

An OSD-based Approach to Managing Directory Operations in Parallel File Systems

Nawab Ali ^{#1}, Ananth Devulapalli ^{*2}, Dennis Dalessandro ^{*3} Pete Wyckoff ^{*4} P. Sadayappan ^{#5}

[#] *Department of Computer Science and Engineering, The Ohio State University
395 Dreese Laboratories, 2015 Neil Avenue, Columbus, OH 43210, USA*

¹ alin@cse.ohio-state.edu

⁵ saday@cse.ohio-state.edu

^{*} *Ohio Supercomputer Center*

1224 Kinnear Road, Columbus, OH 43212, USA

² ananth@osc.edu

³ dennis@osc.edu

⁴ pw@osc.edu

Abstract—Distributed file systems that use multiple servers to store data in parallel are becoming commonplace. Much work has already gone into such systems to maximize data throughput. However, metadata management has historically been treated as an afterthought. In previous work we focused on improving metadata management techniques by placing file metadata along with data on Object-based Storage Devices (OSDs). However, we did not investigate directory operations. This work looks at the possibility of designing directory structures directly on OSDs, without the need for intervening servers. In particular, the need for atomicity is a fundamental requirement that we explore in depth. Through performance results of benchmarks and applications we show the feasibility of using OSDs directly for metadata, including directory operations.

I. INTRODUCTION

Today, many scientific applications [1], [2], [3] generate data of the order of terabytes and petabytes. This has motivated the development of peta-scale file systems in order to accommodate the data requirements of high-performance computing applications [4]. Recent advances in networking [5] and other cluster technologies have enabled the design of large-scale file systems [6], [7], [8], [9].

The predominant focus of current file system research tends to be in the realm of data throughput and redundancy. Metadata management is given less consideration. However, the performance of metadata operations such as file lookups and concurrency management have a significant impact on the overall file system performance. Roselli *et al.* [10] studied file system traces on different environments and concluded that metadata operations, while small in size, account for more than 50% of all I/O operations. In fact, the metadata load is often a major bottleneck [11], [12] in parallel file systems.

With the introduction of a standard for object-based storage devices (OSDs) [13] we are given the opportunity to revisit the architecture of parallel file systems. An OSD is a logical storage device that replaces the traditional block interface with an object interface. An OSD-based file system views a file as a collection of objects rather than as a linear array of blocks.

OSDs also manage their internal data layout, simplifying the design of file systems that store data on OSDs.

Due to the higher functionality available from OSDs, it is reasonable to consider using them as directly attached storage elements in a distributed file system. However, there is a significant mismatch between the requirements of a parallel file system and the operational interface exported by OSDs. Our research is focused on mapping the capabilities offered by OSDs to parallel file system semantics. In previous work [14], [15], we have successfully moved file I/O and file metadata operations from dedicated servers to OSDs. This work addresses the problem of designing the representation of directories on OSDs, and access methods that ensure directory consistency. Enabling both data and metadata operations directly on OSDs simplifies the design of parallel file systems by removing the need for supplemental server nodes, and offers increased scalability, performance and lower management overhead. This is the main motivation for the paper.

Another difficulty in the design of directory trees comes from the desire to spread the metadata load across multiple servers, or OSDs in our case. Adding a file to a directory may involve actions on two different devices. File systems such as PVFS [6] already implement distributed metadata, and rely on ordering guarantees by the servers to ensure that directories are always internally consistent.

Moving directory operations to OSDs offers significant challenges. To ensure correctness, directory operations such as insertion and removal must be atomic. However, OSDs lack atomic operations and do not provide support for locking or transaction semantics. While it is always possible to use a client-side distributed lock manager to guarantee consistency, this solution adds complexity and limits performance.

In this work we present algorithms that enable the use of OSDs to store directories. Since OSDs lack support for atomic primitives, we have designed a Compare-and-Swap operation that is tuned for manipulating directories, along with two representations of directories in native OSD elements. We implement the designs in a modified version of PVFS [6]

and evaluate the performance through microbenchmarks and a metadata-intensive application.

II. BACKGROUND

This section provides a brief discussion of the technologies relevant to our research.

A. Object-based Storage Devices

Object-based storage devices (OSDs) are a next step in the evolution of disk interface technology. Unlike traditional block-based disks that represent storage as a linear array of bytes and export a logical block address (LBA) interface, OSDs export an object interface. Enabling a high-level object interface allows OSDs to take over low-level data layout management from the host operating system as shown in Figure 1. According to the OSD specification [13], an object is a fundamental entity that has attributes and stores data. The object data is made up of a sequence of related bytes. Some object attributes can be user-defined, and all attributes are managed by the device. This capability of extensible attributes gives users powerful semantic control over management of the data.

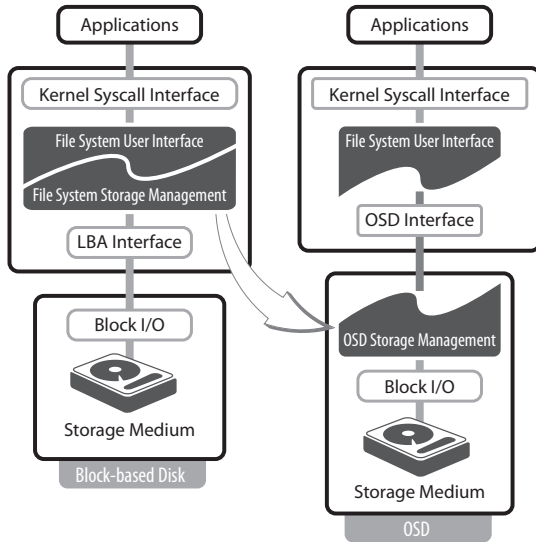


Fig. 1: Comparison of block-based and object-based disks.

B. Parallel Virtual File System

The Parallel Virtual File System (PVFS) [6] is a parallel file system designed for data-intensive high performance parallel applications. Like any distributed file system, PVFS is comprised of clients, servers, and an interconnect. There are two kinds of servers: I/O Servers (IOS) for data, and Metadata Servers (MDS) for managing the file system metadata. PVFS is designed such that it can exploit parallelism for both I/O and metadata operations. Files are striped across multiple data servers to improve data throughput. Similarly, metadata can be distributed among multiple servers for better load distribution.

III. SYSTEM DESIGN

In this section we give a brief overview of the OSD infrastructure developed previously. Further details on the design are available in related papers [16], [17], [14]. The OSD architecture is made up of two entities: the initiator that translates client requests to OSD SCSI commands, and the target that executes the commands. The initiator and target communicate using the iSCSI protocol.

A. OSD Initiator

Clients utilize the OSD infrastructure through an OSD initiator library. It translates client requests into OSD commands, and interprets OSD responses. All client commands are translated into a 200-byte command descriptor block (CDB). Due to the SCSI buffer model and large combinations of items that have to be encoded into the buffer, the translation into an OSD CDB is complex. Thus, the initiator library provides a list-type interface to the clients, removing the burden of CDB translation and buffer management complexity.

The initiator relies on the iSCSI stack in the Linux kernel for communication, but recent kernels still require modifications to enable OSD support. OSD commands rely on extended CDBs that are larger than what stock kernels will support. Also, many OSD commands can generate data flows in both directions, requiring kernel support for bi-directional commands. Lastly, to minimize data copying, we have extended I/O vector support in the kernel SCSI stack. These modifications have been published on the relevant kernel mailing lists.

B. OSD Target

Currently, there are no commodity object-based disks available. This fact, along with the desire to study extensions to the OSD command set, led us to develop a software implementation of an OSD. Our OSD target is made up of the SCSI/iSCSI layer, the OSD command processor, data management, and attribute management entities.

The SCSI/iSCSI layer is responsible for communicating with initiators using the iSCSI protocol. This layer is derived from the open-source *tgt* [18] project. The OSD command processing layer demarshals and executes OSD commands, then marshals the OSD responses. Data management is handled by an underlying file system, *ext3* in our case. The attribute management backend is implemented using SQLite, a light-weight embedded database, that allows for high-level operations on attributes. Details about the design and implementation of the attribute storage component is explored in [17].

C. Integrated Data and Metadata on OSDs

Parallel file systems often segregate data and metadata to remove the metadata operations from the critical I/O path. This has led to file system designs with dedicated metadata and I/O servers [6], [7], [8], [9]. In previous work [15], we made the case for coupling data and metadata using OSDs to mitigate the scalability issues associated with dedicated servers. Figure 2 shows our OSD architecture that combines

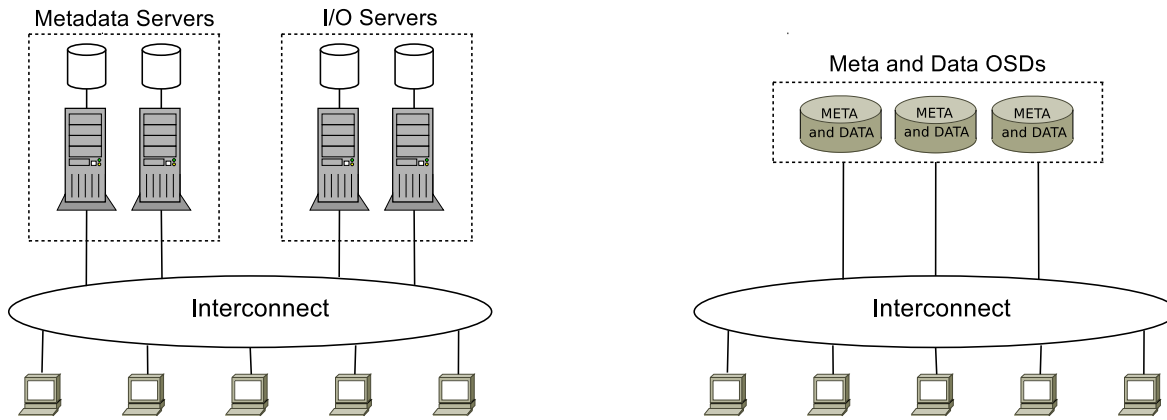


Fig. 2: Parallel File Systems; left: Segregated data and metadata servers; right: Integrated data and metadata on OSDs.

data and metadata, along with the more conventional parallel file system design.

We have exploited the capability of OSDs to store user-defined attributes with object data in order to redesign the way metadata is handled in parallel file systems. Instead of storing the file metadata on a dedicated metadata server, we store it as attributes of the first stripe of data. Since the location of the first data stripe of a file is selected randomly, the metadata is distributed across multiple disks. This architecture will serve as a basis for the following discussions.

IV. REPRESENTING DIRECTORIES ON OSDS

Before we delve into representing directories using OSD primitives, it is important to draw a distinction between file and directory metadata. Metadata in general contains descriptive information about data. In a typical file system such as NFS, the file metadata contains information such as *owner*, *group*, *permissions* and *access times* of the file. Directory metadata on the other hand contains information required to maintain unique file entries in a directory and also to maintain file system correctness during operations which require inserting and removing file entries from the directories. For instance, in a file system which uses locks to maintain file system correctness, the lock can be considered as directory metadata. This work focuses on managing directory metadata in OSD-based systems.

Managing directory operations in parallel file systems offers significant challenges both in terms of performance and scalability. Metadata access constitutes a substantial percentage of all I/O requests [10] and as such even a small improvement in common operations such as `stat` and `lookup` can potentially improve the overall I/O performance of these systems.

A directory is simply a list of name/value pairs called directory entries. A name is the user-visible name of the particular path component, while a value could be a handle to a file, a sub-directory, or a symbolic link to some other place in the file system. Directories are composed into trees by allowing directory entries to reference other directories. The one constraint on directories that make them interesting

is that the names must be *unique*. The same name is permitted in different directories, but in a single directory, no more than one occurrence of a given name may exist.

Since parallel file systems provide concurrent access to multiple clients, they must guarantee file system correctness. This is particularly important for operations that modify the directory hierarchy. For instance, a file creation operation in PVFS essentially translates to the following four steps [19]:

- Create a metadata object for the new file
- Create the data objects to hold data for the new file
- Point the metadata to the data objects
- Create a directory entry for the new file

The steps are ordered in this manner precisely to ensure that there are no dangling directory entries to files that do not exist. The remove operation is ordered similarly to ensure consistency of directories. Error states are handled by careful unwinding of the multiple steps in all cases.

While this ordering works to ensure consistency on metadata servers, it is not sufficient to do so when metadata is stored on OSDs. This is because each of the four steps above may translate into multiple operations at the OSD command level. A particularly troublesome operation is creating a directory entry for a new file. This involves two steps: ensure that the directory does not already contain a file of the same name, then insert a new entry. On a server-based implementation, the server guarantees that a single thread performs both these operations sequentially, with no other directory modifications in between. On OSDs, this is not possible.

The rest of this section discusses various ways of representing directories using OSD primitives, showing the need for some sort of atomicity across the fundamental OSD operations. The next section discusses alternatives for locking. Section VI briefly presents issues with name hashing that are motivated by the representations below.

A. Directory Entries as Object Data

OSDs are built around the concept of objects, which are simply a linear series of bytes referenced by a unique handle. One obvious way to represent a directory on an OSD is to

encode all the directory entries into bytes in an object. Each directory is represented by an OSD object, and the entries are written to and read from the object using regular file I/O operations. While simple, this design has serious drawbacks. The object data might have to be parsed by the client for all directory operations, especially in heterogeneous architecture systems. This adds latency to common operations such as looking up file entries. Further, since all entries are stored in a single object, operations such as insert and remove would likely require reading all the bytes, which could be very slow for large directories.

B. Directory Entries as Object Attributes

Another approach involves storing directory entries as attributes of an object. One of the main advantages of OSDs over block disks is the ability of OSD objects to have user-defined attributes [13]. This feature allows file system designers to extend the semantics of objects and develop new data and metadata storage paradigms. In this design, each directory is represented by an OSD object and the directory entries are stored as user-defined attributes of the object. Attributes are referenced by two 32-bit values, called the *attribute page* and the *attribute number* within the page.

By representing directory entries as attributes of an OSD object, operations such as directory lookups, inserts and deletes are reduced to fetching an attribute, inserting a new attribute value, and deleting an attribute, respectively. The use of a hash function to calculate the location of the attribute results in all directory operations being executed in constant time.

By limiting the location of attributes to a single page, it is possible to use an existing mechanism to fetch all attributes on a page, facilitating directory listings. If four billion entries in each directory are insufficient, a few pages could be used, or a different mechanism to gather all attributes on all pages may be used.

One complication of this approach is the need to handle collisions in the hash space. Each directory entry name is converted into a location in the attribute space, and collisions are possible, albeit unlikely. We use a hash chaining mechanism that builds linked lists in the attribute space to manage collisions, described further in Section VI. Another issue is that good directory performance relies on a good implementation of attributes in the OSD. Given the emphasis on attribute use in the object-based model, hopefully quality implementations will result.

This is the approach that will be implemented and evaluated in the experiments in Section VII.

V. SERIALIZATION

There are two main reasons why we need serialization to manipulate directories in parallel file systems. The first is to ensure the uniqueness of entries in directories. During the file creation phase, an entry for the file is placed in its parent directory. If we do not employ any mutual exclusion algorithm, then it is possible for multiple clients to place the same entry in the parent directory, resulting in an inconsistent file system.

The second reason for locking algorithms is to ensure the consistency of data structures. For instance, during file removal, PVFS starts by removing the entry from the parent directory before deleting the file object. In case the file system is unable to delete the object (e.g. when the object to be deleted is a directory that is not empty), it recreates the entry for the object and places it back in the parent directory. In the absence of a locking algorithm to ensure mutual exclusion, another client could potentially replace the original directory entry and link it to another file object. This is particularly relevant when directory entries and objects live on different metadata servers.

Other directory operations such as `stat` and `lookup` do not require any form of locking because they are single-step operations and also because they are based on the idea of a best-effort service. A successful `lookup` does not guarantee that the file handle is valid because another client could have deleted the file object after the `lookup`. While it is possible to envision a “lookup and lock” operation, current applications have not demonstrated the need for this.

A. Locking Mechanisms

The following sections discuss ways to implement atomicity guarantees in OSDs.

1) *SCSI Reservations*: The SCSI primary command set [20], the basic set of commands that all OSDs and other SCSI devices implement, includes support for reservations. Persistent reservations permit a given client to claim a target device exclusively, locking out commands from other clients. It is useful for applications where a single server is the only one to read or write to a disk. For our purposes, this level of locking is too coarse. Locking the entire device just to update one directory precludes any other operations, such as reading data from a file not even in the directory of interest.

2) *Object Lock*: A simple extension to the OSD protocol would be the addition of a `LOCK` command that works on a particular object, and an argument to indicate whether to lock or unlock the object. If the object was already locked, by perhaps a different client, the command would fail. This option is a straightforward improvement on SCSI Reservations that has the advantage of a smaller locking granularity that is more appropriate for our situation.

In practice, we implemented the object lock function using the compare-and-swap function described next, limiting it to values of zero and one, and using a well-known attribute location to indicate the binary value of “locked” for an object.

3) *Atomic Operations*: Atomic primitives are operations that are guaranteed by the underlying system to complete without being interrupted. Most modern computer instruction set architectures provide some sort of atomic instructions such as `Compare-and-Swap (CAS)` and `Fetch-and-Add (FA)`. These instructions can be used to design synchronization primitives such as semaphores and mutexes.

The current OSD specification [13] does not provide any atomic primitives. Since these primitives are essential to implementing directory operations on OSDs, we have proposed

extensions to the current standard. These extensions include support for device-based compare-and-swap and fetch-and-add operations. Our OSD target implementation includes support for these new atomic operations.

Our CAS extension works on attributes. The client provides two values to the OSD, a compare value and a swap value. It also specifies an object identifier and an attribute page and number on which to operate. The OSD performs the following two steps atomically with respect to other operations on that particular attribute of the specified object: fetch the current attribute value and compare it to the client-provided compare value. If they are equal, update the attribute with the client-provided swap value. In either case, return the original value of the attribute to the client. The comparison is defined byte-wise. Only values of the same length with exactly the same bytes will be equal.

B. Locking Algorithms

Building on the lock and compare-and-swap primitives above, respectively, we designed two different protocols for the directory operations. Both implementations use the same approach to implement directory entry lookup and listings, which do not require any serialization. Section VII shows performance evaluations of these two options.

1) *Lock-based Directory Access Protocol*: The first of two directory access protocols we consider uses the OSD lock operation to implement a multi-client directory access protocol. The protocol to insert or remove an entry (file or sub-directory) essentially consists of the following four steps:

- Lock the directory
- Perform a lookup on the directory entry
- Insert or remove the directory entry
- Unlock the directory

The attribute number is calculated by hashing the object name. The PVFS client makes an explicit call to the OSD for each of the above steps. As the processing time on the OSD for each of these steps is small, communication overheads dominate and result in an execution time of four network round-trip times (RTTs). The remove operation also requires that we retrieve the object handle of the file or sub-directory that we want to delete from the metadata servers. This is an implementation workaround related to PVFS, though it does result in remove requiring five RTTs.

The biggest advantage of this protocol is that it is well-understood and easy to implement. However, every directory operation requires significant communication overhead with the OSDs. Also, as we will see in Section VII, the lock contention associated with multiple clients accessing a particular directory tree simultaneously, severely degrades the file system performance.

2) *Atomic Directory Access Protocol*: The lock-based directory access protocol is handicapped by high communication overhead and lock-contention issues. To mitigate the performance degradation due to the above reasons, we designed an atomic directory access protocol which exploits the CAS primitive described in Section V-A3. The generic nature of the

CAS operation (it compares an array of bytes as opposed to an integer) reduces the directory insert operation to a single message exchange.

The attribute number is calculated by hashing the file or sub-directory name. For insert, the compare value is NULL while the swap value consists of the 8 byte object handle along with the object name. The CAS operation also returns a result attribute. A successful operation will return NULL, the empty slot that was just filled by the new insert. Otherwise, we can detect that the name is already used (or that the hash of two names collided) by examining the previous entry that is returned when the CAS failed.

The remove operation is almost identical to the insert operation except that the compare value consists of the encoded directory entry we are trying to remove and the swap value is NULL. A successful removal will return exactly the item we were trying to remove. Otherwise it will have no effect and return either NULL (the entry was already removed) or some other value (the entry was re-used by a different colliding file name). The remove operation requires one other operation before this to discover the handle, as discussed above in the lock case.

VI. HASH TABLE COLLISION RESOLUTION

As described in the previous sections, we chose to implement directories as OSD objects and directory entries as attributes of that object. The location of the directory entry within the page is determined by the hash value of its name. One problem that arises when dealing with hash tables is the possibility of a collision. In our case, due to the finite number of entries in an attribute page, multiple directory entries may be hashed to the same location. The conventional solution to a collision problem is chaining [21]. When multiple entries collide into the same hash bucket, they are chained abstractly as a linked list. This is the approach we plan on using to implement hash table collision resolution in the OSD target.

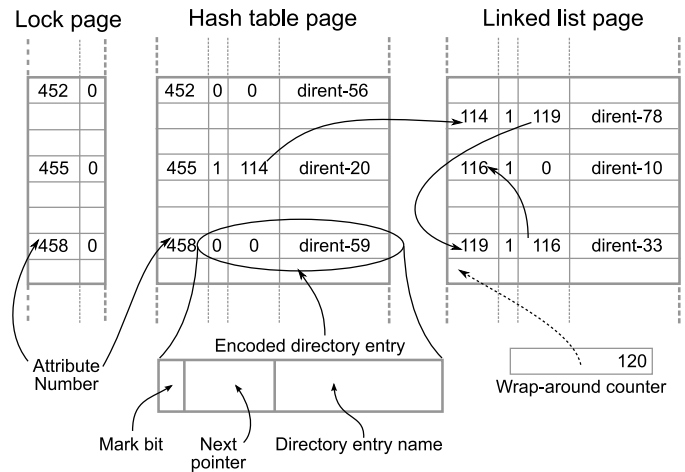


Fig. 3: Hash table data structure.

The encoding of the directory entry and the design of the data structures used is influenced by the hash table collision

resolution scheme. Figure 3 shows the OSD-resident data structures used by our scheme. The hash table page stores the directory entries. This is the entry point into the hash table. A directory entry name is hashed to get an attribute number in this page. The linked list page stores the remaining nodes of the linked list and is only used in the case of collisions. The lock page stores a lock for each entry corresponding to the hash table entry. The purpose of a lock within the lock page is to serialize modifications to linked list in the corresponding hash table entry. There is a wrap-around counter that points to the next available entry in the linked list page. It is implemented as an attribute in some other page.

Each directory entry encoding should uniquely identify a file or directory. The encoded value of the directory entry is stored as an attribute value in the hash table page and, in the case of collisions, in the linked list page. As shown in Figure 3, this encoded value is made up of a mark bit, next pointer, and the directory entry name that includes the 8-byte object identifier for the entry.

Figure 3 shows a snapshot of the state of different data structures. In the common case, for names “dirent-56” and “dirent-59”, there is no collision and the mark bit and next pointer are zero. Rarely, multiple names will hash to the same attribute slot. Here, slot 455 is shared by “dirent-20”, “dirent-78”, and two others. As a result, the next pointer specifies a location in the linked list page that contains the second directory entry for that slot. Further links in the chain are found from there. The mark bit is used to control access to the linked list page so that the fast-path algorithm can update directory entries in a single step.

VII. EXPERIMENTS

We now present experiments conducted to evaluate the performance of directory operations on OSDs. We present results from microbenchmarks and a metadata-intensive application. The experiments were conducted on a Linux cluster with each node consisting of dual AMD Opteron 250 processors, 2 GB of RAM, an onboard Tigon 3 Gigabit Ethernet NIC, and a 80 GB SATA disk. The nodes run a modified version of the Linux kernel, based on release 2.6.24-rc6, and are connected by a single SMC 8648T 48-port switch. The latency experiments use a single PVFS server or a single OSD. All the other experiments use four PVFS servers or four OSDs. We did not experience any hash collisions in these experiments.

A. Latency Microbenchmarks

The first set of experiments evaluates the latency of directory operations as a function of the number of files in the file system. These experiments attempt to measure the scalability of our directory access protocols and atomic primitives. We have compared our OSD-based directory algorithms against a standard server-fronted PVFS installation.

Figure 4 shows the time taken to perform the six fundamental directory operations. These are creation and removal of files and directories, single-entry lookups and directory listings. Results are shown as a function of the number of existing

objects in the file system. Each file system consists of a single node (either PVFS server or OSD) acting as both an I/O and metadata device.

With the exception of PVFS `readdir`, the one notable trend in all the figures is that the latency of directory operations is constant and does not vary with the file system size. This is important because it signifies that our new metadata operations can scale to large file system deployments. The directory listing benchmark (Figure 4f) lists all entries in a particular directory. This involves fetching all the entries from a remote metadata server. Naturally, as the number of files increases, it takes more time to fetch the entries. The time is dominated both in PVFS and the OSD cases by the network data transmission time.

The OSD atomic directory access protocol outperforms the lock-based protocol in most cases. This is due to the fewer number of steps required by the atomic protocol to perform an insert or remove. They perform identically on the lookup and `readdir` tests.

Stock PVFS and the OSD atomic protocols perform similarly for the two insert operations (`create` and `mkdir`). In the case of file and directory removal, OSD atomic requires an extra step to fetch the handle of the file to be removed so that it can perform the hash function that identifies the location of the directory entry. The file remove operation is further slowed down due to the need to remove all user-defined attributes associated with the object before it can be deleted. Similarly, the overhead associated with the database we use for storing attributes (SQLite) results in the OSD atomic protocol having a higher latency for file lookups.

B. Throughput Microbenchmarks

Parallel file systems are often accessed by hundreds and thousands of clients simultaneously. An important measure of file system performance is the number of operations that it can perform per second. The next set of microbenchmarks measures the throughput of some common directory operations as a function of the number of clients. Each file system uses four servers, which handle both I/O and metadata operations. The location of a new file or directory is chosen randomly.

Figure 5 shows the file system throughput for creating and removing files and directories, performing lookups and listing the entire directory.

The file and directory create operations are faster on OSDs primarily because the OSD algorithms do not have to explicitly create a metafile object. Since the file metadata is stored as attributes on the first stripe of data, OSDs save on a single object creation time. This issue can be fixed in PVFS by reordering the create steps and defining a compound operation that will both create the metafile and assign datafiles to it. We then expect the PVFS create throughput to be the same as the OSD atomic protocol.

The OSD atomic protocol has higher throughput for both `mkdir` and `rmdir` because PVFS needs to modify two different databases for creating a directory. Furthermore, the OSD atomic protocol throughput for the `rmdir` operation fails to

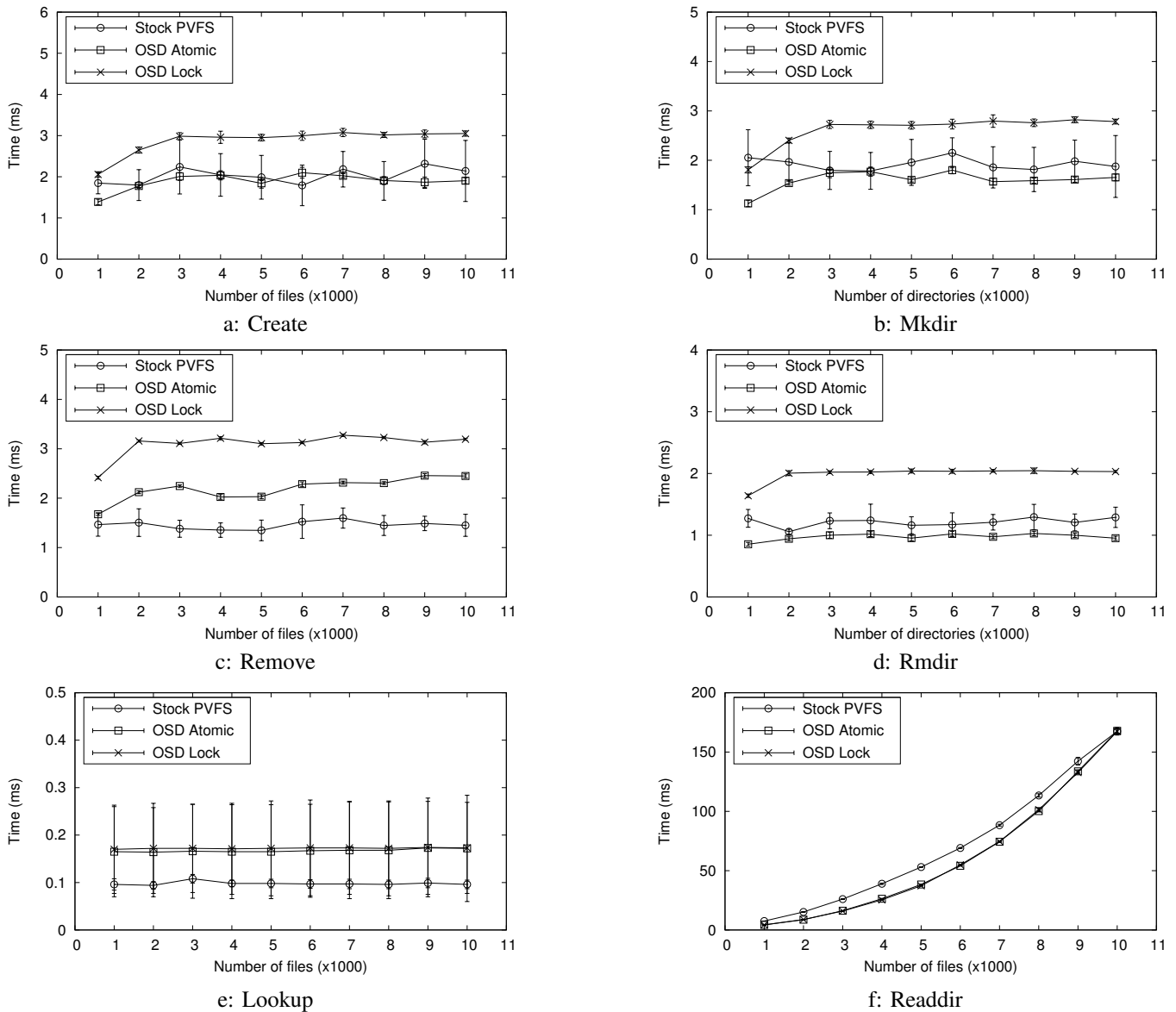


Fig. 4: Directory operation latency as a function of the number of existing objects. 1 PVFS server or 1 OSD.

check that the directory is empty before removing it. We will address this issue in future work.

The OSD atomic protocol avoids the lock contention issues that adversely affect the performance of the OSD lock-based protocol. Further, the atomic protocol uses a finer lock granularity. Instead of locking the entire directory, it only locks a portion of the attribute page associated with individual directory entry. As long as two entries do not resolve to the same hash value, they can be created and removed concurrently. The atomic and lock-based curves for the `lookup` and `readdir` operations are the same because we do not perform any locking during these operations.

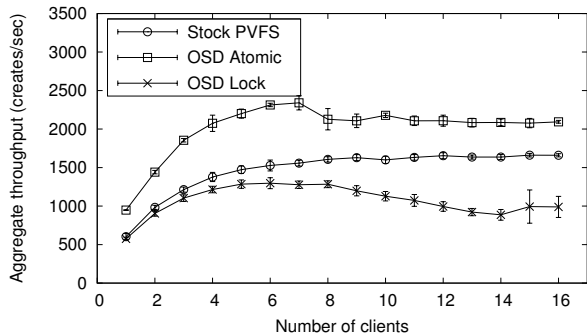
Again, the `lookup` operation throughput is a bit slower for OSDs due to SQLite overheads, and `readdir` throughput is identical for all three cases as the limit is network bandwidth. Note that these tests use an identical number of PVFS servers

or OSDs to make fair comparisons. The directory operations show similar scalability in both cases.

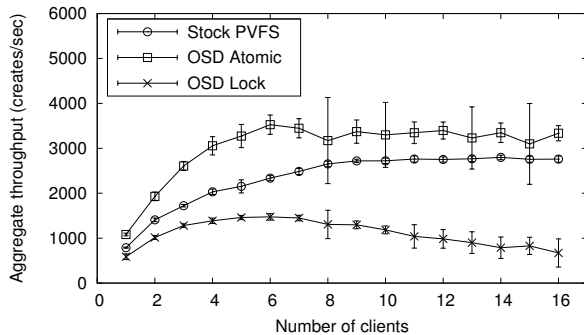
C. Scalable Synthetic Compact Application

Scalable Synthetic Compact Application (SSCA) [22] is a set of HPCS benchmarks which model high-performance computing applications. We used the file I/O component of the benchmark (SSCA-3) to evaluate the OSD directory algorithms. SSCA-3 is I/O and metadata intensive.

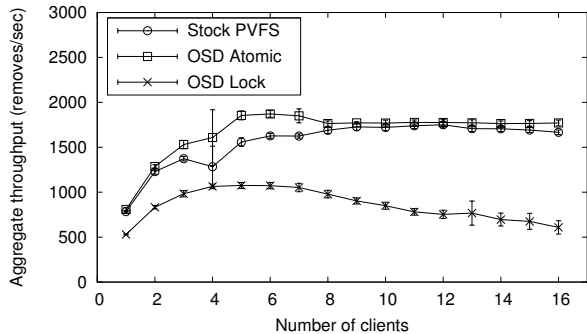
We made two changes to the SSCA-3 application. First, we modified the I/O component to issue MPI-IO calls instead of using the POSIX file I/O interface. This enables us to measure the PVFS file system performance without the overhead associated with the PVFS kernel module. Second, SSCA-3 inexplicably tends to break read and write operations into 4-byte chunks; i.e., while the kernel generates large I/O requests, a subroutine breaks them into a series of smaller reads and



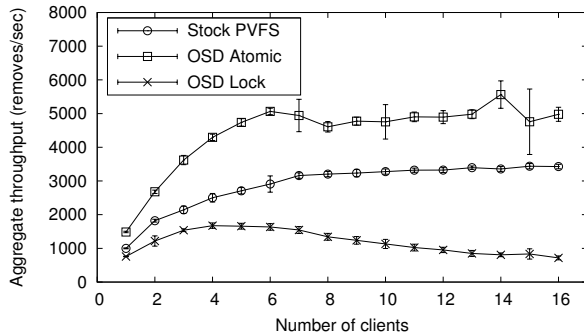
a: Create



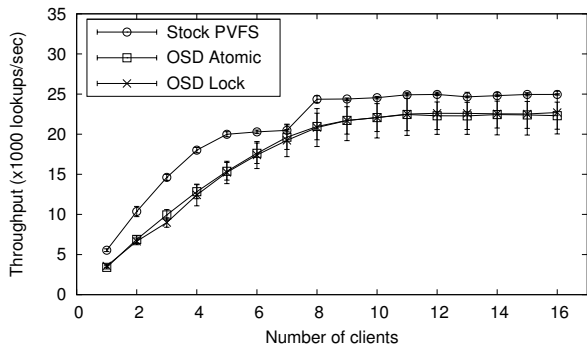
b: Mkdir



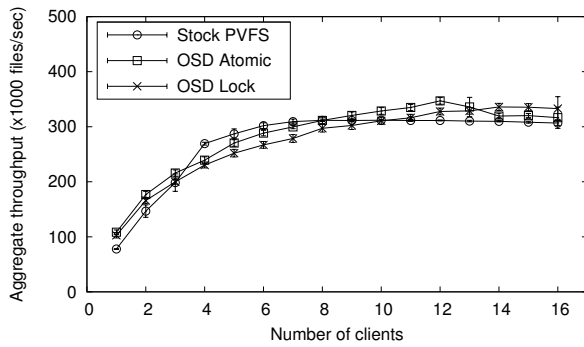
c: Remove



d: Rmdir



e: Lookup



f: Readdir

Fig. 5: Directory operation throughput as a function of the number of clients. 4 PVFS servers or 4 OSDs.

writes. We removed this behavior by aggregating all reads and writes. All experiments were carried out using four servers and one client.

Figure 6 (left) shows the execution time of the SSCA-3 benchmark for the predefined *test1* run. The SSCA-3 *test1* run is dominated by I/O rather than metadata. It creates about 5000 files and writes almost 1 GB of data. The average execution times of the OSD atomic and lock algorithms are identical and they both perform better than PVFS. This is because of the relatively small metadata footprint, most of which can be cached on the clients, factoring out some costly lookup operations. However this means that the overall time is now dominated by a large number of small I/O operations. Since PVFS is not tuned for small message sizes, its performance suffers [14].

Figure 6 (right) shows the execution time of the SSCA-3

benchmark for the predefined *test7GB* run. This configuration creates around 95000 files and writes almost 7 GB of data. We can see that PVFS performs better than both the OSD algorithms. This is because this benchmark is metadata intensive, in particular it is dominated by the lookup operation. Since the number of files is too large to fit in the client-side cache, the algorithms are forced to contact the metadata server for the file handle information. As can be seen from Figure 4e, the lookup latency for the OSD protocols is about 0.08 ms higher than PVFS. This translates to a higher average execution time for the *test7GB* run.

VIII. RELATED WORK

Distributed file systems employ myriad techniques to manage metadata. Some [23], [24], [25], [7] use a single dedicated metadata server to manage a global shared namespace. This design limits scalability and leaves the entire file system

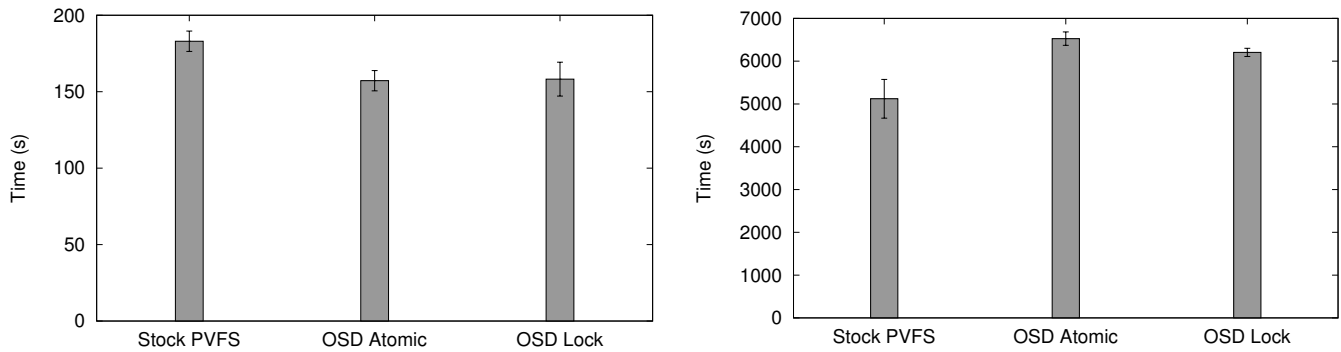


Fig. 6: SSCA-3 execution time; left: SSCA-3 test1; right: SSCA-3 test7GB. 4 PVFS servers or 4 OSDs.

vulnerable to a single point of failure. Others use multiple metadata servers for performance and scalability, employing techniques such as static directory sub-tree partitioning and hashing [26], [27] to delegate metadata management responsibilities to various servers. Hashing diminishes the problem of hot-spots that is often experienced with directory sub-tree partitioning.

The Lazy Hybrid (LH) metadata management scheme [27] combines hierarchical directory management and hashing with lazy updates. Zhu *et al.* proposed using Hierarchical Bloom Filter Arrays (HBA) [28] to map file names to the corresponding metadata servers. One recent work [29] explores multiple algorithms for creating files on a distributed metadata file system for scalable metadata performance.

A recent trend in distributed file system design is to use an object model for managing data and metadata. Ceph [8], PanFS [9] and Lustre [7] use various non-standard object interfaces to store data. However, these file systems still require the use of dedicated I/O and metadata servers. It is important to note that these file systems use objects only at the file system level and not at the disk level. OSDs as we present here are the storage device, and PVFS is the file system.

IX. CONCLUSIONS

We have described a mechanism to use native object-based storage devices as directory servers in a distributed, parallel file system. The algorithms to implement directory operations rely on the ability to perform atomic operations on OSDs. We propose a Compare-and-Swap operation for OSDs that enables an OSD-based file system to guarantee correctness. Now it is possible to implement an OSD-based parallel file system without using any dedicated I/O or metadata servers. Even with the overhead associated with using OSDs implemented in software, the performance of our OSD directory management algorithms is comparable to that of stock PVFS. We expect the performance of OSD-based file systems to improve significantly with the availability of native OSD disks. Hopefully this work will spur interest in the possibilities for scalable distributed storage systems that are enabled by using intelligent storage devices as building blocks.

X. FUTURE WORK

A major consideration in favor of using metadata servers is to limit client access to certain permitted sections of the file system. OSDs offer a fine-grained capability system that can be used to implement this. We believe that the directory schemes presented here are amenable to the enforcement of per-directory permissions, but have yet to implement it.

More forward-looking projects include the implementation of directory operations that go beyond the traditional POSIX ones discussed in this paper. Clients that are not restricted to the usual tree model may find advantages in the OSD implementations, especially those that involve searching based on user-defined attributes.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. 0621484.

REFERENCES

- [1] CERN, "The Large Hadron Collider," <http://lhc.web.cern.ch/lhc/>.
- [2] ESG, "The Earth System Grid," <http://www.earthsystemgrid.org/>.
- [3] SDSS, "Sloan Digital Sky Survey," <http://www.sdss.org/>.
- [4] S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller, "Dynamic metadata management for petabyte-scale file systems," in *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2004, p. 4.
- [5] *InfiniBand Architecture Specification*, <http://www.infinibandta.org/specs/>, InfiniBand Trade Association, Oct. 2004.
- [6] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000, pp. 317–327.
- [7] Cluster File Systems, Inc., "Lustre: a scalable high-performance file system," Cluster File Systems, Tech. Rep., Nov. 2002, <http://www.lustre.org/docs/whitepaper.pdf>.
- [8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of OSDI'06*, Seattle, WA, Nov. 2006, pp. 307–320.
- [9] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas ActiveScale storage cluster—delivering scalable high bandwidth storage," in *Proceedings of the ACM/IEEE SC2004 Conference (SC'04)*, Pittsburgh, PA, Nov. 2004.
- [10] D. Roselli, J. Lorch, and T. Anderson, "A comparison of file system workloads," in *Proceedings of the 2000 USENIX Annual Technical Conference*, Jun. 2000, pp. 41–54.

- [11] F. Wang, S. A. Brandt, E. L. Miller, and D. D. E. Long, "OBFS: A file system for object-based storage devices," in *21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST'04)*, College Park, MD, Apr. 2004, pp. 283–300.
- [12] 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 Twentieth IEEE/Eleventh NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, Apr. 2004.
- [13] R. O. Weber, "Information technology—SCSI object-based storage device commands -2 (OSD-2), revision 1," INCITS Technical Committee T10/1729-D, Tech. Rep., Jan. 2007.
- [14] A. Devulapalli, D. Dalessandro, P. Wyckoff, N. Ali, and P. Sadayappan, "Integrating parallel file systems with object-based storage devices," in *Proceedings of SC'07*, Reno, NV, Nov. 2007.
- [15] N. Ali, A. Devulapalli, D. Dalessandro, P. Wyckoff, and P. Sadayappan, "Revisiting the metadata architecture of parallel file systems," <http://www.cse.ohio-state.edu/~alin/papers/osd-md-techreport.pdf>, The Ohio State University, Tech. Rep. OSU-CISRC-7/08-TR42, 2008.
- [16] D. Dalessandro, A. Devulapalli, and P. Wyckoff, "iSER storage target for object-based storage devices," in *SNAPI '07*, San Diego, CA, Sep. 2007.
- [17] A. Devulapalli, D. Dalessandro, P. Wyckoff, and N. Ali, "Attribute storage design for object-based storage devices," in *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, San Diego, CA, Sep. 2007.
- [18] T. Fujita and M. Christie, "tgt: framework for storage target drivers," in *Proceedings of the Ottawa Linux Symposium*, Ottawa, Canada, Jul. 2006.
- [19] R. Latham *et al.*, "Parallel Virtual File System, Version 2," <http://www.pvfs.org/doc/pvfs2-guide/>.
- [20] R. O. Weber, "Information technology—SCSI Primary commands - 3 (SPC-2), revision 23," INCITS Technical Committee T10/1416-D, Tech. Rep., May 2005.
- [21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.
- [22] HPCCS, "Scalable Synthetic Compact Application," <http://www.highproductivity.org/SSCABmks.htm>.
- [23] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz, "NFS version 3: Design and implementation," in *USENIX Summer Technical Conference*, 1994, pp. 137–152.
- [24] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, "Andrew: a distributed personal computing environment," *Communications of the ACM*, vol. 29, no. 3, pp. 184–201, 1986.
- [25] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 2003, pp. 29–43.
- [26] E. Levy and A. Silberschatz, "Distributed file systems: concepts and examples," *ACM Computing Surveys*, vol. 22, no. 4, pp. 321–374, 1990.
- [27] S. Brandt, E. Miller, D. Long, and L. Xue, "Efficient metadata management in large distributed file systems," in *20th IEEE/Eleventh NASA Goddard Conference on Mass Storage Systems and Technologies*, Apr. 2003.
- [28] Y. Zhu, H. Jiang, and J. Wang, "Hierarchical Bloom Filter Arrays (HBA): A novel, scalable metadata management system for large cluster-based storage," in *Proceedings of IEEE International Conference on Cluster Computing*, San Diego, California, Sep. 2004.
- [29] A. Devulapalli and P. Wyckoff, "File creation strategies in a distributed metadata file system," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*, Mar. 2007.