Dynamic Hashing: Adaptive Metadata Management for Petabyte-scale File Systems[*]

Weijia Li Wei Xue Jiwu Shu Weimin Zheng
Department of Computer Science and Technology, Tsinghua University, China
liweijia00@mails.tsinghua.edu.cn {xuewei, shujw, zwm-dcs}@tsinghua.edu.cn

Abstract

In a petabyte-scale file system, metadata access performance and scalability will significantly affect the whole system's performance and scalability. We present a new approach called Dynamic Hashing (DH) for metadata management. DH introduces the RELAB (RElative LoAd Balancing) strategy to adjust the metadata distribution when the workload changes dynamically. Elasticity strategy is proposed to support the MDS cluster changes. WLM (Whole Lifecycle Management) strategy is presented to find hot spots in the file system efficiently and reclaim replicas for these hot spots when necessary. DH combines these strategies and Lazy Policies borrowed from the Lazy Hybrid (LH) metadata management together to form an adaptive, high-performance and scalable metadata management technique.

1. Introduction

Metadata management plays a substantial role in a petabyte-scale distributed file system. Although the size of metadata is relatively small compared to the overall capacity of the storage system, more than 50% of all file system operations are metadata operations [12]. So a carefully designed metadata management system is needed to avoid potential bottlenecks caused by metadata access.

Nowadays the prevailing system architecture for petabyte-scale storage systems is object-based storage architecture [14]. Object-based storage architecture separates metadata transactions and file data access as depicted in Figure 1. The separated metadata server (MDS) cluster is responsible for handling metadata request. To efficiently handle the workload generated by tens of thousands of clients, metadata should be properly partitioned to take

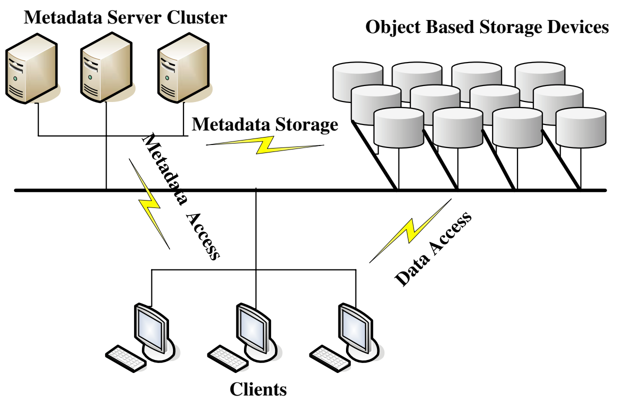

Figure 1. Storage System Architecture

full advantage of the MDS cluster and to distribute metadata traffic. However, because the workload can dynamically change, statically metadata partition can not always balance the workload. In order to deal with the changing workload, an adaptive metadata management scheme is needed.

The size of the MDS cluster can vary too. Metadata management system is responsible for moving metadata when MDSs are added or removed and should minimize the overhead for moving metadata.

In a petabyte scale storage system, tens of thousands of clients may access a same file simultaneously or over a short period of time. Metadata of this file may become a bottleneck in the storage system. It is a common phenomenon in scientific computing workloads and also in other general purpose workloads. A well designed metadata management strategy should be able to eliminate this kind of bottleneck.

We present *Dynamic Hashing (DH)*, an adaptive metadata management technique to provide high-performance, scalable metadata management for extremely large file systems. When workload evolves, DH can dynamically adjust the metadata distribution to adapt to the workload evolution. When new MDSs are added into the cluster or existing MDSs are withdrawn from the cluster, DH can mini-

[*]This work is partially supported by the National Natural Science Foundation of China under Grant No. 60473101 and 10576018; the National Grand Fundamental Research 973 Program of China under Grant No. 2004CB318205

mize the amount of metadata that really needs to be moved and move the metadata in parallel. Finally, DH can identify hot-spots in the file system and eliminate bottlenecks caused by these hot-spots.

2. Related work

According to whether metadata can be dynamically redistributed, previous approaches fall into two classes: static strategy and dynamic strategy. Subtree Partitioning [5], Hashing [4], and Lazy Hybrid [3] are static strategies and Dynamic Subtree Partitioning [15] is a dynamic strategy.

2.1. Static strategy

Subtree Partitioning divides the global namespace into subtrees and allots them to metadata servers. LOCUS [10], NFS [9], AFS [7], Coda [13] and Sprite [8] use this technique to partition namespace. The major disadvantage of this strategy is that the workload may not be evenly distributed among metadata servers.

Hashing applies hash function to the file name or other file identifier to determine on which MDS the file's metadata should be stored. Vesta [4], Intermezzo [1], RAMA [6], zFS [11] and Lustre [2] all use path name hashing to locate metadata. Hashing provides a better balanced load across metadata servers and eliminates hot-spots consist of popular directories. But Hashing is a random distribution. Metadata update operations may incur a burst of network overhead.

The Lazy Hybrid technique is based on Hashing. To address the metadata update problem, LH uses *Lazy Policies* to defer and distribute update cost. If metadata migration is needed because of the update operation, metadata is not moved until the metadata is first accessed. Performing the update and metadata movement at a later time avoids a sudden burst of network activities between metadata servers.

These static strategies can not dynamically redistribute the global namespace to get high throughput in the case of a changing workload. When new MDSs are added or existing MDSs are removed, none of these strategies address how to determine the optimal amount of metadata that should be moved. Finally, they can do nothing for hot-spots which are caused by popular individual files.

2.2. Dynamic strategy

Like Subtree Partitioning, Dynamic Subtree Partitioning assigns subtrees of the global namespace to metadata servers. To cope with the changing workload, Dynamic Subtree Partitioning leverages the *dynamic load balancing* mechanism to dynamically redistribute metadata among metadata servers. Periodically busy metadata servers transfer subtrees of their own to other non-busy metadata servers to balance the workload. The smallest unit of metadata transfer is directory (subtree). Dynamic Subtree Partitioning can eliminate bottlenecks caused by hot-spots consisting of individual files by replication. But it can not reclaim replicas for file metadata which are not popular any more. These cumulate replicas will consume storage space and incur maintenance overhead.

However, clients are ignorant of the metadata distribution. So their requests are directed randomly and forwarded within the MDS cluster. Client ignorance leads to a large amount of forwarding overhead. For example, when the dynamic load balancing mechanism is deployed, nearly 20% of all the requests are forwarded [15].

3. Dynamic hashing

Like pure Hashing and LH, DH uses hashing to distribute metadata throughout the metadata server cluster. DH also borrows Lazy Policies from LH to handle metadata update.

DH introduces three strategies to provide adaptive, high-performance and scalable metadata management. These strategies are RElative LoAd Balance strategy (RELAB) for metadata redistribution, Elasticity strategy for metadata movement and Whole Life-time Management strategy (WLM) for hot-spots. All these strategies make use of a memory data structure called Metadata Look-up Table (MLT). In the following sections, MLT and these strategies are discussed in detail.

3.1. Metadata look-up table (MLT)

Metadata look-up table provides an additional level of indirection between the clients and the metadata servers. Each entry in the MLT contains two fields: the metadata server id field and the entry version field. When a file is accessed by a client, the hash value of the file path name is used as index to the MLT, and the corresponding metadata server id found in the entry indicates the metadata server which stores the metadata for the file.

Let $S = \{s | 0 \leq s < 2^L\}$ represent the hash value space, where L is the bit number of the maximum hash value. We simply abstract the MDS cluster as a set of n servers, identified by $\mathrm{MDS}_1, \mathrm{MDS}_2, \ldots, \mathrm{MDS}_n$. The MLT maps the hash value space S to the metadata server id set. First, the hash value space S is divided into N intervals of equal length, where N satisfies $n \ll N < 2^L$. Then each interval is assigned to an entry in the MLT so there are N entries in the MLT. There is also an MDS id field in each entry. So the hash values relate to the MDS id. By modifying the MDS id field, the metadata distribution is also modified.

Every client and every metadata server has a copy of the MLT. However, all updates for the MLT are furnished by the metadata servers. When the MLT on a server is updated, the update message is broadcast to all the other metadata servers. Making use of the version field in each entry, clients are sent the updated MLT when needed. Every time an entry in the MLT is modified, its version is changed too. When a client contacts a metadata server, it sends entry version along with its metadata request. The contacted metadata server checks whether the entry version is out of date. If the entry is invalid, the server will respond with the updated entry. The client should update its entry and entry version according to the response. So a client only updates its out-of-date entries when these entries are used.

The MLT is kept in main memory and it will not take up too much space. For example, suppose there are 10,000,000 files in the file system. A 20-bit hash function is used and N can be 2^{16}, so there are 65536 entries in the MLT. Assuming that the metadata id field needs 1 bytes and the version field needs 4 bytes, the amount of memory consumed by the MLT is 320KB.

3.2. Relative load balancing strategy

The workload changes all the time. Static distribution of metadata, although carefully designed, can not always achieve the best performance. In order to adapt to the changing workload, DH adopts the RElative LoAd Balancing(RELAB) strategy to dynamically adjust the metadata distribution to maintain an optimal distribution of the workload. The key idea of RELAB is that periodically the MDS nodes exchange heartbeat messages including a description of their current load level. Then busy MDS nodes transfer part of the metadata of their own to non-busy nodes.

Assume the period for MDS nodes to exchange heartbeat messages is T (the same value used in Dynamic Subtree Partitioning can be used here). During the period T, each MDS maintains a special *Entry Access Counter List (EACL)*. Each member in the list has an access counter for an entry in the MLT, storing how many times the entry has been visited during the period T. In addition, each member contains an SAI field which is short for *synthetically access information*. SAI is computed using the following equation:

$$SAI = (1-\alpha) \times SAI + \alpha \times access\ counter \quad \alpha \in (0,1]$$

Unlike the access counter which just stores the current access information of an entry, SAI also takes past access information into account. α acts as a forgetting factor. The bigger the factor is, the less effect past access information will impose on current access information.

To measure how busy an MDS is, the *absolute load level* is used. The absolute load level of a MDS is the number of requests handled by the MDS during the period T. Taking past access information into account, the absolute load level of an MDS is determined by the sum of all SAIs in the EACL. Let ALL represent this sum. If the ALL value of each MDS is equal or close to other MDS, each MDS deals with approximately the same number of requests. That means the load is distributed evenly and is balanced. However, a balanced load may not achieve the largest overall throughput because metadata servers may have different performance. RELAB tries to distribute the workload to the metadata servers according to the throughputs of the metadata servers. Each server is assigned a weight adjustment factor w_i which measure the performance of the server (such as CPU frequency, memory capacity, network throughput, or some combination of the three). The *Relative load level (RLL)* of an MDS is calculated by dividing the absolute load level by the weighting factor. RELAB' goal is to balance the relative load levels for all MDSs. To accomplish this, algorithm illustrated in Figure 2 is executed at the end of each period T to adjust the RLL values.

```
1. Calculate the following equations to get $x_i (i = 1,\cdots,n)$
       $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} ALL_i$        $\frac{x_1}{w_1} = \frac{x_2}{w_2} = \ldots = \frac{x_n}{w_n}$
2. $For\ i = 1\ to\ n$
       $y_i = ALL_i - x_i$
3. $L = \{i \mid y_i > 0\}$,   $S = \{i \mid y_i < 0\}$
   $Sum(L) = \sum_{i \in L} y_i \ \rightarrow\ Sum(L) + Sum(S) = 0$
4. $For\ \ i \in S$, find a subset $C \subseteq L$, satisfying
   $|y_i + Sum(C)| \le |y_i + Sum(O)| \quad \forall O \subseteq L$
5. $Target(j) = i \quad \forall j \in C$
   $L = L - C,\ S = S - \{i\}$
   $if\,(S = \varnothing)\ Target(k) = k \quad \forall k \in L$
   $if\,(L \neq \varnothing\ \&\&\ S \neq \varnothing)\ goto$ 4.
6. $For\ i = 1\ to\ n$
       $if\ \ (y_i > 0\ \&\&\ Target(i) \neq i)$
           Find a set of entries in $EACL_i$. Sum of $SAIs$
                   of these entries is approximate to $y_i$
           Modify these entries to transfer part of
                   metadata from $MDS_i$ to $MDS_{Target(i)}$
```

Figure 2. Algorithm for relative load balancing

The MLT entry size N and metadata server cluster size n is important to the computing time of this algorithm since the average number of elements in EACL is bounded by N and n. For $N = 65536$ and $n = 20$, we implemented this algorithm and executed it on a normal PC (AMD Athlon

1.54GHz, 256M memory), and found that it consumes less than 2ms.

After the adjustment, the RLL values should be approximately equal to each other. Relative load levels are balanced. The workload is distributed according to the throughput of each MDS. This strategy makes better use of better MDS and can provide optimal performance. RELAB takes only one period to balance the relative load of all metadata servers, so the relative load is almost always balanced.

3.3. Elasticity strategy

The MDS cluster can extend or shrink. When MDSs are added or removed, Elasticity strategy can determine how much metadata should be moved for each MDS. After the metadata movement stage, Elasticity ensures that the relative load of each MDS is still balanced.

When new MDSs are added, they can just be treated as existing MDSs, but their ALL values are 0. So Elasticity adopts the same algorithm used in RELAB to determine the amount of metadata that should be moved for each MDS. The expected amount of metadata moved to these new MDSs during the redistribution is exactly the amount which will still hold the assertion that the workload is allotted to MDSs according to their performance. So the amount of moved metadata is minimal to keep the relative load balanced. Besides, because the ALL values of these new MDSs are 0, the metadata is moved from existing MDSs to these new MDSs. Metadata can be moved in parallel to hide some time-consuming disk I/O.

Elasticity utilizes algorithm illustrated in Figure 3 to move metadata when *k* MDSs are leaving. It moves all metadata resided on these candidate MDSs to all other MDSs. At the same time, it tries to ensure that the relative load levels are still balanced among the remaining MDSs.

3.4. Whole lifecycle management strategy

In a peta-byte scale file system, a certain file could be extremely popular, causing thousands of clients to access the same file at the same time or over a short period of time. Thousands of metadata requests for the same file are delivered to a single metadata server directly and will overwhelm the metadata server. Popular files are hot-spots and also bottlenecks in the file system.

3.4.1. Identifying hot-spots Before hot-spots can be eliminated, they should be identified first. We can find out hot-spots by recording access frequencies for popular metadata items. The cache mechanism is leveraged to find popular metadata items. Each cached metadata item is associated with an *access information structure* to recording

```
1. Calculate the following equations to get $x_i (i = 1, \cdots, n-k)$
       $\sum_{i=1}^{n-k} x_i = \sum_{i=1}^{n} ALL_i$        $\frac{x_1}{w_1} = \frac{x_2}{w_2} = \ldots = \frac{x_{n-k}}{w_{n-k}}$
2. For $i = 1$ to $n-k$
       $y_i = x_i - ALL_i$
3. $L = \{i \mid y_i > 0\}$,   $S = \{n-k+1, \cdots, n\}$
   $Sum(L) = \sum_{i \in L} y_i$
4. For $i \in S$, find a subset $C \subseteq L$, satisfying
       $|ALL_i - Sum(C)| \le |ALL_i - Sum(O)|$    $\forall O \subseteq L$
5. $Source(j) = i$   $\forall j \in C$
   $L = L - C$, $S = S - \{i\}$
   if $(S = \varnothing)$ $Source(m) = m$   $\forall m \in L$
   if $(L \ne \varnothing \;\&\&\; S \ne \varnothing)$ goto 4.
6. For $i = n-k+1$ to $n$
     For $j = 1$ to $n-k$
       if $(y_j > 0 \;\&\&\; Source(j) = i)$
         Find a set of entries in $EACL_i$. Sum of $SAIs$ of
                  these entries is approximate to $y_j$
         Modify these entries to transfer part of metadata
                  from $MDS_i$ to $MDS_j$
     Transfer all left metadata in $MDS_i$ to $MDS_1$
```

Figure 3. Algorithm for moving metadata when k MDSs are removed

access frequency for that item. The table below illustrates the components of the access information structure.

access counter	last update time	popularity

The *popularity* field represents how popular the metadata is. The *last update time* field stores the last time when the popularity field was updated. The *access counter* field monitors how many times the cached metadata have been visited since the last update time. When the cached metadata is accessed, its access information structure of the metadata should be updated. Figure 4 shows this procedure.

Actually the access counter stores how many times the cached metadata has been accessed during a time interval of T_p, which is the period to update popularity. α and the power exponent $\lfloor \frac{\Delta T}{T} \rfloor$ are used to limit the effect the initial popularity imposes on the new popularity. To check if the cached metadata is a hot-spot, its popularity is compared to some threshold P_t. If its popularity exceeds that threshold, it is a hot-spot; otherwise, it is not a hot-spot.

This algorithm can identify hot-spots efficiently and easily. To determine if cached metadata is a hot-spot, only one

```
1. Look up the cache for the requested metadata
2. If the metadata is not in the cache
      load it into the cache
      initialize its access information structure
        $access\ counter \leftarrow 0$
        $popularity \leftarrow 0$
        $last\ update\ time \leftarrow T_a$ (arrival time of the request)
3. $\Delta T = T_a$ - $last\ update\ time$
4. $If\ (\Delta T \geq T_p)$
      $popularity \leftarrow (1-\alpha)^{\left\lfloor \frac{\Delta T}{T_p} \right\rfloor} \times popularity + \alpha \times access\ counter$
      $access\ counter \leftarrow 0$
      $last\ update\ time \leftarrow T_a$
   $else$
      $access\ counter \leftarrow access\ counter + 1$
5. Check if the cached metadata is a hot-spot
```

Figure 4. Algorithm for finding hot-spots

simple comparison is needed and the comparison can be done at the same time when the metadata is accessed.

3.4.2. Client awareness Hot-spots are replicated when they are located. The metadata server that holds the original metadata is called the *reality metadata server* for the metadata and the metadata servers that store replicas are called the *shadow metadata servers* for the metadata. Requests for hot-spots are directed to both the reality server and the shadow servers. Each server will only handle a part of all the requests. So shadow servers mitigate the stress of the reality server brought by a popular file.

When a hot-spot is identified and replicated, the reality MDS knows exactly where the shadow servers are. It will upgrade the version of the entry for the popular file. The update is also broadcast to all other servers in the cluster. So far the clients have no knowledge of the replication operation, so all requests for the hot-spot are delivered to the reality MDS. The reality MDS will find that entry versions sent by the clients along with the requests are out of date. So the reality MDS tells the clients to update their entries and send shadow server information of hot-spots which relate to this entry to the clients. Using this strategy, the clients know which files are hot-spot files and where their replicas are stored. When a client accesses a hot-spot file, the client can randomly pick a server from among the reality server and shadow servers and send the request to it.

Because shadow server information is transferred along with requested metadata, no extra network overhead is incurred. Also, clients know where the replicas are so replicas can be accessed directly. Forwarded requests in DH will be much less than in Dynamic Subtree Partitioning.

3.4.3. Replica reclamation As the workload changes, some popular files may become unpopular after a period of time. When a file becomes unpopular, there is no need to store copies of the file metadata on a number of metadata servers. In order to save storage space and to avoid network overhead caused by maintaining replicas consistency, replicas for an unpopular files should be reclaimed.

Suppose a hot-spot has r replicas within the MDS cluster. Each shadow metadata server individually monitors access frequency for the replica on it. If a replica's popularity declines and becomes smaller than the popularity threshold, for example P_t/r, this replica should be freed. The corresponding shadow metadata server should remove the replica and then tell the reality server that it has reclaimed the replica.

When a client request for a hot-spot file arrives at the shadow metadata server which has reclaimed the replica, the shadow server tells the client that this file is not popular any more. Then the client will send its successive requests for this file to the reality metadata server. This approach ensures that all storage space taken up by useless replicas is freed while the file system can still work correctly.

4. Evaluation

Here we take a look at the overhead DH incurs as a whole and compare DH with the other dynamic metadata management strategy– Dynamic Subtree Partitioning.

Suppose there are 10,000,000 files in the system and $N = 65536$, The MLT will take up 320KB memory. The EACL used in both RELAB and Elasticity needs $12N/n$ bytes on average, where n is the number of MDSs in the MDS cluster. Usually n is a small integer. For example, $n = 20$. Then each EACL needs about 40KB. In WLM the memory consumed by the access information structure is about 12bytes per cached metadata item. Because Dynamic Subtree Partitioning also records access frequency, it has this kind of overhead too. Compared to Dynamic Subtree Partitioning, the extra memory needed by DH is about 320KB on each client and 360KB on each MDS.

RELAB runs periodically and will update some MLT entries to transfer metadata. Maintaining MLT consistency will cause some network overhead. Since old entries in clients' MLTs are updated when they are accessed, there is no sudden burst of update activities in the file system. Adding MDSs and removing MDSs happen rarely, so this kind of update overhead is trivial. In WLM, shadow server information are sent along with needed metadata, so no extra network activity is needed.

DH only incurs a little computation overhead. For $N = 65536$ and $n = 20$, we ran RELAB on a normal PC (AMD Athlon 1.54GHz, 256M memory) and it took less than 2ms. Computing time of the Elasticity strategy is approximate to that of the RELAB strategy. WLM only needs a few arithmetical computations per metadata access.

Compared to Dynamic Subtree Partitioning, DH has several advantages. First, it can adapt to a changing workload faster than Dynamic Subtree Partitioning. The smallest unit of metadata transfer in Dynamic Subtree Partitioning is directory. So Dynamic Subtree Partitioning is a coarse adjustment strategy and it will take a long time to balance the load. DH moves metadata based on more accurate computation results and can balance the relative load within one specific time period. Secondly, Dynamic Subtree Partitioning does not consider the metadata movement process when the MDS cluster size changes. DH takes it into account and moves the minimum metadata to keep the relative load balanced. Besides, the metadata can be moved in parallel in DH. Thirdly, Dynamic Subtree Partitioning can identify hot-spots but it does not provide any detailed method. It can replicate hot-spots but can not reclaim these replicas. DH presents an algorithm to identify hot-spots, replicate them when they appear, and reclaim replicas when they disappear. Finally, because clients have no idea of how metadata is distributed in Dynamic Subtree Partitioning, requests are directed randomly. Almost 20% of client requests are forwarded in the MDS cluster. In DH, the client knows exactly which MDS contains the metadata it needs, and there are much fewer forwarded requests.

5. Conclusions

We present Dynamic Hashing, an adaptive metadata management technique to serve pertabyte-scale distributed file systems. DH leverages hashing to distribute metadata among the metadata servers. When the workload changes dynamically, DH introduces the RELAB strategy to quickly adjust the metadata distribution to get high performance. After the adjustment, RELAB ensures that the relative load of each MDS is balanced. Besides, RELAB can balance the relative workload much faster than other metadata management techniques. When the size of the MDS cluster changes, DH uses the Elasticity strategy to move the minimal metadata to keep the relative load still balanced. Finally, DH presents a strategy called WLM to manage the whole life cycle for all hot-spots in the file system. WLM makes use of the cache mechanism to find hot-spots and then replicates them to eliminate bottlenecks. When hot-spots files are not "hot" any more, WLM will reclaim their replicas to save storage space and avoid maintenance overhead. Compared to other metadata management techniques, DH brings a lot of advantages, such as balancing relative workload efficiently, good scalability, no bottlenecks and less forwarded requests, etc, but only incurs a little overhead. We are currently implementing DH and integrating it into our distributed object-based storage system.

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. Oct 23, 2003. http://www.lustre.org.

[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. Feitelso. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, 1996.

[5] E. Levy and A. Silberschatz. Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4), Dec 1990.

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

[7] 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.

[8] 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.

[9] B. Pawlowski, S. Shepler, C. Beame, B. Callaghan, M. Eisler, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, Maastricht, Netherlands, May 2000.

[10] G. Popek and B. J. Walker. *The LOCUS distributed system architecture*. The MIT Press, 1986.

[11] O. Rodeh and A. Teperman. zFS - a scalable distributed file system using object disks. In *Proceedings of the 20th IEEE / 11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 207–218, Apr 2003.

[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, Jun 2000.

[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] R. O. Weber. Information technology-SCSI object-based storage device commands (OSD). Technical Council Proposal Document T10/1355-D, Technical Committee T10, Aug 2002.

[15] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC'04)*, Nov 2004.