Accelerating Distributed Updates with Asynchronous Ordered Writes in a Parallel File System

Youyou Lu*, Jiwu Shu*✉, Shuai Li†, Letian Yi*
*Department of Computer Science and Technology, Tsinghua University, Beijing, China
Email: luyy09@mails.tsinghua.edu.cn, shujw@tsinghua.edu.cn, lonat.front@gmail.com
†Department of Computer Science and Technology, Jiangsu University, Zhenjiang, China
Email: lishuai.ujs@163.com

***Abstract*—Ordered writes mechanism is an efficient and widely used way to guarantee the consistency of distributed updates in a parallel file system. To keep the write order, remote commit operations should not be sent out until the local updates are forced to be stable. However, this can block the execution of applications and significantly degrade the overall performance. Thus, the I/O and network latency of commit requests serve non-negligible cost for file updates, especially for large amount of small files.**

In this paper, we argue that the write order keeping can be handed over from the applications to file systems i.e. the order keeping can be removed from the critical I/O path of applications. We propose the Delayed Commit Protocol that the requests of committing sub-operations are submitted to the commit queue and in the meanwhile the execution flow can be returned back to applications immediately. To reduce the total I/O and network overhead, we use space delegation and adaptive RPC (Remote Procedure Call) compound techniques. Experiments show an up to 2.6x speedup while applying such protocol in a CDN (Content Delivery Network) benchmark. No performance degradation occurs for workloads with large files or conflicted operations.

***Keywords*-parallel file systems, distributed updates, consistency, performance**

I. Introduction

Applications from both data intensive computing and high performance computing demand the scalable access to large and single namespace storage resources. Parallel file systems are getting widely used not only for the performance increasing by parallel accessing mechanism but also for the easy management of large amount of data in a single namespace across multiple nodes. Data and metadata of files as well as file system metadata are distributed to different nodes in a cluster for fast and scalable processing. It is obvious that file operations, such as file creates or file writes, involve many sub-operations on several different nodes. The sub-operations compose a complete logical operation that should be considered in a file system as an atomic operation. In other words, either all or none of the sub-operations should be successfully applied to the file system. Otherwise, the file system will be left inconsistent. Unfortunately, it is expensive and complex to maintain the consistency of such distributed operations.

Some approaches, such as two-phase commit (2PC) and ordered writes, have been proposed for maintaining the consistency property of a distributed file system. While two-phase commit strictly maintains the atomicity property at the cost of complexity and long latency, which is mostly used for distributed metadata operations such as the work done by Slice[1] and Archipelago[2], ordered writes mechanism is more popular as it trades consistency for performance, allowing 'orphan' data. In implementation, files are indexed in a tree-like structure in most modern file systems. The partial order property of the tree-structured file data organization makes it possible to maintain the consistency by keeping the order of sub-operations. The file data will be made durable locally before issuing the metadata commit operations. Thus, even if the system crashes in between the two sub-operations, the file system can still be kept consistent as the 'orphan' data cannot be accessed without corresponding metadata. They can be recycled with garbage collection. Ordered writes approach offers a slightly weaker but acceptable consistency, and is considered as an attractive way to provide the consistency in distributed operations for parallel file systems such as PVFS2[3] and Redbud[4].

In ordered writes, a serial execution is required for consistency in parallel file systems. A typical write operation consists of a network round trip and two disk write operations, for data and its metadata respectively. The file data has to be synchronously flushed to disks before the metadata sent out to the metadata server. If the order is not kept properly, in the case of server crashes after metadata written without file data synchronized, the metadata will describe some invalid or not available data which means the file system is inconsistent and almost sure to cause unexpected results. Thus, the ordered writes mechanism can cause long latencies of distributed operations and bring performance degradation especially for updating large amounts of small files. Considering small data updating such as updating a typical 4KB page size data, the overhead incurred in order to keep strict serial order is sure to be large comparing to the merely updating of 4KB data.

Write back cache policy is quite common in a file system, which tremendously hides the write latency. However, such

978-0-7695-4807-4/12 $26.00 © 2012 IEEE
DOI 10.1109/CLUSTER.2012.38

method cannot be applied while meeting the distributed operations in parallel file systems building on top of multiple nodes. Previous work, such as soft updates[5] and patch dependencies[6], has defined the update dependencies of writes with high complexity. But as the sub-operations are distributed across different servers, the dependencies are hard to achieve with in-memory data structures. Instead, the metadata commit request can be sent only after local writes are flushed to stable storage, which makes the ordered writes synchronous. Considering the discussed issues here, write back policy is not suitable anymore as the buffer manager fails to provide the order of updates in a parallel file system. Thus, file data synchronization will block applications waiting for I/Os to complete.

Since the order of sub-operations is kept by the file system as the file system often uses the application thread context to process the operation[1], disk I/Os and network requests from different operations are often unrelated which could prevent the file system from merging I/O requests or compounding network requests. In fact, the possibility of I/O merge is determined by applications i.e. with the number of application threads increases there are more opportunities for I/O merges. As each network request is sent by one specific application thread, no network requests from multiple files can be combined together. In the very situation of large numbers of small file updates in parallel file systems, there will be plenty of synchronous writes and network requests, leading to poor performance.

Based on the above observations, we propose to hand over the write order keeping task to background file system daemons. This can release applications from synchronously waiting for I/Os to complete. With the delayed commit protocol, we are able to increase the possibility of merging small I/Os and combine RPCs over the network, improving the storage device throughput as well as reducing the network traffic. In this paper, we have made the following contributions.

- We propose Delayed Commit Protocol, which hands over the order keeping to file systems, releasing applications from synchronously waiting.
- We employ the space delegation technique to cluster the space allocated for each client and increase the I/O merge possibility in multi-client parallel file systems.
- We also use the adaptive RPC compound technique to adaptively adjust the number of commit threads and the compound degree according to the client workloads, and the statuses of the network and the metadata server.

In the rest of this paper, we first express the background and motivation of our work in Section 2. We present Delayed Commit Protocol in Section 3 and two techniques, space delegation and adaptive RPC compound, in Section 4. Section 5 gives the implementation and evaluation results. We present related work in Section 6 and conclude in Section 7.

[1]In this paper, we want to distinguish the threads borrowed from the application and the threads spawned by the file system in the kernel. For convenience we call the former as application threads and the latter as file system threads.

II. Background and Motivation

Traditional file systems are passive: file system operations are driven by applications through system calls. The application threads trap into kernel space through system calls from user space, and then follow the file system logic for executing corresponding operations. For example, when the application is about to read, it sends a read system call, traps into the kernel, and executes until the data is returned from the stable storage. It is the application that drives and waits for the operation done. Even with data prefetch enabled, any data prefetch operation is affiliated with a read operation. No file system thread exists to actively prefetch data.

The passiveness of file systems limits the functionality of the file systems, especially for parallel file systems. Lack of global view, each operation is passively driven, losing the chances to cooperatively work with other operations. Network requests from different system calls hardly have the chances to be compounded into one request, for that each request is generated within a different system call. In read or write operations, each application thread is responsible for its own I/Os, reducing the possibility of multi-operation or multi-file I/O merge. However, an active file system, whose operations are taken using daemon threads, can make an optimization from a global view of file system operations. Applications submit read or write operations to the file system, and the file system itself determines when and how to execute these operations. In such a scenario, the file system feels more flexible to execute operations with knowledge of workload I/O characteristics, network or the metadata server statuses, thus compounds nearby network requests, and merges I/Os from multiple files.

Also, some functionality provided by applications, such as order keeping, hurts performance, whereas a daemon thread in an active file system can easily take over the responsibility to keep the order, freeing applications from waiting. The file system keeps the order, making write back policy work when updates have dependencies. Applications submit the read/write operations to the file system and continue to the next steps if the read/write result is not needed immediately. After the file system actually finishes these operations, it acknowledges the applications with the result.

In parallel file systems, each application thread issues an I/O operation to the storage device for each small file update. Since each small file update requires the order of updating file data followed by updating the metadata, the application thread has to wait for the completion of file data updating before issuing the metadata commit request. Network latency is another cause for operation latency besides I/O latency[8].

I/O and network requests from multiple files are independently submitted and the application is forced to wait I/O for order keeping. We propose to hand over order keeping to the file system with daemon threads, making the file system active to execute the operations themselves. We also employ space delegation and adaptive RPC compound techniques to enforce the I/O and network requests merge opportunities from multiple file updates.

III. Delayed Commit

To free the applications from synchronously waiting to keep the write ordering, we hand over the ordering keeping task to the file system daemons by delaying the commit requests. With delayed commit, I/Os and computing achieve more parallelism. Also network requests and I/Os from multi-files gain more aggregation opportunities.

A. Delayed Commit

In delayed commit, applications issue the update requests, typically a local data write followed by a remote metadata commit request, to the file system, and then the applications return. Issued commit requests are inserted into the commit queue if no commit request of this file resides in. Since commit requests of the same file share the in-memory metadata structure, one commit request is enough to commit the metadata of each file. In the meanwhile, a background commit daemon periodically checks out the local file data write completed requests and sends out the remote commit RPC requests. Thus, the synchronous writes and the order keeping are handed over to the parallel file systems without holding back the applications.

In traditional synchronous ordered writes, which we call synchronous commit, the synchronous writes lie in the critical path of the update system call. The steps of I/Os and RPCs during updates are as follows:

1) A local write request to write file data is issued to the storage device with $writepage$.
2) Completion flag is checked in a loop to wait for the write completion.
3) Remote metadata commit RPC is constructed and sent to the metadata server after local write completion.
4) The update succeeds and returns after receiving the RPC reply.

Delayed commit removes the synchronous local write from the critical path of an update operation, and hands it over to the file system as background processing. As soon as the commit task is inserted into the commit queue, the update operation returns, and the application continues running. In order to make local storage devices busy and limit the total commit latency, local data writes are issued with $writepage$ in Linux kernel before the commit request inserted into the commit queue. The following lists the steps of delayed commit.

1) A local write request to write file data is issued to the storage device with $writepage$.
2) The commit request is inserted into the commit queue if no commit request of the same file exists;
3) The update request returns without waiting for completion by handing over the commit task to the file system.
4) Commit requests are handled periodically by background commit daemons.
    a) The background daemons check out the local I/O completed requests.
    b) A compound RPC for remote metadata updates of those requests checked out in the last step is constructed and sent to the metadata server.
    c) The commit succeeds with an RPC reply.

Delayed commit hides the disk latency from the application, which is similar to write back. What is different from write back is that delayed commit keeps the write order in the system background and enlarges the write back chances. Thus, delayed commit gains more by leveraging the client cache. However, due to the volatile characteristic of memory cache, the cached data or metadata may be lost after system crashes. Like soft updates[5] and patch dependencies[6], applications that cannot afford data loss should explicitly call fsync or open the file with SYNC flag. Delayed commit exploits the advantage of memory buffer by keeping the write order with a background daemon. Since the write order is kept, the system remains in the consistent state even after system crashes.

B. Parallelism of Computing and I/Os

In synchronous commit, I/O requests are sent to the block devices followed by a loop to check I/O completion status, which acts as a barrier. As shown in Figure 1(a), the running sequence is restricted to write-barrier sequence, causing dramatic performance degradation. In delayed commit, applications take on the next computing tasks immediately after submitting the I/O requests without any barrier. The file system background daemons take the responsibility of the order keeping to perform I/Os and send commit RPC requests. Applications are freed from I/O waiting, providing the parallelism of computing and I/Os, as shown in Figure 1(b).

More importantly, the possibility of I/O merges increases as the number of issued requests grows. In most cases, small writes are file appends or new file writes, which require space allocation. The allocation policy prefers to allocate new space nearby, which improves data locality. As a result, I/Os are merged to reduce the seek time, which is known to be one source of overhead for small writes.

IV. I/O and RPC Aggregation

Delayed commit allows more file updates submitted at the same time, and thus increases the possibility of I/O merge

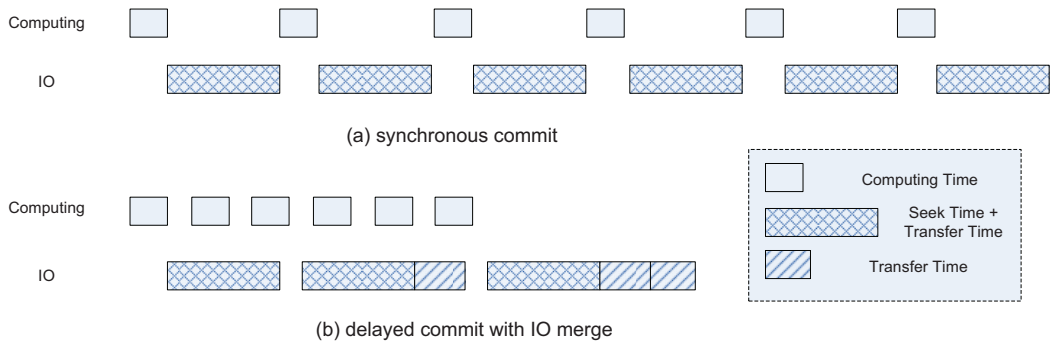


Figure 1. I/O Merge in Delayed Commit

and RPC compound. However, parallel access from different clients simultaneously causes space segments in each client, which decreases the possibility of I/O merge. In delayed commit, we use space delegation, in which each client keeps a continuous physical space, to alleviate the problem. In addition, the adaptive RPC compound technique is also used to reduce the network traffic according to the statuses of the network and the metadata server.

A. Space Delegation

Space allocation requests are often handled by the metadata server (MDS). It is difficult for the MDS to differentiate requests from different clients and allocate adjacent space for requests from the same client. As a result, the physical addresses allocated for successive I/Os often scatter over a large space. Instead, we use the space delegation technique to allocate a chunk for each client in advance and grant the space allocation for small files to each client. Thus, the write operations in a short time window get the continuous physical addresses, increasing the possibility of I/O merge.

We maintain a double-space-pool in each client to manage the delegated space. The two pools are used exchangeably, one active, and the other standby. The active pool serves the current space allocation requests until the free space is not large enough for the running request. Then, the standby pool turns to be the active one, and the former active pool changes to the standby with the space-need flag set. The next layout-get operation will get the new delegated space for the client.

In space delegation, the free space in double-space-pool is guaranteed to be enough for any single allocation. Large file requests, whose request size is larger than the chunk size, apply for the physical space directly from the MDS. Only the small file update requests are served locally. Before each request is sent to the MDS, the free space is checked. If not enough, the space request flag is set and the new space is delegated. Since the request size is smaller than the chunk size, it is enough to allocate space locally.

B. Adaptive RPC Compound

Network congestion is another cause that usually slows down the performance of parallel file systems. Either the metadata server is hard to process floods of requests, or the cluster network becomes congested and causes retransmissions. Network traffic boosts when all the writes are small. Since each update needs an RPC round trip, the network throughput, measured with RPCs per KB, is extremely low. Delayed commit grants the commit task to the file system, and provides the chances to compound RPC requests from different file operations.

In delayed commit, we adjust both the number of commit threads and the number of network requests compounded to one RPC request, a.k.a. the compound degree of an RPC, to adaptively match the client workloads and the network or MDS traffic. The number of commit threads varies in the commit thread pool with the length of commit queue. A new commit thread spawns when the length increment exceeds a threshold. When the length of commit queue decreases, a certain thread terminates to keep proper thread numbers. The thread numbers are kept as follows:

$$ThreadNums_{cur} = \rho * QueueLen_{cur},$$

where $\rho = ThreadNums_{max}/QueueLen_{max}$. More commit threads are spawned when more commit requests are inserted into the commit queue. The commit threads compete for more schedule time for sending committing RPCs while slowing down the incoming commit requests.

Commit thread pool is used to adaptively maintain the chances to issue commit requests, aims to match the ratio of incoming requests. Compared with commit thread pool, compound degree adjustment focuses on the network traffic and the MDS capacity instead of the workload of a single client. The compound degree changes periodically with the knowledge of the network traffic in the cluster and the workload on the MDS. The compound degree increases as the network is congested or the MDS is busy enough, so as to reduce the RPC requests.

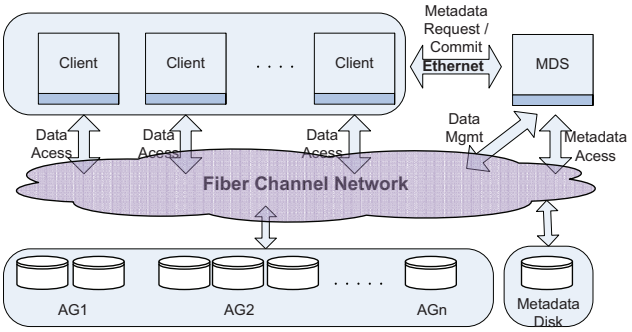


Figure 2. Architecture of the Redbud Parallel File System

V. Implementation and Evaluation

A. Redbud Parallel File System

We implement delayed commit in Redbud. Redbud is a block-based parallel file system[4]. As shown is Figure 2, the metadata server (MDS) handles the storage and processing of metadata. Each client applies for or commits metadata through network RPCs (Remote Procedural Calls) and directly accesses the data storage devices. The metadata get or commit requests are sent through Ethernet, while the metadata or data reads/writes are performed through the fiber channel network.

The mapping of file logical address to the physical address is represented in the form of <file_offset, length, device_id, volume_offset, state>, which is called an extent. A file may have one or more extents, determined by the continuity of the space allocation. The collection of extents in a certain range of a file is called a layout. The client sends a layout-get RPC request to the MDS before data reads/writes. The MDS reads out existing extents, allocates space for new extents, and returns back the layout to the client. After the client finishes data I/Os, it sends a commit RPC request to MDS for updating the metadata.

The metadata server manages the physical storage resources. All storage devices are divided into allocation groups (AGs). An allocation group is the management unit of storage resources. Each AG has its own B+ tree to allocate and deallocate physical space. Multiple AGs provide parallel allocations. Across AGs, flexible allocation strategies can be applied to the metadata server. The default is round-robin .

B. Experimental Setup

Our experiments are conducted on the Redbud deployed in an eight-node cluster, one for MDS and the others for the clients. Each server has a 3.0GHz Intel Xeon CPU core and an 8GB memory. All the servers connect with each other in a 1000Mbps Ethernet for metadata requests and directly access the disk array with a 4Gb fiber channel for data read/write.

In the experiments, we use the following benchmarks for evaluation:

- Filebench[10] is a macro-benchmark which emulates different enterprise application workloads. Fileserver, varmail, webproxy are three typical workloads emulating file servers hosting files, the mail server, and the web proxy server.
- Xcdn is a benchmark emulating the read/write operations of the servers in the CDN (Content Delivery Network) environment.
- NPB (NAS Parallel Benchmarks) consists of several scientific applications using MPI[11]. We use BT (Block-Tridiagonal) for evaluating parallel I/O[12].

C. Overall Performance

We evaluate the five workloads in three different file systems: PVFS2 (Orangefs 2.8.5), NFS 3, Redbud without delayed commit (the original Redbud), and Redbud with delayed commit. Figure 3 shows the performance normalized to the performance in original Redbud.

Redbud is implemented in kernel space and directly access the disk array through the FC (Fiber Channel) network, which gains higher performance than PVFS2 in most cases except in NPB experiment, whereas PVFS2 has been optimized for MPI-IO. Compared with NFS3, Redbud offers better or comparable performance. The exception is the 32KB xcdn experiment, in which small file writes are randomly scattered over the whole namespace. In this case, client cache is useless and the state maintenance poses a negative effect on the performance. Another reason lies in the distributed updates in Redbud, while NFS3 has no distributed operations and all data or metadata updates are performed by the NFS server. For large file reads/writes, NFS server might be the bottleneck as it has to process all the data or metadata and forward data to clients. Redbud has freed the MDS from data processing, in which clients directly access data through the FC network. Thus, in the case of large file reads/writes, Redbud performs much better. Also, due to the elimination of the central data processing and storage unit like NFS server, Redbud scales much better compared with NFS3.

In small file writes, delayed commit has improved the performance of Redbud. In varmail and webproxy, delayed commit achieves 1.5x performance gains. This is because more operations are issued with asynchronously submitting commit requests for parallel execution, resulting in the shorter average latency for close operations. In 32KB xcdn test, Redbud with delayed commit achieves a 2.6x speedup compared with original Redbud, due to the parallelism of computing and I/Os, and more chances enabled for merging I/Os and compounding network requests, which is close to NFS3. In 1MB files test, delayed commit also brings improvements, which shows that performance does not degrade in large files test.

For NPB benchmark, written data is read out into memory to verify the correctness at the end of the program. The

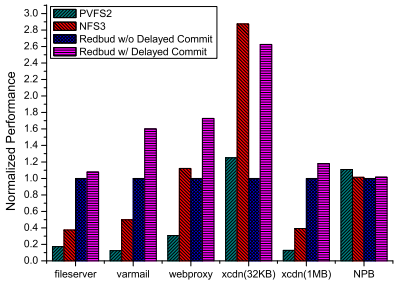


Figure 3. Performance Comparison of PVFS2, NFS3, Original Redbud, and Redbud with Delayed Commit

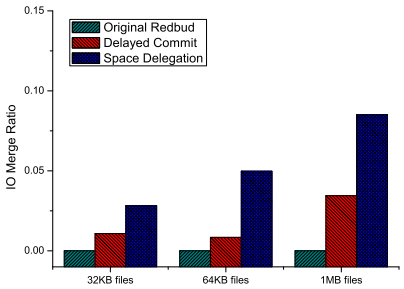


Figure 4. I/O Merge Ratio

read operations may include those requests that haven't been committed, and these read operations are known as conflict operations. In Figure 3, no significant difference appears between the four file system configurations, which shows that delayed commit does not hurt the conflict read operations.

D. I/O Merge

In xcdn experiment, we vary file sizes from 32KB, 64KB to 1MB in three different Redbud configurations: the original Redbud, Redbud with delayed commit but no space delegation, Redbud with delayed commit and space delegation. In this experiment, the space delegation size is set to 16MB. The three configurations are represented with Original Redbud, Delayed Commit, and Space Delegation respectively in Figure 4.

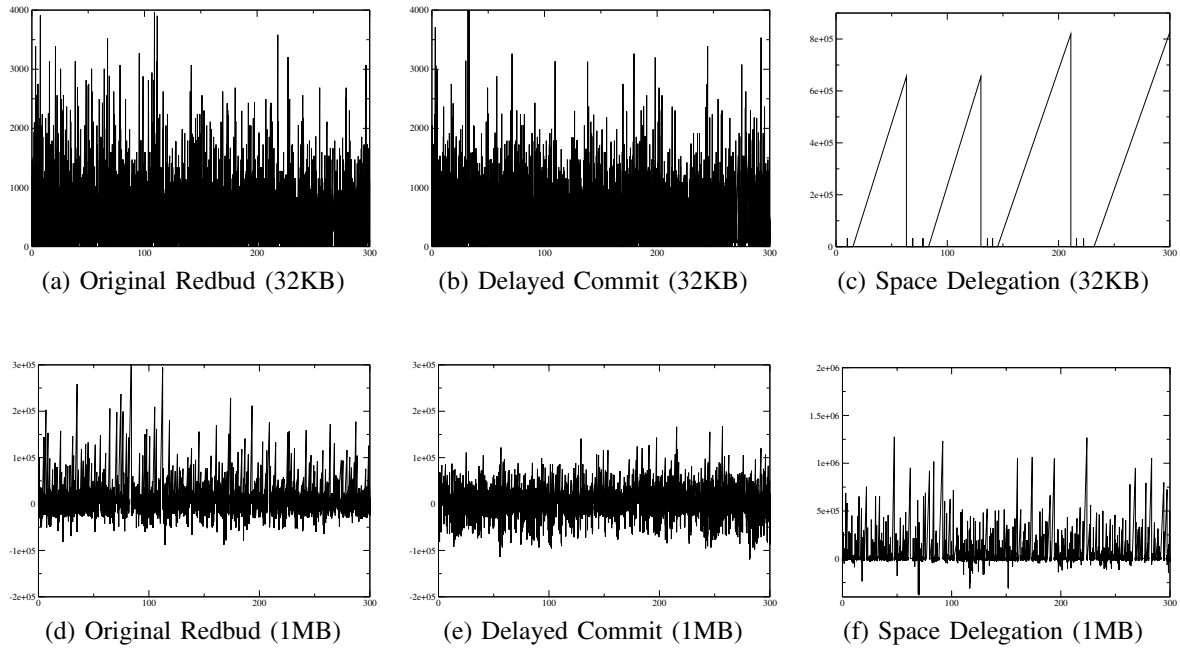
Figure 4 shows that the original Redbud has no I/O merge, while delayed commit brings the I/O merges, and space delegation improves the I/O merge ratio 2.8 to 5.9 times. The increase of the I/O merges ratio in delayed commit comes from the parallel I/O executions. In original Redbud, the order is strictly maintained by application threads, making I/O requests serially executed. Since ordering is kept by the file system in delayed commit, more I/O requests are issued simultaneously, resulting in parallel executions of I/Os and a higher I/O merge ratio. However, in parallel file systems, clients apply for space concurrently, leading to a difficult decision in the metadata server. The space allocated to different clients mix together, reducing the sequential access possibility. In space allocation, the metadata server allocates a continuous space to each client and each client allocates space locally. The write requests in the same client have the space allocated nearby, further increasing the I/O merge possibility, as shown in Figure 4. Larger files have a higher I/O merge ratio, possibly because the large requests reside in I/O queue longer than small requests.

We collect block traces using blktrace for analyzing the changes of block-level I/O characteristics under the three configurations of Redbud. Figure 5(a,b,c) show the results of 32KB file experiments under three different configurations. There is no significant difference between Figure 5(a) and (b), which shows seek operations are still common in delayed commit. It is the parallel execution of I/O requests that contributes to I/O merge ratio. Figure 5(c) exposes few seek operations except some long disk seeks shown as spikes. This proves that space delegation decreases seek operations dramatically. Both parallel execution and the reduction of seek operations contribute to I/O merge ratio. Figure 5(d,e,f) shows the result of 1MB file experiments. The result is similar to 32KB file experiments. Space delegation has increased the possibility of I/O merge by allocating to concurrent write requests the continuous space locally, which is represented with less dense waves in Figure 5(f).

E. Adaptive RPC Compound

In the experiments, we trace the birth and death of the number of commit threads and the commit queue length. The maximum of commit thread number is set to 9. The results are shown in Figure 6: the left Y axis is the number of commit threads and the right Y axis is the commit queue length. It shows that the number of commit threads adaptively changes according to the commit queue length.

In varmail experiment shown in Figure 6(a), there are two spikes at the time 162s and time 351s. At the two spikes, commit requests are flooded to commit queue, causing a sharp increase in commit queue length. The number of commit threads increases, and after a short time the commit queue length returns to normal state. For the other periods, the commit queue length ranges from about 20 to 40, and the number of commit threads changes from 1 to 5 to keep the commit queue length as smooth as possible. Webproxy experiment shows a similar result in Figure 6(c). The commit queue length keeps around 50 with an average commit thread number 5, except that in the last period of webproxy run, the commit queue length increases to more than 400, leading to a maximum commit thread number. Figure 6(b,d) show the results of fileserver and xcdn run. The commit thread number



(a) Original Redbud (32KB) (b) Delayed Commit (32KB) (c) Space Delegation (32KB)

(d) Original Redbud (1MB) (e) Delayed Commit (1MB) (f) Space Delegation (1MB)

Figure 5. Disk Seeks under 32KB and 1MB workloads in Redbud without Delayed Commit, Redbud with Delayed Commit, Redbud with Delayed Commit and Space Delegation

reaches quickly and keeps to the maximum almost all the time as heavy updates appear in the two runs. However, the commit thread number keeps to only one in the NPB experiment as not many commit requests inserted into the commit queue.

Figure 7 shows the performance impact of compound degree by varying the number of server daemon threads. On average, performance is poorer in parallel file systems with fewer server daemon threads. The output of each client with one server daemon thread in the metadata server is around 2.3MB/s, while the output of 8 or 16 server daemon threads reaches to 2.6MB/s. The compound degree of three, in which three network requests are compounded, shows an increase of 0.2MB/s, 0.2MB/s, 0.1MB/s for server daemon threads 1, 8, 16 respectively. A reduction in network requests brings more benefits for metadata servers with fewer server daemons. The floods of network requests make the server with few server daemons busy, thus the compound operations bring more benefits. The compound degree of six shows the same improvement as compound degree three. High compound degree more than three does little help in xcdn experiment, because I/O is slower compared with network requests. Figure 7 also shows degradation in 16 server daemons compared with 8 daemons, which is probably caused by multi-thread contention.

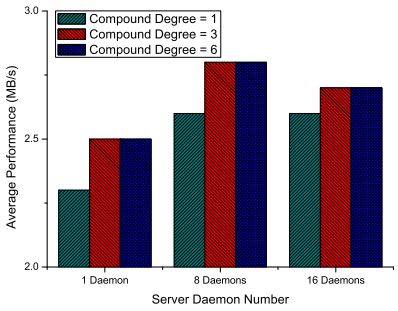


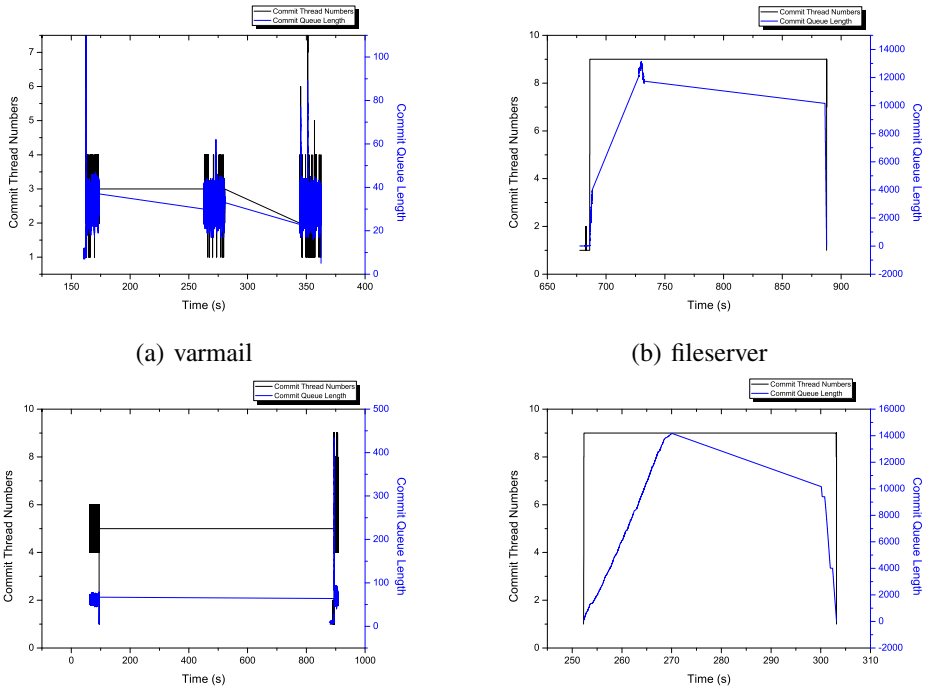
Figure 7. Compound Degree

VI. Related Work

Some work has explicitly defined the dependency in write buffer and use either client or server side buffer to improve write performance. In parallel file systems, I/O or network aggregations are also studied to optimize distributed operations. We will discuss these optimizations in this section.

A. Write Back and Write Dependency

Write back policy is widely used to hide I/O and network latency in parallel file systems. Most parallel file systems, including Lustre[13], GPFS[14], PanFS[15], have implemented client-side buffering. Similarly, NFS v3 implements the server-side buffering by sending asynchronous WRITE requests to the NFS server with a later COMMIT request to flush uncommitted write data to stable storage[9]. In Lustre, each write request requires a journal transaction committed first, which has a negative impact on Lustre performance. Researchers from Oak Ridge National Laboratory (ORNL) use solid state devices to externally store the journals and



(a) varmail (b) fileserver

(c) webproxy (d) xcdn

Figure 6. The Relation of Number of Commit Threads and Commit Queue Length

remove the synchronous journal commit by asynchronously replying to the client first[13].

Write dependencies are quite common in file systems. Soft updates is well known for keeping the order of updates by avoiding duplicated and synchronous writes in file system journaling while maintaining file system consistency[5]. Frost etc have explicitly expressed the "write-before" relationship in the patch structure and made it file system agnostic, which also supports asynchronous writes[6]. Literature[16] employs backpointer-based consistency to avoid ordering writes, but the required consistency checking on read is not suitable for distributed environments.

Similar to soft updates and patch dependency, delayed commit makes the dependency file system agnostic and hands over the order keeping to the file system. But delayed commit focuses on the order of updates in distributed operations in parallel file systems. Delayed commit also enables asynchronous writes, but without changing the order, which is different from asynchronous journal commits in Lustre.

B. Aggregations and Compounds

In high performance computing, parallel applications explicitly reorganize output files using IO libraries to reducing small files performance degradation. Parallel netCDF is a scientific I/O interface allowing parallel writing to the netCDF datasets, thus enables gains from collective I/O optimizations[17]. Adaptable IO system (ADIOS), developed at ORNL, provides developers with a simple API to flexibly select optimal I/O routines with metadata configured in an XML file[18]. However, the benefits of IO libraries are limited to parallel applications in high performance computing, whose outputs are regular and predefined.

Some efforts have been made in file systems to collectively merge I/Os or compound network requests to amortize the overall cost. Since version 3 of NFS, READDIRPLUS has been introduced to eliminate the LOOKUP requests by fetching both file handles and attributes simultaneously[9]. In NFS v4, RPC requests for a single operation are compounded into one RPC, named compound operation, thus reduce the number of network requests[19][20]. Optimizations on PVFS2 mainly focus on metadata operations, for performance and consistency issues. Carns etc. use collective communications across servers, freeing the clients from the consistency checking[21]. In file create operations, metadata handle and metadata setattr operations are compounded in [22] and metadata is delayed to be synced to database when metadata server is busy in [23].

Different from IO libraries, delayed commit provides a general solution to optimize multiple file operations in the file systems. Previous optimizations in file systems focus on the operations of a single system call or in a single file. Delayed commit takes operations from multiple operations and multiple files into consideration, and makes a global decision for optimization.

VII. Conclusion

Maintaining the write order in parallel file systems is costly, which requires applications to wait for the completion of synchronous I/Os. We propose Delayed Commit to make the dependencies agnostic to the file system. The file system keeps the write ordering, releasing applications from synchronous writes. Applications continue its executions with more I/O requests issued simultaneously, which enables the parallel execution and global optimizations. We employ space delegation and adaptive RPC compound techniques to increase the I/O merge ratio and reduce the network traffic. Space delegation pre-allocates a continuous space to each client, and the client allocates space locally. As a result, the writes in each client are clustered nearby, reducing disk seeks and enforcing I/O merges. Adaptive RPC compound technique adjusts the number of commit threads and compound degree according to client workloads and the statuses of the network and the metadata server. Experiments using benchmarks from different aspects show the improvement brought by delayed commit and the benefits from the two techniques of space delegation and adaptive RPC compound.

Acknowledgment

This work was supported by the National Science Foundation for Distinguished Young Scholars of China under Grant No.60925006, the National Science and Technology Support Plan of China under Grant No.2011BAH04B02, and research fund of Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology.

References

[1] D. Anderson, J. Chase, and A. Vadhat. Interposed Request Routing for Scalable Network Storage, In Proceedings of the 4th conference on Symposium on Operating System Design and Implementation (OSDI'00), Berkeley, CA, USA, 2000.

[2] M. Ji, E.W. Felten, R. Wang, and J.P. Singh. Archipelago: An Island-Based File System for Highly Available and Scalable Internet Services, in Proceedings of the 4th USENIX Windows Systems Symposium, August 2000.

[3] PVFS2, http://www.pvfs.org/

[4] L. Yi, J. Shu, Y. Lu, W. Wang, W. Zheng. MiF: Mitigating the Intra-file Fragmentation in Parallel File System, In Proceedings of International Conference on Parallel Processing (ICPP'11), September 2011.

[5] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: a solution to the metadata update problem in file systems. In ACM Transaction on Computer Systems, May 2000.

[6] C. Frost, M. Mammarella, E. Kohler, A. Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized file system dependencies. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP'07). New York, NY, USA.

[7] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In Proceedings of the twentieth ACM symposium on Operating systems principles (SOSP'05). New York, NY, USA.

[8] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G. A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In Proceedings of the ACM SIGCOMM 2009 conference on Data communication (SIGCOMM'09). New York, NY, USA.

[9] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, D. Hitz, NFS Version 3 Design and Implementation, USENIX Technical Conference, 1994.

[10] Filebench, http://sourceforge.net/projects/filebench/

[11] NAS Parallel Benchmarks, http://www.nas.nasa.gov/publications/npb.html

[12] P. Wong, R. F. Van der Wijngaart, NAS Parallel Benchmarks I/O Version 2.4, NAS Technical Report NAS-03-002, January 2003.

[13] S. Oral, F. Wang, D. Dillow, G. Shipman, R. Miller, and O. Drokin. Efficient object storage journaling in a distributed parallel file system. In Proceedings of the 8th USENIX conference on File and storage technologies (FAST'10), Berkeley, CA, USA, 2010.

[14] F. Schmuck, R. Haskin, GPFS: A Shared-Disk File System for Large Computing Clusters, in Proceedings of the Conference on File and Storage Technologies (FAST'02), Monterey, CA, January 2002.

[15] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08), Berkeley, CA, USA, 2008.

[16] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Consistency Without Ordering, in Proceedings of 10th USENIX Conference on File and Storage Technologies (FAST'12), San Jose, CA, February 2012.

[17] J. Li, W. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, M. Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface, in Proceedings of 2003 ACM/IEEE Conference on Supercomputing, November 2003.

[18] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In Proceedings of the 6th international workshop on Challenges of large applications in distributed environments (CLADE'08), New York, NY, USA, 2008.

[19] S. Shepler, B. Callaghan, D. Robinson, et al. Network File system (NFS) version 4 protocol. IETF RFC 3530, April 2003

[20] G. Goodson, B. Welch, B. Halevy, D. Black, and A. Adamson. NFSv4 pNFS extensions. Technical Report draft-ietf-nfsv4-pnfs-00.txt, IETF, October 2005.

[21] P. H. Carns, B. W. Settlemyer, and W. B. Ligon, III. Using server-to-server communication in parallel file systems to simplify consistency and improve performance. In Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC'08). Piscataway, NJ, USA, 2008.

[22] A. Devulapalli, P.W. Ohio. File Creation Strategies in a Distributed Metadata File System, IEEE International on Parallel and Distributed Processing Symposium (IPDPS'07), March 2007.

[23] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, T. Ludwig. Small-file access in parallel file systems, IEEE International Symposium on Parallel and Distributed Processing (IPDPS'09), May 2009.