Dynamic Metadata Management for Petabyte-scale File Systems[†]

Sage A. Weil *sage@cs.ucsc.edu*
Kristal T. Pollack *kristal@cs.ucsc.edu*
Scott A. Brandt *scott@cs.ucsc.edu*
Ethan L. Miller *elm@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 and scalability. We present a dynamic subtree partitioning and adaptive metadata management system designed to efficiently manage hierarchical metadata workloads that evolve over time. We examine the relative merits of our approach in the context of traditional workload partitioning strategies, and demonstrate the performance, scalability and adaptability advantages in a simulation environment.

1 Introduction

A compelling architecture for large distributed 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, which is responsible for maintaining the file system namespace, to receive permission to open a file and information specifying the location of its contents. Subsequent reading or writing takes place independent of the MDS cluster by communicating directly with one or more object-based storage devices (OSDs) [4, 9], which intelligently manage their own on-disk storage and enforce security policies. Although the size of metadata are relatively small compared to the overall size of the system, metadata operations may make up over 50% of all file system operations [19], 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 exhibit a higher degree of interdependence, making the design of a scalable system much more challenging.

The metadata server cluster in such a system should efficiently maintain file system directory and permission semantics for a variety of workloads, including both scientific computing applications and general purpose computing. Such workloads and file systems may involve files ranging from a few bytes to multiple terabytes, directories containing millions of files, and many thousands of clients accessing either disparate or identical files. Furthermore, as the character of the workload may change over time, the MDS cluster must be able to continually adapt to current demands by dynamically repartitioning workload to maintain high system performance and long-term scalability.

We describe a dynamic metadata management system to efficiently distribute responsibility for metadata for extremely large file systems across a cluster of servers. We utilize a dynamic subtree partitioning strategy to continually adapt the metadata distribution to current demands and facilitate traffic control, while simultaneously preserving and exploiting locality. Finally, we evaluate competing metadata management strategies by simulation on the basis of overall system performance, adaptation to file systems and workloads that evolve over time, and ultimately their ability to scale to efficiently manage metadata for large storage systems. Our results demonstrate the effectiveness of dynamic subtree partitioning over both static partitioning and hash-based approaches.

2 Background

We examine distributed metadata management in the context of a petabyte-scale (10^{15} or 2^{50} byte) storage system being designed at the University of California, Santa Cruz to handle both general-purpose and scientific computing workloads by exporting a POSIX-compliant interface [12, 25]. This architecture will consist of tens of metadata servers (MDSs), thousands of object-based storage devices (OSDs), and potentially hundreds of thousands of clients. Applications of such a system will 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 System Architecture

One of the primary design features of object-based storage systems is the separation of metadata from data management. By decoupling metadata operations, which depend

[†]0-7695-2153-3/04 $20.00 (c)2004 IEEE

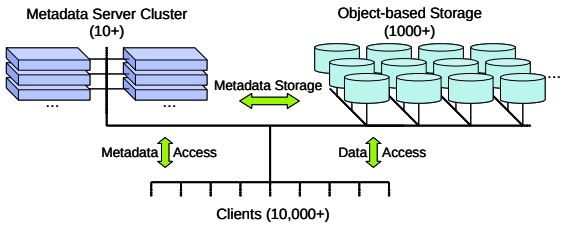

Figure 1: Storage system architecture.

on a more complicated and interdependent set of semantics, from file data I/O, which is trivially parallelizable, the I/O bottleneck typical of conventional file servers is avoided. Intelligent object storage devices (OSDs) simplify file system design by handling block-level allocation internally and presenting a simple object-based interface. Clients first communicate with a separate metadata server cluster which is responsible for managing the file system namespace and directory hierarchy, file and directory permissions, and the mapping from files to objects. Clients then read or write data by contacting OSDs directly, allowing very efficient data transfers between large numbers of clients and OSDs.

2.1.1 Data Distribution

File data is striped and replicated across a large number of objects on a large number of OSDs to maximize I/O throughput and data safety. Unlike data storage in traditional file systems, which typically seek to group related files together, data is distributed to OSDs based on a deterministic pseudo-random algorithm that guarantees a probabilistically balanced distribution of data throughout the system [11].

A critical feature of our file data distribution algorithm is that the sequence of object identifiers and OSD devices can be recalculated by the client—without interaction with the MDS cluster—given a single small input value for the algorithm, such as an inode number. In our system, that value is augmented by a replication group identifier to facilitate reliable replication and recovery [28]. Although the details of this distribution and reliability mechanism are not described here, the implication is that the file to object mapping that must be defined for every file has a fixed size of only a few bytes. This simplifies metadata management and storage by avoiding cumbersome block or object lists typical of most file systems.

2.1.2 Metadata Partitioning

The metadata workload must be effectively partitioned across the cluster of metadata servers such that average case behavior properly utilizes available resources and the system can efficiently cope with extreme workloads, such as thousands of clients opening the same file or writing to the same directory. The partition should be such that cache overlap between servers is minimized, thus maximizing the MDS cluster's ability to mask 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. More significantly, as the size of the storage system grows, the MDS cluster should be able to expand to encompass additional servers with minimal effort required to redistribute workload.

2.1.3 Storage

Ultimately all metadata must be stored on some sort of permanent disk storage. Metadata for a petabyte file system that may contain more than a billion files might consume a terabyte or more 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 requested information, and all updates must be saved to a stable store such as disk. Generally speaking, a shared metadata store (*e.g.* consisting of OSDs) is necessary and appropriate for many partitioning strategies (including ours), and offers fundamental advantages over directly-attached storage by easing MDS failover and utilizing more generic hardware.

2.2 Workload

Because file read and write operations involve primarily the client and one or more OSDs, the metadata cluster need only concern itself with a relatively restricted set of operations. Metadata transactions fall into two categories: operations like `open`, `close`, and `setattr` are applied to metadata records for files and directories ("inodes"), and operations like `rename` and `unlink` manipulate the directory entries defining the file system namespace and hierarchy. A few typical sequences of operations tend to represent a majority of a file system's metadata workload: `open` followed by `close`, and `readdir` followed by many `stat`s [19]. 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.

Because the MDS will not be able to keep all metadata in RAM, the cluster design must scale such that its collective cache can mask sufficient read operations to reduce I/O demands and response times to a reasonable level.

3 Related Work

This work draws from a number of areas including distributed metadata management architectures, file system simulation and file system workload characterization.

3.1 Metadata Distribution

With the emergence of large-scale storage architectures that separate metadata management from file read/write operations, metadata management has become an interesting research problem on its own. When designing a metadata server cluster, the partitioning of the metadata among the servers is of critical importance for maintaining efficient MDS operation and a desirable load distribution across the cluster. Current approaches used in traditional and more modern distributed file systems serve as a basis for discussion of the performance and scalability issues involved.

3.1.1 Subtree Partitioning

Networked file systems have traditionally partitioned workload and storage by statically assigning portions of the directory hierarchy to different file servers; this is the approach taken by systems based on NFS [17], AFS [15], Coda [20], Sprite [16] and countless others. *Static subtree partitioning* typically requires a system administrator to decide how the file system should be distributed and manually assign subtrees of the hierarchy to individual file servers. A static distribution simplifies the clients' task of identifying servers responsible for metadata, and servers are able to process requests without communicating with (or even being aware of) other nodes because subtrees of the hierarchy are treated as independent structures.

This strategy can allow a storage system to scale for breadth, but not depth. A statically partitioned cluster can only accommodate file system expansion if growth is such that data remains evenly partitioned across available servers, or new data is written exclusively to new servers allocated new portions of the hierarchy. File systems do not typically expand in such a regular fashion, however, requiring manual redistribution of the hierarchy to accommodate new data or even increased client demand for existing data. Furthermore, if client workload is not evenly distributed across all file data, static partitioning is vulnerable to imbalance as individual servers can be overloaded by "hot spots" of popularity in certain parts of the hierarchy.

3.1.2 Hashing

To provide good load balancing across metadata servers, many recent systems distribute files across servers based on a hash of some unique file identifier, such as an inode number or path name. Vesta [5], Intermezzo [1], RAMA [14], zFS [18] and Lustre [2, 22] all hash the file pathname and/or some other unique identifier to determine the location of metadata and/or data. As long as such a mapping is well defined, this simple strategy has a number of advantages. Clients can locate and contact the responsible MDS directly and, for average workloads and well-behaved hash functions, requests are evenly distributed across the cluster. 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 only at the expense of a more costly average case. The Google file system [8] attempts to address this problem by using read-only replication of metadata on *shadow metadata servers* to alleviate hot-spots formed by many clients trying to read the same file, at the expense of vastly simplified metadata semantics. A hashed metadata distribution also makes the process of expanding the MDS cluster to accommodate growth more difficult because the size of the desired output range suddenly changes.

More significantly, distributing metadata by hashing eliminates all hierarchical locality, and with it many of the locality benefits typical of local file systems. Some systems distribute metadata based on a hash of the *directory* portion of a path only to allow directory contents to be grouped on MDS nodes and on disk. This approach facilitates prefetching and other methods of exploiting locality within the metadata workload.

Even so, to satisfy POSIX directory access semantics, the MDS cluster must traverse prefix (ancestor) directories containing a requested piece of metadata to ensure that the directory permissions allow the current user access to the metadata and data in question. Because the files and directories located on each MDS are scattered throughout the directory hierarchy, a hashed metadata distribution results in high overhead, either from a traversal of metadata scattered on multiple servers, or from the cache of prefixes replicated locally. Prefix caches between nodes will exhibit a high degree of overlap because parent directories inodes must be replicated for each MDS serving one or more of their children, consuming memory resources that could be caching other data.

3.1.3 Lazy Hybrid

Lazy Hybrid (LH) metadata management [3] seeks to capitalize on the benefits of a hashed distribution while avoiding the problems associated with path traversal by merging the net effect of the permission check into each file metadata record. Like other hashing approaches, LH uses a hash of the file's full path name to distribute metadata.

To alleviate the potentially high cost of traversing paths scattered across the cluster, LH uses a dual-entry access control list that stores the effective access information for the entire path traversal with the metadata for each file. We have found that this information can usually be represented very compactly even for large general-purpose file systems. LH need only traverse the path when access controls need to be updated because an ancestor directory's access permissions are changed, affecting the effective permissions of all files nested beneath it. Similarly, renaming or moving a directory affects the path name hash output and hence metadata location of all files nested beneath it, requiring metadata to be migrated between MDSs. Previous trace analysis has shown that changes like this happen very infrequently [19] and it is likely that they will affect small numbers of files when they do occur. Moreover, it is possible to perform this update at a later time to avoid a sudden burst of network activity between metadata servers, by having each MDS maintain a log of recent updates that have not fully propagated and then lazily update nested items as they are requested.

LH avoids path traversal in most cases, provided certain metadata operations are sufficiently infrequent in the workload. Analysis has shown that update cost can be amortized to one network trip per affected file; as long as updates are eventually applied more quickly than they are created (changes to directories containing lots of items could trigger potentially millions of updates with a single update), LH delivers a net savings and good scalability. Like other file hashing approaches, it avoids overloading a single MDS in the presence of directory hot-spots by scattering directories. However, in doing so the locality benefits are lost while the system remains vulnerable to individually popular files. More importantly, the low update overhead essential to LH performance is predicated on the low prevalence of specific metadata operations, which may not hold for all workloads.

4 Dynamic Metadata Management

We present a metadata cluster architecture utilizing a dynamic subtree partitioning strategy for distributing metadata across a cluster of metadata servers. Subtree-based partitioning is a natural approach to partitioning a hierarchy and provides a number of advantages over hashed distributions, including greater MDS independence and greater locality of reference within the workload. Although it is more difficult to create and maintain a good partition, a dynamic partitioning strategy must be employed to adapt to a changing file system and workload. Moreover, dynamic partitioning has a number of advantages over other techniques in terms of metadata storage, traffic control, and flexible resource utilization policies.

4.1 Hierarchical Partition

Central to the dynamic subtree partitioning approach is the treatment of the file system as a hierarchy. The file system is partitioned by delegating authority for subtrees of the hierarchy to different metadata servers. Delegations may be nested: /usr may be assigned to one MDS, for instance, while /usr/local is reassigned to another. In the absence of an explicit subtree assignment, however, the entire directory tree nested beneath a point is assumed to reside on the same server.

Implicit in this structure is the process of hierarchy traversal in order for nested inodes to be located and opened for subsequent descent into the file hierarchy. Such path traversal is also necessary to verify user access permissions for nested items as required by POSIX semantics. Although this process may seem costly for locating a file deep within the directory hierarchy, the locality of reference typical of both scientific and general purpose computing workloads [6, 26] allows those costs to be amortized over subsequent accesses to the same directories. More importantly, unlike LH permission management, a hierarchically defined structure allows the system to move or change the effective permissions of arbitrarily sized subtrees of the directory tree by modifying the subtree's root directory with fixed cost. Likewise, individual subtrees of the hierarchy are fully independent from their siblings; semantics are dependent only on the prefix (ancestor) directories leading to the root of the file system.

To allow client requests (and the path traversal required to properly respond to them) to be processed efficiently, each MDS caches prefix inodes for all items in the cache, such that at any point the cached subset of the hierarchy remains a tree structure. That is, only leaf items may be expired from the cache; directories may not be removed until items contained within them are expired first. This allows permission verification for all known items to proceed without any additional I/O costs, and for hierarchical consistency to be preserved.

4.2 Authority and Collaborative Caching

Metadata updates must be serialized at some point within the metadata cluster such that atomicity and consistency are maintained. A significant body of research has investigated distributed locking and consistency strategies in potentially unreliable environments, but experience has shown that the simplest solutions perform the best, provided they can scale appropriately. To maximize average case performance, each metadata item has a well-defined authority MDS based on the hierarchical partition that is responsible for serializing updates, committing changes to stable storage, and managing cache consistency and coherence when that record is replicated on other nodes in the MDS cluster.

If a MDS node receives a request for a portion of the hierarchy it is not responsible for, it will ordinarily forward the request to the authority. If it needs to replicate some metadata (either because the directory is popular and flagged for replication, or the MDS needs some prefix metadata to traverse to a subtree of the hierarchy it is responsible for), it will request the relevant inode(s) from the authority. Once an item is replicated in another MDS's cache, the authoritative MDS is responsible for communicating updates to maintain cache coherence. Similarly, if a node discards an inode for which it is not authoritative from its cache, it will notify the authority, who will then be free to remove its own copy from memory. This approach ensures that within the MDS cluster the state of the file system remains consistent and well defined.

In certain cases, updates to metadata can be distributed. For instance, fields like modification time and file size are monotonically increasing for most operations, such that replicas serving concurrent writers can periodically send their most recent value to the authority, which retains the maximum value seen thus far and initiates a callback for the latest information on client reads (*e.g.* a `stat` to check file size). This approach is taken in the GPFS file system to facilitate shared parallel write access to files typical of scientific computing workloads [21].

The requirements for client consistency are potentially less demanding. A system serving hundreds of thousands of clients will require a significant amount of memory to maintain the state necessary to provide clients with strong consistency guarantees with callback-based cache coherence at the data block level, as with the Sprite [16] and Coda [20] file systems. Conversely, the weak consistency provided with fully stateless approaches such as NFS (v3 and earlier) can cause a number of problems for client applications. We believe that relatively simple (and inexpensive) metadata coherence strategies may prove sufficient for, although the appropriate approach is likely to be dependent on specific workload requirements and outside the scope of this paper.

4.3 Load Balancing

It order to adapt to file system evolution and changing workload demands the MDS cluster must adjust the directory partition to maintain an optimal distribution of its workload. A dynamic distribution is necessary because both the size and popularity of portions of the hierarchy change over time in a non-uniform and unpredictable fashion. The metadata partition is modified over time by allowing MDS nodes to transfer authority for subtrees of the directory hierarchy. Periodically the MDS nodes exchange heartbeat messages that include a description of their current load level. At that point busy nodes can identify portions of the hierarchy that are appropriately popular and initiate a double-commit transaction to transfer authority to non-busy nodes. During this exchange all active state and cached metadata are transfered to the newly authoritative node, both to maintain consistency and to avoid the disk I/O that would otherwise be required for the newly authoritative node to re-read it and which would be orders of magnitude slower.

In our prototype, a busy node will initially try to redelegate entire trees that were delegated to it before delegating subtrees of its workload to other nodes. This helps keep the overall partition as simple as possible. Further investigation of distributed algorithms for exchanging subtrees of the hierarchy is warranted to minimize the complexity of the partition, as there is a small overhead associated with each delegation because the authority must cache the containing directory (prefix) inodes for each of its subtrees. Nevertheless, a dynamic subtree partition is always an improvement over a statically hashed directory distribution, where individual directories are essentially randomly delegated.

The smallest unit of authority delegation with a purely subtree-based partition is a directory. If a single directory becomes extraordinarily large or busy, however, it may be undesirable or inefficient for it to reside on a single MDS. To address such a situation an individual directory's contents can be hashed across the cluster, such that the authority for a given directory entry is defined by a hash of the file name and the directory inode number. Because the delegation of authority is well defined for any given file name, individual MDS nodes can act authoritatively and independently for all directory operations except `readdir` (and `rename`, which often requires communication between two directories). Although a similar directory hashing approach is utilized in Lustre [22, 2], we propose that the decision to hash (or unhash) a directory be dynamic: as directories grow or become popular it may become appropriate to hash them, but if they shrink or become less popular they should be consolidated on a single node for more efficient manipulation and storage.

The distribution of workload across a cluster typically seeks to balance load. Such a distribution is dependent on an appropriate load metric: a number of MDS resources may limit MDS performance, including memory, CPU, and network utilization. Although distributing metadata based on MDS throughput might equalize relative performance of all MDS nodes, this may not maximize overall cluster efficiency because different nodes may be bound by different resource constraints. Not all portions of the hierarchy—or the cluster's current working set—may be equally busy; nodes managing widely but sparsely utilized portions of the hierarchy may be limited by memory while nodes serving busy "hot spots" may be bound by CPU or network and utilize a small fraction of their available cache. A robust load balancing strategy might seek to equalize utilization of all resources across the cluster.

A balanced distribution may not always be ideal, however. A "fair" distribution of workload consuming a scarce resource like memory may only ensure that all

MDS caches operate equally inefficiently, for example. Our experimentation indicates that maximizing total cluster throughput is not necessarily achieved by a balanced workload distribution. Furthermore, it may not be appropriate to assume—as most systems do—that all metadata is equally important. Hashed partitioning strategies probabilistically equalize resource utilization by pseudo-randomly distributing metadata across the cluster, but in reality MDS performance will favor items near the root of the hierarchy because those items (and their prefixes) are most likely to be cached. In contrast, a dynamic distribution algorithm can be predicated on *any* hierarchical performance metric, and need not be based on vanilla balancing. Policies can be formulated that prioritize active portions of the file system at the expense of archival data, for instance, by adjusting the subtree partition appropriately—a flexibility that is not possible with hashed distributions that ignore file system structure.

4.4 Traffic Control

To effectively adapt to a changing workload, the MDS cluster must also cope with situations where a large number of clients access the same file or directory in the hierarchy at the same time, either over some period of time or even suddenly and without warning. Unfortunately, extremely popular files and directories and sudden "flash crowds" are common in both scientific computing workloads (where large numbers of nodes may be acting in unison) and general purpose workloads where large numbers of users access similar files due to external events. If tens of thousands of clients access a single MDS simultaneously, that node will not be able to handle the request workload efficiently.

The fundamental problem is client knowledge of the metadata partition: if all clients know where to access any given piece of metadata at any time (based on a well-defined hashing strategy, for instance) then there is nothing to prevent them from simultaneously accessing the same item. Similarly, if clients are ignorant of the metadata distribution, then their requests must be directed randomly and forwarded within the MDS cluster, or pass through some sort of proxy, in either case requiring an extra network hop for all requests. Ideally, one would like a combination of the two situations: access to unpopular items to be directed at the authoritative MDS nodes, and access to popular items to be directed at many or all nodes (each replicating the popular metadata) to distribute traffic.

A dynamic subtree partitioning strategy can control how client requests are directed by using clients' initial ignorance of the metadata distribution to achieve near-ideal traffic flow for both popular and unpopular metadata. MDS nodes monitor the popularity of metadata using a simple access counter whose value decays over time, or any other measure or estimate of the extent to which an item appears in client caches (precision isn't necessary). All responses sent to clients include current distribution information—that is, which MDS nodes the client should contact in the future—for the metadata requested and their prefix directories, which are then cached on the client. For unpopular items, the MDS cluster will tell clients to direct future requests only at the authoritative node, while for popular items the client is told the item is replicated on many or all nodes. Because the popularity metric approximates the prevalence of an item in all client caches, the MDS cluster can effectively bound the number of nodes believing any particular file or subtree of the file hierarchy is located in any one place at all times, thus avoiding potential flash crowds before they can occur while still allowing most requests for unpopular data to be directed efficiently.

This strategy works for both explored and unexplored portions of the hierarchy. Because client requests are directed based on the deepest known prefix, any potential flood of requests will initiate from a set of mutually known (and thus popular) directories—in extreme cases, the root directory, which is known to all clients and consequently highly replicated.

4.5 Directory Locality

A hierarchical partition of metadata facilitates the exploitation of locality within the workload because the existing structure of the file system is preserved. We exploit workload locality by storing directly related information together whenever possible, and prefetching potentially related information—inodes within the same directory—to more efficiently satisfy requests in typical workloads.

Traditional file systems store inodes in a large table, typically split into "cylinder groups" in order to keep them close to their associated directory entries on disk [13]. A global inode table also facilitates the ability to create multiple "hard" links from different file names (in different directories) to the same inode, while the distribution of the table across a disk serves to minimize expensive disk seeks when a directory lookup is invariably followed by an operation involving a file's inode. This general approach to metadata storage has persisted in various forms for decades, despite the fact that the overwhelming majority of files are only linked by a single directory entry.

Instead, we store inode metadata directly with the directory entries that link to them, much like the embedded inodes used in C-FFS [7]. In the process of performing a `readdir` or directory lookup, the MDS cluster simultaneously fetches embedded inodes that subsequent transactions will likely require, without any additional disk seeks or table lookups. In addition to making single lookup operations faster, the entire contents of directories can be prefetched into cache, such that multiple lookups within the same directory—common in scientific and general purpose computing workloads—can be

be satisfied with no further disk I/O. Prefetched metadata is inserted near the tail of the cache's LRU list to avoid displacing known useful information with information that is only potentially useful, reducing the window in which it can be used.

Embedding inodes in directories also avoids the difficult problem of efficiently and consistently managing a distributed table sparsely populated by potentially billions of inodes. The lack of a globally indexed inode table has certain implications, however, most notably that an alternative (though simpler) mechanism for allocated unique identifiers must be employed. Further, if inodes are no longer globally locatable by inode number, the contexts in which inode numbers may be revealed becomes limited to those in which the containing directory is known and accessible. When inodes are exposed to clients (when a file is opened for read or write, for instance), the MDS must take care to remember where the inode is stored in the hierarchy and on disk, and to retain inodes that are deleted while still open.

Similarly, the usual POSIX treatment of multiple hard links to the same file or directory requires modification, as a directory entry linking to an inode already embedded in a different directory will have no index with which to locate it. This problem is fundamentally complicated by the fact that an inode's location within the file system hierarchy will change if any of its containing parent (prefix) directories are moved or renamed—a path name is not a reliable identifier for the same reasons that symbolic links are not durable. Fortunately, hard links are rare enough that most operations can avoid a lookup in an inode table containing only multiply-linked inodes (a table that would much cheaper to maintain because of the its vastly reduced size). Instead of slower access from both names, however, the locality benefits of embedded inodes can still be partially exploited by locating the inode with the most-popular directory entry and providing a mechanism with which to locate it. This can be accomplished with a global table mapping inode numbers to parent directory inode numbers, and populating it only with multiply-linked inodes and their ancestor directories. Combined with a reference count of all such nested items, embedded inodes can be located by recursively identifying containing directories. The table is easily modified when directories are moved around the hierarchy, and the reference counts facilitate the inclusion of only those directories and inodes that are necessary. (This is similar to the structure used in C-FFS, except that C-FFS does not use a counter and subsequently has to include all directories in the lookup table.)

4.6 Storage

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 utilize a bounded log structure for the immediate storage of updates on each metadata server. 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 preloaded with millions of records on startup or after a failure. The use of NVRAM in the metadata servers can further mask the latency of writes to the log or other storage.

Ideally, long-term data layout should be optimized for reads such that expected access patterns allow related records to be fetched without additional disk seeks. In the WAFL file system [10], this strategy is abandoned in favor of a write-anywhere approach; the authors found that simply writing metadata to disk in the order it is modified 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. We store directory contents, along with embedded inodes, together to better match expected file system usage patterns. As directory sizes vary widely, variably sized objects located in a collection of OSDs provide an appropriate shared storage medium. Directory contents (entries and inodes) can be stored in a B-tree-like structure (similar to XFS [23]) that allows incremental updates (small numbers of creates or deletes) with minimal modifications to on-disk structures (rewriting changed B-tree nodes). The tree structure also facilitates copy-on-write techniques for safe updates and advanced file system features like snapshots. OSDs appear to be an appropriate choice for the short-term per-MDS logs as well, as shared access facilitates takeover in the case of a node failure.

5 Evaluation

To validate our design decisions and evaluate the effectiveness of specific design choices we have implemented our dynamic metadata management system within an event-driven simulation environment, along with static subtree partitioning, hashing of either files or directories, and Lazy Hybrid metadata strategies to serve as points of comparison.

Our simulations validate a number of our design hypotheses. We show that subtree partitioning strategies allow metadata servers to operate with higher efficiency by avoiding duplication of data between nodes and more effectively utilizing available memory. We show the utility of exploiting directory locality in improving MDS performance by reducing I/O demands. We demonstrate the ability of a dynamic partitioning strategy to control flash

crowds by exploiting client ignorance. And finally we illustrate that the distribution of workload is a complex issue and that traditional load balancing and homogeneous distributions are not always ideal.

5.1 Simulation

Although simulation-based analysis provides a practical mechanism to evaluate file system design behavior, the accuracy of the results are very much dependent on the detail with which the file system code is implemented [24]. The focus of our simulation efforts is on MDS behavior and workload generation, and not on underlying disk storage behavior. Although a significant body of research has investigated the use of accurate disk simulation for storage system evaluation [27], the distributed metadata management systems we are evaluating can exist on any underlying disk subsystem. For this reason, we simplify the storage simulation to reflect average disk latencies and transactional throughputs only. In contrast, our metadata server prototype implements or simulates most features of the system design, including metadata updates, callback-based cache coherence (within the MDS cluster only), embedded inodes, a two-tiered storage mechanism, dynamic subtree partitioning and load balancing, and traffic control. The hashing and static subtree servers implement subsets of this functionality to accommodate the different partitioning mechanisms.

Because metadata is already quite small, it is not feasible to simulate a full scale system consisting of tens of MDS nodes and millions of files—a simulated server maintains most of the same state a real system would. Instead, we have run our simulations on much smaller file systems with less MDS memory, somewhat fewer clients and appropriately throttled I/O rates. Within this environment we demonstrate efficiency, performance and scaling behavior over a range of system variables, including MDS cluster size, cache size, and file system size.

The initial metadata partition for dynamic and static subtree partitioning is created by hashing directories near the root of the hierarchy. In a dynamic subtree partitioning system, this is clearly a suboptimal strategy, as even small subtrees near the root of the hierarchy are still scattered, but it facilitates testing by generating a relatively even distribution quickly with only a minor performance penalty. The prototype real-time workload distribution algorithm attempts to balance load by redistributing metadata based on a single load metric (a weighted combination of node throughput and cache misses). Although this approach is primitive, and (our experience has shown) a poor choice for maximizing total cluster throughput, it is sufficient to show the promise of a dynamic partitioning strategy over the alternatives.

5.2 Workload

The effectiveness of file system simulations relies heavily on the type of workload utilized. In general, using a trace from a real system is preferable to a generated workload as it more accurately reflects realistic access patterns. Although a number of file system traces analyzed in the literature are publicly available, correct metadata server simulation requires both a trace of file system activity and a snapshot of file system metadata [24]. Instead, we chose to simulate client workload based on prior research characterizing file system usage, executed against snapshots of actual file systems (which are more readily available). This approach provides us with a larger body of available content to feed our simulations as well as allowing us to easily scale our simulated client workload.

The metadata operations comprising our generated client workload are based primarily on a study of a 1997 trace of a general-purpose workload [19]. Our simulated clients submit different types of metadata operations with frequencies that mimic observed general usage patterns. Client activity is also engineered to favor operations within localized areas of the file hierarchy to resemble typical general-purpose workloads [6].

Our simulations of scientific computing workloads are based on a recent analysis [26] of file system traces from scientific computing clusters at Lawrence Livermore National Labs in 2003. This analysis found bursts of activity for which all the nodes access the same file or a set of files in the same directory. The extreme locality of reference exhibited between many clients accessing similar files presents a more difficult challenge to metadata management than general purpose workloads in which clients exhibit individual (but independent) locality, as the system must facilitate highly concurrent access to individual files ordinarily residing on individual servers.

5.3 Performance and Scalability

Initially, we evaluate the relative performance and scalability of the different metadata management strategies by fixing MDS memory and scaling the entire system: file system size, number of MDS servers, and client base. Figure 2 shows the performance degradation of individual MDS nodes for different system sizes under a predominately static (file system and client) workload. Dynamic and static subtree partitioning show the best performance, the only difference between the two being that the static strategy does not employ load balancing to adjust the initial partition. In a real workload environment, a static partition is unlikely to be practical as file systems and workloads evolve over time and hierarchies are not typically as easily partitioned as our workload (a large collection of home directories). The apparent performance penalty for load balancing is interesting, however, and is discussed in detail in Section 5.3.2. More significantly, the perfor-

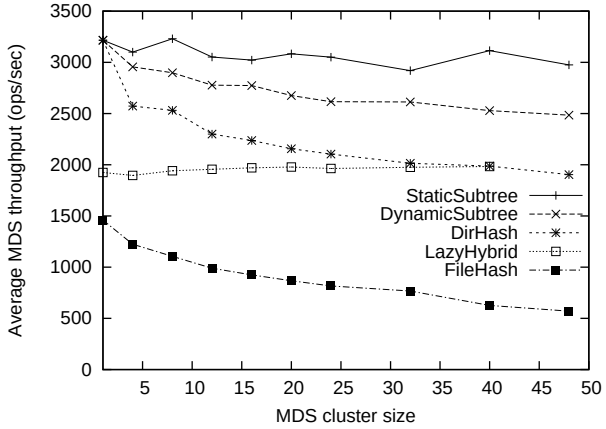

Figure 2: MDS performance as file system, cluster size, and client base are scaled.

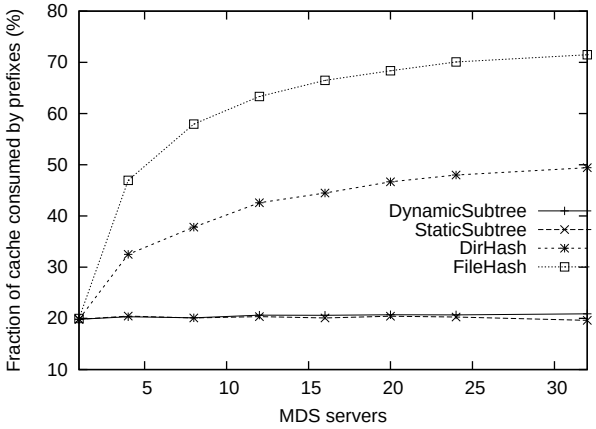

Figure 3: Percentage of cache devoted to prefix inodes as the file system, client base and MDS cluster size scales. Hashed distributions devote large portions of their caches to prefix directories. The dynamic subtree partition has slightly more prefixes than the static partition due to the re-delegation of subtrees nested within the hierarchy.

mance of file and directory hashed distributions degrades more quickly than subtree based partitions due to inefficiencies analyzed in Section 5.3.1.

File hashing and lazy hybrid distributions show significantly lower performance due to inefficient metadata I/O operations, which involve disk requests to load individual inodes into cache. In contrast, the subtree and directory hashing partitioning strategies exploit the presence of locality in the workload by embedding inodes and storing entire directories together on disk to allow efficient lookups and prefetching. The benefits of this approach are best seen by contrasting the performance of the directory and file hashing strategies, which are otherwise identical.

Lazy Hybrid performance is interesting because it scales almost linearly due to its ability to avoid performing most path traversals under the evaluated workload. However, this ability is predicated on the rarity of modifications to the directory permissions and hierarchy which must be (lazily, but eventually) propagated to potentially large quantities of metadata.

5.3.1 Prefix Caching

The performance of metadata partitioning strategies is tightly linked to metadata cache efficiency. One of the primary factors affecting cache utilization is the need to cache prefix inodes of ancestor directories for the purposes of path traversal. The overhead associated with caching prefix inodes for hashed partitions is particularly high because directories are scattered throughout the hierarchy and the prefix directory inodes to locate them must be replicated widely throughout the cluster. Figure 3 shows the percentage of MDS cache associated with prefix inodes as file system, client base and cluster size scale (as in Figure 2). The utilization for the static subtree partitioning represents a baseline for the file system simulated and is related to the ratio of directories to files, the average branching factor and average file depth. The dynamic subtree partition devotes slightly more cache to prefixes

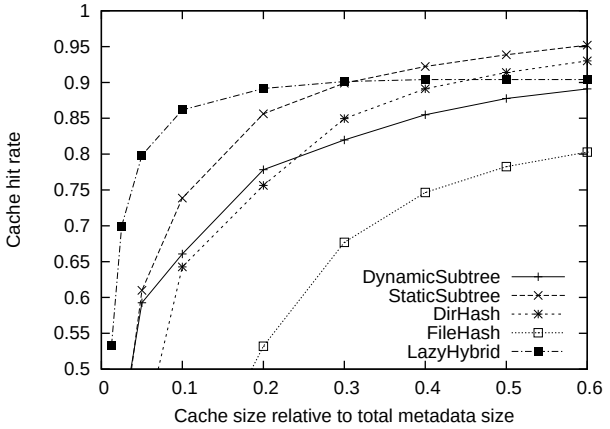

Figure 4: Cache hit rate as a function of cache size (as a fraction of total file system size). For smaller caches, inefficient cache utilization due to replicated prefixes results in lower hit rates.

to anchor subtrees nested within the hierarchy that have been re-delegated to other MDS nodes to balance load.

The consumption of cache memory by prefix inodes has the effect of decreasing the cache hit rate and thus overall MDS performance. The extent to which this affects performance is related to the average depth of directories in the hierarchy; obviously, a mostly flat namespace is more easily distributed—Lazy Hybrid tries to artificially flatten the namespace to achieve this effect. Prefix cache overhead is also greater for smaller cache sizes both because memory is more scarce and because the demand for prefixes for path traversal is related to the distribution of requests throughout the file system, not just factors proportional to the size of the cache. Figure 4 shows how cache performance varies with the cache size, expressed as a fraction of the total size of the file system's meta-

data. Note that the convergence of the hit rates as cache size increases is predicated on the degree of locality in the workload; a more random distribution of requests will result in a performance similar to smaller cache sizes.

5.3.2 Dynamic Partitioning and Workload Evolution

Static and dynamic subtree partitioning strategies show similar performance, and in many cases the total throughput for a static partition is actually better. The difference between these two mechanisms as tested is that the dynamic approach implements a simple load balancing algorithm to dynamically redistribute workload and replicate popular items. There are two key points to be made about the relative performance and merit of these approaches.

The most interesting observation is that a perfectly balanced distribution of load may not be ideal, depending on the overall benefit metric being used. If cluster performance is measured simply with the overall cluster throughput, a perfect load balance is actually counterproductive, and tends to ensure that all nodes achieve equally mediocre performance. In contrast, serving portions of the hierarchy disproportionately can result in extremely high cache hit rates for certain nodes and extremely fast response times for some clients, while overburdened nodes have very poor cache performance and go to disk for most requests. This effect highlights the fact that load balancing and "fairness" are not always the best approach, and is reflected by irregular static partitioning performance in Figure 2 due to an irregular partition. A dynamic partitioning strategy can ultimately implement any kind of workload distribution policy—even taking a laissez-faire hands-off approach resembling a static partition, though more likely implementing policies favoring more important portions of the hierarchy.

The more fundamental problem with a static partition of the hierarchy is that neither file systems nor file system workloads are static. Directory hierarchies grow in non-uniform ways over time, and the distribution of client requests changes even faster. Although hashed approaches can rely on well behaved hash functions to maintain a good distribution, subtree approaches are intentionally coarse for simplicity and efficiency, and subsequently must intelligently adjust to workload demands. Figure 5 shows the relative performance of a dynamic and static subtree partitioning approach under a changing workload. After a short time, about half of the clients change their local region of activity and create new files in portions of the hierarchy served by a single MDS. Although the static approach results in heavy load and MDS saturation on a single busy MDS while other nodes remain relatively idle, the dynamic strategy adapts by re-delegating subtrees in the busy node's workload to non-busy nodes. Despite the primitive load balancing algorithm employed, average MDS throughput is significantly higher under the dynamic approach because the cluster can adapt.

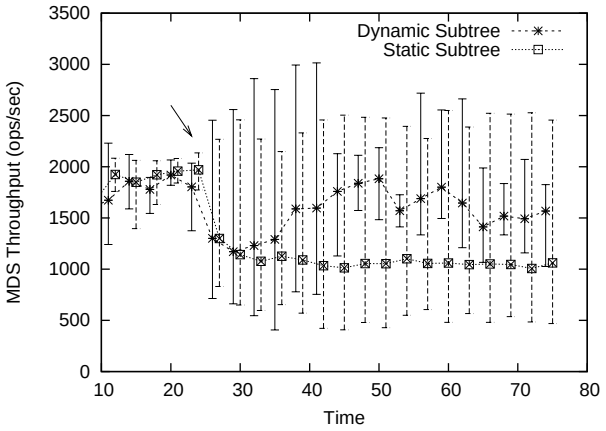

Figure 5: The range and average throughput of MDSs is shown under a dynamic workload. When clients migrate and create files in new portions of the hierarchy, a static subtree distribution remains unbalanced, while the dynamic partition re-balances load and achieves higher average performance by migrating newly popular portions of the hierarchy to non-busy nodes.

5.3.3 Client Ignorance

One consequence of a dynamic and (to a lesser degree) static subtree partitioning is that clients are initially ignorant of the location of metadata in the MDS cluster. Until clients discover new portions of the hierarchy their requests may be directed at non-authoritative MDS servers, incurring some forwarding overhead. Similarly, the movement of metadata between MDS nodes as the workload partition changes over time will result in misdirected requests. Although forwarding requests within the clusters interconnect network is likely to be cheap, it will have some effect on observed client latency. Figure 6 shows forwarded requests for the dynamic workload above in which client activity shifts to a new portion of the file system. When static partitioning is employed, forwards reflect only clients initial process of discovering new portions of the file system, while a dynamic approach additionally requires them to rediscover metadata that is migrated between nodes due to load balancing. The spike at time 25 represents a shift in workload as clients move to and modify new portions of the file system, while the higher level thereafter under dynamic partitioning is due to the load balancing algorithm being used.

5.4 Traffic Control

One of the key advantages of a dynamic partitioning strategy is the ability to manage client ignorance to prevent simultaneous access by tens of thousands of users from overwhelming an individual metadata server. Figure 7 shows the number of requests processed over time by individual nodes in the MDS cluster when 10,000 clients simultaneously request the same file, a scenario typical of many scientific computing workloads. Requests are

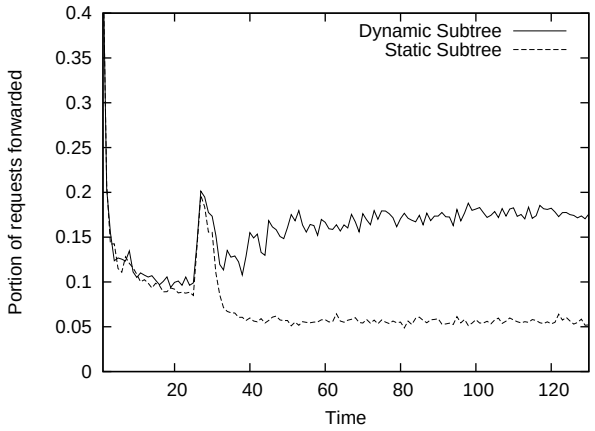

Figure 6: Forwarded requests for static and dynamic partitioning under a dynamic workload. The spike represents a shift in workload, while the difference after that point highlights overhead due to client ignorance of metadata movement from dynamic load balancing.

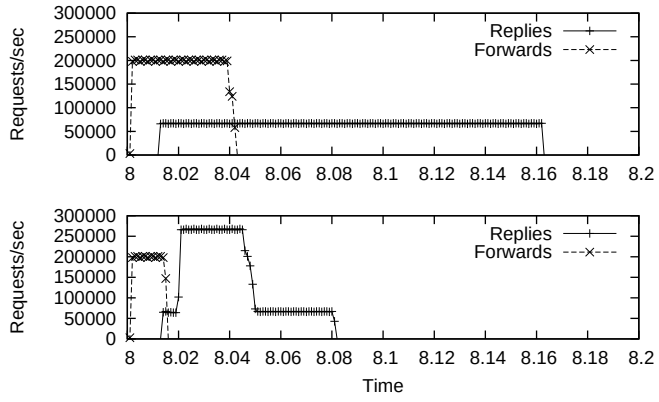

Figure 7: No traffic control (top): nodes forward all requests to the authoritative MDS who slowly responds to them in sequence. Traffic control (bottom): the authoritative node quickly replicates the popular item and all nodes respond to requests.

directed randomly because clients do not already know which MDS node is responsible for the file. Without traffic control (top), MDS nodes simply forward requests to the authoritative node who is quickly saturated and slowly (and, in real situations, inefficiently) responds. When traffic control is enabled (bottom), the authority quickly recognizes the file's sudden popularity and replicates the metadata on other nodes.

The response time from when the flash crowd begins until it is effectively distributed across the cluster is dependent on a number of factors, including the replication threshold, the rate at which client requests can be received and then forwarded by MDS nodes, and the latency of I/O requests that may be required to load the requested metadata into the cache. This response time could be reduced if non-authoritative MDS nodes recognized the sudden flood of requests and preemptively cached the metadata being requested without waiting to be told to do so, or if the authoritative node noticed the flood of requests before waiting for the metadata to be loaded from disk.

6 Conclusions

We have presented a metadata server cluster designed to service a petabyte-scale distributed file system. Our system utilizes a dynamic subtree partitioning strategy to distribute workload while maximizing overall scalability. We utilize embedded inodes to exploit the locality of reference present in both scientific computing and general purpose workloads and to simplify storage. Metadata is stored using a two-tiered strategy, initially writing updates to a log for fast commits to stable storage and for quick recovery and cache warming, and later committing directory contents to an object storage pool. We leverage the dynamic metadata distribution and collaborative caching framework to avoid flash crowds by preemptively replicating popular metadata and distributing client requests to file system hot spots. Further, our simulations indicate that uniform workload distributions inherent in hashing approaches are often not ideal, and that static subtree partitions fail to adapt to file system and workload evolution over time. Finally, we show that a dynamic partitioning approach can accommodate a variety of load distribution policies and can effectively accommodate heterogeneous system growth, scaling in terms of both the total file system size and client base.

7 Future Work

Ultimately the performance of a dynamically partitioned system will depend on good algorithms to appropriately distribute the workload based on load balancing or other policies. Although the simple load balancing algorithm utilized by our simulation prototype is sufficient to demonstrate the adaptability and scalability advantages of a dynamic approach relative to static subtree and hashed distributions of metadata, more intelligent algorithms and heuristics may be necessary to control incremental redistribution of a changing directory hierarchy and workload in a real system.

Furthermore, to fully validate our simulation findings, a fully distributed working prototype will be necessary to evaluate real-world performance at scale. We are currently completing development of metadata cluster server software for an object-based distributed file system to test our design in a large cluster environment.

Finally, thorough performance evaluation will require testing based on a wider range of workloads. The use of actual workload traces with matching file system metadata snapshots would allow us to evaluate system behavior based on more realistic workloads. For full scale eval-

uation, more realistic synthetic workloads will need to be generated to reflect expected workload characteristics at scale.

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] L.-F. Cabrera and D. D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, 1991.

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

[6] R. A. Floyd and C. S. Ellis. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):238–247, 1989.

[7] G. R. Ganger and M. F. Kaashoek. Embedded inodes and explicit groupings: Exploiting disk bandwidth for small files. In *Proceedings of the 1997 USENIX Annual Technical Conference*, pages 1–17. USENIX Association, Jan. 1997.

[8] 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.

[9] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, San Jose, CA, Oct. 1998.

[10] 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.

[11] 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.

[12] D. Long, S. Brandt, E. Miller, F. Wang, Y. Lin, L. Xue, and Q. Xin. Design and implementation of large scale object-based storage system. Technical Report ucsc-crl-02-35, University of California, Santa Cruz, Nov. 2002.

[13] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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, June 2000.

[20] 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.

[21] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 231–244. USENIX, Jan. 2002.

[22] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, July 2003.

[23] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 1–14, Jan. 1996.

[24] C. A. Thekkath, J. Wilkes, and E. D. Lazowska. Techniques for file system simulation. *Software—Practice and Experience (SPE)*, 24(11):981–999, Nov. 1994.

[25] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long. OBFS: A file system for object-based storage devices. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, Apr. 2004. IEEE.

[26] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, Apr. 2004.

[27] J. Wilkes. The Pantheon storage-system simulator. Technical Report HPL-SSP-95-14, Storage Systems Program, Computer Systems Laboratory, Hewlett-Packard Laboratories, Palo Alto, CA, May 1996.

[28] 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.