In-Flight Data Management for Distributed Storage Systems

Kendrick Boyd Carlos Maltzahn
University of California, Santa Cruz
Storage Systems Research Center
boydk@lawrence.edu carlosm@soe.ucsc.edu

September 8, 2006

Abstract

Distributed storage systems suffer performance loss due to increased file access latency when multiple clients share the same file. This is especially noticable with extremely large files which take a long time to transmit between server and client. To increase performance for true and false file sharing we propose in-flight data management: an extension to standard distributed storage where a client's local cache is made available for reading and writing by other clients.

1 Introduction

High performance clusters used for massive simulations often require enormous amounts of storage space that must be available to thousands of clients with minimal file access latency. To provide a large-scale distributed storage system capable of servicing high performance clusters the Storage Systems Research Center (SSRC) at the University of California Santa Cruz (UCSC) is developing Ceph. The ultimate goal is that Ceph will provide multipe petabytes of storage with billions of files ranging from bytes to terabytes in size and accessible by thousands of clients [3].

Ceph relies on several key ideas to provide such capabilities. The first is the separation of metadata and data. Metadata is stored on a small cluster of machines that are solely devoted to metadata. These metadata servers handle all client requests for metadata allowing the data storage devices to only store data and no metadata. The second idea is the use of object based storage. Files are divided into objects and objects becomes the primary structure which is handled by the storage system. Files are mapped to objects using a universally known algorithm called CRUSH which allows clients to automatically know where a specific object is stored once the client has contacted the metadata server and received information regarding the file's striping method. The objects are stored on Object Storage Devices (OSDs) which include a processor and hard drive(s). Since each OSD has a processor, it performs the overhead of mapping the object onto memory blocks and allows the client to only worry about the objects and files. In addition, OSDs automatically back-up objects amongst themselves and monitor each other to identify and recover from OSD failures without interruption in service [3].

Currently, Ceph (like most distributed storage systems) requires all data requests to be satisfied from the OSDs. For many applications and situations this causes no problems, however, when a client opens a file for writing and then other clients want to read or write that data, either clients must wait for the new changes to be written to the OSD or they must work with old data. For a terabyte sized file, the time to transmit the changes back to the OSDs could create an inordinate delay which vastly increases latency. We refer to such data that is being edited by a client and is thus cached locally at the client as "in-flight data". When multiple clients want access to in-flight data, performance immediately drops as clients must wait to obtain write access or for the new data to be available from the OSDs.

We believe that better management of in-flight data will provide considerable performance increases, especially for simulations where the ideal is to have all clients read and write to the same data file. To accomplish this we propose that clients should be able to read and even write data not only to the OSD but also to other client's caches. Adding this peer-to-peer aspect to a distributed storage system should provide a framework which can provide both a general performance boost and customizable implementation variables for fine tuning the storage system to a specific situation.

2 Overview

In this section we outline the design space for an in-flight data management implementation. In all further discussion of in-flight data management we make a few assumptions about the system. First is that locking granularity is by file region, so all open requests must also include a byte offset from the beginning of the file and a byte length or a signal that the file is being opened for appending. After a specific region is opened, the associated file descriptor can only access bytes of the file in that specific region or,if it was opened for append, add data to the end of the file. The second assumption is that all clients store a copy of the entire file region in a local cache after opening a file. Both read only and readwrite opens are cached and all subsequent read or write requests use the local cache. Finally this discussion assumes an object based storage system and is specifically geared toward implementing in-flight data management in Ceph. However, the same concept could be easily adapted for other distributed file system paradigms.

During our investigation we identified several variables for precisely how in-flight data management might be implemented. The variables we investigated, discussed below, all pertain to one of 3 broad categories within distributed storage systesms: forwarding, coherence, or failure recovery.

2.1 Forwarding

Once we decide to make a client's local cache available to other clients we have to determine precisely when another client's cache would be used. The first variable is which types caches can be forwarded to so that the data comes from that cache. The options are to send requests to clients with read only caches, readwrite caches, or both. Sending requests to read only caches only lessens the potential data transfer load at the OSD while forwarding requests to readwrite caches allows the latest write to be read by another client without waiting for the region to be transferred back to the OSD.

The next variable is which types of requests the OSD may forward to clients. Once again the options are read only, readwrite, or both. Forwarding read only requests to a client cache allows the latest changes to be read if the cache is readwrite. If the cache where the data is obtained is read only then the only gain is that the OSD must transfer less data since a client is providing the data. If readwrite requests are forwarded then clients can share or transfer locks with other clients. A dangerous situation arises if readwrite requests can be forwarded to read only caches since that would imply that the read only cache could actually be modified by another client and then those changes would need to be written to the OSD by a client with a read only lock. Hence readwrite requests should not be forwarded to a read only cache.

The third variable for forwarding is when open requests should be forwarded by the OSD. In general, requests will be forwarded if some client has the file region open already (as long as the already opened file region is stored in a cache type that can be forwarded to). However there are 2 options for precisely when requests should be forwarded to a client's cache. The first option is to forward the request only while the data is being written back to the OSD (only applies to readwrite caches). So as a file is written back to the OSD, the OSD would send any open requests (read only) to the client that is writing data to the OSD so that the read only open can be satisfied immediately from the readwrite cache. At any other time, including if a client had a file open and was modifying it but had not yet closed the file, the data (but necessarily the last write) would be available only from the OSD. The second option is to forward all requests to the client from the moment it opens the file until it closes the file and finishes transmitting any changes back to the OSD. This allows any modifications the client might make to a readwrite cache to be immediately available to any other client since other clients are obtaining the data directly from the client's local cache. This second option would have to be used for any read only caches that are being forwarded to since there are no modifications to write to the OSD upon file close.

The last variable for forwarding requires some further discussion about what occurs if a file is heavily shared. As clients open file regions and obtain the data and locks from another client a tree of dependencies is created. We call this a sharing tree since it shows the sharing that is occuring. The final variable is then if any limits should be placed on the size of this sharing tree. The first option is to simply let the tree grow arbitrarily as the opens occur. By putting no limits on the sharing tree it is possible for the tree to become very complex with an extremely large height. This could lead to difficulties, especially with recovery after a client fails. So another option for the sharing tree is to limit it in some way, most likely by specifying a maximum height. Then any open requests that would push the sharing tree over the limit must wait until other clients close so the limit is not crossed.

2.2 Coherence

After a client has obtained a copy of a file region from another client's cache, there are four variables associated with coherence. The first variable is whether the cached copy should be dynamic or static. A static cache is never changed regardless of what modifications other clients might make to the file region that is cached. A dynamic cache will reflect modifications made by other clients. If a static cache is desired then there are no other relevant variables for coherence but if a dynamic cache, which will reflect the latest changes to the data, is required then there are two other variables.

The first of these variables specific to keeping an updated dynamic cache is when changes should be seen by another client. More specifically, if Client 1 has the file region opened for readwrite and Client 2 has the same region cached for read only, how frequently should modifications that Client 1 makes be transmitted to Client 2. The simplest option is that Client 2 is only informed of changes once Client 1 closes the file region. Another option is for every modification that Client 1 makes to be immediately indicated to Client 2. A third option is for Client 2's cache to be synchronized with Client 1 at specific time intervals.

The final variable for coherence is how modifcations are transmitted to a dynamic cache. [1] outlines several such methods used in shared-memory multiprocessors, of which two are useful for this application. Both of these methods assume that Client 1, with the readwrite cache, has a list of all clients with caches that overlap with its own readwrite cache. In the first method, all other clients with overlapping caches are updated every time the modifications should be transmitted (on close, on write, or time interval). So every time a modification is made by Client 1 the changed data is transmitted to Client 2 and any other clients with a read only cache of the changed data. Since the new data might never be read before Client 2 closes the file region, another method is to invalidate the modified region of the cache. With invalidation, Client 1 only notifies Client 2 (and any other overlapping caches) that a small section of the file has changed and thus Client 2's cache for that section is invalid. So the changed data is not transmitted until Client 2 issues a read for the invalidated region.

2.3 Failure Recovery

Distributed storage systems which use in-flight data management are expected to be most prevalent in large high performance clusters. With thousands of clients, failures occur frequently so we need a mechanism to minimize the loss of in-flight data modifications and allow the rest of the cluster to continue to function normally when a client fails. Here we list some features that will facilitate recovering from client failures.

One option is to lazily write modifications up to the root of the sharing tree. This will leave the OSD free of extra network traffic but still allow new changes to be stored by multiple clients when a client is writing to another client's cache. There would then be a system call to allow a selected file region to be flushed to the OSD so that a client could guarantee that its changes are now stored securely on the OSDs.

Another option is for the client to store some portion of the sharing tree with the actual cached data. Besides storing the entire sharing tree for the region, a promising possiblity that we found is to store the sharing tree below the current client and the path to the root. This reduces the size of the tree that is stored but still allows the tree to rebuilt even if the root client crashes.

3 Related Work

In-flight data management brings together elements from 3 different areas. The first are distributed storage systems, such as Ceph (described in detail in [3]), which we propose extending to allow reading and writing to client's buffered caches. [1] provides an excellent overview of the second area which is coherence mechanisms from shared-memory multiprocessors. Finally in-flight data uses a peer-to-peer concept for client interactions by allowing clients to read and write from other client caches in an unstructured manner. [2] also brings together these ideas for another method of implementing in-flight data management in Ceph.

4 Chosen Design

After exploring the design space described in 2, we chose some solution requirements for an in-flight data management implementation in Ceph and selected the options which would best fulfill those requirements. The requirements that we selected for this design are: last write is seen by any subsequent read by any client, reduced latency for opening file regions that are in-flight, scalable, and minimal data loss due to client failures.

4.1 Forwarding

We choose to only forward to readwrite caches but to forward both read only and readwrite requests to the readwrite cache. This allows clients to write to another client's cache and should always reduce latency for opening in-flight data. We do not allow requests to be forwarded to read only caches for two reasons. First, it completely avoids the dangerous situation if a readwrite request is forwarded to a read only cache. Second, forwarding read only is only useful for reducing the load on the OSDs which is not something upon which we are focused for this design and it could be easily added if load-balancing is necessary. Thus read only caches are always leaves in the sharing tree. We only allow a readwrite request to obtain a lock from a client if the entire region that is being opened is contained in a single client's readwrite cache. Further thought would need to investigate what extra problems, especially during failure recovery, might arise if a client could obtain its readwrite cache from multiple clients. However, read only caches may obtain data from any combination of multiple clients and OSDs. Note that at any moment only one client may write to a specific file region so that when a client 2 opens region A for readwrite and obtains the lock from client 1, client 1 cannot modify region A until client 2 closes. Since we allow clients to write to other client caches and we require the latest write to be read by all clients we will forward requests to a cache from the moment the file region is opened until the the data has been transmitted back to the OSDs or another client after the close system call. Finally, we allow the sharing to grow arbitrarily deep. We expect a sharing tree will very rarely have a height more than 3 or 4 in almost all applications so the tree shouldn't become too big or complex. Additionally, if we specify a cutoff size after which no more forwards are made, then a client whose open would extend the tree beyond the cutoff will have an enormous latency for the open as it waits for the sharing tree to become small enough to allow the open to occur without making the tree too tall.

4.2 Coherence

The coherence options were mostly dictated by the solution requirement we made that all clients must always read the last written data. Clearly if this is to be true the cache must be dynamic and change as other clients modify the file region. Also, since the last write must always be read we cannot wait for the client that modified the file to close it so clients must be informed of the change after every write call. Even the method of updating is partially dictated since we cannot use a time interval update because the clients need to be notified immediately of any modifications. That leaves either updating automatically or invalidating regions and synchronizing only when necessary. Since many changes made to a file region won't actually be read by a client with that region cached we choose to use the invalidation method. In addition, this should help make in-flight data management more scalable since on every modification to its local cache, a client must only send a small message to its list of overlapping caches and not all of the changed data all at once. Thus, each readwrite cache will also store a list of all clients with a cache that intersects the readwrite region so that the client with the readwrite cache can notify other clients when parts of their cache are invalid. We don't expect that a read only cache needs to store where it obtained the data, so all it must store is a list of the invalid regions in the cache and, for convenience, an identification of the client which invalidated that region. Then if the invalidated region is read the client can first query the client which invalidated the region for the updated information and if that clietn has already closed the cache then the update will start from the OSD like an open call.

4.3 Failure Recovery

Here we outline one method for recovery after client failure but it is just an idea at this point and needs further investigation to clarify details and confirm it is a feasible method. The first step for this recovery method is to identify the root of the sharing tree as the authority for this file region. Then, every readwrite cache will also store the sharing tree below itself and the path to the root (thus the root stores the entire sharing tree). We then use a lazy write to propagate any modifications to a client's local cache up the tree to the root, but not to secure storage on OSD unless a specific flush call is issued. This allows the root to rebuild the tree and redirect readwrite caches to write their changes to the correct client's cache to rebuild the sharing tree around the failed client (when the failed client is not the root of the tree). When such a client fails, there should not be any lost in-flight data because any modifications were lazily written up the tree to the root. The problem then becomes what to do when the root fails. In this event, the rebuilding of the sharing tree becomes the responsiblity of the OSD. To rebuild the tree the OSD first needs to locate, probably using a broadcast message, the subtrees that are still functioning below the failed root. Then, using the paths to the failed root which are stored with the readwrite caches, the OSD can redirect the caches to rebiuld the sharing tree into multiple trees with new roots. When the root fails, some in-flight data will be lost if it was modified by the root or by a client which wrote its modifications to the root and then closed. We find that since the data is "in-flight" and still in local caches such data loss is acceptable since if a client needs to guarantee that a modification is securely stored a flush command can be used. This recovery method allows copies of modified data to usually be stored on different clients without burdening the OSD with more traffic except when a root client fails. Thus this method should be scalable unless the root client which is now the authority for the region gets too much traffic.

5 Conclusion

Although we have not yet implemented any in-flight data management in Ceph, we see the potential for decreased file access latencies for many applications. We think this will have the most impact on high performance computing and believe it will make it easier for large simulations to write all data to a single file with minimal performance degradation. The next step for in-flight data management is to confirm and quantify any performance gains in actual applications and determine how the changing the design options effects performance. Another important aspect that must be investigated further is recovery from client failures. We only looked at one method, outlined in 4.3, but it still needs work filling in details on the algorithms and to insure that the recovery can be performed in a timely manner while allowing the rest of the cluster to continue normal operation. A third aspect that must be determined is how metadata information such as last modified and file size is stored and kept updated. Although there is still much work to do on in-flight data management, it is a promising addition to distributed storage systems which moves the storage system one step closer to existing entirely within the clients.

6 Acknowledgements

This work was completed as part of UCSC's SURF-IT summer undergraduate research program, an NSD CISE REU Site. This material is based upon work supported by the National Science Foundation under Grant No. CCF-0552688.

References

[1] D. J. Lilja. Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons. *ACM Computing Surveys*, 25(3):303–338, Sept. 1993.

[2] A. Pozner and T. Kaldewey. In-flight data management in large scale storage systems - crossing the boundary towards peer to peer systems. Personal communication.

[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. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.