terms—Parallel file system, asymmetric

1 Introduction

In the SAN environment, a parallel file system is often built to manage shared disks. These file systems usually assume a conventional block I/O interface, such as read/write blocks, with no particular intelligence at the disks. They are often termed as *block-based* parallel file systems.

To simplify the disk management, existing block-based parallel file systems opt to take a *symmetric* approach, where metadata is blended with file data in their on-disk images [4], [7], [19], [25], and clients of these parallel file systems are connected to the disks that contain metadata as well as data content. For example, the implementation of block-model pNFS in Linux uses the same on-disk format with Ext4 [4].

However, such symmetric parallel file systems are prone to system-level failures, as metadata on shared disks can be damaged by a malfunctioning or malicious client. In most deployed block-based file systems, this threat is inevitable, as a large number of drivers that typically communicate with the kernel are installed in their clients. Unfortunately, previous studies have shown that errors of such drivers may overwrite file system structures through conventional block interface and destroy on-disk data [22], [23], [30]. Despite of the effort from testing communities, with tens of thousands of drivers, such errors still exist even in production softwares.

One option for ensuring the safety of metadata in parallel file systems is to outfit a front-end server for each disk (or disk array) to emulate a intelligent device, such as NASD [33] and TSD [34], which can effectively verify each operation of a client. However, the resulting storage systems significantly increase the economic cost and the power of cluster in practice [32]. In addition, interposing bulky, general-purpose servers between the network and disks potentially increase the latency of metadata access. In contrast, building a parallel file system directly on disks without intelligence avoids these shortcomings [7] [19].

Accordingly, we present an *asymmetric* block-based parallel file system, Redbud. Redbud leverages the zoning technique [9] of SAN to prevent clients from accessing the disks that store metadata. Specially, the disks storing file data are used to construct data zones and the clients can only connect to the data zones. On the other hand, metadata zones are constituted by all disks that store metadata, and they are managed by a *metadata server (MDS)*. Such design can effectively improve the reliability of the system by using more reliable devices or better controlled drivers for metadata storage to prevent namespace from being damaged by malfunctions.

However, in an asymmetric block-based file system, managing metadata on a centralized MDS faces challenges in providing high performance and availability. Towards this end, we make the following contributions in designing an asymmetric block-based file system:

- We propose an *embedded directory (ED)* mechanism to optimize the metadata access in the asymmetric block-based file system. Centralized metadata management allows potential optimizations in organizing metadata for commonly seen metadata operation pairs, such as lookup-stat. The *embedded directory* mechanism stuffs both file inodes and layout structures, which map file logic offsets to on-disk blocks,

- *L. Yi, J. Shu, Y. Zhao, Y. Lu and W. Zheng, are with the Department of Computer Science, Tsinghua University, East Main building 8-201, Beijing 100084, China. E-mail: lonat.front@gmail.com, {shujw, yingz, zwm-dcs}@tsinghua.edu.cn.*
- *Y. Qian is with Oracle Corporation. E-mail: Yingjin.Qian@sun.com.*

Manuscript received 21 Dec. 2011; revised 4 Nov. 2012; accepted 12 Dec. 2012. Date of publication 10 Jan. 2013; date of current version 24 June 2014. Recommended for acceptance by P. McDaniel.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.6

into the parent directory content, so that the content of the directory and its files/sub-directories are on the same or adjacent blocks, decreasing the number of disk accesses in the intensive metadata workloads.

- We propose an *adaptive layout prefetching (ALFP)* and an *on-demand pre-allocation (OD)* algorithm to deliver high I/O throughput for both exclusive file access and heavily concurrent file access pattern. With the help of MDS, the *adaptive layout prefetching* algorithm allows clients to prefetch file layouts to reduce layout requests, while carefully monitoring the competitions of layouts among clients to avoid frequent revocations of fetched layouts. The *on-demand pre-allocation* makes file pre-allocation being aware of concurrent process streams that write the same file at the same time. By doing so, clients can predict the pre-allocation size and effectively mitigate the fragmentation of files.
- We propose a *Line-least-square Curve Fitting (LCF)* based timeout algorithm to improve system availability. Failure detection in an asymmetric block-based parallel file system becomes an urgent and critical issue: if the requests to MDS cannot be responded in a timely fashion, the client needs to decide whether to send the requests again or to claim a MDS failure. The LCF-based timeout algorithm takes the workload on the MDS into consideration, and adaptively predict the response time (and thus the timeout value) of future requests, improving the availability of system.

We implemented a Redbud prototype system, and tested it in a SAN environment consisting of 33 nodes. Redbud was evaluated and compared with several state-of-the-art file systems using various benchmarks, real applications and simulators. Our evaluations show that Redbud achieves scalability as the cluster size increases, and achieves high file availability when faults occur on clients.

Compared with the traditional approaches, the embedded directory leads to 11-31 percent improvement of metadata access throughput; although the adaptive layout prefetching slightly decreases the exclusive access performance, it improves concurrent performance by up to 40 percent; the on-demand pre-allocation reduces the fragmentation effectively, leading to a performance increase of about 18 percent.

We next present the overview of the Redbud system in Section 2. Section 3 presents how Redbud manages and indexes metadata. The adaptive layout operations are described in Section 4. Section 5 presents system failure tolerance. Section 6 evaluates Redbud through detailed experiments. Related work is the subject of Sections 7 and 8 summarizes our research and presents the future work.

2 Redbud Overview

2.1 Architecture

Fig. 1 shows all main components of Redbud system architecture. Assisted by a dedicated *Metadata File System (MFS)*, a Redbud *Metadata Server* running on a node collectively manages the storage of all metadata and exports the entire file system directory tree to all *clients*. Since all metadata is stored on a dedicated backend storage (metadata zone) and can only be accessed through MDS, it can be built on well-chosen devices, for example, more reliable devices to prevent namespace from being damaged by device failures. The MFS is a module that runs inside the Linux kernel, and therefore Redbud MDS can use the default Linux-VFS caching policy. Similarly, the Redbud client residing on a client node also runs inside the kernel and provides file system interface.

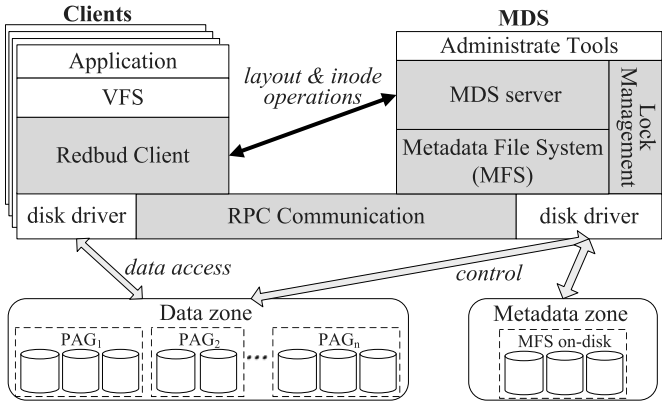


Fig. 1. System architecture of Redbud.

File data backend storages (data zones) are shared by all cluster nodes in Redbud. Typically, clients and MDS connect to data block devices via refined fiber channel. Data block devices are further organized as *parallel allocation groups (PAGs)*, each one of which has its own free space management and mapping structure, to provide high concurrency.

2.2 Metadata Types and Usage

Metadata in Redbud file system falls into three major types. First, *global file system metadata*, such as superblock, is managed and duplicated internally by MFS as in traditional file systems. Second, *free space metadata* tracks all unused blocks in PAG; for each PAG, MFS indexes its free space using a B+tree, in which the key is the starting block number of a contiguous free disk space and every leaf node corresponds to a contiguous free space in the PAG. Third, *file level metadata* is also managed in MFS and includes two components for each file in Redbud: (1) *inode* that contains the file attributes (such as mtime and size) and (2) *layout* that maps file logic address to physical address on devices.

The file layout is organized as an array of *segments*, each of which corresponds to a part of the file placed in a continuous disk space. A segment is presented as a tuple of $<$*file logic address (64 bit), length (48 bit), PAG address (80 bit)*$>$ using a total of 24 bytes, which means the file data of a size of *length* starting from *file logic address* is located at *PAG address*. A PAG address consists of a 16-bit PAG ID and a 64-bit offset within that PAG. Assuming the size of each block (i.e., the minimum space that can be represented by one segment) is 4K bytes, the worst-case additional space cost of using segments is 0.5 percent and occurs when files are at 4 KB-size or fragmented into blocks. Because the average file size is ever-increasing in network file systems [12], [18], this layout representation will be more space efficient in the future.

TABLE 1
Typical RPC Procedures in Redbud

RPC procedures	arguments	return values
open	file name	file attributes
layoutget	prefetched/min layout range	layout (segments)
revoke	revoked layout range	Yes/No
lookup	file name	Yes/No
getattr	file name	file attributes
setattr	file name and file attributes	Yes/No
readdir	dir offset	file name

2.3 Collaborations

Redbud clients interact with the out-of-band MDS via remote process call (RPC) protocol [10] implemented in the RPC communication module. Table 1 shows the typical RPC procedures in Redbud.

Redbud supports congregations of common RPC pairs. For example, clients can aggregate the *readdir-getattr* pair for *ls* operations, and the *open-layoutget* pair for accessing a file promptly after opening it; most *lookup* and *setattr* are also congregated in a request to modify file attributes (such as chown and utime operations) promptly after determining whether a file exists. Since clients always need to interact with MDS to get metadata for each operation, such congregated operations are preferred to reduce interaction cost and are actually seem commonly from real world applications.

With the help of RPC procedures, the modules of Redbud are used collaboratively to complete a certain task. We use the task of writing a file as an example to illustrate more on this point. To write an empty file in Redbud, a client first sends a request (*open-getlyout*) to MDS to ask for opening the file and where to write new data; on receiving the request, the MDS server accesses on-disk file inode to obtain the attributes, and then selects a PAG and queries its free space metadata to allocate some new blocks for the file; then, both new segment and attributes are sent back to the client, and the client writes the new data at PAG address indicated by the segment.

3 Metadata Management

As all metadata cannot be loaded into memory on MDS, MFS must aim to access disk efficiently for some metadata access-intensive applications. In this section, we present how metadata of directories and index is designed to minimize disk access count.

3.1 Embedded Directory

Congregated RPC pairs rise opportunities for exploring disk bandwidth fully on MDS. That is, if the metadata needed by each operation of a congregated operation pair is contiguously placed on the disk, multiple disk accesses of metadata can be merged into a single one. For example, for performing an open-layoutget request on a file, if its segments are placed adjacent to its inode, MFS can construct a single disk request to obtain both its inode and segments. Towards this end, Redbud introduces the *embedded directory* mechanism, which contiguously places all metadata of a specific file (including its inode and layout) into the *directory content blocks (DCBs)* of its parent directory.

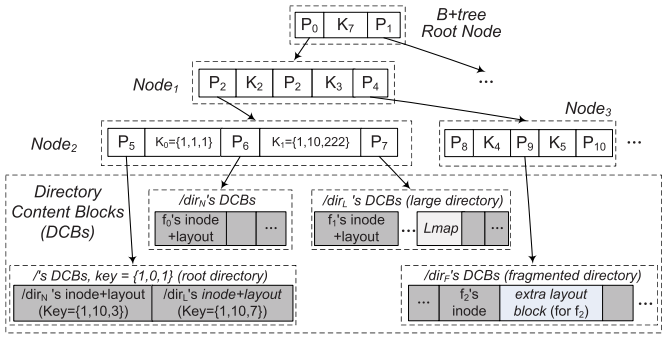


Fig. 2. Organization of MFS's B+tree. P *indicates a pointer to a tree node or a directory content block;* K *stands for a key in the B+tree.*

With ED, when creating a directory, we need to pre-allocate blocks for its directory content to prepare for future sub-file creations, and store the pointer to its directory content in its inode. When creating a sub-file, MFS allocates a new block in the reserved directory blocks to write its inode. While extending this file, the layout is first put after its inode in the same block. Depending on the number of sub-files/sub-directories and the size of each sub-file, directory content blocks may grow differently according to the following three cases:

Normal: In this case, each sub-file/sub-directory takes one block in the directory content (see the content blocks of "/" and "$/dir_N$" in Fig. 2). Deleting a file in a directory does not release the block in the directory content immediately. All freed files are batched and *lazy-free* is performed on freed blocks in the same directory.

Fragmentation: When a file system suffers from fragmentation, storing a large file may generate a large number of segments, and we need additional blocks for storing the layout (see the content blocks of "$/dir_F$" in Fig. 2). In this case, a dedicated *fragmentation degree* is maintained in every directory's inode structure. The degree value is simply the ratio between the number of segments and the number of files in the directory. Upon creating a file, if serious fragmentation is detected (when the degree value exceeds a predefined threshold), an extra layout block is thus preallocated and used to stuff layout segments to be generated.

Large directory: The embedded directory mechanism also supports large directories that are normal in large data centers. We introduce a *local directory map table (Lmap)* block in directory content to provide a mapping from a file/directory name hash value (using the n low-order bits of MD5 hash value) to the logical offset of its block containing its inode in the directory content (see the content blocks of "$/dir_L$" in Fig. 2). By doing so, looking up a sub-file/sub-directory inode can be done directly by looking up the Lmap table without traversing the entire directory content. In particular, for every 128 inodes in a large directory, we construct a Lmap table block in the directory content, and multiple Lmap blocks may exist for really large directories.

Unlike the traditional file systems, if a client only issues the *readdir* requests (without *stats* request), Redbud opts to read all content in the directory. This may enforce MDS to read more data (more content blocks) than the traditional method. However, previous studies show that accessing several contiguous 4 KB disk blocks costs a relatively small

amount more than accessing only one block (e.g., 1 percent for 56 KB extra). Since our directory content blocks are groups of contiguous blocks, the extra cost for acquiring all content of a directory would be quite small.

3.2 Namespace Index

Balanced trees have been proven to be successful at managing file system metadata because of their rapid indexing properties [14], [15]. In particular, we design a balanced tree to organize content blocks of directories only, since metadata of files are embedded in their corresponding directory content.

As shown in Fig. 2, MFS builds a directory index with a dedicated *B+tree*, where all inodes and file layouts of a directory are stored in the blocks (i.e., in the *directory content blocks*) pointed by the keys in leaf nodes. The internal nodes are composed of keys and pointers to their child nodes; there is always one more pointer than keys in every node. For example, P_0 points to the objects that have keys smaller than K_0, P_1 to those $K_0 \leq keys < K_1$. Therefore, when accessing a specific directory, one traverses B+tree using its key from the root node and gets its metadata's pointer from a leaf node.

Our goal of directory index design is to explore as much directory locality as possible at runtime. By *runtime directory locality*, we mean the set of directories that a user is working on at runtime. If we place the metadata of those directories close to one another on disks, we can achieve better I/O performance for the user at runtime by avoiding moving the disk head all over the place. In particularly, we make the following two assumptions to capture runtime directory locality: a user tends to access sub-directories of a same directory at runtime; a user tends to access directories created by the same user at runtime. Therefore, our metadata placement tries to place the metadata of directories by considering two priorities: placing sub-directories of a same directory together as our primary priority; placing directories created by one user together as our secondary priority.

Given such metadata placement strategy, we want to design *keys* for all directories, such that if two directories' metadata are closely placed on a disk, their key values are also closely located in the B+tree. By doing so, *runtime directory locality* also ensures better performance of searching in the B+tree, because directories visited by a user only involves nodes nearby, and we do not need to look for directories all over the B+tree.

In particular, we design the key for each directory as a 128-bit tuple, with three components: *{a parent directory ID (54 bits), a stream ID within the cluster (20 bits), a directory ID (54 bits)}*. The *stream ID* aims to uniquely identify the user that creates the directory in the cluster, and it is the conjunction of a user ID and a client ID. The *directory ID* is the physical address of the first content block in the directory. The root directory ("/") has the specific key {1, 0, 1} (for convenience, we use decimal digit here).

It is easy to verify that our key value generation preserves *runtime directory locality*. Sub-directories of a same directory share the first 54 bits in their key values, hence their key values are more nearby in the B+tree than those of sub-directories from different directory. Similar argument also applies for directories created by one user. Finally, directory IDs that reflect actual physical locations of directory content blocks on disks enforce the consistency between the metadata placement and the key placement furthermore. As shown in the Fig. 2, when stream 10, which has created "$/dir_N$" (whose key is {1, 10, 3}), intends to create a new directory named "$/dir_L$" under "/", MFS first lookups from the root node and acquires the key of its parent directory ("/"). Then the parent ID of "$/dir_L$" is filled with the directory ID of "/" (which is 1 in the example). Since the key of "$/dir_L$" has the same parent directory ID and stream ID as "$/dir_N$", the two keys are close and they reside in the same sub-tree of B+tree (the sub-tree is the $Node_2$ in the figure). With this organization, searching keys locates in the same directory can be performed locally. For example, after accessing "$/dir_N$", accessing "$/dir_L$" in the same directory only needs to traverse from $Node_2$.

Unlike the modern file systems [15], Redbud utilizes the B+tree to index only directories, preventing the frequent creation and deletion of normal files to cause splitting/merging nodes. As a result, the whole tree is locked only in the "mkdir/rmdir" operations that are scarce in typical workloads [18].

4 Adaptive Layout Mechanisms

Different applications in large data centers introduce diversified file access patterns. For example, recent studies show that scientific applications require inter-file parallel to be accessed by multiple nodes across a cluster [7]. Contrarily, 76.1 and 97.1 percent of files are only opened by one client in finance and engineering environments, respectively [12]. With the help of a centralized MDS server, we design two adaptive layout mechanisms to achieve high performance and minimize the number of interactions between clients and MDS.

4.1 Layout Prefetching

When a client needs to access the content of an existing file, it first sends a *layoutget* request to the MDS server to fetch the segments of the file. Once obtaining the layout, the client may hold the segments until the file is closed. On the other hand, Redbud guarantees single-client-node equivalent POSIX semantics for file system operations across the cluster. That is, multiple clients in Redbud can concurrently hold the segments that correspond to the same file range for read operations. However, if a client wants to write a range of a file, Redbud does not allow the other clients to hold or obtain the segment that corresponds to the written range. In this case, the MDS server can send a *revoke* message to a client, asking the client to purge the corresponding layout range from its memory; and we say a *conflict* occurs in this case.

To decrease the number of messages for requesting segments, Redbud allows clients to predict the file ranges to be accessed and prefetch related segments from MDS in a layoutget request. However, a client should determine the size of a prefetching range carefully: a large size may increase the conflict probability, consequently, the segments will be revoked by the MDS server in the future; contrarily,

if the prefetching range size is too small, client may be enforced to send a large number of layoutget requests even for sequential and exclusive access. Hence, we introduce an *adaptive layout prefetching* algorithm on the client side, which considers both recent layoutget and revoke operations, to determine the prefetching range size. In general, ALPF would increase the prefetching range size if past requests show a trend of sequential and exclusive access, and decrease the size otherwise.

For each file, a Redbud client maintains a *history list* structure, which records historical layoutget and revoke requests on the file. For each request *req*, an entry consisting of its time stamp t_r and its layout range r_{req} is recorded in the history list. The importance of a request *req* is determined by a decay function:

$$f_{req} = \begin{cases} log_n(ct - t_r); & \text{if } (ct - t_r) \leq 1 \\ |log_n(ct - t_r)|; & \text{if } 1 < (ct - t_r) \leq n^{-1} \\ 1/|log_n(ct - t_r)|; & \text{if } (ct - t_r) > n^{-1}, \end{cases} \quad (1)$$

where ct is the current time, both ct and t_r are in seconds, and $n < 1$ (the default value of n is 0.8 in our implementation).

Now, when a client needs to send a layoutget request, ALPF first identifies two sets of requests from the history list S_g and S_r, corresponding to the set of *layoutget* requests and *revoke* requests that worked on the nearest continuous range before the range of the current layout-get request, respectively. Then the prefetching range size (denoted as ps) is calculated as follows:

$$ps = Max\left(0, init * \left(\eta * \sum_{req \in S_g} f_{req} + \theta * \sum_{req \in S_r} f_{req}\right)\right), \quad (2)$$

where η is a positive coefficient and θ is a negative coefficient; the *init* is the initial prefetching range size; in our implementation, the default *init* is 16 KB, and the default values of η, θ are 1, -1 , respectively.

As a result, if a client frequently performs layoutget operations in a sequential workload, ALPF enlarges the prefetching size for future operations; on the contrary, if there are some revoke requests in the history list, the prefetching size is gracefully decreased to reduce potential conflicts. For example, in the case of sequential exclusive write, no revoke request is in the history list for the file, so the prefeching range size grows fast and the client can prefetch the whole file's layout soon.

After determining the prefetching size, for every layoutget request, two types of ranges are sent to the MDS server: the *min-range*, which corresponds to the range of the current read()/write() system call, and the *prefetch-range*. On receiving the layoutget request, the MDS server first checks if the prefetch-range has overlaps with the ranges held by the other clients. If it has, the MDS server will grant as large prefetching range as possible, without revoking from the others.

However, min-range may also have overlaps with the held ranges, and this conflict type is referred as *mandatory conflict*. In this case, the MDS server must revoke the overlapped ranges from other clients. However, if mandatory conflicts on a small region are frequent, revoke messages could be the performance killer of a parallel file system. With regards to this, the MDS server divides a file into multiple 16 KB-regions. If too many mandatory conflicts occur in the same region, Redbud gives up the prefetching on the region.

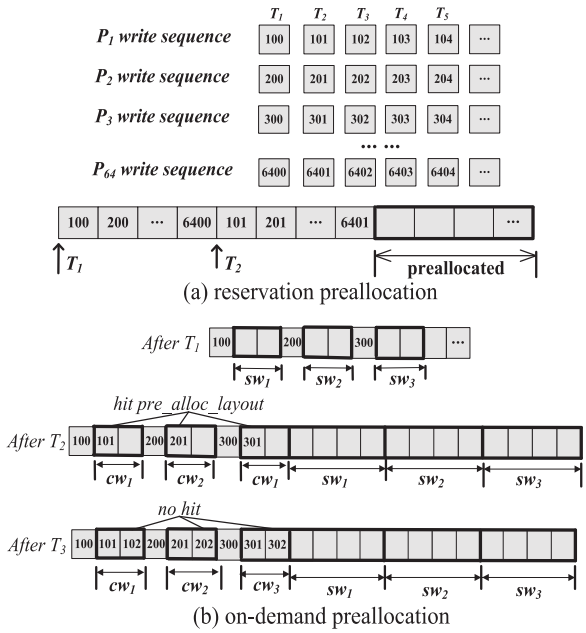


Fig. 3. Examples of traditional and on-demand pre-allocation. *sw and cw stand for the current window and sequential window, respectively.*

4.2 On-Demand Pre-Allocation

Now we consider the case for allocating new layout segments for a file when clients try to extend it. To place a normal file contiguously on disk, the core idea of traditional pre-allocation in local file systems, such as Ext3/4 [16], is that, for every file that is being extended, the allocator 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 file are allocated from that range, instead of from any arbitrary free space. If the subsequent workload is dominated by sequential write from a single process, this *reservation* ensures the contiguous placement.

However, reliance on traditional pre-allocation algorithm have limited the performance of parallel file system. In particular, the reservation algorithm cannot reduce the fragmentation inside individual file that is written by multiple processes concurrently. Fig. 3a shows a fragmentation example of it. In the example, 64 processes (denoted as P_1, P_2, ..., P_{64}) concurrently extending-write on a file. Assume each process writes sequentially with the speed of one block at each time T_n. For example, P_1 writes logic block number 100 at T_1, and 101 at T_2, and so on. The traditional pre-allocation would reserve a range of on-disk blocks for all processes to write into. Consequently, logic blocks $100, 200 \ldots, 6{,}400$ are placed together in the reserved space in the order of arrival time, and logic blocks 100 and 101 are far apart. As a result, the mapping from logic block address to the physical block is fragmented and subsequent (even sequential) access to this file incurs a mass of disk head interferences.

Redbud introduces *on-demand pre-allocation* to mitigate the fragmentation when multiple processes extend the same file. In OD, the file allocator first distinguishes write streams, which is essentially a process that has opened the shared file in the cluster. Every write stream is identified by a unique *stream ID*, which is a tuple of $<$*client node ID, process ID*$>$ to uniquely represent a process running on a client node. It then performs pre-allocation for every write stream independently.

OD maintains two data structures for pre-allocation: a *current window (cw)* and a *sequential window (sw)* for each write stream. Similar to the traditional reservation, a current window contains the blocks which have been persistently pre-allocated to a stream (i.e., appeared as a segment in the layout of the file). However, unlike the traditional reservation, not all file contents can be written into a current window by a stream. Instead, a current window also corresponds to a range of file logic blocks. If a stream writes a file logic block outside the range of the current window, it will not be written into the current window. A sequential window, on the other hand, is used to predict the future extending requests. Disk blocks in a sequential window are temporarily reserved for a stream, and other streams cannot allocate any blocks from the sequential windows. Similarly, a sequential window corresponds to a range of file logic blocks as well. A sequential window becomes a current window, when a stream starts to write file logic blocks within the range of a sequential window. At that time, all the blocks in the sequential window are allocated permanently for the file.

OD is designed for sequential write, so the file logical blocks in a sequential window immediately follow those in a current window. If a stream writes outside both its current and sequential window, we say a *layout_miss* happens, and use a counter *miss* to record how many such misses have happened. In general, high miss numbers indicate that the access pattern of a stream is not sequential write at all and OD should be turned off immediately. In particular, we employ two triggers for sequential window reservations:

- *layout_miss*: this trigger is hit if the current write block is not located in the current or sequential window; or if the corresponding stream performs extending operation on the file for the first time.
- *pre_alloc_layout*: this trigger is hit only if the current extending request locates in the sequential window, which means the sequential window becomes the current window and a new sequential window needs to be reserved for this stream for further extending operations.

Algorithm 1 depicts the pseudo code of the OD algorithm. As the algorithm shows, we adapt the sequential window size *sw_size* to the varying workload to obtain more continuous placement.

An example of on-demand pre-allocation performing on a shared file is shown in Fig. 3b. At time T_1, since it is the first time for streams to write the shared file, the allocator first initiates a sequential window for each stream. At T_2, requests (101 and 201) of P_1 and P_2 arrive. Since the requested block locates in the sequential window, the request hits *pre_alloc_layout*. Therefore, the allocator turns the original sequential window to be the current window, and then reserves a sequential window with an enlarged size. The subsequent requests (102 and 202) of P_1 and P_2 at T_3 hit neither the *layout_miss* (the block it wrote locates in the space of the current window) nor the *pre_alloc_layout* (it does not reside in the current sequential window). Therefore, the allocator neither moves the sequential window ahead nor increases the counter *miss*. From the example we can see that when sequential write progresses, all write requests from a stream locate in the range indicated by its current window. Reasonably, it is useful to reserve more contiguous on-disk blocks for subsequent sequential write requests. After the current window becomes full, the sequential window moves forward to reserve more blocks.

Algorithm 1: On-demand pre-allocation algorithm

```
miss = 0;
for each request do
    if first written by this stream then
        sw_size = write_size * scale;
        sw_size = min(sw_size, max_reserv_size);
        sw_start = write_logic_block;
        Request to setup a sw of sw_size for logic blocks
        starting from sw_start;
    if miss exceeds a threshold then
        //other access patterns;
        //turn off OD of this stream;
        sw_size = 0;
    if hit layout_miss then
        //logic address not in the cw;
        miss++;
        sw_size = sw_size/scale;
    else if hit pre_alloc_layout then
        //logic address is in the sw;
        Make the sw being the cw;
        sw_start = sw_start + sw_size;
        sw_size = sw_size * scale;
        sw_size = min(sw_size, max_reserv_size);
        Request to setup a sw of sw_size for logic blocks
        starting from sw_start;
```

5 Adaptive Timeout

In an asymmetric parallel file system, clients must constantly detect if MDS fails as they communicate with MDS for each metadata access. *Static timeout* is widely adopted in modern parallel file systems, in which clients reconnect MDS and resend their requests if previous requests are not responded within a fixed amount of time. Such strategy assumes that service time of every request in the systems should not exceed the predefined upper bound; if the request cannot be responded in the service time, the request is considered failed. This approach is applicable in small scale environment [10], [17]. However, applying it in a large asymmetric parallel file system, where burst access is normal, would complicate recovery. For example, when layouts are committed by a large number of clients simultaneously, the MDS server would suffer from huge disk access latency and service time thus cannot be determined easily. Obviously, a small timeout value is reckless: when the timeout is reached, thousands of clients may try to reconnect to MDS and resend un-committed requests to recovery mistakenly. Contrarily, using a large timeout value, clients

cannot cope with MDS failures in time and the file system therefore hangs longer than expectation, damaging the system availability [1].

We introduce *adaptive timeout* in Redbud. The main idea is that, since workloads in data centers may be evolving constantly, the service time of every individual request should be determined based on the current load of MDS: when MDS becomes congested because of burst requests, the estimated service time should be increased; otherwise, if the workload is reduced, the value should be decreased. Obviously, if the service time can be predicted explicitly, we can determine a timeout value correctly.

Again, we ask each Redbud client to record historical request service time values and estimate a timeout value for itself. By doing so, the MDS server does not have to compute estimated service time values for all clients. In particular, the service time is defined as the time range between the time point when a client sends a RPC request and the time point when the client receives the response. This choice enforces clients to predict the end-to-end service time, considering both network latency time and MDS's processing time.

Historical service time values are tracked by a *time window*, which spans multiple seconds, in a client. We split a time window into N *sub-windows*: assuming the length of a time window is H, the length of each sub-window, L, is equal to H/N. Since a large number of requests may be accomplished in each sub-window, we record the average service time of them and present them using a pair of time values: $< t_i, v_i >, (i = 0, 1, \ldots, N)$, where t_i is the time stamp of sub-window i and v_i is the corresponding average service time.

Based on the recorded historical values, we consider two strategies to estimate service time: *MAX* and *line least square curve fitting*. The MAX approach use the maximum value among the records kept in a time window: $estimate(v) = max(v_i), (0 \leq i \leq N)$. This simple approach introduces little cost (either implementation efforts or CPU load) on estimating but with limited prediction ability. The LCF method, on the other hand, is adopted to perform curve fitting on historical values to predict the timeout. Such fitting is reasonable as service time changes gradually, instead of rapidly, even in burst workloads. This is mainly because that the cost to handle a request on disks is nearly constant, service time is therefore proportional to queuing time (at MDS) which is determined by the queue length. The estimation of service time for current requests is given as below:

$$estimate(v) = f(t) = a_0 + a_1 * t; \tag{3}$$

$$where: a_0 = \left(\sum_{i=0}^{N-1} v_i\right)/N - a_1\left(\sum_{i=0}^{N-1} t_i\right)/N;$$

$$a_1 = \frac{\sum_{i=0}^{N-1} t_i v_i - \left(\sum_{i=0}^{N-1} t_i \sum_{i=0}^{N-1} v_i\right)/N}{\sum_{i=0}^{N-1} t_i^2 - \left(\sum_{i=0}^{N-1} t_i\right)/N}. \tag{4}$$

While adaptive timeout can effectively improve accuracy of timeout prediction, it cannot completely eliminate false positives. Therefore, on detecting a timeout, the client sends a *probe* message to MDS to check if it really fails. If MDS is not failed, it immediately responds the probe message after receiving it. Note that this probe message is similar to traditionally lightweight "heartbeat" message; however, since our adaptive timeout algorithm makes predictions more accurate, network traffic incurred by probe messages can be significantly reduced.

6 Evaluation

Our experimental evaluation answers four questions: (1) What is the overall performance and the data availability of the asymmetric block-based parallel file system? (2) How much improvement can metadata placement gain over the traditional methods? (3) How much improvement can adaptive mechanisms gain over the traditional approaches? (4) What is the impact on system availability when the LCF-based approach is used?

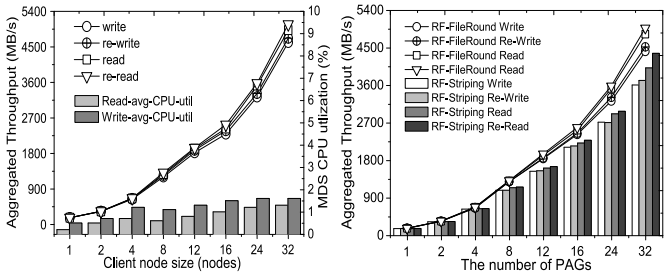
6.1 Experimental Setup

All experiments were performed on up to 33 nodes, including a metadata server. Each node was configured with Intel Xeon processor (4 cores) running at 1.60 GHz and 1024 MB physical memory. All nodes were running Linux 2.6.32 kernel. With plugged Qlogic2432 card, each machine was connected to the Silk Worm fabric switcher by its own 400 MB/s point to point link. Communication between clients and MDS is GbE constructed by Catalyst Ethernet switches. Shared disk devices are fiber disks sitting in a series of individual JBOD arrays. According to the measurements of our micro-benchmarks, peak performance (accessing the outer tracks of the disks) of individual disks is about 172.2 MB/s for sequential read and 178.3MB/s for sequential write, and the average performance (accessing the middle tracks of the disks) of individual disks is about 137.1 and 138.8 MB/s for sequential read and write operations. In our experiments, each PAG is configured with one disk and MDS is also built using one disk.

We also compared Redbud with three state-of-the-art parallel file systems: (1) two object-based asymmetric file systems, Lustre [5] and Ceph [3]; (2) a pNFS implementation in Linux (published in Linux kernel 2.6.32), which was a symmetric block-based file system. In our experiments, MDS and each object storage device (OSD) node of Lustre was configured with one disk, and each OSD of Ceph also uses one disk. The version of Lustre and Ceph is 1.6.0 and 0.46, respectively.

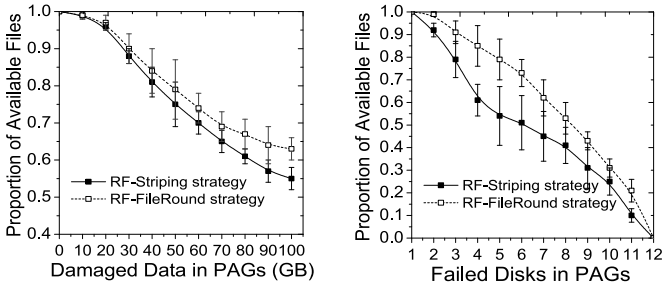
6.2 Overall Evaluation

In this section, we measure the aggregated throughput of Redbud when scaling either clients or storage nodes and the data availability of Redbud when some faults occur. To evaluate the overall performance of Redbud, we ran an *IOzone* [13] worker on every client to generate four types of normal operations: *read, write, re-read and re-write*; the read/write is to access file after the file is created and the re-read/re-write is to access the existing file repeatedly. Except for special declaration, we conducted the experiments of



(a) Increasing the clients. (b) Increasing the PAGs.

Fig. 4. Performance as the cluster size increases. In the (a), the number of PAGs was equal to the number of clients. In the (b), we configured 10 clients to access their own files.



(a) Increasing the damaged data. (b) Increasing the failed disk.

Fig. 5. File availability as the faults increase. *Redbud was built with 10 and 12 PAGs in (a) and (b), respectively.*

IOzone with the request size of 32 KB. We use two typical file data distribution of parallel file systems: (1) *RF-Striping* strategy, in which MDS splits a file into 32M-parts and every part is then placed on an individual PAG, and (2) *RF-FileRound* strategy, in which MDS first selects a PAG in "round-robin" fashion when creating a new file and then places the whole file on the selected PAG.

6.2.1 Data Access Scalability

Fig. 4a shows how throughput varies when more client nodes are added to Redbud. We can observe that, except for slight performance lost, aggregated throughput increases when scaling the cluster size. Generally, write operations must perform more metadata updates such as layout change, therefore, they performed poorer than read operations which only updated inode attributes. From the figure, we can also find that the Redbud gains more overall performance in the re-write workload than the write one. The reason is that, to write a new file (write workload), MDS must update layouts and free space metadata before responding to the clients; re-write operations, on the other hand, never update them because the layouts need not to be changed before removing file. Besides, as far as each situation is concerned, it is shown that, with the increase in the client count, the CPU utilization of MDS increases, whereas the average CPU load is still lower than 2 percent when scaling to 32 clients.

Fig. 4b illustrates performance results as increasing the PAGs. We can observe that a roughly linear growth of performance as increasing disk. Fig. 4b also gives a comparison between two data placement strategies in Redbud. Compared with RF-Striping strategy, RF-FileRound strategy provides a ranged growth of aggregated throughput when all clients perform intensive IO. The main reason is that, when striping every file on all disks, the disk head interleave incurs penalty because every disk is simultaneously accessed by all clients in the cluster.

6.2.2 Data Availability

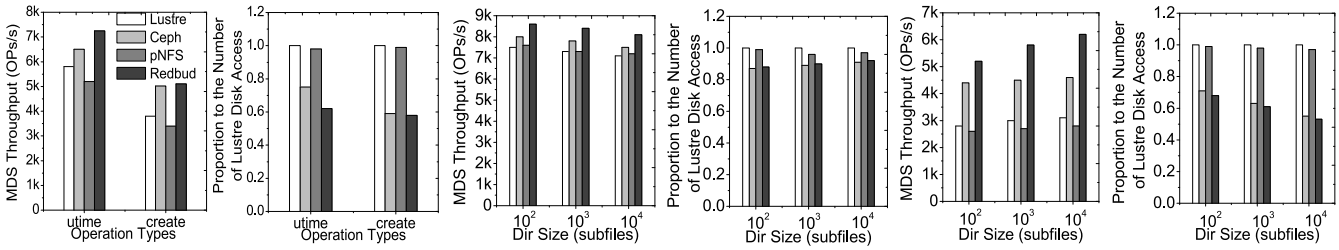
Since Redbud prevents the clients from damaging file system consistency by isolating metadata storage in the MDS access domain, it is meaningfully to examine how file availability degrades when the faults occur on PAGs. We generated a file system snapshot according to the analysis of NetApp large scale network file systems [12], which were used by over 1,500 employees. The snapshot had about 1 TB file data and one million files. We injected two types of faults: (1) *data overwritten*: we bypassed the file system and randomly overwrote the data on the PAGs; (2) *disk failure*: we remove PAGs after snapshots have been stored in system. In both cases, we measured the percentages of the files whose *all* contents are available after faults have been injected.

Fig. 5a shows the file data availability result. From the figure, we can observe that Redbud works quite well, with the number of unavailable files proportional to the amount of damaged data, in contrast to a symmetric file system where a overwritten sector or disk crashes would lead to the whole file system unavailable because of metadata corruption [4] [19]. In fact, availability sometimes degrades slightly less than expected from a strict linear fall-off, this is because a slight imbalance in data placement across disks and within directories. Fig. 5b shows the result of file data availability when removing disks. We can find that RF-File Round strategy leads to better file availability than the distribution with RF-Striping strategy in average. This is mainly because the large files are usually striped in multiple disks with RF-Striping strategy, and removing one of these disks will make the files unavailable.

6.3 Metadata Service

To evaluate the performance benefits of our method for metadata placement, we tested Redbud, Lustre, Ceph and pNFS in our environments. Lustre uses traditional directory placement by exporting an Ext3; similar to Lustre, pNFS stores all data and metadata on an Ext4 that is a improved version of Ext3; Ceph, on the other hand, embeds inodes in its directory and eliminates object-level metadata using CRUSH algorithm; besides, by borrowing idea from log-structure file systems, Ceph performs copy-on-write for each write operation to optimize small writes. In all system setups, 10 clients simultaneously accessed the MDS.

We first used *Metarates* application [27], which was an MPI application that coordinated file system's metadata accesses from multiple clients. We generated three types of workloads, and every client works in its own directory: first, a *concurrent create* workload in which every client created 50,000 zero-byte files simultaneously in its own directory, and a *concurrent utime* workload in which every client modifies the access time of created file; second, we used a *concurrent delete* workload in which every client simultaneously performs *unlink()* operation on all files in its own directory;



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

Fig. 6. Comparison of Metadata Access Performance.

finally, a *concurrent readdir-stat* workload that performed both *readdir()* and *stat()* operation on the sub-files concurrently in the created directory.

Next, we used two benckmarks, PostMark [2] and Polygraph [35], and three different applications, tar, make and make clean, to test performance of small file access that is a metadata intensive workload. Three applications were intended to approximate activities common to software development, using files (or tar.gz packets) of Linux kernel code. PostMark was configured by file-counts of 100,000 and transaction-count of 500,000, mimicking business transactions environments. Polygraph, on the other hand, targeted IO performance evaluations of web services: in our runs, Polygraph generated two typical directory structures: 8*8*1 and 8*8*16, in which each configuration had 1 and 16 files in their leaf-directory respectively, and depth of directory was 8 with 8 subdirectories in each directory; after the generation, Polygraph read the files from the file systems to verify the data correctness; the file size in Polygraph was randomly distributed in range of 10KB-1MB.

6.3.1 Metadata Access

We conducted the experiments to measure the metadata access performance by running 4 Metarates processes on each client. Since the reduction of disk access count contributes to the expected improvement, we also examine the disk access count by intercepting the disk access in the general block layer of kernel on the MDS and OSD.

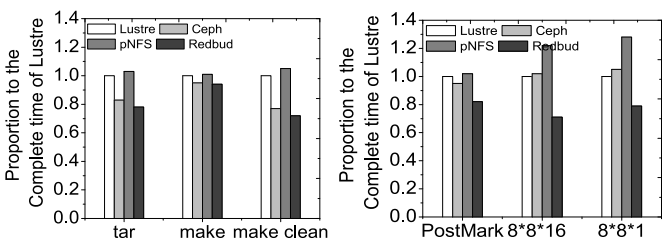
The right bar graphs in the subfigures of Fig. 6 show that, the proportion of Redbud's disk access count to the other approaches in four workloads varies greatly. First, except for utime workloads, the disk access count of Ceph is close to that of Redbud. Utime operations introduce higher proportion of disk access count in Ceph than that of Redbud, this is because that the cleaning process of Ceph's OSD needs to periodically perform garbage collection, incurring many disk accesses. Second, the improvement of Redbud for deletion workload is much less than that of pNFS and Lustre. The main reason is that, in deletion operation, the embedded directory only eliminates the disk access of the updates on the inode bitmap blocks. The disk access count of the other operations, on the other hand, can be further decreased by avoiding to access the inode blocks. Finally, 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 linux kernel: the size of prefetching window is gradually enlarged when it correctly predicts the blocks to be used [28]. This optimization causes the system using embedded directory algorithm to essentially merge more disk requests of the readdir-stat operations into some larger ones.

The left graphs in the subfigures of Fig. 6 also provide a performance comparison between Redbud and the other file systems. It is interesting that, under the deletion/readdir-stats workloads, though disk access count of Ceph does not appear to be much different from that of Redbud, the results show the performance of Redbud is higher than that of Ceph. The reason is that, MDS in Ceph acts as a cache-of-metadata and OSD is a simulated device using a server, therefore, accessing metadata needs a large number of network interactions between MDS and OSD. The results also indicate that the performance improvement gained from Redbud's metadata placement ranges from 8 percent (10,000 sub-files for every directory in the delete workload) to 31 percent (10,000 sub-files for every directory in the readdir-stat workload).

6.3.2 Small File Access

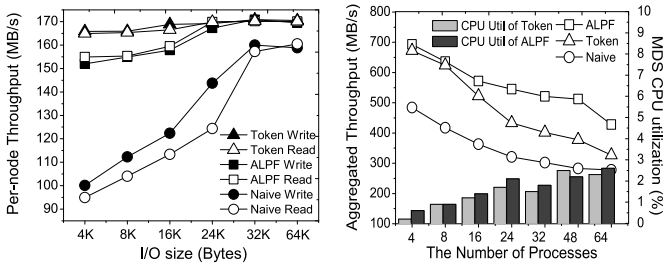
In this group of experiments, we intended to quantitatively characterize the performance of small file access by running a process on each client. Fig. 7 shows the executive time proportion of two benchmarks and three different applications. We can observe that Redbud gains 3-14 percent reduction than the other file systems in execution time for programs, including *PostMark*, *tar* and *make-clean*. Compiling the kernel program (*Make* workload), on the other hand, generates CPU-intensive workload in our environment, therefore, we see a much smaller improvement.

The improvement of Redbud for webservice benchmark, however, is much higher. First, Redbud outperforms Ceph with a improvement of up to 32 percent, the main reason is that, when the processes concurrently generated intricate



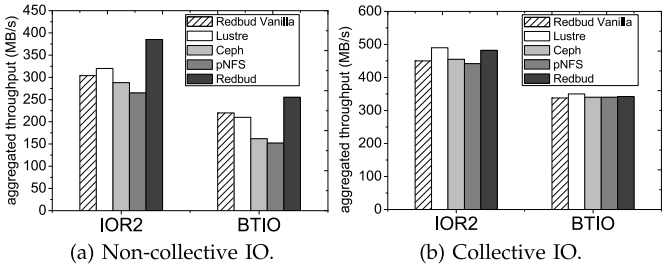
(a) Development workloads. (b) PostMark and PolyGragh.

Fig. 7. Executive time proportion of benchmarks and applications. *Both 8*8*16 and 8*8*1 are the different polygraph configuration as described in text.*



(a) Performance of IOzone (b) IOR2 results as client nodes increase. All data are striped in eight disks.

Fig. 8. Results of adaptive layout prefetching. *In our experiments, we used the default parameters of Redbud to determine the* ps, $\eta = 1, \theta = -1, n = 0.8$.



(a) Non-collective IO. (b) Collective IO.

Fig. 9. Throughput results of macro-benchmarks. *All of them are running on a 16-nodes cluster and each node runs 4 processes. All data are striped in 8 PAGs.*

directory structures, as Ceph writes data in a copy-on-write fashion, the metadata locality is inherently damaged, and the performance of subsequent read access is therefore compromised. Second, the observed improvement over Lustre is also related to exploiting directory locality in Redbud; since a new directory is always allocated in a different group in Ext3, traversing a deep directory in Lustre therefore needs to move the disk head all over the place; Redbud, on the other hand, constrains a user's directory in an index subtree, effectively decreasing the number of disk positions. Finally, the symmetric block-based file system, pNFS, performs worst in all cases; by further analyzing the characteristics of disk access, we find that metadata accesses heavily interleave with data accesses in pNFS, because pNFS colocates metadata and file data on the same disk; since these interleaves potentially squeezes out file system's metadata from disk cache, the performance of metadata access, which is critical under these workloads, is seriously impacted.

6.4 Layout Management

We also chose two widely used benchmarks: IOR2 [20] and BTIO [21], which are MPI applications and used by parallel file system vendors and users, to evaluate the effectiveness of layout prefetching and on-demand preallocation algorithm. IOR2 and BTIO launches MPI processes to concurrently write a large amount of data to one file (each of m MPI processes is responsible to write 1/m of a file),then read its own written data back to verify the correctness of the data; as soon as the written data is available, each process reads the data written by the other processes. The average request size of IOR2 (more than 32 KB) is larger than that of BTIO (less than 16 KB). We also profiled the executions of them using either non-collective I/O or collective I/O, and the size of collective-I/O requests is around 40 MB, much larger than the size of requests with non-collective I/O.

6.4.1 Layout Prefetching

In this section, we compare ALPF of Redbud with two traditional prefetching approaches: *naive* and *token* algorithm. For the naive approach, any *write()* request will enforce client to conservatively get the *precise* layout range to be written [4], [19]; for the token approach, a node accessing a file will aggressively acquire a range token for the whole file, and when the other nodes write the same file, MDS needs to revoke the part between the requesting offset and the last byte held by the former [3], [5], [7]. Revoke requests in all algorithms in the client is handled by a specific thread.

We first conduct experiments on one PAG and run an IOzone worker on a client to generate exclusive access pattern. Fig. 8a shows the read/write throughput of Redbud as the request size increases. Since the token algorithm allows the client to cache the layout of the whole file locally, it is obviously the best case for the exclusive access pattern, and outperforms the ALPF about 5-11 percent. However, ALPF significantly improves the performance than conservative naive approach by predicting the layout request ranges, especially with small request size; improvement is decreased as the request size becomes large, this is mainly because the network interaction overhead incurred by getting layout is amortized into huge disk request latency.

We next evaluate the concurrent access performance, using IOR2 benchmark. As shown in Fig. 8b, since the number of the network interaction is increased as more work threads join in the experiments, performance of all algorithms is decreased. ALPF outperforms token algorithm about 4-40 percent. The reason is that clients change their write range frequently in the workload [11] and token algorithm brings a large number of revoke messages into the system. Comparison of CPU utilization in MDS using different algorithms is also shown in the Fig. 8b. The results indicate that, ALPF adds less cost to the MDS than token algorithm. This is because our ALPF implementation leverages the clients to predict the prefetching range, reducing the CPU overhead of MDS.

6.4.2 On-Demand Pre-Allocation

To evaluate the effectiveness of on-demand pre-allocation, we also compared Redbud to: (1) a Redbud version using reservation method for file data preallocation; (2) Ceph, Lustre and pNFS, which use delay allocation, Ext3 and Ext4 reservation method, respectively.

As shown in Fig. 9a, Lustre performs similarly to the Redbud version using reservation method, and runs with OD pre-allocation maintain higher throughput (about 18 percent) than Lustre by mitigating the file fragmentation. Besides, Redbud outperforms Ceph by up to 56 percent; the main reason is that, in these runs, since the written data generated by a client needs to be read by the other clients, the newly written data must be flushed into disks immediately, thus reducing the effectiveness of delayed allocation in

TABLE 2
Statistical Results in Runs

Mode	Apps	Segs	MDS CPU Util
Vanilla	IOR2	2023	3.2%
	BTIO	1332	5.1%
Reservation	IOR2	1242	3.0%
	BTIO	701	4.2%
On-demand	IOR2	231	1.1%
	BTIO	106	1.0%

Ceph. Compared with BTIO, the improvement for IOR2 is smaller, as shown in Fig. 9b. This is because that, in IOR2, the request size is larger, and each process accesses adjacent data in its access scope. We can also find that the program's throughput with collective I/O performs is much better than its non-collective version because the latter version generates smaller requests; this may makes the effectiveness of on-demand preallocation being disappointed in this case.

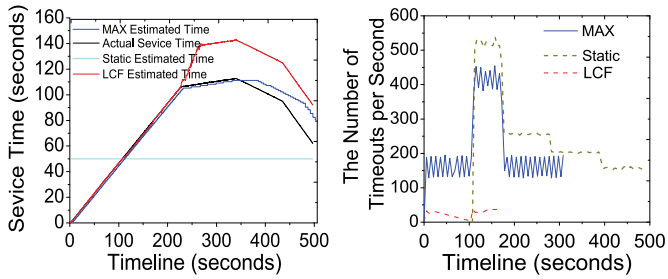
In the runs, we also measured the metadata overhead of Redbud using different pre-allocation algorithms. Table 2 shows the number of segments generated by the programs and the average CPU utilization of MDS without using collective-IO. The *Vanilla* mode indicates pre-allocation is not used. The results show that, with Vanilla mode, files are severely fragmented, and the runs generate significantly more segments than that of the other modes. We can also observe that *OD* approach has the potential to reduce the segments count (for both reads and writes) by a factor of 5-10 compared to the same file system with traditional *reservation* pre-allocation. With the less segments to be operated in a block-based parallel file systems, the less CPU load is involved in MDS as revealed from the results. Since the increased metadata overhead causes less efficient mapping, we expect more benefits can be gained from OD pre-allocation in these programs as the system scales.

6.5 Handling Timeout

We then evaluate LCF-based adaptive timeout handling mechanism. To evaluate the effectiveness of LCF-based adaptive timeout algorithm in large scale system, we used the simulator [31] developed for parallel file systems to simulated 32,000 clients to access a MDS. We generated workload to mimic the burst workload of typical the data centers [49] [11] [5]: we divided the clients into 400 groups and every group contained 80 clients; and in the first 200 seconds of our experiments, every client in one group sent 100 metadata requests (*create* and *truncate* operation which enforce MDS to write layout's changes into MFS) to MDS every second and then waited for responses.

We compared our adaptive timeout algorithm with: (1) *Static* method, in which the timeout value is 30 seconds for every RPC requests as the modern file systems [3], [4], [5]; and (2) *Max* method mentioned in Section 5.1.

Fig. 10a shows the estimated service time using different methods. In the initial seconds (0-240 seconds) of the experiment, the actual service time increases rapidly, this is because in this stage the clients continuously send requests and MDS queues most of them. We can also find that the service time predicted by the LCF is slightly larger than the actual service time at most time, and it can vary the estimated service time as the workload evolves. The MAX



(a) Estimated service time. (b) The number of timeouts

Fig. 10. Comparison Results of different timeout handling approaches. *LCF uses 50 seconds for the size of its time window that is divided into 10 sub-windows. The actual service time in the figure indicates the accurate service time for every request.*

method, on the other hand, generates smaller prediction values than the actual service time in the initial seconds; this is because in most time, it obtains the maximum value from the past service times, which are always smaller than the service time to be increased. These prediction deviations of every method directly lead to different timeout ratio: Fig. 10b shows the average number of the detected timeout along the timeline. The Static method can only completely eliminate timeouts in the initial 110 seconds. When the actual service time increases rapidly, MAX method incurs more timeout than LCF because it leads to high false positive rate using the smaller service time. We can also find that LCF approach still brings some timeout in the initial 120 seconds. This is mainly because the records kept in the sub-windows are the average values of past requests. To address this problem, we may perform more accurate prediction by fitting the curve using the service time of every request instead of the average value in sub-windows; however, this may introduce more memory and CPU overhead because of a mass of requests in burst workloads.

7 Related Work

Asymmetric parallel file systems. Some file systems designed around NASD such as Lustre [5] uses general-purpose local file system (Ext3) to build its namespace; PVFS2 [29], which is based on file-storage, employs the Berkeley-DB to store and index metadata and distributes file data on multiple storage servers. HDFS [26] and Google File System [8], which are also based on the file-storage, store their metadata in a dedicated name node and all file data is placed in the data servers. Since asymmetric architecture isolates the metadata in MDS access domain, it can eliminate the probability of damaging the metadata by buggy software installed on clients. We believe this elimination is especially useful to the SAN environments where any nodes can touch the shared disks using the conventional block interface.

Traditional block-based file systems for SAN environments are in *symmetric* fashion: CXFS [25] is the cluster version of XFS [14] that allows multiple nodes to access data on shared disks; implementation of pNFS in block-mode for Linux [4] exports Ext4 to achieve parallel access based on standard NFSv4.1 protocol. Both GFS2 and GPFS [7] distribute all data by a "network storage pool" layer; to use multiple disks, a logic volume is often configured and all metadata and data is blended in the volume. In this paper,

our experiments demonstrate that the asymmetric block-based parallel file system can gain data access scalability as well as the traditional symmetric ones.

Adaptive mechanisms: To maintain the POSIX read/write atomic semantics, two traditional approaches are often used: Token and Naive algorithm are used in [5], [7] and in [4], [19], respectively. Redbud introduces adaptive approaches and our experiments demonstrate that these approaches make a good tradeoff between the gained performances of exclusive and share access pattern.

Reservation pre-allocation algorithm is widely used in most parallel file systems [4], [5], [19]. Since this approach is not aware of the concurrent write streams, it leads to the file fragmentation in parallel application. The other widely adopted approach uses the ideas borrowed from LFS: the object storage servers in Ceph [3] aggressively perform copy-on-write. Assuming that free blocks are always available, this approach works well for write activity. Our experiments have shown that the performance of read traffic can be compromised in many cases. Delayed allocation is also proposed in these file systems to postponed allocation to page flush time [14]. However, it does not fit application with explicit sync requests. On the other hand, our experiments show that on-demand pre-allocation effectively mitigates more fragments than these approaches.

Most parallel file systems implement the static timeout algorithm in their protocol to handle server failure [1], [4], [6], [8]. However, as shown in our experiments, the static approach incurs a mass of false positives in burst metadata workloads and our adaptive timeout mechanism overcomes this problem.

Metadata storage. Classical local file systems have proposed to place related metadata nearby on disk to explore access locality [15], [16]. However, the goal of embedded directory is to place *all* related metadata as contiguously as possible. Ceph [3] embedded inode into the directory to decrease the network interaction. This embedding works well in object-based file systems. However, by also stuffing the file mapping in the directory content, our work on embedded directory seeks an approach to target block-based parallel file systems which must explicitly use file layout mapping. Most parallel file systems [3], [5], [6], [7] opt to explore the directory locality of the workload; Redbud also carefully integrates the user's information into the index to expect to exploring the runtime locality.

8 Conclusion and Future Work

In this paper, we present and evaluate an asymmetric parallel file system. Unlike the traditional block-based file systems for SAN environments, Redbud decouples data and metadata storage to improve system reliability, and we are glad to see this architecture can gain scalable performance.

We tackle the challenges of asymmetric architecture in providing high performance and availability. Towards overall performance end, we introduce the embedded directory to reduce the number of disk accesses effectively, and two adaptive layout algorithms to make system adaptive to various access pattern; to improve system availability, we also propose the LCF-based timeout algorithm to predict the service time by taking the current MDS's current load into consideration, reducing the false-positive of traditional approaches. Our experiments demonstrate the effectiveness of our design and implementation.

Since the asymmetric architecture allows us to leverage the solid state drives (SSD) [24] to improve the performance of metadata access, we plan to optimize the performance of MFS built on SSDs. Although embedded directories and index tree potentially decrease the number of SSD writes that suffer from large latency, we intend to further improve the MFS performance on SSD by distinguishing the hot and cold metadata in MDS at runtime. The distinguishing allows us to co-locate the cold metadata in the same page of SSDs, avoiding the frequent erasure of the page because of overwriting the hot metadata.

Besides, we are working on developing a quality of service architecture to allow aggregate class-based traffic prioritization [3]. A number of other enhancements are also planned, including improved allocation logic and checksums or other bit-error detection mechanisms to improve data safety.

Acknowledgments

The authors would like to thank the anonymous reviewers for their suggestions on improving this paper. This work was supported by the National Natural Science Foundation of China (Grant No. 60925006), and the National Science and Technology Support Plan of China(-Grant No.2011BAH04B02), and the National High Technology Research and Development Program of China (Grant No. 2012AA011003).

References

[1] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow, "The NFS Version 4 Protocol," *Proc. Second Int'l System Administration and Networking Conf. (SANE '00)*, p. 94, 2000.

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

[3] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, and C. Maltzahn, "Ceph: A Scalable, High-Performance Distributed File System," *Proc. Seventh Symp. Operating Systems Design and Implementation (OSDI '06)*, 2006.

[4] http://www.pnfs.com, 2014.

[5] P.J. Braam, "The Lustre Storage Architecture," http://www.lustre.org/documentation.html, Cluster File Systems, Inc., 2004..

[6] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, and B. Mueller, "Scalable Performance of the Panasas Parallel File System," *Proc. Sixth USENIX Conf. File and Storage Technologies (FAST '08)*, pp. 17-33, 2008.

[7] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," *Proc. Conf. File and Storage Technologies (FAST '02)*, pp. 231-244, 2002.

[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP '03)*, pp. 29-43, 2003.

[9] SAN Zoning Methods, http://www.comptechdoc.org/docs, 2014.

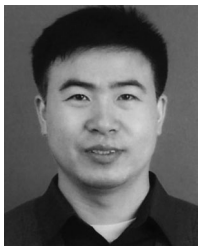
[10] A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*, vol. 2, no. 1, pp. 39-59, 1984.

[11] 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," *Proc. 21st IEEE Conf. Mass Storage Systems and Technologies (MSST '04)*, pp. 139-152, 2004.

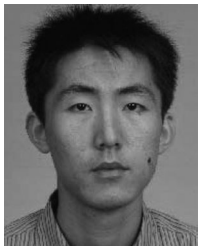
[12] A.W. Leung, S. Pasupathy, G. Goodson, and E.L. Miller, "Measurement and Analysis of Large-Scale Network File System Workloads," *Proc. USENIX Ann. Technical Conf.*, pp. 213-226, 2008.

[13] http://www.iozone.org/, 2014.
[14] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS File System," *Proc. USENIX Ann. Technical Conf.*, 1996.
[15] H. Reiser, "ReiserFS," www.namesys.com, 2004.
[16] S. Tweedle, "EXT3, Journaling File System," July 2000.
[17] M.D. Schroeder and M. Burrows, "Performance of Firefly RPC," *ACM SIGOPS Operating Systems Rev.*, vol. 23, no. 5, pp. 83-90, Dec. 1989.
[18] D. Roselli, J. Lorch, and T. Anderson, "A Comparison of File System Workloads," *Proc. USENIX Ann. Technical Conf.*, pp. 41-54, June 2000.
[19] http://sourceware.org/cluster/gfs/, 2014.
[20] http://visa.cis.fiu.edu/tiki/IOR2, 2014.
[21] http://www.nas.nasa.gov/Resources/Software/npb.html, 2014.
[22] M.M. Swift, B.N. Bershad, and H.M. Levy, "Improving the Reliability of Commodity Operating Systems," *ACM Trans. Computer Systems Archive*, vol. 23, no. 1, pp. 77-110, Feb. 2005.
[23] F. Zhou, J. Condit, Z. Anderson, and I. Bagrak, "SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques," *Proc. Seventh USENIX Symp. Operating Systems Design and Implementation (OSDI '06)*, 2006.
[24] http://www.samsung.com/Products/Semiconductor/SSD, 2014.
[25] SGI CXFS Clustered File System, Datasheet, SiliconGraphics, Inc., 2000.
[26] http://hadoop.apache.org/, 2014.
[27] www.cisl.ucar.edu/css/software/metarates/, 2014.
[28] F. Wu, H. Xi, and C. Xu, "On the Design of a New Linux Readahead Framework," *ACM SIGOPS Operating Systems Rev.*, vol. 42, pp. 75-84, July 2008.
[29] www.pvfs.org/, 2014.
[30] A. Kadav, M.J. Renzelmann, and M.M. Swift, "Tolerating Hardware Device Failures in Software," *Proc. 22nd ACM Symp. Operating Systems Principles (SOSP '09)*, 2009.
[31] Y. Qian, E. Barton, T. Wang, N. Puntambekar, and A. Dilger, "A Novel Network Request Scheduler for a Large Scale Storage System," *Computer Science*, vol. 23, no. 3/4, pp. 143-148, 2009.
[32] http://www.enterprisestorageforum.com/storage-technology/4-storage-technologies-lost-to-the-recession.html, 2014.
[33] G.A. Gibson, D.F. Nagle, K. Amiri, J. Butler, F.W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka, "A Cost-Effective, High-Bandwidth Storage Architecture," *Proc. Eighth Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 92-103, Oct. 1998.
[34] G. Sivathanu, S. Sundararaman, and E. Zadok, "Type-Safe Disks," *Proc. Seventh Symp. Operating Systems Design and Implementation (OSDI '06)*, 2006.
[35] http://www.web-polygraph.org/, 2014.

Letian Yi received the bachelor's and master's degrees in computer science from the National University of Defence Technology in 2005 and 2008, respectively. He is currently working toward the doctoral degree in the Department of Computer Science and Technology at Tsinghua University. His current research interests include mass storage, parallel file systems, computer networks, and distributed systems.



Jiwu Shu received the PhD degree in computer science from Nanjing University in 1998. In 2000, he finished the postdoctoral position research at Tsinghua University and has been teaching at Tsinghua University since then. He is now a professor in the Department of Computer Science and Technology, Tsinghua University. His current research interests include cloud storage, performance, security and reliability for storage system, parallel/distributed processing technology.



Youyou Lu received the bachelor's degree from Nanjing University in 2009. He is currently working toward the doctoral degree in the Department of Computer Science and Technology at Tsinghua University. His current research interests include mass storage, parallel file systems, and distributed systems.

Ying Zhao received the bachelor's degrees in computer science from Peking University and the PhD degree in computer science from the University of Minnesota. She is currently an associate professor of Tsinghua University. Her current research interests include data mining, performance and reliability of parallel/distributed systems.

Weimin Zheng received the master's degree from Tsinghua University in 1982. He is the director of the Institute of High Performance Computing Technology, Department of Computer Science and Technology, Tsinghua University. His research covers parallel computer architecture, parallel computing, compiler techniques, and network storage.

Yinjin Qian received the master's and PhD's degrees in computer science from the National University of Defence Technology. He is currently a researcher in Oracle. His current research interests include performance and reliability of parallel/distributed systems.