File System Scalability with Highly Decentralized Metadata on Independent Storage Devices

Paul Hermann Lensing
Seagate
Universitat Politècnica de Catalunya
paul.lensing@gmail.com

Toni Cortes
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
toni.cortes@bsc.es

Jim Hughes
Seagate
james.hughes@seagate.com

André Brinkmann
Universität Mainz
brinkman@uni-mainz.de

ABSTRACT

This paper discusses using hard drives that integrate a key-value interface and network access in the actual drive hardware (Kinetic storage platform) to supply file system functionality in a Supercomputing environment. Taking advantage of higher-level drive functionality to handle metadata on the drives themselves, a severless system architecture is proposed. The key technique discussed in this paper to avoid performance degradation with highly decentralized metadata is to skip component traversal during the lookup operation. It relies on hierarchical namespace operations such as directory moves to be extremely rare to be effective. We therefore present Supercomputing file system traces about the frequency and characteristics of hierarchical namespace operations. Scalability implications are reviewed based on a fuse file system implementation.

Categories and Subject Descriptors

D.4.3 [**File Systems Management**]: Distributed file systems

Keywords

Object sorage, key-value storage, file system, distributed, parallel, supercomputing, file system design, direct lookup

1. INTRODUCTION

Key-value and object storage has considerably changed the storage landscape and has replaced standard file systems in many environments. It achieves excellent scalability by using a flat namespace to reduce interdependence between devices of a storage network as well as simplifying logic for data/metadata placement. Handling metadata management operations on individual object storage devices removes pressure from (or, ideally, any need for) centralized metadata servers [15]. However, existing approaches to reintegrate a hierarchical namespace and standard file system functionality on top of key-value stores[21, 18, 24] again rely on a separate metadata server (or server cluster) and also push functionality beyond basic key-value operations into the object storage (e.g. failure detection, managing replicas).

SC '15 November 15–20, 2015, Austin, Texas USA

This paper investigates another direction, where simple, independent object storage devices are used to coordinate all metadata management as well as to store the data. Metadata operations are in the critical path and can be a major bottleneck in parallel file systems [16, 19]. Fully distributing them among all hardware resources of the system is an attempt to increase metadata scalability, especially regarding Supercomputing use-cases such as highly concurrent create operations.

The architecture presented in this paper has been motivated by Seagate's announcement of Kinetic storage devices (see Section 2), which offer direct network access using a native key-value protocol. While the concept of simple, independent object storage devices is well established [2, 14], a hardware implementation at the drive level has not been available until this time.

Standard file system lookup behavior is to read in the file path directory-by-directory, each time discovering the location of the next path component. Without a central metadata server, this is clearly not a performant approach (average directories per path increase with system size, different devices have to be contacted for individual path components). Skipping component traversal during the lookup operation is the key technique explored in this paper to avoid performance degradation with highly decentralized metadata: The flat namespace provided by the underlying key-value storage is used directly for the lookup operation by setting a file's path as the key to its metadata.

Section 3 introduces the proposed file system architecture and its implementation on top of a flat key-value namespace and discusses the tradeoffs between simplifying metadata lookup and the additional costs to handle path permissions and hierarchical functionality such as directory moves in such as system. We show empirical data in Section 4 supporting the notion that hierarchical file system operations, while fundamental to file system functionality, are not frequent compared to other operations and that the advantages of optimizing `critical-path` functionality like lookups thus outweighs these extra costs, at least in the Supercomputing environment. Scalability results are discussed in Section 5. Measurements are performed using both a set of real Kinetic drives as well as emulating a higher number in the Amazon EC2 Cloud. Related work is discussed in Section 6.

2. KINETIC DRIVES

Unlike traditional hard drives, Kinetic drives do not expose a block interface over a local protocol such as SAS or SATA. Instead they expose a key-value interface over Ethernet, so that each Kinetic drive becomes an independent, native key-value store. Kinetic drives promise cost savings due to not having to buy, power, and administrate general purpose computing hardware to provide key-value storage. Replicating key-value functionality on each drive ensures that failures and performance bottlenecks are isolated to the drive in question without affecting other drives. Further, eliminating the server before the drives allows a failed drive to be replaced by another drive anywhere in the network, without having to be physically co-located (as is the case with standard raid configurations). However, any system built on top of these Ethernet drives cannot rely on custom software running on the Kinetic devices. Furthermore, there is some overhead added on a per-device basis by the key-value management and Kinetic drives currently only offer a bandwidth of about 50% of the obtainable bandwidth when using the traditional block interface. While many of the discussed capabilities are clearly not primarily targeted at HPC / Supercomputing, we believe the robustness aspect and scalability potential to be worth an evaluation of the devices in this context.

The Kinetic drive interface provides (besides basic put, get, and delete) three features, which are essential to our file system architecture:

- `Atomic operations` on keys in combination with key-versioning can provide test-and-set behavior: A put or delete operation will only succeed if the remote key version equals the key version that is supplied by the client. With this functionality, concurrency issues of a distributed / parallel file system that are traditionally handled by metadata servers can instead be implemented by directly using the storage backend (see also [7]).
- `GetKeyRange` requests return all keys whose alphanumeric names are in a supplied range, a feature used to implement scalable directories.
- Each drive has a `cluster version` and will only accept requests of clients supplying the current cluster version. This can be used to implement global knowledge of the cluster state: Changing the cluster version of a drive ensures that clients having an outdated view on the cluster state (e.g., cluster configuration change or a drive failure in a replication group) cannot continue operation using their outdated view of the cluster.

3. SYSTEM OVERVIEW

The system consists of two main components. A fuse client that exposes a POSIX interface to the user and a cluster of key value storage devices that store all data and metadata (see Figure 1).

Data. File data is chunked into one megabyte blocks, which are stored in data keys. The name of a data key is constructed using the file inode number and the chunk number. Due to this fixed pattern for key-names, allocation metadata beyond file size is not required.

Metadata. The full path of a file is used as a metadata key-name in order to facilitate skipping component traversal during the lookup process. To enable independent allocation of inode numbers by every client, clients reserve a 16 bit number range when connecting to the system the first time (if a client runs out of inode numbers it reserves a new block). While pseudo-random data placement is an established method, a common critique of pseudo-random metadata placement is that it sacrifices locality for load distribution [21]. When skipping component traversal metadata locality does not affect the lookup operation. While there are still use cases where locality can be taken advantage of (e.g. prefetching, directory reads, compare Section 3.5), the reduced metadata inter-dependency of the direct lookup approach drastically reduces the importance of locality.

Keys: Placement and Redundancy. There already is a huge set of literature on key placement in a dynamic, multi-target key-value environment, e.g. [22, 5, 13]. This topic is not the focus of this work and will not be discussed in detail. The implementation uses a simple hash-based placement strategy using the keyname to assign keys to devices and an even simpler write-all read-any replication scheme for redundancy.

3.1 Direct Lookup Implementation

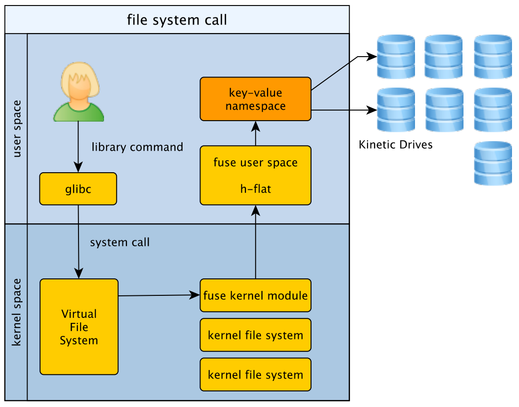


Figure 1: System Architecture & IO-Path

Figure 1 displayed the path a file system request takes from the user till it hits the drive. The standard component-based lookup logic is implemented in the Virtual File System (VFS). Direct lookup is not natively supported by current operating systems. Changes to the VFS itself require a kernel patch and thus introduce a high barrier for usage as well as limited portability. For this reason we choose an alternative option, manipulating requests before they hit the VFS layer. Path based glibc file system calls (e.g., open or chmod) are intercepted and the file paths are modified to hide all path components below the file system mount point from the VFS. This interception is achieved using library preloading.

```
/a/mountpoint/path/to/file
            →
/a/mountpoint/path:to:file
```

As a result, all files of the file system appear to the VFS as direct children of the file system mountpoint, disabling component-based VFS functionality. Inside the actual file system implementation these path modifications are reversed in order to obtain the correct keys. The main drawback of the approach is that it is not completely universal: Though very uncommonly done, user programs can use system calls directly. This cannot be similarly intercepted without kernel hacks.

3.2 Hierarchical Functionality

Some file system functionality such as moving a directory is inherently hierarchical and conflicts with a lookup approach that is decoupled from the hierarchical file system namespace. The approach taken in this paper is to support this functionality by using additional metadata, either stored on a per file basis or even system wide. A simpler variant has already been proposed in [11], we will briefly summarize it in the following and extend it for a distributed system with multiple clients.

Hierarchical Access Permissions. The standard component based lookup approach independently evaluates access permissions for every directory in a file's path. By the time the permissions of the file are checked, it is already verified that the current user has the required access permissions for every directory in the path. This is clearly no longer the case when component traversal is skipped. To support component-based access permissions with a direct lookup strategy, the complete permission set has to bet stored with each file. These *path permissions* can be evaluated in addition to the file access permissions, leading to a behavior equivalent to checking permissions on a component-by-component basis. This approach is taken in similar manner by most past work where skipping component traversal has been considered [4, 20, 12, 11]. Storing path permissions is very compact: Even for long paths the number of different directory permissions are usually very limited (e.g., all directories in a user's home directory tend to belong to that user), some supporting empirical data is presented in [11].

Directory Permission Changes. When the access permissions of an existing directory change, it can change the path permissions of all files located in the directory or one of its sub-directories. For the traditional component based lookup approach this is not problematic at all, as the path permissions are evaluated independently for each component. Storing the complete path permissions for each file to enable direct lookup is, however, problematic in this case: From a performance stand point it is not feasible to update an arbitrary number of path permissions in large scale scenarios.

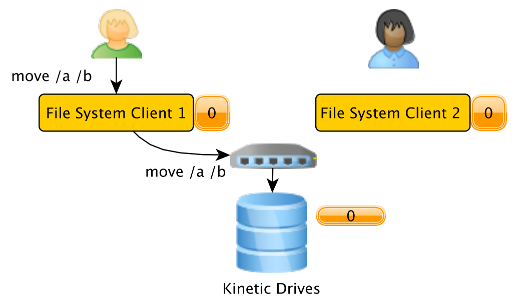
This arbitrary cost can be at least broken up and postponed by storing a logical timestamp together with the directory path as file system wide metadata. When any file or directory is accessed, the most recent permission change that occurred in the accessed path is extracted from this metadata and compared to the accessed file's path permissions. Only if the path permissions are older than the most recent permission change in the path is it possible that they are outdated. In this case they are re-computed.

Symbolic Links. A single symbolic link to a directory creates alternative paths to everything contained in that directory. When using component based lookup, the symbolic link is naturally discovered during the lookup process (it is, after all, a path component) and can be followed. To enable symbolic link discovery without component traversal, the path of a symbolic link is stored as file system wide metadata. Before a direct lookup operation this file system wide metadata is queried. If the queried path contains a symbolic link that link is followed and the actual lookup operation executes on the modified path.

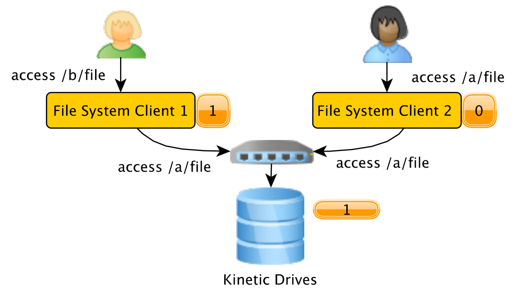
Director Rename and Move Operations. Similar to a symbolic link, a directory rename or move creates a new valid path to existing files and directories. In addition the previously correct path to all these files and directories becomes invalid and has to be reusable. Once again, it is not feasible from a performance perspective to actually move the metadata of all files to the location indicated by their new path for the direct lookup approach, although from a purely functional point of view this would solve the problem. Storing two path mappings as additional metadata is enough to provide the functionality:
`/new/path/to/dir` → `/old/path/to/dir`
`/old/path/to/dir` → `/fs/internal/name`
Following these mappings during a lookup operation leads to correct behavior without having to move any metadata. Internally the old directory path is still used for metadata keys, but this is completely invisible to the user.

Extension to Multiple Clients

The extra metadata has to be made available to multiple clients in a consistent manner. To not limit scalability and re-introduce a central servers into the lookup process, the system-wide metadata has to be cached at every client. We have chosen to concentrate on asynchronously updating this cached metadata, as synchronous updates (e.g. using a consensus based protocol) would introduce new performance and scalability issues. Asynchronous updates enable an architecture of completely independently operating clients, but leave the consistency problematic unsolved. Situations where the local knowledge is not equivalent on all clients are at least temporarily unavoidable. **This changes file system semantics**. The extra metadata available at different clients can be considered snapshots of the global information at different points of time. We call the resulting file system semantics snapshot semantics.



(a) Time Point A



(b) Time Point B

Figure 2: Snapshot Semantics

Clients have different *views* on the file system based on their local knowledge of hierarchical operations performed in the system. This concept is demonstrated in Figure 2 using a directory move operation as an example. After the move has been performed in 2a by Client 1, it can (and must) use the new directory name when accessing files in the moved directory in 2b. Client 2 does not know anything about this move operation because its local snapshot containing hierarchical information has not been updated in 2b. From its point of view the move operation simply did not happen. It continues to access files using the old directory name without issues. It cannot use the new directory name. Only when its local snapshot is updated will it learn about the directory move operation. From then on it will exhibit the same behavior as Client 1.

Implementation. The hierarchical file system operations and symbolic link creations are journaled to the key-value space in special keys outside the file system namespace (one key per operation). Similarly, deleting a moved directory or symbolic link is journaled. Every client builds an internal snapshot of path mappings by replaying the journaled operations. The snapshot is kept in an in-memory hash table for quick reference during a lookup operation. Periodically a complete snapshot is stored in addition to journaling the operations so that it can be chosen as a starting point by clients that are out-of-date by a lot. This limits the worst case time to obtain an up-to-date snapshot which is especially relevant for new clients.

System Consistency. Directory moves with snapshot semantics can pose a challenge to system consistency: In the above example, even though Client 2 is not aware of the directory move at time point B it cannot be allowed to create a file at path `/b`. To prevent this scenario, Client 1 creates a special key at the metadata location of a file or directory called `/b` as part of the directory move operation in time point A. This key specifies the snapshot version of the move operation. If a client encounters such a key, it is required to update its snapshot to the specified version as a minimum. After updating the snapshot, a client will never encounter this special metadata again: In the example, any use of path /b will be internally remapped to point to metadata at path /a.

The same mechanism is used to keep the `readdir` operation consistent. Parent directories keep the snapshot version corresponding to the newest directory move that happened inside of them as part of their metadata and clients are forced to update when they attempt an ls operation using an older snapshot. This prevents a client from not able to access a directory entry that has been returned by ls.

Extended Operation. Frequently, a snapshot that stores a specific view is much more compact than the number of executed operations suggests. For example, moving a directory `/a` to `/b` adds two path mappings to the snapshot. Moving the same directory to `/c` only updates the path mappings, the snapshot stays at the same size. Deleting the directory or moving it back to path `/a` would result in an empty snapshot even though three operations would have been journaled at this point. To further limit the size of snapshots that have to be cached at clients, some of the operations can be applied to the flat namespace in the background. If all metadata keys affected by a directory move are migrated or all metadata-keys that are affected by a directory permission change are updated, the corresponding extra metdata can be removed: Requests to the file system return the expected results at this point without any manipulation. The migration of multiple metadata keys is not an atomic operation, during the migration part of the affected keys will have the original key-name, while another part already has updated key-names. Directory moves that are currently migrated have to be marked correspondingly in all client snapshots to ensure that both possible key-names are tried during a lookup operation before starting the migration (enforced through minimum update intervals).

Summary. In essence the approach enables symbolic links with a direct lookup approach and defers the non-static costs of applying hierarchical file system operations to a flat namespace. Metadata migration (directory moves) is deferred completely and path permission updates (directory permission changes) are on-access. In distributed scenarios the approach has an impact on overall file system semantics, leading to snapshot-semantics for the provided file system functionality.

3.3 Caching

Client side caching is usually explicitly managed by metadata servers: Clients can obtain read or write caching permissions for both metadata and data, which can be dynamically revoked by the management service. In our system there are no metadata servers or other active services that could potentially take over a similar role. However, skipping component traversal during the lookup process dramatically reduces dependency on metadata caching: Except for a direct hit, the state of the cache does not influence performance at all. This allows a very simplistic caching strategy to be efficient: We use a read-only LRU cache with item expiration in order to prevent unnecessary I/O when metadata is requested multiple times in a short time period. Item expiration can be customized to adjust the trade-off between cache efficiency and system agility (a client might, for example, check for file updates with stat calls), it is set to one second as a default. We completely forego write caching and use versioned, atomic puts to prevent write conflicts as described in the following Section (Section 3.4).

While reduced metadata caching suggests itself to take advantage of direct lookup, more sophisticated schemes for data caching are completely applicable to the system and have been omitted solely for simplicity's sake. Instead, data caching follows the same concept as metadata caching, extended only by aggregating continuous writes to one megabyte chunks for performance reasons.

3.4 Serialization and Concurrency Control

In Section 2 we briefly discussed kinetic capabilities. In combination with the no-write caching strategy, versioned keys and atomic puts allow concurrency control without explicit locking. Consider two clients trying to update the same key. Both clients read the key, obtain the key-value and the key-version. Both clients update the key-value, create a new key-version (using a uuid to prevent both clients from creating the *same* new version) and try to write it back using a put command. The first request to arrive at the drive succeeds, the second request fails as the key-version that is supplied as the expected key-version is no longer current. The client whose request failed now obtains the key-value for the up-to-date key-version and re-dos the user requested operation using the new key-value. For most POSIX functionality this adequately solves concurrent access without using locks, as there is no defined order for two requests spawning from different clients. Serialization is done when necessary by creating temporary lock keys.

3.5 Directories

While directories are not accessed during regular lookup operations, they are required to support readdir functionality. One use case in Supercomputing is the parallel creation of files in a single directory by a large number of clients. Performance is usually limited by directory accesses: Clients lock the directory during file creation to ensure that a file with the same name is not created multiple times. This property is already enforced in our case by the key-value backend in combination with a fixed mapping between file path and key name: If multiple clients try to create the same key only one client will succeed. To take advantage of this property and avoid directories becoming a bottleneck for file creation, directory entries are not implemented as directory data (this would sequentialize write access of multiple clients trying to append a file name to the directory data).

Instead, a new key is generated to represent a directory entry. The key-name is constructed using the directory inode number and the file name. Readdir functionality is supported by key-range requests (compare Section 2) for all keynames starting with the directory's inode number. In this way, the generation of directory entries is not serialized by a central data structure. The number of Kinetic drives where directory entry keys of a single directory are stored is a configurable property, the ideal value depending on expected directory sizes. To minimize writes to directory metadata, the system can be mounted in a POSIX_RELAXED mode. This disables mtime and ctime timestamp updates for directories when files are created or deleted. Files can thus be created without taking locks or writing to directory metadata.

3.6 Security

Standard Kinetic access control is per-drive, not per-key. Access control according to file system semantics is enforced by the file system client, not the drives themselves. Assuming a potential attacker could steal the authentication of the Kinetic user for the file system he could use the open Kinetic protocol to communicate with the drives directly (and get / put / delete keys), bypassing the file system client. While security measures beyond standard Kinetic identification are not implemented in the file system client at this point, we want to briefly discuss the topic due to its importance.

Kinetic access rights can be granted for limited scopes of keynames. It is, for example, entirely possible to create a user John who solely can write keys that start with `/home/john`. Since metadata keynames mirror path names in our system, this feature can be used to limit access in a way that is enforced by the drives themselves. However, this form of access control is less fine grained than file system semantics and very rigid: If the access permissions for a user change the corresponding access control has to updated on every single drive in the system that user has access to.

A simple way to keep Kinetic credentials completely out of clients hands is using proxy servers. An untrusted client would simply forward file system requests to a proxy server, which has the real file system mounted and executes the operation on it and returns the result to the client. This method would actually work reasonably well and allow a client to distribute calls among multiple proxy servers due to the minimalistic caching requirements (compare Section 3.3), but it introduces a level of indirection into the system an requires additional hardware.

A more sophisticated approach that allows clients to retain direct access to the drives is proxy authentication. It allows a third-party authentication server to resolve access control on a per-key basis (the client passes a token to the Kinetic drive that is used by the drive to verify access using the authentication server). This method is not implemented in Kinetic at the time of writing but is planned for future firmware releases.

3.7 POSIX Limitations

Asynchronously updating the client-side snapshots of hierarchical file system operations as described in Section 3.2 leads to snapshot semantics and corresponding limitations regarding POSIX compatibility. While introducing this limitation was a conscious choice on our part to increase system scalability, a second limitation to POSIX compliance is inherent to any approach skipping component traversal: A lookup operation of non-existing files always returns a *file does not exist* error code, while the reported error code using traditional lookup depends on the component traversal (e.g., no access permissions to a directory, a path component is not a directory, etc.). If the file system is mounted in relaxed posix mode as described in 3.5 to increase file create performance, directory time stamps are not updated correctly. Finally, library preloading effectively limits maximum path length below POSIX standards as the whole path is seen as a filename by the operating system as described in 3.1.

3.8 Recovery

File system clients are at the same time the only actors in the system and considered to be unreliable (can crash or be disconnected from the network at any time). As many metadata operations are not atomic, this can leave the file system in an invalid state. To address this problem, multi-step metadata operations are ordered so that a crash will always leave a dangling directory entry. Partial operations can be rolled back using the file system check (fsck) tool. As mentioned in the introduction of Section 3, a basic replication strategy has been implemented to handle device failures.

4. EMPIRICAL DATA

The feasibility of storing additional metadata to support hierarchical file system operations as described in Section 3.2 directly depends on the frequency that these operations occur. The costs of applying an operation to the flat namespace correlates with the number of affected files. Intuitively it seems reasonable that directory moves and directory permission changes are far less frequent compared to other file system operations and are more frequently done on relatively new (and small) directories. There is, however, no supporting literature. These operations simply have little impact on traditional file system performance and therefore have not been the focus of file system studies. This section presents empirical data from the GPFS file system of the Barcelona Supercomputing Center. The following information is of special interest:

- Operation count by target of operation; differentiating between operations on directories and files.
- For operations targeting a directory, information about that directory. The most relevant is the *directory size*, defined as the total number of files and directories located in the directory's subtree.

	Login Nodes	Compute Nodes
lookup	52.04	10.59
access	20.42	13.46
read	14.75	26.65
write	6.62	43.67
open+close	3.93	2.34
readlink	1.15	2.67
...		
symlink	10^{-5}	10^{-4}
dir permission	10^{-6}	10^{-8}
dir rename	10^{-7}	0

Table 1: Operation Frequency (percentages)

4.1 Procedure and Challenges

GPFS traces contain the executed operation as well as the id of the user and the involved inode numbers. The traces do not, however, directly contain the information we wish to obtain. The only way to obtain this information is by reverse-mapping the inode numbers to file system paths and then do a tree-walk on a target directory's subtree to discover the *directory size*. This is an intrusive process since reverse mapping inode numbers to paths is not directly supported: The file system namespace had to be searched in a brute-force manner (limiting the search to the part of the namespace that traced user id has access rights for helped reduce the impact).

Directory information also has a temporal component. Many directories and files will eventually be deleted or the directory size can change drastically. It is therefore impractical to do the mapping and analyzing post-processing steps after all traces have completed. Instead, it was performed periodically (every 30 to 60 minutes) throughout the tracing period. Even with this periodic approach, it should be noted that due to the nature of post-processing in a dynamic environment, it cannot be guaranteed that the directories in question were left untouched after the move operation. The traces were taking as samples of overall file system activity, not all file system activity in the whole supercomputer was traced; it is therefore not possible to exclude the possibility that files were created or removed from a directory or one of its sub-directories. In a handful cases, a target had even been deleted from the file system by the time the post processing step was taken.

4.2 Results

The GPFS file system of the Barcelona Supercomputing Center has been traced for a combined 8694 hours on 1064 compute nodes and a combined 1487 hours on 4 login nodes, encompassing over 1.7 billion traced file system operations. Table 1 shows the relative frequency of the most commonly used file system operations as well as of the hierarchical operations. The big picture is as expected: The automated workloads on compute nodes tend to be more focused on file read and write operations, while the primarily user generated workloads on login nodes are heavier on metadata operations. By far the most common operation relevant to direct lookup is the creation of symbolic links, with a total of 60574 symbolic links being created in the trace period. We will examine directory moves an permission changes separately in the following.

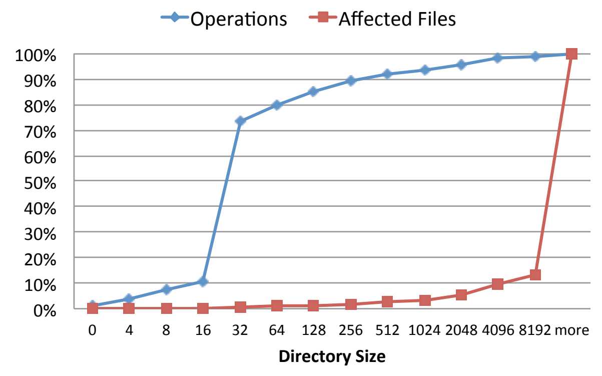


Figure 3: Cumulative sizes of moved directories

Directory Moves. The traces contain a total of 329,188 rename / move operations. Of these, 1,538, less than half a percent, targeted a directory. All observed directory move operations originated on login nodes. Figure 3 shows the size distribution of the moved directories. Over 60% of moved directories had a size between 16 and 32, over 90% have 512 or less files. Conversely, over 97% of files potentially affected by a move are in bigger directories.

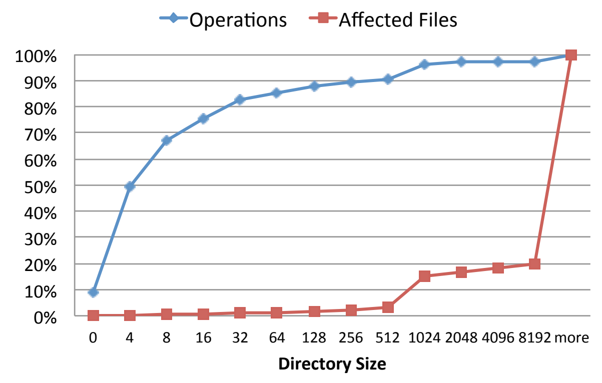


Figure 4: Cumulative sizes of directory permission changes

Directory Permission Changes. Unfortunately, the post processing step was only executed for a part of the setattr operations. In total 16% of 1.6 million setattr operations were further analyzed. 1,940 of the 262,969 analyzed operations, less than 0.75 percent, were chmod or chown operations on directories that could potentially have changed path permissions. Only a single operation originated on a compute node. Figure 4 again shows the size distribution of affected directories and repeats the overall trend observed for directories: Most operations are executed on relatively small directories, while the vast majority of files that would potentially be affected by applying the path permission change to the flat namespace are located in big directories that make up a small percentage of operations.

4.3 Conclusion

The empirical data has confirmed the assumption that directory move operations and directory permission changes are rare operations. Traced rename and setattr operations make up around 0.1 % of total traced operations, and well below 1% of these target directories. In addition, the traces show that 90% of affected directories were of size 512 or smaller. The required metadata migration or updates to apply these operations to the flat namespace is still trivial. One more interesting, if intuitive, observation is that all examined operations, with the exception of a single permission change, originated on login nodes. This is especially relevant for a supercomputing / distributed computing environment. Starting a job with a static snapshot of hierarchical information on compute nodes (disallowing move or directory permission changes) prevents these nodes from getting different views of the file system which might otherwise impact a distributed computing job. In contrast to the hierarchical functionality, symbolic links are far more frequent and cannot be applied to the flat namespace. This suggests that limiting the number of symbolic links a user is allowed to access in the file system (and thus the number of entries in that user's snapshot) might become a necessity for big deployments.

5. EVALUATION

Kinetic drives are not yet in mass production at the point of time this paper is written and the number of available prototype drives is limited. In addition to a cluster of 64 Kinetic drives we therefore use amazon EC2 to run higher numbers of clients and simulated drives.

We heavily focus on evaluating scalablity properties of the system, as a fair comparison with current Supercomputing file systems is near impossible (especially since it is not currently possible to use Kinetic hardware with these systems). Current results on metadata performance, e.g. as presented by Data Direct Networks for Lustre[1] show that scalability remains an open issue (e.g. creating files in a shared directory shows performance degradation of over 50 % compared to creating files in many directories).

Benchmarks. The Defense Advanced Research Projects Agency (DARPA) specified a set of 14 workloads to evaluate storage system scalability. An open source implementation of these test cases has been released by Cray[2]. We use two of these test cases to evaluate some base performance metrics of the implementation: The hpcsio03 benchmark tests multi-stream data throughput with large block I/O and the hpcsio04 benchmark is used to test file creation rates in combination with small write performance by creating files and writing a random amount of data (up to 64 KB) to them. In addition we use the mdtest[3] HPC benchmark to examine pure file creation performance in a single directory, as direct lookup promises increased scalability for this special case (see Section 3.5).

[1]http://www.eofs.eu/fileadmin/lad2014/slides/03_Shuichi_Ihara_Lustre_Metadata_LAD14.pdf
[2]http://hpcs-io.cray.com/
[3]http://mdtest.sourceforge.net

5.1 Hard & Software

The following software versions have been used for the evaluation on both environments: h-flat file system 0.1, kinetic-cpp-client 0.1.0, mdtest 1.9.3, hpcsio 1.1.

Kinetic Cluster. The Kinetic cluster consisted of two servers running the file system client and 64 Kinetic drives. The two client machines were both equipped with dual Intel Xeon E5-2630 CPUs, 64 GB of main memory and an Intel 82599EB dual port 10 gigabit Ethernet controller and running Ubuntu Linux 12.04. The 64 Kinetic drives were running firmware 2.0.5 and were located in three enclosures, each attached with a dual 10 gigabit connection to the network. Gcc version 4.7.3 has been used to compile the file system and benchmark utilities.

EC2. As examining scalability is the main goal of using EC2 instead of absolute performance, instance types are chosen from the very low-end (and cheap) tier. To simulate a Kinetic drive the official java Kinetic simulator (snapshot 0.6.0.2) has been started on m1.small (1 virtual CPU, 1.7 GiB main memory) instances, which have been running Amazon Linux with a 3.10 kernel and OpenJDK 1.7.0_65. Clients have been started on m3.medium (1 virtual CPU, 3.75 GiB main memory) instances with Ubuntu 14.04 and gcc 4.8.2.

Benchmark Configuration. All benchmarks use mpi to scale beyond single process / single client machines. POSIX functionality is typically serialized per thread (e.g., file creation), so we have oversubscribed mpi processes per client to test system performance instead of per-process performance. Benchmarks on the Kinetic cluster are started with 128 processes per client while the smaller EC2 clients run 32 processes per client. Cache size for HPCSIO benchmarks is limited to 1 GiB on EC2 clients. Workloads are scaled on a per-drive basis. Benchmarks have been started with library preloading to enable direct lookup (Section 3.1) and use relaxed posix mode for directory timestamp updates (Section 3.5). Unless explicitly stated otherwise, all benchmarks use a cache configuration with one second item expiration. The simple replication technique implemented for the system assigns drives to fixed size replication groups. We used a replication group size of one (effectively disabling replication) to maximize system scale (key placement targets) for the available number of drives, as the replication overhead is constant and does not change the scalability results.

- HPCSIO 03: File size is adjusted to write 1 gigabyte per drive in the scenarios using real Kinetic drives and 256 megabyte per drive in the EC2 scenarios (due to different base performance), the O_DIRECT option is specified to minimize context switches. Due to space constraints, only the first phase of the benchmark (pure write) will be considered.

- HPCSIO 04: Standard options are used (50 second runtime, 10 directories per process), the O_DIRECT option is specified to minimize context switches

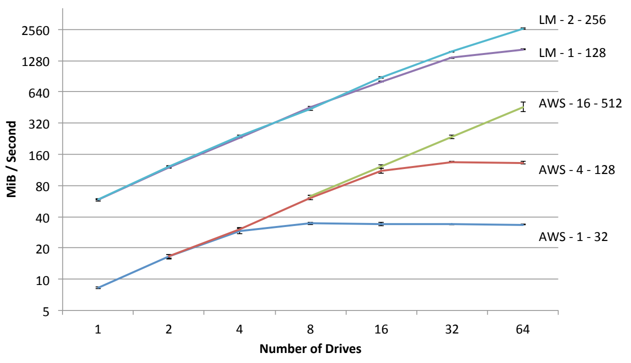


Figure 5: HPCSIO 03 - Write Throughput

- mdtest: 8192 files are created per drive, the directory-entry clustersize (drives used for directory entry keys of a single directory) is set to the entire cluster of drives for these benchmarks.

5.2 Results

Benchmarks are named based on the following scheme:
`location-#clients-#processes`
where location is either Longmont (`LM`) or the Amazon Cloud (`AWS`). The real system is compared directly with the simulated performance for up to 64 drives. Additional simulation results are shown for 256 and 512 drive scenarios. All displayed values are means of 10 benchmark runs, error bars indicate the 95% confidence intervals.

Data Throughput

Achieved write throughput for the HPCSIO 03 scenario is displayed in Figure 5. The per-drive write performance drops from roughly 55 MB/s on the real drives to 7.5 MB/s in the EC2 simulation and a single Longmont client can saturate 4 times as many drives as a single EC2 client. Regarding the scaling properties of the system, however, the scenarios display very similar characteristics: Throughput scales linearly with the number of drives, until it bottlenecks based on client performance and / or available network bandwidth and remains constant for a particular client configuration from there on. At least in the case of the Longmont clients this bottleneck is not directly hardware related. Neither network resources nor available drive bandwidth are exhausted when the clients start to bottleneck. Instead, client performance is limited by the high number of context switches required for fuse along with the inherent serialization in fuse. While this limits single-client performance it does not affect overall system scalability.

Metadata Throughput

The HPCSIO 04 benchmark (Figure 6) distributes creates across many directories and performs small data writes while the mdtest benchmark (Figure 7) is used to create files in a single directory without any data writes. Differently from the previously considered data throughput, the per-drive performance for these IOPs based workload is a lot closer when comparing the real drive with the simulator and the

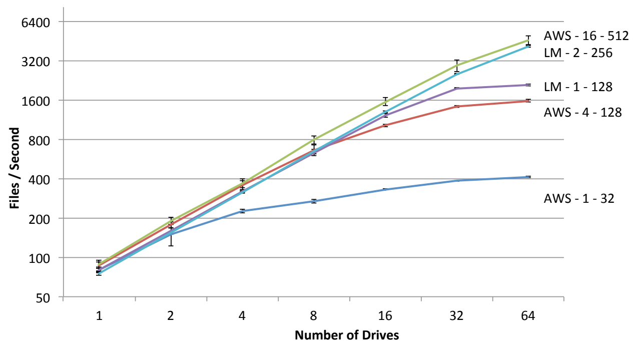


Figure 6: HPCSIO 04 - Creates and Small Writes

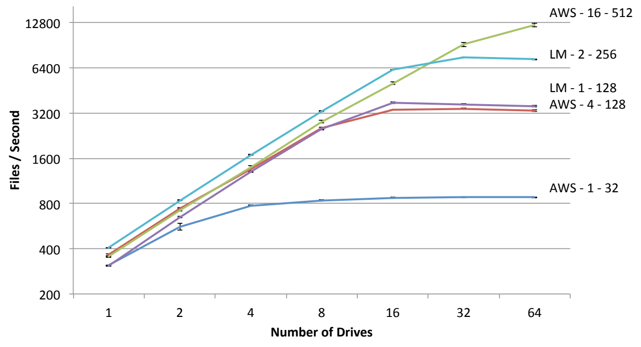


Figure 7: mdtest - Creates in Single Directory

16 client `AWS` scenarios outperforms the 2 client `LM` scenarios. From a scaling perspective, however, we observe similar behavior to data throughput. Performance scales linearly until client performance becomes a bottleneck. In the HPCSIO 04 scenarios (Figure 6) this bottleneck is not absolute. The system can take advantage of added drives, even though performance does not scale linearly anymore after the client performance starts to peak. Network bandwidth is not an issue for metadata benchmarks and workload distribution to the drives can be temporarily uneven due to being purely hash-based. It is intuitive that adding drives will prevent a single drive being temporary overloaded and thus result in a higher overall throughput. This behavior can, however, not be observed when creating files in a single directory (Figure 7), which will be discussed in the following section supported by additional data from larger-scale simulations.

Larger Simulations

To evaluate system scalability without focusing on potential bottlenecks such as limited client performance, we compare average per drive performance with a fixed client-to-drive ratio (four drives for every client) in scenarios of different scale in the Amazon cloud. This includes the previously discussed scenarios with one, four and 16 clients and is extended by 64 clients and 128 clients.

Figure 8 shows normalized per-drive performance for the benchmarks. The hpcsio benchmarks scale reasonably when taking into account confidence intervals and the shared nature of EC2 resources. The mdtest benchmark, which creates files in a single directory, however, shows a significant performance degradation in the highly-scaled examples. This is an effect of the item expiration for the read-only metadata client cache: All clients will invalidate the directory metadata once a second and have to delay further file creation until a fresh copy of the directory metadata is obtained. While the same is true for any number of clients, a higher number of clients creates a higher number of requests, which will skew the system load distribution, as all requests for the same metadata target the same kinetic drive. As shown in Figure 8, this performance degradation is not present when extending or disabling the item expiration time as described in Section 3.3. While this might not be acceptable in all use cases, Supercomputing scenarios that do no rely on the stat file system call for synchronization can take advantage.

6. RELATED WORK

There are a number of systems that have experimented with skipping component traversal during the lookup approach. The Vesta Parallel File System [6] places metadata at I/O nodes and uses path based hashing to select a master node responsible for the file metadata, but does not handle path permissions, hard links or supply a solution to directory moves beyond brute-force. Brandt et al [4] use a hashing based scheme to select one of several metadata servers for a given file path and employ a lazy on-access metadata update strategy in case of directory renames. To avoid the directory-rename problematic for hash based metadata placement entirely, some approaches [20, 12] substitute a directory id for the directory path and place metatdata based on the id. The directory id remains unchanged in case of a rename, but an additional path-to-id mapping has to be done during the lookup operation which relies on additional metadata. Storing path permissions in addition to normal file permission to enable skipping component during the lookup operation is a universal approach for the mentioned systems (except Vesta), usually in the form of dual-entry access control list (ACL) structures. The id-based metadata placement approach has also been used while keeping normal component traversal for the lookup operation [8].

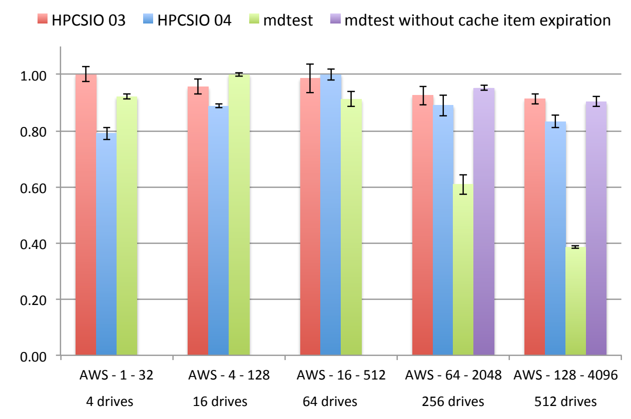


Figure 8: Normalized Per-Drive Performance

In some cases it is possible to sidestep metadata related bottlenecks by reducing the number of files in a system by storing many (logical) files in one big file. Metadata functionality in this case has to be supplied by the file structure itself; examples for this approach are Facebook's photo store Haystack [3] and the Hierachical Data Format [4].

Not taking into account skipping component traversal, the file system shares a lot of similarities with the Ceph file system [21]. Both file systems build on a global object namespace, which is provided in Ceph's case by the RADOS [23] distributed object store for the whole OSD cluster. Further, both systems omit allocation metadata in favor of algorithmic / pattern based construction of object names for file data and use a hash-based function on the object name to resolve object placement. The use of pseudo-random data placement strategies based on hashing object names or ids is a common strategy in object storage networks in general (e.g., [17]).

Another strategy for metadata placement on multiple targets besides pseudo-random hash-based placement is the use of bloom filters [25, 10] to map file names to metadata servers.

Using the capabilities of a osd based storage backend to get rid of explicit metadata servers is an idea that is also proposed by [1]. They use object attributes to support directory functionality while keeping the overall file system architecture conventional. Another approach to overcome restrictions of centralized metadata servers is introducing programmability into the object storage devices [9].

7. CONCLUDING REMARKS

By deferring the non-static costs of hierarchical file system operations in a flat namespace, skipping component traversal in a highly distributed key-value environment without metdata servers can be supported. Skipping component traversal, in turn, drastically reduces reliance on metadata caching. This allows a very light-weight file system client implementation that avoids the complexity of keeping caches on multiple clients consistent, which is especially beneficial in a Supercomputing / HPC scenario with many parallel clients. Single client performance is quite limited, a point that could potentially be addressed in the future with SSD based Kinetic devices. The observed linear scaling properties of distributed metadata in a flat namespace are what makes the approach truly interesting. Without central components there remains no obvious bottleneck for system wide performance, even when increasing system size dramatically. A further consequence of breaking the separation of data and metadata handling is that is that bottlenecks in one-sided usage scenarios are prevented as data and metadata share a common hardware resource,

APPENDIX

A. AVAILABILITY

The h-flat file system source code is available at

`https://github.com/Seagate/h-flat`

[4]http://www.hdfgroup.org/

Additional information about the kinetic ecosystem can be accessed at

`https://developers.seagate.com`

B. REFERENCES

[1] N. Ali, A. Devulapalli, D. Dalessandro, P. Wyckoff, and P. Sadayappan. An osd-based approach to managing directory operations in parallel file systems. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster)*, pages 175–184, 2008.

[2] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. Towards an object stor. In *Proceedings of the 20th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, page 165, 2003.

[3] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60, 2010.

[4] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue. Efficient metadata management in large distributed storage systems. In *Proceedings of the 20th IEEE Conference on Mass Storage Systems and Technologies (MSST)*, page 290, 2003.

[5] A. Brinkmann, S. Effert, F. Meyer auf der Heide, and C. Scheideler. Dynamic and redundant data placement. In *Proceedings of the 27th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2007.

[6] P. F. Corbett and D. G. Feitelson. The vesta parallel file system. *ACM Trans. Comput. Syst.*, 14(3):225–264, 1996.

[7] A. Devulapalli, D. Dalessandro, and P. Wyckoff. Data structure consistency using atomic operations in storage devices. In *Proceedings of the 5th International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2008.

[8] Y. Fu, N. Xiao, and E. Zhou. A novel dynamic metadata management scheme for large distributed storage systems. In *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications HPCC*, pages 987–992, 2008.

[9] R. Geambasu, A. A. Levy, T. Kohno, A. Krishnamurthy, and H. M. Levy. Comet: An active distributed key-value store. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 323–336, 2010.

[10] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian. Scalable and adaptive metadata management in ultra large-scale file systems. In *Proceedings of the 28th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 403–410, 2008.

[11] P. H. Lensing, T. Cortes, and A. Brinkmann. Direct lookup and hash-based metadata placement for local file systems. In *Proceedings of the 6th Annual International Systems and Storage Conference (SYSTOR)*, 2013.

[12] Z. Liu and X.-M. Zhou. A metadata management

method based on directory path. *Ruan Jian Xue Bao(Journal of Software)*, 18(2):236–245, 2007.

[13] A. Miranda, S. Effert, Y. Kang, E. L. Miller, A. Brinkmann, and T. Cortes. Reliable and randomized data distribution strategies for large scale storage systems. In *Proceedings of the 18th International Conference on High Performance Computing (HiPC)*, 2011.

[14] D. Nagle, M. Factor, S. Iren, D. Naor, E. Riedel, O. Rodeh, and J. Satran. The ANSI T10 object-based storage standard and current implementations. *IBM Journal of Research and Development*, 52(4-5):401–412, 2008.

[15] E. B. Nightingale, J. Elson, J. Fan, O. S. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–15, 2012.

[16] D. S. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 41–54, 2000.

[17] J. R. Santos and R. R. Muntz. Performance analysis of the RIO multimedia storage system with heterogeneous disk configurations. In *Proceedings of the 6th ACM International Conference on Multimedia*, pages 303–308, 1998.

[18] P. Schwan. Lustre: Building a file system for 1,000-node clusters. In *Proceedings of the Linux Symposium*, page 9, 2003.

[19] 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 scientific computing applications. In *Proceedings of the 21st IEEE Conference on Mass Storage Systems and Technologies (MSST)*, pages 139–152, 2004.

[20] J. Wang, D. Feng, F. Wang, and C. Lu. MHS: A distributed metadata management strategy. *Journal of Systems and Software*, 82(12):2004–2011, 2009.

[21] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, 2006.

[22] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. Grid resource management - CRUSH: controlled, scalable, decentralized placement of replicated data. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC)*, page 122, 2006.

[23] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2nd Parallel Data Storage Workshop (PDSW)*, pages 35–44, 2007.

[24] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, pages 17–33, 2008.

[25] Y. Zhu, H. Jiang, J. Wang, and F. Xian. HBA: distributed metadata management for large cluster-based storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 19(6):750–763, 2008.