Intelligent Metadata Management for a Petabyte-scale File System

Sage A. Weil *sage@cs.ucsc.edu*
Scott A. Brandt *scott@cs.ucsc.edu*
Ethan L. Miller *elm@cs.ucsc.edu*
Kristal T. Pollack *kristal@cs.ucsc.edu*

University of California, Santa Cruz

Abstract

In petabyte-scale distributed file systems that decouple read and write from metadata operations, behavior of the metadata server cluster will be critical to overall system performance. We examine aspects of the workload that make it difficult to distribute effectively, and present a few potential strategies to demonstrate the issues involved. Finally, we describe the advantages of intelligent metadata management and a simulation environment we have developed to validate design possibilities.

1 Introduction

A compelling architecture for petabyte-scale storage systems involves decoupling metadata transactions from file read and write operations. In such a system a client will consult a metadata server (MDS) cluster, responsible for maintaining the file system namespace, to receive permission to open a file and information specifying the location of its content. Subsequent reading or writing would take place independent of the MDS cluster by communicating directly with one or more Object-based Storage Devices (OSDs), which intelligently manage their own on-disk storage and enforce security policies. Although the size of metadata is relatively small compared to the overall size of the system, metadata operations may make up over 50% of all file system operations [12], making the performance of the MDS cluster of critical importance. Furthermore, while the overall capacity of the OSD cluster can easily scale by increasing the number of (relatively independently operating) devices, metadata exhibits a higher degree of interdependence, making the design of a scalable system much more challenging.

We have identified a number of key design issues that will significantly affect the performance of a metadata server cluster in such a system. First and foremost these include the effectiveness of MDS caching, which will be critical for sustaining high throughput and masking slow disk performance. The distribution of client metadata requests among servers will affect observed cache effectiveness and the ability of the system to cope with extreme workloads. Similarly, the mechanisms for ultimately storing and retrieving metadata from stable (disk) storage will be affected by the partitioning strategy and the I/O demands of requests not satisfied by caching.

We describe four partitioning strategies and examine their strengths and weaknesses in light of specific design choices. We contend that the metadata system requirements are such that the most promising approaches require intelligent management of metadata distribution, allowing adaptation to changing workload characteristics. Finally, we describe a simulation environment we have developed to evaluate the specific trade-offs exhibited by these choices, and future research.

2 Background

The Storage Systems Research Center at UC Santa Cruz is currently researching a petabyte-scale (10^{15} byte) storage system designed to handle both general-purpose and scientific computing workloads by exporting a POSIX-compliant interface. This architecture will consist of tens of metadata servers (MDSs), thousands of object-based storage devices (OSDs), and potentially hundreds of thousands of clients. Intelligent OSDs (which will most likely consist of a hard disk, a commodity CPU, and a network interface) simplify file system design by handling block-level allocation internally and presenting a simple object-based interface—file data will be striped across many such objects on many OSDs. Applications of such a system currently include scientific computing environments, the Internet Archive, and large data centers, whose storage demands may well be typical of distributed file systems in a few years time.

2.1 Metadata

Each file or directory metadata record handled by the MDS cluster will include information that is normally stored in an inode, including owner, file mode, c/mtime, and file size. We will utilize a deterministic algorithm to generate a sequence of OSDs and object identifiers for reliably distributing objects (and their replicas) across the

storage cluster [7, 14]. This allows the information necessary to locate file data to be reduced to a fixed size value to serve as the algorithm's input, eliminating the need for a variable-sized data structure to describe object allocation (*e. g.*, a block list) and allowing file metadata to be stored in a record on the order of 64 or 128 bytes.

Handling directory contents is a bit more complicated, since file names are variable length and individual directories must be able to efficiently contain only a few or many thousands of files (or more). For each directory, the MDS cluster must define a mapping of file names to metadata records.

2.2 Workload

Because file read and write operations involve only the client and one or more OSDs, the metadata cluster need only concern itself with a relatively restricted set of operations. Basic metadata operations include `open`, `close`, `stat`, `getattr`, and `setattr` while directory operations include `readdir`, `create`, `link`, `unlink`, and `rename`.

A few typical sequences of operations tend to represent the vast majority of a file system's metadata workload: `open` followed by `close`, and `readdir` followed by many `stat`s [12]. In addition to efficient operation in the general case, the system must additionally handle the extreme usage patterns common to scientific computing applications and less common "flash crowd" behavior in general purpose workloads, including many thousands of clients opening the same file or creating files in the same directory.

With metadata records on the order of 128 bytes and an MDS cluster size of ten servers containing 4 GB of RAM each, the cluster would be capable of collectively caching on the order of 300 million metadata records or directory entries, assuming minimal cache overlap between machines and efficient memory usage. A multi-petabyte file system may contain billions of files, so the cluster design must scale such that its cache can mask sufficient read operations to reduce read bandwidth to a tractable level. These I/O requests will appear relatively random and involve seeks by the underlying disk storage.

3 Design

A number of general design considerations present themselves. A distributed metadata server cluster requires that the workload be partitioned among some set of hosts such that the size of the cluster can be scaled to handle increased client transactions. The underlying storage mechanism must also facilitate independent reads and updates to the metadata itself.

3.1 Workload Partitioning

The metadata workload needs to be effectively partitioned across the cluster of metadata servers such that average case behavior results in balanced utilization and the system can efficiently cope with extreme workloads, such as thousands of clients opening the same file or writing to the same directory. Furthermore, the partition should be such that cache overlap between servers is minimized, thus maximizing the overall effectiveness of the MDS cluster at masking I/O requests to the underlying metadata storage subsystem. The system can be augmented with a failover mechanism such that a failed node's workload is redistributed among other servers or assumed by a standby.

3.1.1 Consistency and Caching

Metadata updates must be serialized at some point within the metadata cluster such that atomicity and consistency are maintained. Since collaborative locking schemes tend to be expensive, we believe a solution where metadata records have a well-defined authority that is responsible for serializing updates and writing to (potentially shared) stable storage is best. Experience has shown that such simple solutions tend to exhibit the best performance in distributed caching environments by simplifying consistency and coherency strategies: the authoritative node can process updates and send invalidate messages (or disable caching) as necessary.

We currently assume that it is undesirable for the MDS cluster to track the contents of client caches, *e. g.*, by issuing leases, because of the resulting memory demands on the cluster. This particular design choice warrants further consideration than is given in this short position paper.

3.1.2 Path Traversal

The traversal of paths containing files that are being accessed typically serves two purposes. In most file systems, each directory in the path is opened in order to identify the inode for the next successively nested object. Traversal also allows the file system to verify that the current user has permission to visit the directory or file in question. Clients cache recently visited prefixes, allowing them to avoid unnecessary and costly consultation with the file server. This traversal is also necessary within the MDS cluster in order to verify that a client has permission to access the files it requests.

3.1.3 Preservation of Locality

Both general-purpose and scientific workloads exhibit significant locality of reference, within both individual client workloads and across groups of clients that tend

to access similar files in similar directories. Ideally, the MDS cluster should maximize the effectiveness of its cache by distributing client requests to exploit both forms of locality in the workload. This partition should be such that cache overlap between servers is avoided, thereby maximizing the amount of metadata that can be cached and minimizing the I/O resulting from cache misses. This is particularly important for effective prefix caching of path traversals, which, in the worst case, may increase memory demands by requiring many additional directory records to be loaded and cached to handle a single request deeply nested in the directory tree. Such locality in individual MDS workloads allows the costs of path traversal to be amortized over all subsequent accesses to the same directory subtree.

3.1.4 Hotspot Development

Clients should ideally contact the metadata server that is most likely to be caching the metadata they are looking for. For an incoherent (low-contention) workload, this yields optimal performance, since cache miss rates are minimized. However, this approach can cause problems for coherent scientific workloads in which thousands of clients may open the same file at once: if all clients agree on where a particular file should be found and decide to open it simultaneously, that MDS may become overloaded by the sudden development of a "hot spot." A similar situation can develop when a large number of clients create files in the same directory.

3.2 Storage

Ultimately all metadata must be stored on some sort of permanent disk storage. Metadata for a 2 PB file system that may contain more than a billion files may consume on the order of 100 GB of disk space. This is likely to be too large to reside completely in the collective RAM of the metadata server cluster. Ideally the MDS memory caches will satisfy most reads, but they will periodically need to go to disk to retrieve the requested information, and all updates must be saved to a stable store such as disk.

3.2.1 Short-term Log

All metadata transactions must be quickly written to stable storage for safety. Since a significant portion of reads are expected to be satisfied by the metadata in-memory caches, the primary demand will be on raw write bandwidth. We believe a bounded log structure to be most appropriate for the immediate storage of updates on each metadata server, where entries that fall off the end of the log without subsequent modifications are written to a second, more permanent, tier of storage. With a log size on the order of the amount of memory in the MDS, such an arrangement has the convenient property that the log represents an approximation of that node's working set, allowing the memory cache to be quickly pre-loaded with millions of records on startup or after a failure. The use of NVRAM in the metadata servers could further mask the latency of writes to the log or other underlying metadata storage.

3.2.2 Permanent Storage

Entries that fall off the end of the metadata logs must be stored elsewhere in the system. Ideally, data layout should be optimized for reads such that expected access patterns allow related records to be fetched without additional disk seeks. An obvious approach is to exploit the hierarchical file structure as a source of locality and store metadata for objects in the same directory together. This is typically the approach taken in general purpose file systems for directory entries, while inodes are stored in the same cluster group when possible. In the WAFL file system [6], this strategy is abandoned in favor of a write-anywhere approach; the authors found that simply writing metadata to disk in the order it was written preserves some temporal locality, which can be similarly advantageous. However, in a system with 100,000 clients or more, we expect any temporal correlation with future access patterns to be insignificant.

The second tier of storage may or may not allow shared access by all nodes in the metadata server cluster. Shared access has the advantage of making it possible for any MDS to read or modify the metadata it needs without bothering other MDS nodes. On the other hand, shared on-disk structures make concurrent updates and consistency more difficult.

4 Approaches

We present four approaches that illustrate some of the tradeoffs related to individual design choices, their interdependencies, and the potential benefits of intelligent metadata management.

4.1 Static Subtree Partitioning

Traditionally network file systems have partitioned workload and storage by simply assigning portions of the directory hierarchy to different file servers; this is the approach taken by NFS [11], AFS [9], Coda [13], Sprite [10] and others. This strategy allows a storage system to scale for breadth, but not depth. Although it typically results in good cache performance due to the enforced locality of a particular server's workload, changes in that workload,

as when many clients access a few files, result in a poor distribution of load among file servers.

4.2 Static Hashing

A more consistent approach to partitioning workload is to distribute metadata based on a hash of some unique identifier, such as an inode number for file or directory records or the parent directory's ID and filename for directory entries [1, 4, 8]. As long as such a mapping is well defined, this simple strategy presents a number of advantages. Clients can contact the responsible MDS directly and, for average workloads, load is evenly distributed across the cluster as long as the hash function is well-behaved. Further, hot-spots of activity in the hierarchical directory structure, such as heavy create activity in a single directory, do not correlate to individual metadata servers because metadata location has no relation to the directory hierarchy.

However, hot-spots consisting of individual files can still overwhelm a single responsible MDS. Directing requests at random servers would allow the cluster to distribute the sudden load by caching the metadata elsewhere, but at the expense of a more costly average case. More significantly, distributing metadata by hashing eliminates all hierarchical locality, and with it many of the locality benefits typical of local file systems. Similarly, hashing results in worst-case performance for the prefix cache, which subsequently expects little to no overlap on individual MDSs between prefix directories added to the cache for each file metadata query. The result is substantial overlap between metadata servers when more than one file is accessed in a single directory, since those files will likely not reside on the same MDS, but in both cases will require a path traversal to verify that the client has permission to access them.

4.3 Lazy Hybrid

Lazy Hybrid (LH) metadata management [3] seeks to merge the net effect of the permission check's path traversal into each file metadata record, eliminating any need for path traversal on the MDS. Subsequent work has shown that it may be possible to do so in a probabilistically concise form (0 or 1 ACL records on average, 2 in unusual cases) based on analysis of a general purpose file system. Metadata for file and directory objects can then be located, modified, and possibly stored on a particular metadata server based on a hash of the full path name.

Although this eliminates the need for path traversal and avoids the related scalability issues in the general case, modifications to the name, ownership or mode of a directory invalidate the path permission vectors for the entire directory subtree it contains. Our trace analysis indicates that the need for propagation of such changes are rare and may effect reasonably small numbers of files for common usage patterns. Regardless, metadata servers need to maintain a list of prefixes for which path permission vectors are no longer valid until updates can be applied to the entire subtree. Further analysis is required to determine what the costs of maintaining such a cache would be given a wider range of workloads, the costs of lazily propagating changes (to potentially millions of files on a larger file system) in order to retire items from the invalidate list, and if those costs compare favorably to a path-traversal approach and (potentially) simpler metadata location strategies.

4.4 Dynamic Subtree Partitioning

The loss of locality in hashed distribution schemes is worrisome because it degrades the performance of the prefix cache, which past file system experience has shown to be quite effective for most workloads. Lazy Hybrid avoids a traversal by trading it for a log of invalidated prefixes that must be applied to the file system either lazily or in the background. Because the costs of such updates are unclear (but potentially quite large) for a large file system, and because more subtle benefits of hierarchical locality (*e. g.*, efficient prefetching) are still lost, we propose a fourth strategy that attempts to leverage the existing hierarchical structure of the file system and intelligently manage metadata distribution based on workload demands.

Instead of locating metadata within the cluster based on a single attribute such as an inode number or full path name, metadata location can be a function of its parent directory. If MDS load, directory size, or other factors permit, metadata for objects within a small directory can be located on the same MDS as the parent directory. If not, directory content can be multiplexed across the entire cluster based on a hash of the parent directory and file name; a similar approach is taken by Lustre [2]. This decision can be a function of any number of factors, including directory size or past and present usage patterns. Since the path must be traversed to validate permissions anyway, there is no additional cost of using parent directory metadata to locate child records. Furthermore, workload permitting, entire subtrees of the directory hierarchy can be managed on a single metadata server, avoiding inefficient duplication of metadata in the other MDS prefix caches, and maximizing the benefit of currently cached metadata.

A hierarchical distribution mechanism also allows the MDS cluster to optimize for the overwhelmingly common case of a single link to a single inode by collocating the inode with the directory entry. Among other things, this allows a MDS to exploit locality of access within a particular directory by loading its entire contents in one op-

eration. For instance, file accesses are often followed by further accesses to the same directory, and a `readdir` is usually followed by a `stat` of every file. The rare case of an inode with more than one link might simply degenerate to a static distribution based on inode.

The primary challenge with this strategy is intelligently adapting the metadata partition to changing workload demands. The most basic case of a single MDS becoming too busy can be resolved by simply shifting one or more large subtrees to another MDS; this approach requires a shared-storage mechanism that allows such a migration of responsibility. Hot-spot development is more difficult, because a small directory on one MDS may suddenly become busy, or a particular file may become suddenly popular. We believe that such situations may be avoided by maintaining a soft-state "popularity" value for each directory that indicates how many clients have opened it, and therefore know how to locate metadata for its contents. By increasing a popularity value each time a directory's metadata is read and allowing it to decay over time, the value can serve as an estimate of how many clients' caches indicate the current (or recent) locations of contained metadata. When the popularity of a directory gets too high, the MDS can alter the current distribution by replicating the record in multiple servers' caches and responding with an alternate distribution. This guarantees that any subsequent requests will be distributed across the cluster, by virtue of the fact that at any time the number of clients who know where to find any one item is limited. In contrast, the more common case of relatively small sets of clients (maybe only thousands) sharing less popular files allows them to directly contact the MDS with the metadata they need.

5 Evaluation and Further Work

These approaches involve a number of different methods for addressing different aspects of the system requirements. Although we have presented three viable options, many of the individual mechanisms described can be incorporated into other approaches. There are a number of specific dependencies and conflicts, however. For instance, an adaptive distribution is incompatible with a static hashing mechanism unless popularity monitoring is employed only when revealing metadata locations to clients for flash crowd avoidance. Combining metadata records (inodes) with directory entries works only if file metadata distribution is based on the containing directory, but would be possible even if directory distribution were based on static hashing. A lazy hybrid permission compression mechanism may be useful for condensing portions of the directory structure, perhaps the first five levels of nesting, while requiring path traversal for points beyond. Finally, a consistency model that requires the MDS cluster to know what metadata items are cached (leased) by individual client nodes would avoid the need for a popularity estimate.

The large size of our target file system makes evaluation of such models particularly challenging. Generating a workload to evaluate performance will require significant work to create realistic synthetic client activity. The largest distributed file system described in the literature, the Google file system [5], services a highly specialized workload and is thus not useful for our purposes.

To evaluate performance ramifications of some of these choices, we have constructed an event-driven simulator package to allow fast prototyping and testing of MDS cluster models. The package has primitives to simulate caches (on metadata servers, clients, and disks), disk performance, network delays, and processor utilization. We plan to evaluate specific performance trade-offs related to:

- Cache overlap penalties for duplicate prefixes across metadata nodes with static hashing distributions
- LH permission vector propagation costs
- Performance benefits or load balancing issues related to combining inodes and directory entries
- Effectiveness of a popularity metric in avoiding flash crowds (based on scientific workload traces)
- Overall I/O demands (read and write) and locality of reference within that request stream
- MDS overhead associated with tracking client leases on metadata

Although fully simulating a petabyte scale system may prove impossible, we plan to demonstrate with a series of smaller simulations that our design can be expected to scale appropriately.

Another key area for further research attention is the underlying permanent storage schema for the entire MDS cluster. Although logs can satisfy the immediate write throughput and latency demands for individual metadata servers, a second-tier collaborative storage strategy will be necessary to allow the dynamic redistribution of server load, in particular with a dynamic subtree partitioning scheme.

References

[1] P. Braam, M. Callahan, and P. Schwan. The intermezzo file system. In *Proceedings of the 3rd of the Perl Conference, O'Reilly Open Source Convention*, Monterey, CA, USA, Aug. 1999.

[2] P. J. Braam. The Lustre storage architecture, 2002.

[3] S. A. Brandt, L. Xue, E. L. Miller, and D. D. E. Long. Efficient metadata management in large distributed file systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 290–298, Apr. 2003.

[4] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, 1996.

[5] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, Oct. 2003. ACM.

[6] D. Hitz, J. Lau, and M. Malcom. File system design for an NFS file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–246, San Francisco, CA, Jan. 1994.

[7] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Parallel & Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, NM, Apr. 2004. IEEE.

[8] E. L. Miller and R. H. Katz. RAMA: An easy-to-use, high-performance parallel file system. *Parallel Computing*, 23(4):419–446, 1997.

[9] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, Mar. 1986.

[10] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, Feb. 1988.

[11] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*, pages 137–151, 1994.

[12] D. Roselli, J. Lorch, and T. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, San Diego, CA, June 2000. USENIX Association.

[13] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[14] Q. Xin, E. L. Miller, T. J. Schwarz, D. D. E. Long, S. A. Brandt, and W. Litwin. Reliability mechanisms for very large storage systems. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 146–156, Apr. 2003.