Direct Lookup and Hash-Based Metadata Placement

Impact on Architecture, Performance and Scalability of Local and Distributed File Systems

Paul Hermann Lensing

Direct Lookup and Hash-Based Metadata Placement

by Paul Hermann Lensing

A thesis submitted in partial fulfillment of the requirements for the

DOCTORAL DEGREE IN COMPUTER ARCHITECTURE

Advisors:	Toni Cortés
	André Brinkmann
University:	Universitat Politècnica de Catalunya (UPC)
Department:	Department of Computer Architecture (DAC)
Pre-dissertation Committee:	David Carrera
	Juan José Costa
	Ramón Nou
Dissertation Committee:	Julita Corbalan
	Juan Piernas
	Shadi Ibrahim

Departament d'Arquitectura de Computadors

UNIVERSITAT POLITÈCNICA DE CATALUNYA

ABSTRACT

New challenges to file systems' metadata performance are imposed by the continuously growing average number of files existing in file systems. The traditional component based lookup approach can become a performance bottleneck for many workloads.

An alternative to component based lookup is the direct lookup approach. It uses hash-based metadata placement to enable direct computation of the metadata location and to completely skip component traversal.

This thesis evaluates the implications of the direct lookup approach on file system architecture, performance and scalability.

It includes an analysis of file system traces focusing on the frequency and characteristics of hierarchical operations. These properties had not been sufficiently explored in existing literature and are fundamental to the direct lookup design.

Full direct lookup file systems are implemented and evaluated for both local and distributed scenarios.

Table of Contents

List of Figures

List of Tables

Chapter 1

Motivation

In order to access (create / read / write) a file in a file system, the location of the file's metadata has to be discovered inside the system. This discovery process is called name resolution and is performed by the lookup operation.

Standard file system behavior is to read in the file path directory-by-directory, each time discovering the location of the next path component until the target data can be located.

The costs of this hierarchical lookup operation grow linearly with the number of path components. In the worst case, when no path components are already cached, this leads to I/O for every single path component.

Distributed file systems show another layer of this issue: When scaling beyond a single metadata server, the namespace has to be distributed across multiple metadata servers. Multiple nodes may have to be contacted in order to execute a single lookup operation.

An alternative approach to component based lookup is the direct lookup approach. Instead of discovering the location of a file's metadata using information contained in its parent directory, the approach relies on computing the metadata location. Because this process does not rely on the information

normally gained through component traversal, no directories have to be accessed during the lookup operation. The cost of a direct lookup operation, in contrast to the hierarchical lookup operation, thus does not depend on the path.

The advantage of a fixed cost lookup approach versus an approach where cost grows linearly with the number of path components increases with the size of a file system.

File system studies [ABDL07, GML98, MB11, Sat81] show that the average file size in the last decades has risen by almost 5 orders of magnitude slower than storage capacity. The number of files and directories existing in file systems, as well as the depth and complexity of the resulting hierarchical namespace, have grown continuously. Distributed file systems have grown even faster. Modern systems aggregate tens of thousands of hard drives to provide hundreds of petabytes of storage in a single namespace.

Since the lookup operation is required every time a file is accessed it fundamentally defines file system metadata performance. The high locality of accesses in many scenarios can conceal that lookup costs grow linearly with path size, even for the number of files encountered in today's file systems. However, when more randomized access patterns exist, as are typically observed in multi-user scenarios [ABCd96, BCF+99] caching can become infeasible. In this case metadata operations become the major single influence on overall file system performance [RLA00, WXH+04].

The hypothesis of the thesis is that the a file system designed for the direct lookup approach can provide improved metadata performance characteristics

and address the scalability limitations of file systems relying on hierarchical lookup.

To prove this hypothesis, the following objectives are achieved:

1. Discover challenges and show theoretical viability.
   Chapter 2 introduces the approach and discusses its conceptual advantages and challenges. A feasability analysis using empirical data obtained from real world file system traces is presented in Chapter 3.

2. Implement a local filesystem with direct lookup functionality.
   Chapter 4 discusses the design, implementation and evaluation of a local file system, focused on maximizing metadata performance.

3. Implement a distributed filesystem with direct lookup functionality.
   Chapter 5 discusses the design, implementation and evaluation of a distributed file system, focused on maximizing system scalability.

Related work is discussed in Chapter 6 and Chapter 7 provides an overview of achieved results.

Chapter 2

The Direct Lookup Approach

2.1 Introduction

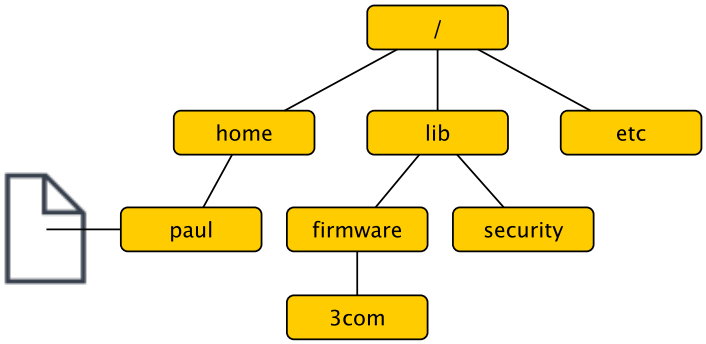


Fig. 2.1: Hierarchical Namespace

Figure 2.1 shows an example file system namespace containing a hierarchy of directories and a single file.

One of the most fundamental tasks of a file system is to allow a user to access files. In order to provide this functionality, the system has to match the file path which is provided by the user to its internal representation of the corresponding file metadata (which, in turn, contains all information needed to access the file data). This matching process of file path to file metadata

is called name resolution and is performed by the lookup operation.

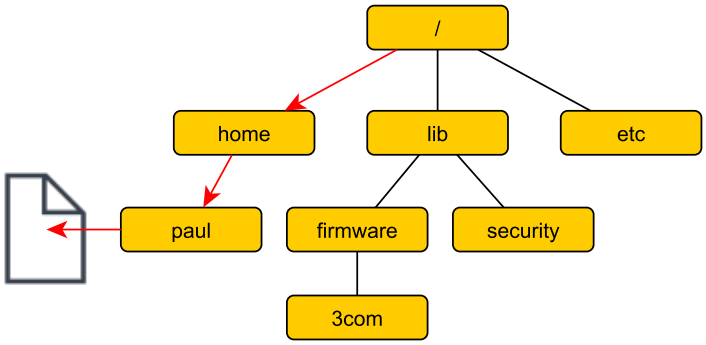


Fig. 2.2: Hierarchical Namespace with Hierarchical Lookup Operation

The lookup process matches the hierarchical nature of the file system namespace. When accessing the example file, the lookup process will search for the name of the first path component in the root directory. After finding out where the metadata for the `home` directory is stored, it can read in the corresponding metadata and data. The data of a directory contains the metadata location of all files and directories stored in that directory. This process keeps repeating with all path components until the metadata for the requested file is read in. A cache of recently used directory entries (dentry cache) is used to accelerate the lookup operation. This can drastically reduce required disk accesses but the cache hit rate depends on the locality of the workload as well as the overall size of the directory tree relative to the available cache size.

The component-by-component lookup path from the file system root directory to the example file is visualized in Figure 2.2. This hierarchical / component-

based lookup technique is employed across all wide-spread operating systems and file systems in use today.

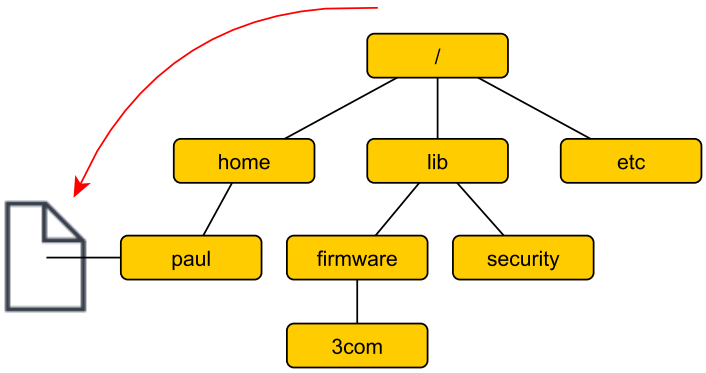


Fig. 2.3: Hierarchical Namespace with Direct Lookup Operation

A fundamentally different technique to provide lookup functionality is the direct lookup approach. It decouples the lookup operation from the hierarchical nature of the namespace as depicted in Figure 2.3: Instead of searching for metadata component by component, the location of the file metadata is directly determined by the file path.

Hash-based Metadata Placement The main technique to enable direct lookup is pseudo-random metadata placement. The metadata location is computed by using a hash function on the file path. Given the same file path, the same location can be recomputed later on in order to access existing metadata.

A common critique of pseudo-random placement strategies is that they sacrifice locality for load distribution [WBM+06]. However, 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), the reduced metadata inter-dependency of the direct lookup approach drastically reduces the importance of locality.

2.2 Advantages

The lookup operation is required every time a file is accessed. It fundamentally defines file system metadata performance.

In the worst case, when nothing relevant is cached, hierarchical lookup leads to two accesses to the the storage backend for every single path component:

- read in component metadata
- read in component data

The directory data contains the location of the next path component's metadata.

This behavior can become a scalability issue as the total cost increases linearly with the number of components in the file path and the average number of components in a path increases with overall file system size.

Distributed file systems show another layer of the same issue when scaling beyond a single metadata server: The namespace has to be distributed across multiple metadata servers, potentially requiring multiple nodes to be contacted in order to execute a single lookup operation.

The direct lookup approach directly computes the location of metadata. No path components have to be accessed during the lookup operation. The number of directories in the path as well as the size of directories therefore do not impact the operation at all. Likewise, the number of servers in a distributed system does not impact performance when the correct destination server can be computed.

In effect, the direct lookup approach decouples file system performance from the existing directory hierarchy.

This characteristic does not only hold for the lookup operation itself.

File creation performance, for example, is usually highly influenced by directory size. As it is vital for the hierarchical lookup approach to quickly search directory data for a specific directory entry, directory data is usually stored using some type of sorted tree structure. Inserting a new directory entry into a sorted structure already containing millions of directory entries (and writing the modified structure back to the storage backend) is logically more work compared to no directory entries pre-existing. When metadata location is computed, directory data does not have to be searchable to provide metadata access. It can thus be implemented in a way that pre-existing size does not impact creation performance.

Another potential benefit of computing the metadata location is that the location itself can be used for concurrency resolution: Normally a directory structure is locked during file creation to prevent multiple files with the same name being created at the same time by different users. As the computed metadata location would be the same for all these creation requests,

concurrency resolution for create requests can potentially be moved from the actual directory. This can allow many concurrent users creating files in the same directory without impacting each others performance.

2.3 Challenges

In the following the challenges of providing direct-lookup functionality are presented, as well as a high-level view of the techniques used to address them. Details regarding the implementation of the techniques both for local and distributed use are discussed in Chapter 4 and Chapter 5 respectively.

2.3.1 Operating System Support

Figure 2.4 displays the path a file system request takes from the user to the file system in a Unix operating system. The standard component-based lookup logic is implemented in the Virtual File System. For each file system operation all path components to the requested file are traversed to reach the relevant file metadata.

The hierarchical lookup logic is thus hard-coded into the operating system kernel and cannot be modified by individual file systems. While this is understandable from a historical perspective (since it has been the only used approach), it does complicate the evaluation of alternative approaches.

Two alternative ways were identified to allow direct lookup functionality despite lacking operating system support.

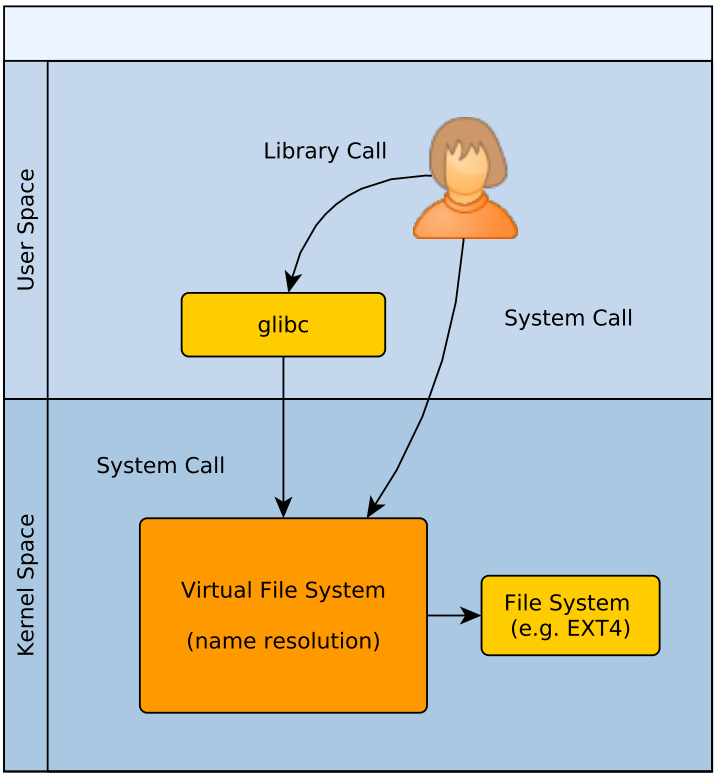


Fig. 2.4: Path of a File System Request: From User to File System

- Add functionality to the operating system kernel by modifying the name resolution in the Virtual File System.
- Hide path components from the hierarchical name resolution to bypass the hardcoded functionality.

Changing the Virtual File System It is possible to change the name resolution in the Virtual File System layer to skip component traversal and to call the real file system implementations with absolute paths. Obtaining

the absolute path (even when the file system request by the user contains a relative path) is straightforward, as the current directory is always known during name resolution.

The main downside of this approach is that the Virtual File System is part of the core Linux kernel. Accordingly, any changes require the kernel to be recompiled. This introduces a high barrier of usage. Most people are understandably reluctant to modify the kernel itself, and compiling and installing a custom kernel is not as effortless as using the package manager.

The approach additionally limits portability, as the Virtual File System code changes between different Linux kernel versions and patches would have to be written for all target kernels.

However it is universal in the sense that all possible requests to the file system will be treated equally and optimal from a performance perspective (it does not add any new layers in the lookup path).

Library Preloading The second option is to manipulate requests before they hit the Virtual File System layer.

This is achieved using the library preloading functionality provided by the dynamic linker in Unix systems (via the `LD_PRELOAD` environment variable on Linux and the `DYLD_INSERT_LIBRARIES` environment variable on OSX). It allows to load a user specified shared library before all other shared libraries which are linked to an executable and therefore to override any kind of library function.

Using this technique path based glibc file system calls such as open or chmod can be intercepted. The file path supplied to the function is then modified so that all path components below the file system mount point are hidden.

```
/a/mountpoint/path/to/file
```

$$\rightarrow$$

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

The above example shows how path components below the mountpoint are hidden by substituting the slash character[1]. After the path modification, the standard implementations of the intercepted functions are called.

As a result of the path modification, all files of the file system appear to the Virtual File System as direct children of the file system mountpoint, disabling component-based Virtual File System functionality. When the request using such a modified path enters the file system, the character substitution can be reversed to obtain the original path.

This approach is highly flexible, as library preloading is not restricted to a particular operating system kernel version and can be activated and deactivated during normal operation of the operating system.

The main drawback of the approach is that it is not completely universal: Though very uncommonly done, user programs can use system calls directly as depicted in Figure 2.4. System calls cannot be intercepted similarly to library calls without kernel hacks.

[1]The colon character was chosen as a substite for human-readability of the substituted path, in a production setting the chosen character would not be a character that is legal to use in file names

2.3.2 Hierarchical File System Functionality

Some file system functionality is inherently hierarchical and as such conflicts fundamentally with a lookup approach that is decoupled from the hierarchical file system namespace.

Besides hierarchical access permissions, the problem cases are single file system operations in the hierarchical namespace that can affect the lookup operation for many files and directories (such as a directory move operation changing the path to many files). A brute-force implementation of this functionality would result in arbitrary runtime for these operations, not an acceptable solution for a general purpose file system.

The approach taken in this thesis is to store a small amount of additional metadata for these operations to provide the functionality without the arbitrary performance penalty. The amount of additional metadata that needs to be stored in real-world scenarios is examined using empirical data in Section 3.

2.3.2.1 Path Permissions

The standard component based lookup approach independently evaluates access permissions for every directory in a file's path. When the permissions of the target file are checked during the lookup operation, it is therefore already verified that the user has the required access permissions for every directory in the path.

This is clearly no longer the case when component traversal is skipped during

the lookup operation.

To support component-based access permissions with a direct lookup strategy, the complete permission set that is normally enforced by directory traversal has to be stored with each file. These *path permissions* can then be evaluated in addition to the file access permissions during a lookup operation, leading to behavior equivalent to checking permissions on a component-by-component basis.

Generation of Path Permission Sets Path permissions are inherited from the parent directory during file creation. Since all direct children of a directory share the same path permission set, it does not have to be recomputed when multiple files or directories are created within a single directory.

Whenever a directory is created the inherited path permissions are extended by any additional access restrictions that might have been introduced. If direct children are now created inside this directory, this extended path permission set is inherited.

Example In the following, this mechanism is demonstrated using a short example with POSIX [2] access permissions. The same concept applies when using more fine grained access lists.

POSIX security attributes allow for a very limited number of access permis-

[2]Portable Operating System Interface. Standard specified by the IEEE Computer Society for maintaining compatibility between operating systems: http://pubs.opengroup.org/onlinepubs/9699919799

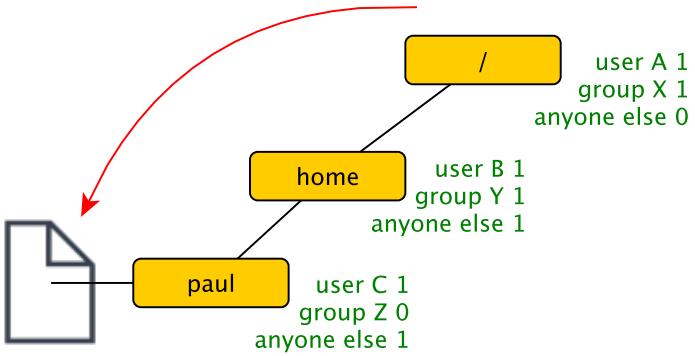


Fig. 2.5: Path with POSIX Permissions

sions and restrictions for a single path component: A specific user and / or group ID can either be required or excluded. The root directory is accessible by everyone.

Figure 2.5 shows an example path including the POSIX directory execute permissions specifying the access permissions of each directory in the path. The path permission for the individual components follow:

- Only user A and group X have access to any child of the root directory. The corresponding path permissions thus contain the restriction: user A $\vee$ group X
- As everyone is allowed access to the `home` directory, no new restrictions apply and the path permissions stay the same.
- Group Z is excluded from access to the `paul` directory. The new path permissions for all children of the `paul` directory thus are: (user A $\vee$ group X) $\wedge \neg$ group Z

Figure 2.6 extends on Figure 2.5 by displaying the above discussed path permissions in addition to the directory POSIX permissions.

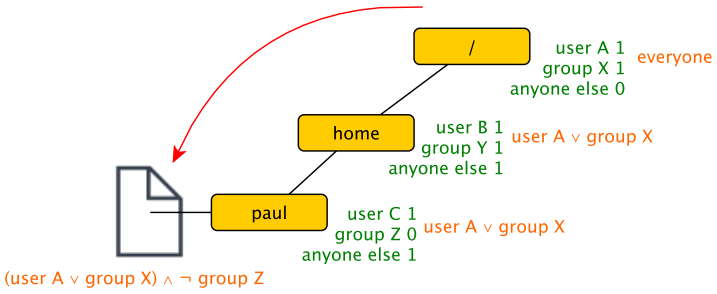


Fig. 2.6: Path with POSIX Permissions & Path Permission Sets

2.3.2.2 Changing Access Permission of Existing Directories

When the access permissions for an existing directory are changed, this can implicitly change the path permission of all its children in the hierarchical namespace; all files and directories in that directory's subtree. In the worst case this affects the path permissions of every single file in the file system.

For the traditional component based lookup approach this is not problematic at all, as access permissions are evaluated independently for each component.

Storing the complete path permissions for each file to enable direct lookup is, however, problematic when directory permissions are changed. Updating all affected path permissions would lead to a runtime proportional to the number of affected files. An arbitrary runtime file system operation is however not acceptable from a performance and usability point of view.

The alternative to access permission changes of directories having an arbitrary runtime is to postpone the required path permission updates. This is the approach taken in this thesis. Path permission updates are postponed to the point of time a file is accessed by the user.

Description of technique A logical timestamp of when a directory permission change occurred is stored together with the directory path as global 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 logical time stamp of 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 and have to be recomputed. In this case parent directories are queried recursively until valid path permissions are found, and permissions along the queried path are updated (or re-validated if they did not change).

Example Assume the file `/d1/d2/file` was created at time point `[time3]`. Its path-permission set is valid for any time point `<= [time3]`.

At this point there is no entry in global metadata. Lookup operations on the file can complete by evaluating the file's path-permissions.

The permissions of `/d1` are now changed at time point `[time4]`. A corresponding entry is inserted in the global metadata: `/d1` $\rightarrow$ `[time4]`.

A lookup operation on `/d1/d2/file` now detects that the stored path permissions are invalid. The path permissions are valid only for `<= [time3]` but the minimal required logical time stamp for any path containing `/d1` is `[time4]`.

Parent directory path permissions are checked recursively, until path permissions valid for `[time4]` are found and all invalid path permissions are updated. Accordingly, after the lookup operation, both `/d1/d2` and `/d1/d2/file` have path-permissions valid for `[time4]`.

2.3.2.3 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.

As components are not accessed at all in the direct lookup approach (compare Figure 2.3), it is not possible to discover symbolic links in the same manner.

To enable symbolic link discovery without component traversal, symbolic links are stored as global metadata.

Before a direct lookup operation this global 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.

Example Assume a symbolic link is created at `/symlink` to the directory `/d1/d2`. A corresponding entry in the global metadata is created:

`/symlink` → `/d1/d2`.

A lookup operation for `/symlink/file` will now remap its path to `/d1/d2/file` using the global metadata before executing.

Hard Links Hard links to directories are forbidden in UNIX systems, only hard links to files have to be provided. Functionality can therefore be provided without global metadata, by placing an internal forwarding hint at the path indicated by the hard link that references the actual file metadata. The reference is followed during the lookup process. All hard links of a file lead to the same metadata, providing full hard link functionality.

The cost of this approach is an indirection step. The direct lookup operation thus becomes a two step instead of a one step process for hard links.

2.3.2.4 Directory Renames and Moves

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.

Instead of moving any metadata, the move operation is again stored in global metadata. Storing two path mappings as additional metadata is sufficient to

provide full functionality: From the target path to the old path and from the old path to a unique fs-internal name. Following these mappings during a lookup operation leads to correct behavior without having to move any metadata, even when re-using a directory name.

Example Assume existing directories `/path/to/dir1` and `/path/to/dir2`.

After moving `/path/to/dir1` to `/path/to/dir1.tmp` the following mappings exist.

$$\begin{aligned} \text{/path/to/dir1.tmp} &\rightarrow \text{/path/to/dir1} \\ \text{/path/to/dir1} &\rightarrow \text{/unique\_name1} \end{aligned}$$

If a user attempts to access `/path/to/dir1.tmp`, the file system internally changes the accessed path using the mapping-table to `/path/to/dir1`. In effect, the *old* path is used internally to correctly access the corresponding metadata, while the *new* path is visible to the user.

If a user attempts to access `/path/to/dir1`, the path gets remapped to `/unique_name1`. The user will thus correctly receive a *"file does not exist"* error message, even though the corresponding metadata object has not been moved during the move operation.

Continuing, moving `/path/to/dir2` to `/path/to/dir1` changes the mapping table to:

$$
\begin{aligned}
\text{/path/to/dir1.tmp} &\rightarrow \text{/path/to/dir1} \\
\text{/path/to/dir1} &\rightarrow \text{/path/to/dir2} \\
\text{/path/to/dir2} &\rightarrow \text{/unique\_name2}
\end{aligned}
$$

Again, a file system user can immediately start using the new name of the directory to access it as well as all files in it. For example, accessing `/path/to/dir1/file` would be internally remapped to `/path/to/dir2/file` and access the correct metadata object (if a file with that name exists).

Also the moved from path immediately becomes inaccessible for the user, as access attempts are remapped to `/unique_name2`.

Finally moving `/path/to/dir1.tmp` to `/path/to/dir2` changes the mapping table to:

$$
\begin{aligned}
\text{/path/to/dir2} &\rightarrow \text{/path/to/dir1} \\
\text{/path/to/dir1} &\rightarrow \text{/path/to/dir2}
\end{aligned}
$$

After each move operation, the user can access files using the new directory name and can no longer access them using the old name as is expected behavior. The fact that no metadata objects are actually moved by the file system itself is not visible to the user. Internally, the file system remaps paths using the available global metadata to achieve correct behavior without having to move an arbitrary number of metadata objects.

An overview of existing mappings after each step of the above example is shown in Table 2.1.

mv /path/to/dir1 /path/to/dir1.tmp
`/path/to/dir1.tmp` → `/path/to/dir1`
`/path/to/dir1` → `/unique_name1`
mv /path/to/dir2 /path/to/dir1
`/path/to/dir1.tmp` → `/path/to/dir1`
`/path/to/dir1` → `/path/to/dir2`
`/path/to/dir2` → `/unique_name2`
mv /path/to/dir1.tmp /path/to/dir2
`/path/to/dir2` → `/path/to/dir1`
`/path/to/dir1` → `/path/to/dir2`

Table 2.1: Hierarchical Functionality: Overview of Path Mappings

Example 2 This second example follows the remapping algorithm more closely and shows how it can resolve a user path where multiple path components have been renamed. Assume existing directory `/path/to/dir1` and two move operations having been executed:

- moving `/path/to/dir1` to `/path/to/a`
- moving `/path/to` to `/path/b`

The following mappings exist after the move operations.

/path/to/a	→	/path/to/dir1
/path/b	→	/path/to
/path/to/dir1	→	/unique_name1
/path/to	→	/unique_name2

When the user attempts to access `/path/b/a`, path remapping is applied to transform the user-path to a system-path. The algorithm attempts to match each sub-path from the complete path down to the first path component.

1. Check if a mapping exists for `/path/b/a`: No.
2. Check if a mapping exists for `/path/b`: Yes.
   Apply mapping to user path: `/path/b/a` → `/path/to/a`
3. Restart matching process with full remapped path.
   Check if a path mapping exists for `/path/to/a`: Yes.
   Apply mapping: `/path/to/a` → `/path/to/dir1`
4. Check mappings for `/path/to/dir1`, `/path/to` and `/path`.
   Reuse mappings are not followed after a move mapping has been found, so no further matches exist.

After the path remapping process the user path `/path/b/a` has successfully been remapped to the system path `/path/to/dir`.

2.3.3 Reducing the Global Metadata Footprint

The common element of the above discussed approaches to support hierarchical functionality is to store additional metadata. Except for basic path permissions, this additional metadata is file system global: It needs to be available for every lookup operation.

To reduce the amount of stored metadata, some of the operations can be applied to the file system backend by a background process.

If all metadata of files affected by a directory move are migrated or all metadata of files that are affected by a directory permission change is updated, the corresponding extra metadata can be removed: Requests to the file system return the expected results at this point without any manipulation.

While arbitrary runtime operations are not acceptable during normal file system execution, *arbitrary* can often be very short. For example, moving a directory that only has two files inside could be applied to the file system backend by moving three metadata objects. After these metadata moves complete, the additionally stored metadata can be removed from the global metadata.

In this way, it becomes possible to quickly apply short-runtime file system operations and reduce the global metadata requirements.

2.3.4 POSIX Implications

A limitation to POSIX compliance inherent to any approach skipping component traversal is that a lookup operation of a non-existing file will always return 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.).

This property can be a security issue: It can be used to test for the existence of any file by any file system user.

This cannot be easily addressed without taking further degradation of POSIX compliance into account. One non-POSIX solution would be, for example,

to encrypt file paths internally, preventing anybody from accessing a file (or even finding out if it exists) unless they knew the encryption key.

In this thesis it will assumed that the non-POSIX error codes are unproblematic and implementations will avoid degrading POSIX compliance by implementing a solution.

Chapter 3

Feasibility Analysis: Empirical Data

3.1 Introduction

The approaches introduced in the previous chapter to provide full hierarchical functionality with direct lookup functionality rely on additional metadata being available, both for access permissions and operations affecting the hierarchical namespace.

The feasibility of storing additional metadata every time a hierarchical operation is executed directly depends on the frequency that these operations occur. The costs of applying an operation to the file system backend and thus shrinking the amount of additional metadata required as described in Section 2.3.3 correlates with the number of files affected by the operation.

Intuitively it seems reasonable that hierarchical operations such as 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, little supporting literature. [APG11] similarly argue that

directory rename operations are infrequent and provide a 21 hour trace from a single server as evidence which contains only 5 directory rename / move operations. The authors do not further analyze the operations themselves.

For traditional file system studies of hierarchical file systems it makes little sense to differentiate between different operation targets, as hierarchical operations have little to no impact on system performance.

To address the extremely limited literature on the topic, this Chapter provides empirical data obtained both from tracing and static analysis of real-world file systems. Section 3.2 analyses frequency and size of hierarchical operations and Section 3.3 considers the impact of storing full path permissions. Results are summarized in Section 3.4.

3.2 Hierarchical Operations

Section 2.3.2 describes how hierarchical file system operations are supported with the direct lookup approach by storing additional metadata.

To analyze the hypothesis that hierarchical operations are extremely rare in real-world file system workloads, file system traces from the GPFS file system of the Barcelona Supercomputing Center were taken and evaluated focusing exclusively on hierarchical operation frequency and size. The size of a hierarchical operation equals the *directory size* of the operation's target directory, defined as the total number of files and directories located in that directory's subtree.

In total, tracing was done 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.

3.2.1 Procedure

Operation frequency information can be directly obtained from GPFS traces: it is possible to differentiate between inodes corresponding to files and inodes corresponding to directories.

However, there is no direct way to extract operation size information from the traces. Only the following information is available for each traced file system operation:

- the operation type
- inode numbers involved in the operation
- the user id

The only way to obtain operation size 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 a highly intrusive process. Reverse mapping inode numbers to paths is not directly supported by GPFS. Instead, the file system namespace had to be searched in a brute-force manner to link an inode number to a directory path (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 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.

Throughout the tracing period, the inode-path mapping as well as the discovery of directory size was performed periodically every 30 to 60 minutes.

Limitations of Post-Processing 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.

Additionally, the traces were taken as samples of overall file system activity. Not all file system activity in the whole supercomputer could be traced. Following, it is possible that files were created or removed from a directory or one of its sub-directories between the point of time the operation was traced and the post-processing was done.

In a few cases, a target directory had even been deleted from the file system by the time the post processing step was taken.

These inaccuracies inherent to the use of post-processing, however, affect only a very small percentage of operations and therefore do not skew the overall trends obtained from analyzing the file system traces.

Operation	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 3.1: Empirical Data: Relative File System Operation Frequency

3.2.2 Results

As mentioned in the introduction of Section 3.2, over 1.7 billion file system operations were traced in total.

Table 3.1 shows the relative frequency of the most commonly observed file system operations as well as the relative frequency of the hierarchical operations for both login nodes and compute nodes.

The distribution of the most frequent operations is very intuitive: The automated workloads on compute nodes are primarily focused on file read and write operations, while the primarily user generated workloads on login nodes are heavier on metadata operations.

Directory Moves The traces contain a total of 329,188 move operations. 1,538 of these operations target a directory. All move operations targeting a directory originated on login nodes. In total, the paths to over 2.3 million files and sub-directories were changed by these 1,538 directory move operations.

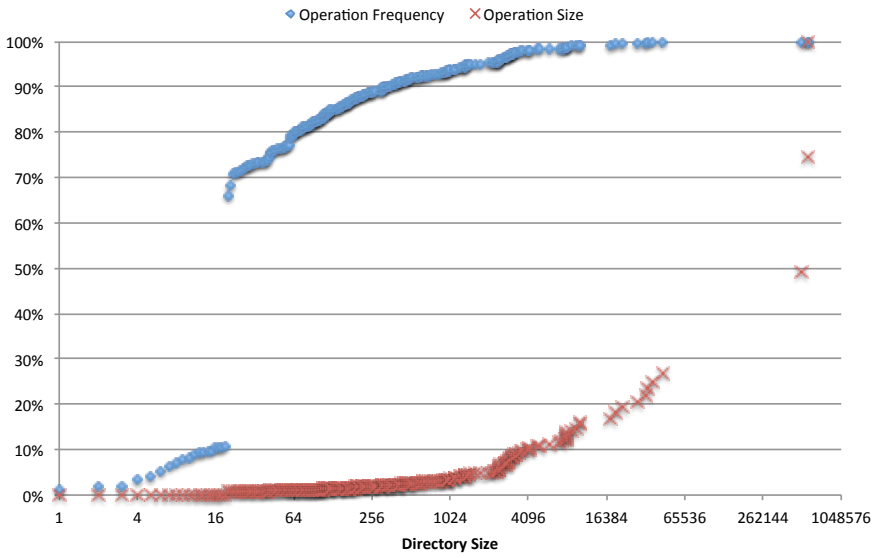


Fig. 3.1: Directory Move: Cumulative Frequency and Size Diagram

Figure 3.1 shows the cumulative frequency and size distributions of moved directories (note that the x axis is displayed in log-scale). The cumulative operation frequency displayed in blue shows the **cumulative percentage of operations** that have been performed up to a certain directory size. The cumulative operation size displayed in red shows the **cumulative percentage of files** that are affected by operations performed up to a certain directory size. The size of a directory is defined as the total number of files and directories located in that directory's subtree.

The initial assumption that hierarchical operations are most frequently done on relatively small directories is supported for the case of move operations.

1,359 directory move operations (90%) are performed on directories containing

less than 335 files and sub-directories. These 1,359 move operation change the path of a total of 49,343 files and sub-directories. This corresponds to an average directory size of 36.7 and to only 2.1% of the files and sub-directories affected by all directory move operations.

The massive spike in the frequency distribution at a directory size of 20 was shown to result from a single experiment reorganizing its results, which were stored 20 files to a directory.

The three largest move operations (less than 0.2% of operations) target directories containing over half a million files each, totaling over 73% of overall affected files and sub-directories.

The approach of applying hierarchical operations to the storage backend using a background process in order to shrink the required global metadata described in Section 2.3.3 is very applicable given the observed operation size distribution. A directory move operation can be applied by migrating all affected metadata. The cost of applying a directory move operation thus grows linearly with operation size. The 90% smallest move operations require less than 40 metadata migrations per move operation. A single such operation should thus be applied in less than a second by any file system implementation.

In fact, assuming a throughput of 100 metadata migrations per second for the background process, even the largest observed move operation could be applied to the backend in less than 90 minutes. While this is bordering on the limits of being practicable, it shows that except for extreme outliers almost all directory move operations can be expected to be applied to the

storage backend over time. The amount of generated global metadata that has to be stored permanently can thus be limited to a small fraction.

Directory Permission Changes The traces contain a total of 1.6 million setattr operations. To limit impact on the production environment, only a subset of these could be further analyzed by post-processing. Out of the 262,969 further analyzed operations, 1,940 were operations on directories that could potentially have changed path permissions.

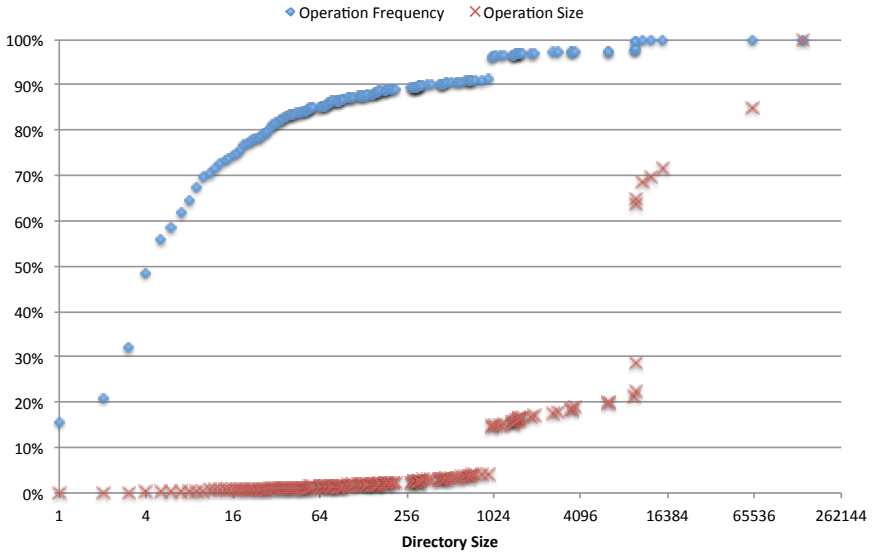


Fig. 3.2: Directory Permission Change: Cumulative Frequency and Size Diagram

Figure 3.2 shows the cumulative frequency and size distributions of directories targeted by permission changes. The cumulative operation frequency displayed in blue shows the **cumulative percentage of operations** that

have been performed up to a certain directory size. The cumulative operation size displayed in red shows the **cumulative percentage of files** that are affected by operations performed up to a certain directory size. The size of a a directory is defined as the total number of files and directories located in that directory's subtree.

The overall trends observed for moved directories are repeated: 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 storage backend are located in big directories.

90% of operations target directories of a size of 256 or smaller, potentially affecting the path permissions of a total of 26,797 files and subdirectories. This corresponds to an average directory size of 15.4 and to 2.8% of the files and sub-directories affected by all directory permission change operations.

Compared to the previously discussed traces of directory move operations, the average size of operations is significantly lower. Otherwise the observed numbers align quite closely. Following, the same argumentation regarding the feasibility of shrinking global metadata by applying operations to the storage backend applies.

There are a number of factors that further simplify applying directory permission changes to the backend.

- As shown in Section 3.3, only very few path permissions exist in real file systems. A change to the access permissions of a directory does not necessarily change path permissions: The access restrictions might

already have been enforced in a parent directory.

- If the directory permission change did affect path permissions, path permissions of files and subdirectories are automatically updated when a file is accessed by a file system user. It therefore becomes cheaper to apply an operation over time, as more and more permission sets are already updated during normal file system operation.

Symbolic Links The traces contain a total of 60,574 symbolic link creations.

In contrast to directory move operations and directory access permission changes, it does not matter if the target of a symbolic link is a directory. Symbolic links only manipulate the path, they do not actually have a fixed target. A symbolic link can be created pointing to a file, but later end up pointing to a directory if the file is deleted and a directory is created with the same name.

It is therefore not possible to constrain the entries in the global metadata to symbolic links pointing to directories. This makes symbolic link creation by far the most frequent operation regarding global metadata creation.

In contrast to directory move operations or directory access permission changes, symbolic links also cannot be applied to the storage backend.

3.3 Path Permissions

As introduced in Section 2.3.2.1, full path permissions are required in order to retain hierarchical access permissions when using direct lookup.

Path permissions have to be evaluated to decide if access to a file is granted. This can only be achieved without performance degradation if the path permissions are either already in memory by the time the regular file metadata is read in or if the path permissions are stored together with the rest of the file metadata. The practicality of either approach depends on the size of path permissions.

In theory, the maximum size required to store path permissions for a file is only limited by the number of components in its path: Every component can introduce different constraints that have to be stored.

However, intuitively, the average size is much lower as many directories in a path share the same access permissions (e.g. directories in the home directory of a user usually belong to that user).

To examine how big this effect is in reality, two multi-user file systems at

	File System A	File System B
files	10,770,676	6,419,483
directories	1,271,791	618,686
unique users (directories)	452	310
unique groups (directories)	115	84
unique path permissions	638	391
total path permission entries	821	489

Table 3.2: Empirical Data: Feasibility of Path Permissions

the Universitat Politècnica de Catalunya have been analyzed. Table 3.2 displays size information about the analyzed file systems, the number of unique POSIX users and groups encountered in the systems, as well as the resulting path permission sets.

File system A contains over 1.2 million directories and over 10 million files, but only 638 different unique path permission sets are required to express path permissions for all directories and files in the file system. The number of observed path permissions correlates much more with the number of existing POSIX users and groups in the file system than the number of directories: For every existing user or group there are 1.12 path permissions. For file system B the same correlation can be observed, with 0.99 path permissions per existing user or group.

The average size required to express all existing access restrictions for a file or directory is below 1.3 entries for both file systems: The majority of observed path permissions only contain a single entry.

As a consequence of the number and size of existing path permission sets, caching all existing path permissions is trivial for a local file system (even for a system containing two orders of magnitude more unique path permissions compared to the observed systems).

For large scale distributed file systems, storing path permissions together with regular metadata is preferable. Not only does this avoid cache size problems (caching of all path permissions still seems reasonable for many larger scale scenarios), it avoids potential cache inconsistencies when using multiple independent file system clients.

3.4 Conclusion

Due to the limited number of users and groups existing in real-world hierarchical paths, path permissions can be stored extremely efficiently and do not limit system scalability.

Considering operations affecting the hierarchical namespace, symbolic links are the main concern regarding scalibility. They are the most frequent hierarchical operation and cannot be applied to the storage backend. This suggests that limiting the number of symbolic links may become a necessity for big deployments. This could be achieved by

- Introducing a quota for symbolic links and minimizing usage.
- Storing symbolic links on a per-user level instead of file system global. Each user would then have to cache only its own symbolic links.

For directory move operations and directory permission changes the empirical data has confirmed the assumption that these operations are rare. Traced rename and setattr calls make up only 0.1 % of total traced operations, and well below 1% of these target directories.

The second assumption that directories targeted by these operations are most frequently small has also been confirmed. With the exception of extreme outliers, all directory move operations and directory permission changes can eventually be applied to the storage backend and the corresponding entries can be removed from global metadata.

Combined, both of the above properties support the feasibility of using global metadata to enable hierarchical functionality without a hierarchical lookup operation.

Chapter 4

Local File System Evaluation

4.1 Introduction

To examine the implications of the direct lookup approach for local file systems, a kernel-level Linux file system has been developed.

The decision to implement a full kernel-level file system instead of a user-space implementation using fuse was made to obtain performance measurements that are comparable with commonly used real-world file systems such as EXT4 and XFS.

The file system developed for this evaluation is called *Direct Lookup File System*, DLFS for short. It is intended to support both traditional hard drives and solid state drives. Performance results are discussed in Section 4.8.

In order to achieve the functionality of directly locating metadata, the physical on-disk location of the file metadata is computed based on a hash of the full file path. Note that no central data structures, such as a hash table, are required for this process. Such a lookup will incur only a single access to the storage device, regardless of where the file is located in the file set. Besides this one-access characteristic for file system lookups, parallel accesses

can take further advantage because concurrency bottlenecks introduced due to directory hierarchy are avoided. Not all implications of the direct lookup approach are positive, however. As the physical location of the metadata depends on the used hash function instead of file set characteristics, a higher randomization of accesses has to be expected compared to the traditional approach. Furthermore additional complexity is introduced to handle hash collisions, to achieve a non-fragmented data layout and support POSIX access permissions.

4.2 Data and Metadata Layout

Fig. 4.1: Local Data and Metadata Layout

There are two fundamentally different ways the storage area of the underlying block device is used by the DLFS file system, which is reflected in the block device layout displayed in Figure 4.1.

Hash buckets store metadata and (optionally) small file data while *big file spaces* store only file data.

The functionality required of a layout implementation are:

- unambiguously assign file paths to buckets
- arbitrarily assign a big file space to an inode

The layout implementation is not part of the core file system. Different layout implementations can be plugged into the file system similar to how different file system implementations can be used by the Virtual File System.

In the scope of this evaluation the most basic layout possible will be used: a number of sequential hash buckets of homogeneous size followed by a single big file space as shown in Figure 4.1.

As the number of hash buckets is static with this simple layout, it has to be configured at file system creation time similar to traditional file systems which use static metadata structures (e.g. ext-2/3/4).

4.2.1 Hash Buckets

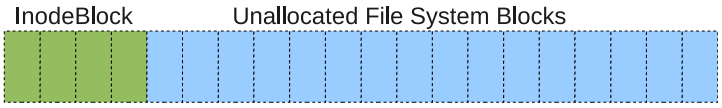


Fig. 4.2: Local Metadata Layout: Empty Hash Bucket

Metadata The file system needs to be able to differentiate between multiple files that are hashed to the same bucket.

As the full path of a file can be arbitrarily long, it is not practical to store it for each inode. Instead, a second hash value, the identification hash, is used. The possibility of hash collisions (more than one file with the same identification hash assigned to the same bucket) is discussed in Section 4.2.3.

The actual inode representing the file metadata is stored in a four kilobyte sized *inode block* structure contained in the hash bucket.

Figure 4.2 shows the contents of an example hash bucket containing a single inode block (assuming a file system block size of one kilobyte). If more files are assigned to a hash bucket than can be stored in a single inode block, additional inode blocks are allocated on demand.

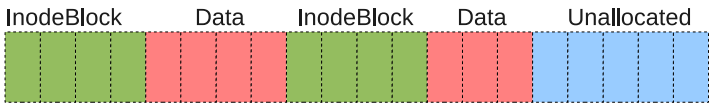


Fig. 4.3: Local Metadata Layout: In-Use Hash Bucket

Data Similar to many existing file systems, direct addressing of data blocks for small files is supported by storing a configurable number of block numbers directly in the inode.

These data blocks can be allocated in the same hash bucket the metadata is located. This is an optimization aimed at hard drives: After reading in the inode block where the file metadata is located, access to these data blocks will be covered by drive-readahead in most cases, allowing rapid access to data of small files.

Figure 4.3 shows a partially filled hash bucket: In addition to a second inode block, multiple file system blocks have been allocated for small file data of inodes assigned to the bucket. Allocation inside a hash bucket is managed using an allocation bitmap contained in the primary inode block structure.

If space is required to allocate additional inode blocks or if a file grows beyond the configured threshold value for co-locating data with metadata, data blocks allocated in the hash bucket are migrated to a big file space.

If no free space should exist in the hash bucket, allocation that would normally occur inside the bucket is moved to a big file space.

4.2.2 Big File Space

If a file grows beyond a size that can be directly addressed by its inode, standard indirect addressing techniques are used.

Data blocks of such files are allocated in a big file space. A big file space never contains inodes, and only contain data blocks of small files if they could not be allocated in the same bucket as the inode itself.

While not evaluated in this thesis, intuitively this should have a positive effect on data fragmentation inside the big file space. A segment (also called extent) based addressing scheme was chosen in order to minimize metadata required for indirect addressing.

Indirection blocks are located in the big file space directly ahead of the first extend they reference to minimize hard disk seeks in case of sequential data access.

One interesting opportunity that follows from the different allocation processes of data blocks for small and big files is to introduce the possibility of different allocation granularities. It is possible to have fine grained allocation inside hash buckets, reducing internal fragmentation for small files due to a small file system block size, while - at the same time - benefiting from the advantages of a coarse allocation granularity (such as minimizing required file system metadata) for data allocated in the big file space.

4.2.3 Hash Collisions

One problem of the approach is the possibility of a collision of identification hashes for multiple files assigned to the same bucket.

If such a collision occurs, the file system cannot differentiate between these file paths and will therefore incorrectly assume that both paths correspond to the same inode.

Let us consider the probability for a hash collision for a single bucket assuming a hash function returning uniformly random values. This probability p is

$$p(m,n) = 1 - \prod_{k=1}^{m-1} \left(1 - \frac{k}{n}\right), \tag{4.1}$$

where n is the number of possible hash values and m the number of inodes in the hash bucket.

Following, the probability of at least one hash collision ocurring when assigning 50 inodes to a hash bucket (filling a single four kilobyte sized inode block) and using 64 bit hash values for the identification hash is

$$p(50, 2^{64}) = 1 - \prod_{k=1}^{49} \left(1 - \frac{k}{2^{64}}\right) = 6.6 \times 10^{-17}$$

The same scenario but using a 128 bit identification hash:

$$p(50, 2^{128}) = 1 - \prod_{k=1}^{49} \left(1 - \frac{k}{2^{128}}\right) = 3.6 \times 10^{-36}$$

While considering the collision probablity in a single, isolated hash bucket

adequately represents the probability of a hash collision when creating a single file, the system-wide collision probablity should be considered as well. The accumulated probability of at least one hash collision for h hash buckets, each containing m inodes with n possible hash values for the identification hash can be computed as follows.

$$p(h, m, n) = 1 - \left(\prod_{k=1}^{m-1} \left(1 - \frac{k}{n} \right) \right)^{h} \tag{4.2}$$

To use a realisitic example, consider a scenario where 15 million files are created in a file system containing three hundred thousand buckets. The probability of at least one hash collision occurring during the entire creation process using 64 bit and 128 bit hash value sizes is respectively

$$p(50, 2^{64}, 300000) = 1.99 \times 10^{-11}$$
$$p(50, 2^{128}, 300000) = 1.08 \times 10^{-30}$$

It is straightforward to arbitrarily lower the colission probability further by increasing the size of the identification hash.

Putting these values into context: Storage device manufacturers commonly specify the probability of unrecoverable read errors (URE) as 10^{-14} for hard drives and 10^{-16} for solid storage drives. In other words, it is more likely that any given data block of a file cannot be read back than that there is a hash collision during file creation, even when using 64 bit[1] identity hashes in

[1]URE probabilities are per sector and not per drive. Per-file a collision is less probable than an URE even with 64 bit hash values.

a populated file system.

4.3 Hierarchical Functionality

As introduced in Section 2.3.2, the approach to provide hierarchical functionality on top of a flat namespace relies on additional metadata being present during the lookup operation.

Using this metadata, paths of moved directories can be re-mapped before the file system lookup executes and path permission sets can be verified not to be outdated. This allows all hierarchical functionality to be executed in constant time while maintaining direct lookup performance characteristics.

For the local scenario, the approach can be implemented straightforwardly.

mv /a/b/c /d/c		
`/d/c`	→	`/a/b/c`
`/a/b/c`	→	`/unique_name1`

Table 4.1: Hierarchical Functionality: Path Mapping

Table 4.1 shows the path mappings resulting from a single directory move operation. The DLFS implementation utilizes an uthash[2] hashtable as an in-memory store to store these path mappings.

The computational overhead of accessing an element in the hashtable equals the costs of the hash function used internally by uthash and is independent of the number of stored items.

[2] http://uthash.sourceforge.net/

When checking whether the hashtable contains information for a specific path, each possible sub-path has to be checked separately. For example, assuming the path mappings displayed in Table 4.1, attempting to access a hypothetical path `/d/c/dir/file` would lead to the following accesses in the hash table:

check: /d	does not exist
check: /d/c	remap → /a/b/c
check: /a/b/c/dir	does not exist
check: /a/b/c/dir/file	does not exist
done	

Accordingly, the computational costs and the number of memory accesses increase proportional with the number of path components.

In absolute numbers, however, these computational costs are insignificant compared to costs of IO operations and do not impact overall file system performance. Access to the hash table only occurs for lookup operations inside DLFS, there is no impact on cache lookups inside the Virtual File System.

Required memory largely depends on the length of the paths stored as entries in the hash table, which in turn depends on the the file set. A realistic average entry size is in the order of 100 bytes.

4.4 Direct Lookup Implementation

Section 2.1 introduces two implementation options to support direct lookup behavior: Modifying the name resolution in the Virtual File System and library preloading.

While a kernel patch is the less flexible approach, it is preferable from a pure performance perspective. It does not introduce any additional layers or bottlenecks into the lookup process. It was chosen as the more suitable approach for this evaluation, which focuses on the comparison of performance with other kernel-level file systems.

At the time of development kernel version 2.6.35 was the latest Linux release considered stable and was thus used as a patch target.

The name resolution is implemented in the kernel by the link_path_walk function. The function is called with a path supplied by the file system user. This can be either a relative or an absolute path. As the function name suggests, it then *walks* the path, looking up every path component until either an error is encountered or the final inode has been found.

Figure 4.4 displays highly abbreviated pseudo code for the link_path_walk function.

The kernel patch introduces the FS_DIRECT_LOOKUP file system flag. During the path_walk the currently active mountpoint is checked for this file system flag. If it is set, the entire rest of the path is used as a single component and will be used in a single lookup operation.

```
link_path_walk (path) {

  while(more path components exist){

    if(current_mountpoint->fs_flag & FS_DIRECT_LOOKUP) {
      component = prepare_direct_lookup(path);
    }
    else{
      component = extract_next_component(path);
    }

    inode = lookup(current_directory, component)
    check_permissions(inode);
    follow_symlink(inode);
  }

  return inode;
}
```

Fig. 4.4: Kernel Name Resolution: Pseudo-Code for `link_path_walk`

The prepare_direct_lookup function ensures that the path assigned to `component` does not contain any duplicate slashes or '.' or '..' components. (eg, `/mountpoint/dir//./../file` would be translated to `/mountpoint/file`).

If the FS_DIRECT_LOOKUP flag is not set, hierarchical lookup will be used by using only the next path component for the lookup function (again handling '..' etc.).

The Virtual File System keeps a cache of previously looked up path components called *dentry* cache. If a lookup request cannot be served from the cache, it invokes a call into the underlying file system.

If an inode is successfully looked up access permissions are verified and, if

the inode should refer to a symbolic link, that link is followed.

Using a file system flag to enable direct lookup functionality allows file systems that rely on the hierarchical lookup approach to coexist with file systems that support direct lookup.

It also allows mixed hierarchical / direct lookups on paths. If the DLFS mountpoint is a directory in a traditional file system, the path_walk function will use component based lookup for the beginning of the path, and direct lookup once the DLFS mountpoint is entered.

4.5 Metadata Caching

All on-disk metadata of the file system is either stored in page sized structures (e.g., on-disk inodes in inode blocks) or is directly page sized (e.g., allocation bitmaps in the big file space).

It is therefore very straightforward to use the page cache directly to cache this metadata. A metadata address space is created during the mount of the file system for this purpose, and all metadata I/O is performed using it.

The most obvious advantage of directly caching on-disk data structures is the effect on metadata updates. Writing back a dirty inode simply causes a flush of the dirty inode block. If the corresponding inode block were not already cached, a read-modify-write cycle would be required to update the inode block on the disk.

There also is a positive effect for all other metadata operations since a single

inode block is shared by many inodes: When creating or accessing a file, there is a chance that the inode block corresponding to the file path is already in memory, thereby eliminating the need to access the storage device. The number of inode blocks that can be simultaneously cached is of course limited by the available main memory.

In-memory inodes are managed as usual by the Virtual File System.

There is no need to store inode numbers, as they are not required to uniquely identify metadata on the drive. However, assigning random inode numbers can lead to VFS cache inconsistencies, as multiple in-memory representations of the same inode can occur when an inode whose dentry VFS object has already been removed from the cache is re-accessed before the in-memory inode itself has been destroyed. To prevent this issue, inode numbers are computed based on the location of the on-disk inode.

4.6 Directories

Hash based metadata placement and direct lookup fundamentally change how directories affect file system performance. During the lookup process, directories are not used and consequently don't affect performance: It does not matter if a file is the only file of a directory or shares it with a billion other files, or even where it is located in the directory tree. In fact, the only file system operation that requires directory entries is readdir / ls.

File Creation In traditional file systems the directory of a to-be-created file has to be checked to decide if the filename already exists. This becomes unnecessary when the hash-based placement approach is used, as name collisions are detected during the allocation of the inode:

As described in Section 4.2.1, inodes are created in a hash bucket and contain an identification hash based on the file path. If the user attempts to create a file, the file system checks if the corresponding inode already exists in the hash bucket associated with the file's path, and - if so - returns an error. As this check is done atomically, it also serves to resolve concurrent creation attempts of the same file.

This allows the creation process to be optimized in DLFS: Simply appending the file name to the directory data structure can be done efficiently independently of the directory size.

File Deletion The simple append structure chosen for directory data to allow fast metadata creation introduces an overhead when deleting a file: The directory entry corresponding to the file has to be found in the unsorted list of directory entries[3]. Additionally, the data structure might need to be compacted when many entries are deleted.

These characteristics are at least partially mitigated by the very small size of directory entries: An entry consists of only the file name. Assuming an average file name length of ten characters, one megabyte of directory data

[3]The tree-structures other file systems employ to sort directory entries allow finding a specific entry much more efficiently.

can store almost one million directory entries.

Implementation Internals Another consequence of skipping component traversal is that the normally available knowledge of the structure of the directory tree is lost and some of the functionality normally done in the VFS layer has to be taken over by the file system itself. When a file is created or removed, it can no longer be guaranteed that its directory is already cached. The directory is, however, required in these cases:

- Attempting to create or remove a file in / from a nonexisting directory does not make sense and should result in an error.
- When a file is removed, the file name has to be removed from the directory.
- When a file is created, the file name has to be added to the directory. Additionally, the path permissions of the file have to be inherited from its parent directory.

To support uncached directories, on-demand lookup of the parent directory of a file is implemented for file creation and deletion.

While not feasible for a general purpose use-case, it is possible to disable directories and just use a flat namespace. This can be reasonable, for example, if all file paths are computed and the file system is simply used in the manner of a key-value store but using the POSIX interface. All directory related overhead for metadata creation and removal in the file system is avoided in this case.

Summary The performance of both file creation and file access is independent of directory size and location. File deletion can incur an overhead proportional to directory size but it only becomes significant for extremely large directories.

4.7 POSIX Conformity

DLFS passes the POSIX test suite[4] with one exception: Since inode numbers are dependent on the location of the inode (see Section 4.5), and the location - in turn - depends on the metadata placement algorithm and thus the file path, the inode number changes when a file is moved or renamed.

This causes some tests to fail, even though it does not explicitly violate the POSIX standard.

4.8 Benchmarking

Benchmarking Methodology: Metadata Frequently used metadata benchmarks such as *metarates*[5] oversimplify access patterns by constraining accesses to a few (or even only one) directory and creating all files used for the benchmark during the actual benchmark run.

In contrast, the aim of this evaluation is to analyze more realistic access patterns on pre-existing file sets.

[4] http://www.tuxera.com/community/posix-test-suite/
[5] http://www.cisl.ucar.edu/css/software/metarates/

The tool *Impressions* [AADAD09] can be used to create file sets with realistic properties (e.g., size of files and their distribution in the directory tree, size of directories at various depths in the tree).

To take advantage of this opportunity modify *metarates* is modified to handle pre-existing file sets: Instead of computing file paths during runtime, access lists are created independently before starting the benchmark.

This decouples the knowledge of the file set from the benchmark; as the modified metarates does not have to store any representation of the directory tree or a complete file list, arbitrarily large file sets can be used even on low memory machines. Other advantages are the possibility to directly use trace data to specify accesses and that the complexity of computing artificial access patterns in no way impacts the performance of the benchmark itself.

This modification is called *listrates*[6] in the following. The access patterns used for access list generation are based on the Zipf distribution [Zip29], commonly observed in multi user scenarios such as web servers [ABCd96, BCF+99], as well as the uniform random distribution to show how file systems handle the case of minimal metadata locality.

Benchmarking Methodology: Data Measurements regarding data performance of big files are taken using the established *IOR*[7] benchmark tool. The performance of very small files is considered separately in order to analyze the optimizations of the individual file systems for this case.

[6] https://github.com/plensing/listrates
[7] http://sourceforge.net/projects/ior-sio

4.8.1 Hardware, Software and Configuration Specifics

One of the most critical hardware components for file system benchmarking is the storage device.

The relative performance characteristics exhibited by hard drives (HDD) and solid state drives (SSD) are fundamentally different. While sequential access outperforms random access on a HDD by nearly two orders of magnitude, this factor shrinks below one order of magnitude for SSDs. This is highly important when comparing the performance of file systems, as a reduction of the total number of drive accesses at the cost of higher randomization can lead to completely different results.

Benchmarks are shown for a consumer-level SSD (OCZ VERTEX 2) and HDD (WDC WD5002ABYS). A partition of the size of the SSD (120 GB) is used on the HDD; both to limit the time required of the various storage benchmarks as well as to allow the same benchmark configurations to be performed on both devices. The test machine has two Intel Xeon E5520 CPUs.

On the software side, the Linux kernel 2.6.35.9 is employed, modified as mentioned in Section 4.4 to support direct lookup functionality. Further version numbers are: IOR version 2.10.3 and Impressions version 1.0.

The standard input parameters for file set creation with Impressions were kept with the exception of the total file set size (set to 100 GB), and a slight skew towards smaller files to increase the total number of files (μ input parameter to the lognormal size distribution decreased by two and pareto tail

disabled). The resulting file set contains 2,555,453 files in 253,089 directories.

Ext-4 and XFS are chosen as comparison basis, as these are widely used state of the art file systems.

In order to keep performance comparisons as fair as possible with the non-journaling DLFS, journaling is disabled for Ext-4, and the delayed logging feature is enabled for XFS. No other file system configuration parameters were changed from the standard values. All file systems were mounted with the noatime and nodiratime parameters.

While the DLFS layout is kept extremely simple as introduced in Section 4.2, its configuration can influence performance: If many more hash buckets exist than necessary, it becomes less probable to find a required inode block in the cache. If not enough hash buckets exist for the existing file set, additional inode blocks have to be allocated dynamically, which can lead to multiple device accesses during a lookup operation.

Considering the size of the benchmark partitions, a bucket number of 100,000 is a reasonable configuration value given an initial space of 50 inodes in each bucket. In addition, benchmarks with 10,000 buckets and 1 million buckets are performed to show the impact unsuitable layout configurations can have on file system behavior.

In order to limit the total number of benchmarks, all benchmarks are performed with a constant number of four threads.

All displayed results are mean values of 10 benchmark runs and show 95% confidence intervals.

4.8.2 Metadata Performance

Due to the limited size of both the available storage space and the file set used for the benchmarks, it is important to use equally limited cache settings to obtain meaningful results.

Results are shown for 512 MB and 1024 MB main memory, which corresponds to a primary memory to secondary memory ratio of 1/240 and 1/120. For a one terrabyte drive these ratios would correspond to four and eight gigabytes of main memory respectively (note, however, that cache size is not equal to main memory size, as the Linux kernel requires some space (around 200 MB) for other purposes).

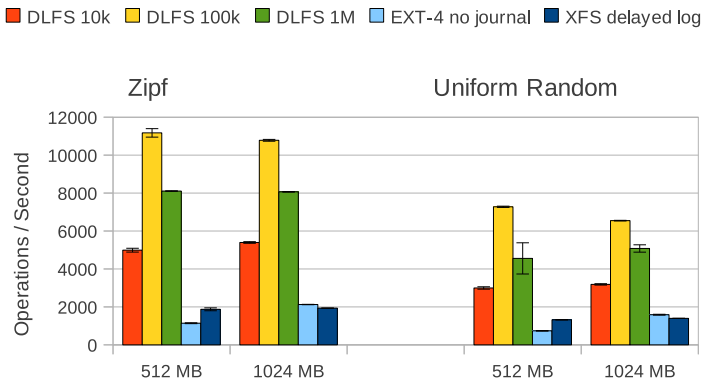
All results were obtained by performing 20,000 accesses using the given access distribution. In the hot cache scenarios, the cache was warmed up prior to the benchmark run by metadata operations of the respective type until no further performance improvement could be observed.

4.8.2.1 Metadata Access

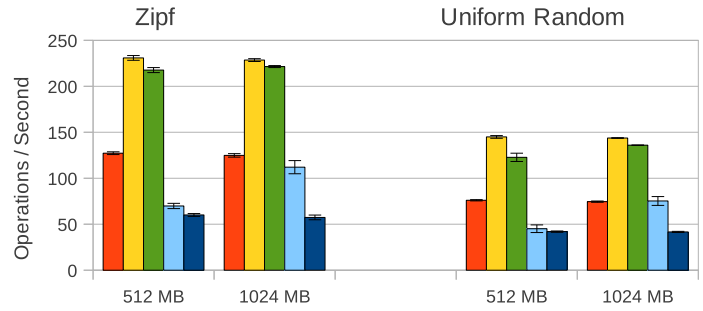
Measured performance is displayed in Figure 4.5 (cold cache) and Figure 4.6 (hot cache; note that the broken diagram uses logarithmic scale for the top half in order to display the widely diverging results).

The main factors influencing file system performance are a combination of cache size, storage technology, and access pattern.

In the following, the influence of these factors is discussed independently.



(a) SSD Cold Cache



(b) HDD Cold Cache

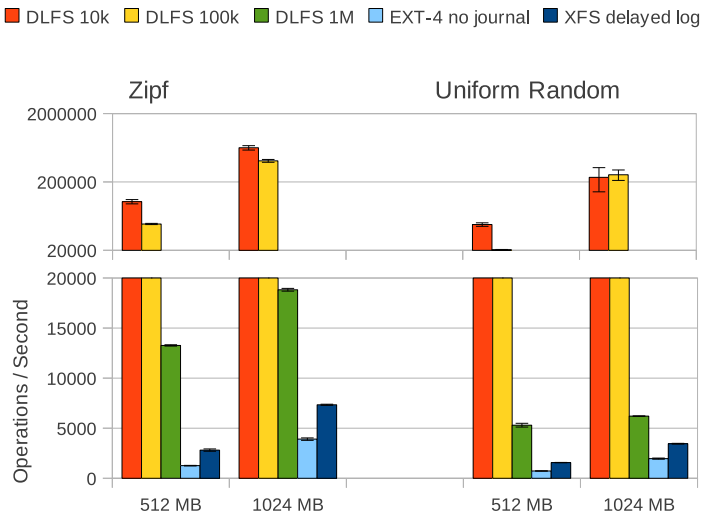
Fig. 4.5: Metadata Access Performance: Stat, Cold Cache

Layout	Initial IB	Additional IB	Total IB
10 k	10,000	68,188	78,188
100 k	100,000	3,412	103,412
1 M	1,000,000	0	1,000,000

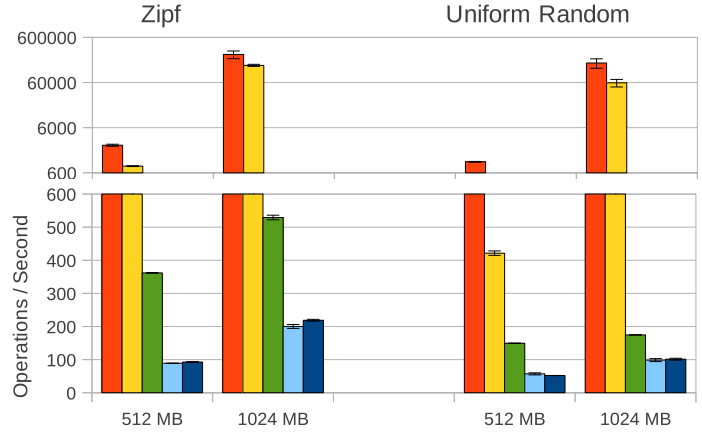
Table 4.2: Number of Inode Blocks after FileSystem Creation and Additional Dynamic Inode Blocks Created During File Allocation

Effect of Cache Size Logically, an increase of the cache size beyond the total size of encountered metadata cannot influence file system performance.

Considering the cold cache scenarios (Figure 4.5), the total amount of encountered metadata for 20,000 accesses is quite limited.



(a) SSD Hot Cache



(b) HDD Hot Cache

Fig. 4.6: Metadata Access Performance: Stat, Hot Cache

In fact, both XFS and DLFS already achieve maximum performance with 512 MB RAM, indicating that they can cache all encountered metadata. In contrast, Ext-4 shows higher cache requirements. It achieves almost twice the performance in the 1024 MB RAM scenario.

To understand the performance difference of the various DLFS configurations, consider the number of existing inode blocks summarized in Table 4.2.

The maximum caching requirements are proportional to the total number of existing inode blocks.

As not all inodes can be stored in the primary inode block of a hash bucket for the 10k configuration, it can take multiple accesses to read in the correct inode block. This leads to the 10k configuration performing worse than the other configurations.

The disadvantage of the 1M configuration compared to the 100k configuration is due to caching effects: With both configurations a one-access lookup can be expected, but since the scenario reads in 20,000 inodes, the cache is not completely empty after the first access. The probability that the inode of an accessed file is located in the same inode block as the inode of a previously accessed file is obviously higher the fewer inode blocks exist in the file system.

In contrast to cold cache performance, hot cache performance (Figure 4.6), is highly influenced by cache size across all file systems.

Ext-4 and XFS are limited in cache usage by the traditional lookup approach, as it requires caching directory data in addition to metadata. Nevertheless, the absolute performance of these file systems triples when increasing the main memory from 512 to 1024 MB RAM.

The performance gain for DLFS is far more drastic, however, especially for the 10k and 100k DLFS configurations.

Due to not having to cache directory data and the relatively low number of

inode blocks a large percentage of overall metadata can be cached, leading to many pure in-memory lookups.

Effect of Storage Technology The relative performance advantage of DLFS in the cold cache scenario (Figure 4.5) is more than twice as big for the SSD compared to the HDD. Considering the high randomization of drive accesses due to DLFS file system design and the different random vs. sequential drive characteristics, this is expected.

Because of the one-access nature of direct lookup operations, there is a clear worst case lookup performance for the hash-based metadata placement approach that is based on hardware characteristics and not - as in traditional file systems - mainly on file set characteristics.

The same impact of storage technology can be observed for XFS. XFS performance can compete with Ext-4 on the SSD, while it falls back to almost half Ext-4 performance on the HDD.

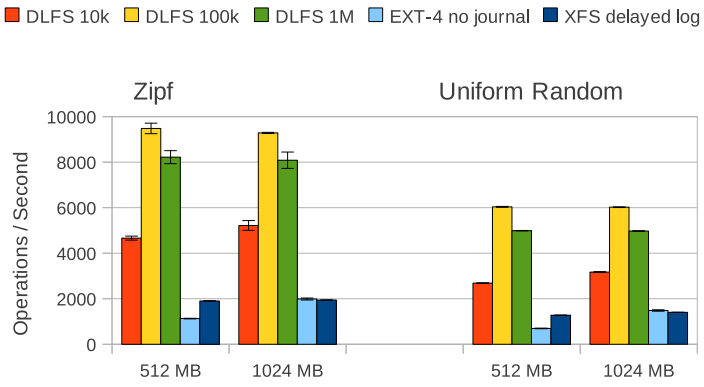
In the hot cache scenario (Figure 4.6), in contrast, the relative advantage of DLFS is actually greater on the HDD for the 10k and 100k layout configurations.

Not having to access the disk at all for many lookups outweighs the performance degradation of a higher randomized access pattern for the remaining accesses. This is not true for the 1M configuration, as only a smaller percentage of inode blocks can be cached in this scenario.

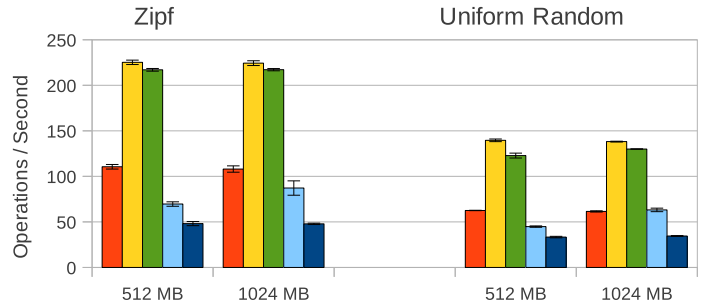
Effect of Access Pattern Uniform random accesses are slower than Zipf distributed accesses for all file systems.

As multiple accesses to the same file will not result in multiple disk accesses, this behavior is intuitive. The relative performance difference is similar across file systems and storage technologies.

4.8.2.2 Metadata Update



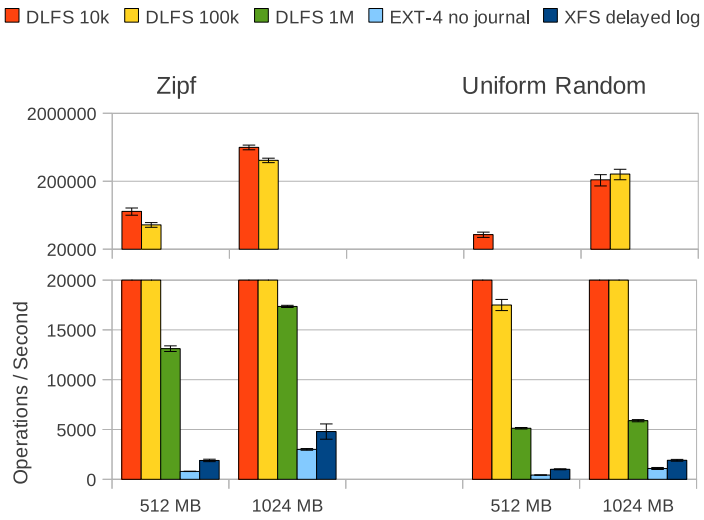
(a) SSD Cold Cache



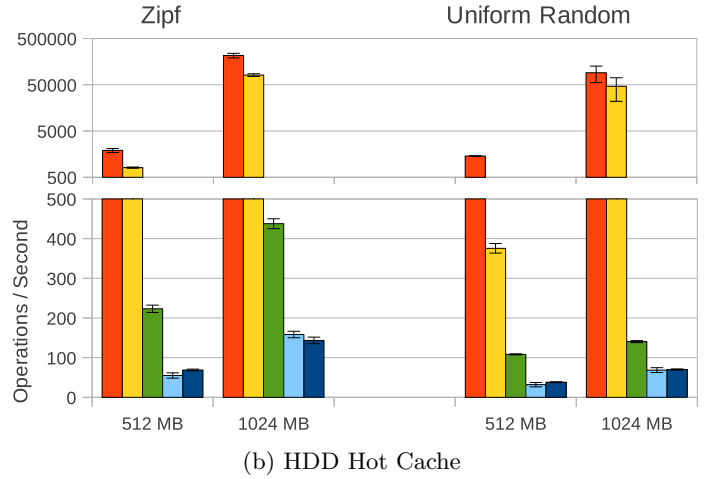
(b) HDD Cold Cache

Fig. 4.7: Metadata Update Performance: Utime, Cold Cache

Figure 4.7 (cold cache) and Figure 4.8 (hot cache) display the observed



(a) SSD Hot Cache



(b) HDD Hot Cache

Fig. 4.8: Metadata Update Performance: Utime, Hot Cache

metadata update performance.

Cold cache results are almost identical to metadata access, with absolute performance numbers decreasing only slightly. This is easily explained as a large percentage of the updated metadata is simply not yet flushed to

the secondary storage device at the end of the 20,000 metadata update operations.

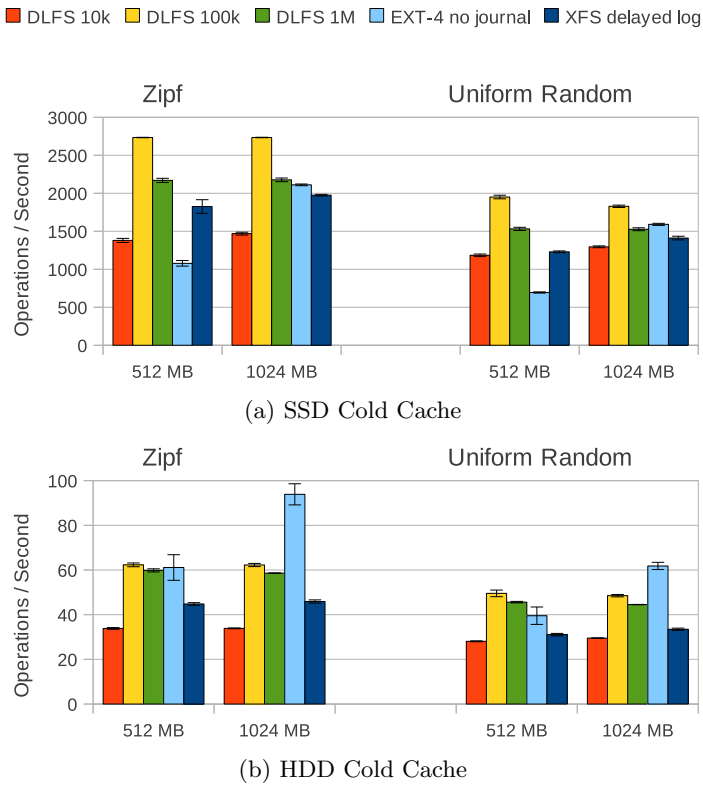
The hot cache results, in contrast, show the performance impact of persisting the updated metadata. The benchmark is continuously updating metadata, so the system is under continuous pressure to flush dirty inodes back to secondary storage. The expected loss of performance compared to metadata accesses can be observed for all file systems. The update performance in the hot cache benchmarks lies between 60% and 80% of the metadata access performance.

Despite this discrepancy in absolute performance values, the overall results are quite similar to metadata accesses discussed in Section 4.8.2.1, showing that metadata lookup and not metadata writeback dominates performance. The same effects for cache sizes, storage technologies and access patterns as discussed previously apply for the metadata lookup.

4.8.2.3 Metadata Creation

For metadata creation the access distributions have a slightly different meaning. Instead of choosing the files themselves according to the distribution, the directories the files are created in are chosen (after all it makes no sense to create the same file multiple times).

It should be noted that if directories are disabled as described in Section 4.6, metadata creation becomes equivalent to the metadata update operation from a performance point of view.



(a) SSD Cold Cache

(b) HDD Cold Cache

Fig. 4.9: Metadata Create Performance: Create, Cold Cache

If, however, full directory support is required DLFS loses some of its advantages when creating files instead of only accessing them. The directories the files are created in have to be read in, resulting in additional lookups as well as reducing the cache size available for metadata.

As these steps are part of the traditional lookup process, there is no additional effort involved for Ext-4 and XFS. Figure 4.9 and Figure 4.10 show the measured results for metadata creation, which are, as anticipated for the above reasons, less clear-cut than previous results.

In the cold cache scenario (Figure 4.9), the relative performance of the differ-

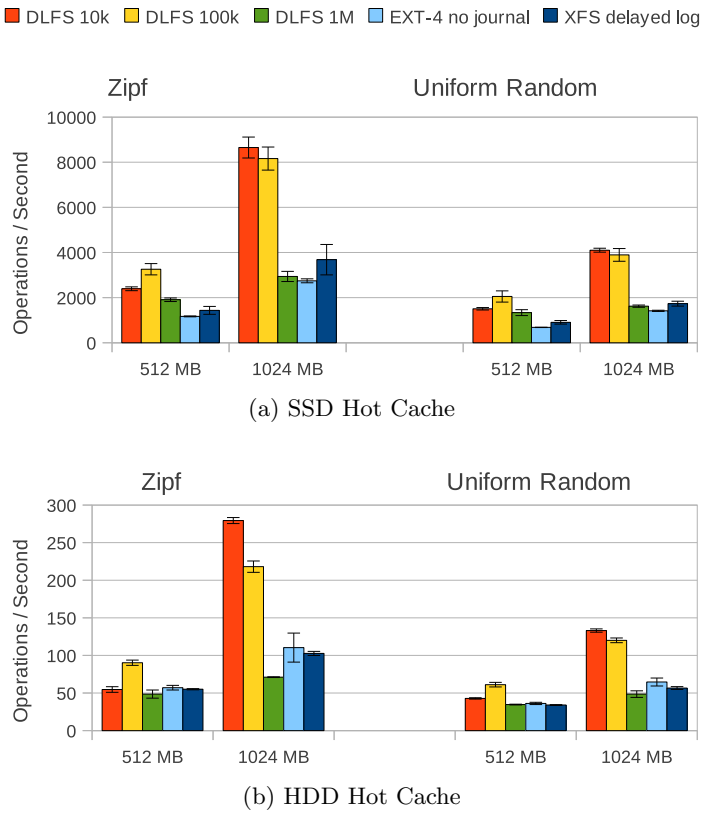


Fig. 4.10: Metadata Create Performance: Create, Hot Cache

ent DLFS configurations remains similar to previous observations. Differently from previous benchmarks, however, it is not the clear winner in absolute performance and Ext-4 can take the performance lead on the HDD.

In the hot cache scenario (Figure 4.10), DLFS performance characteristics differ from previous observations.

As directory metadata and data now have to be cached, the previously achieved high percentage of in-memory inode blocks is not reached. The drastic performance gain observed in previous hot cache benchmarks is not

repeated, although DLFS performance still profits from the available cache compared to Ext-4 or XFS.

Another difference to previous results is that the 10k DLFS configuration performs worse in the 512 MB RAM scenarios than the 100k configuration. The cache is limited by read in directory data so much that, contrary to all previous metadata benchmarks, the lesser overall number of inode blocks of the 10k configuration does not outweigh the possibility of multiple drive accesses in case the correct inode block was not found in the cache.

4.8.3 Data Performance

For big files, the storage of the allocation metadata as well as the allocation process itself takes place solely in the big file space as discussed in Section 4.2.1. Differently from the metadata benchmarks, the number of buckets of the DLFS file system layout therefore cannot influence data performance for big files.

In order to obtain small file performance independent of the lookup performance, the small file benchmarks measure the completion time of a read request sent directly after a small file is opened successfully. As just a single hash bucket will be accessed for each measurement, the total number of buckets in the file system is also irrelevant.

For this reason the DLFS layout configuration is not shown in the benchmarks of file system data performance.

4.8.3.1 Big File Performance

In order to obtain meaningful results that do not solely rely on the cache the total file size is set to twice the main memory size of 512 MB.

The I/O transfer size is set to 4 KB for sequential and random accesses, as large transfer sizes do not adequately measure random access performance.

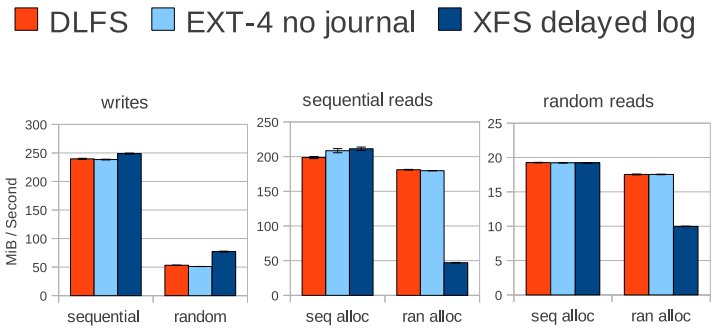
All other parameters were left at the initial IOR values (all threads share a single file).

The results of the various benchmarks are summarized in Figure 4.11. It is important to note that read accesses are differentiated for sequentially and randomly allocated files, as XFS in particular shows highly different performance depending on the allocation type.

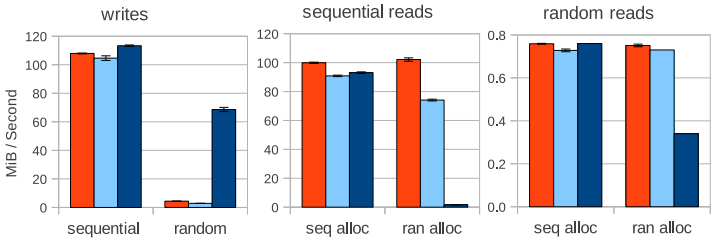
Write Performance Ext-4 and DLFS performance is straightforward. Random writes performed by IOR translate directly to random write requests issued to the physical device; performance therefore equals random write bandwidth.

XFS on the other hand employs a more complex block allocation strategy, where data blocks are only allocated when the data is flushed to the storage device. If the allocated file fits completely into the cache, this enables XFS to perform completely sequential allocation even if the file data is written randomly.

In this benchmark, however, the file data does not fit into the cache. As a result, XFS assumes that the missing data does not exist and ends up



(a) SSD



(b) HDD

Fig. 4.11: Data Performance: Big Files

allocating the data in small data extends (fragmented with respect to relative position inside the file). However, because the logical blocks mapped to this fragmented file data are sequential, writes on the device are much faster compared to the other file systems due to asynchronous random / sequential write bandwidth of the storage devices.

Read Performance Ext-4 and DLFS read access is almost equivalent for both the randomly and sequentially allocated cases.

XFS on the other hand suffers from its allocation strategy. As the file data is stored in a very fragmented way, sequential access is limited by the random read performance of the underlying storage device. Random access is also

slower compared to the other file systems for this case. Due to the large number of data extents, the B-Tree used to store extents is equally extensive. This leads to a large number of accesses to the storage device in order to read in the relevant metadata.

Once again, it should be noted that XFS only shows this behavior for randomly allocated files bigger than the cache size.

4.8.3.2 Small File Performance

Each file will only be accessed once in this benchmark. Additionally, cached metadata is not important for performance (measurements only start after the inode is read in successfully), so cache size is irrelevant for the results.

Multiple scenarios are analyzed in order to measure the impact of data placement and readahead strategies of the file systems.

All scenarios use 1,000 4 kilobyte files, which are accessed sequentially in the order they were allocated.

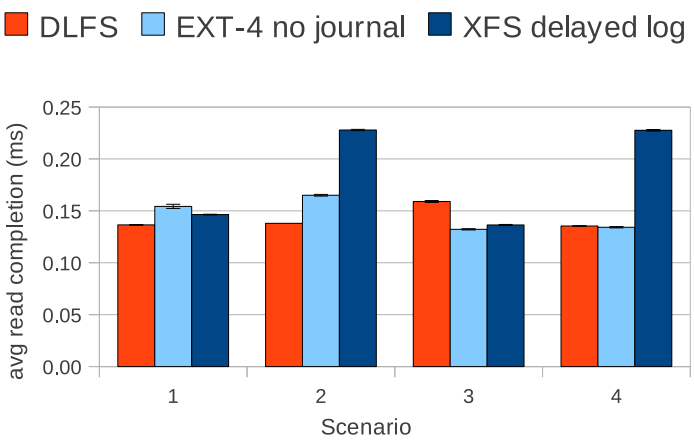
Some scenarios additionally use 4 megabyte files to showcase non-ideal cases.

1. One directory per small file, no big files are allocated.

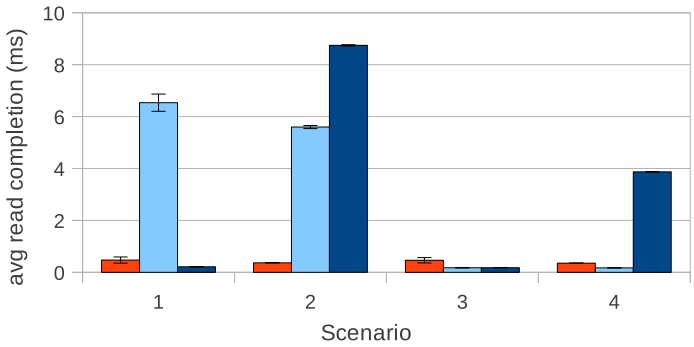
2. One directory per small file, one big file is allocated after each small file.

3. One directory containing all files, no big files are allocated.

4. One directory containing all files. Small and big files are allocated alternately.



(a) SSD



(b) HDD

Fig. 4.12: Data Performance: Small Files

The results displayed in Figure 4.12 show a very different picture for the two tested storage media.

On the SSD results are fairly homogeneous across file systems and scenarios. XFS shows the highest variance, with scenarios not containing big files

performing $\sim 60\%$ faster compared to scenarios that contain big files.

On the HDD, in contrast, results differ by a factor of more than 40. This difference is explained by the performance characteristics of the storage media: While the random access performance on the HDD depends directly on the distance of the accessed data to the current drive-head position (as well as hits in the drive readahead cache having a huge influence), the SSD performance is mostly independent from the data position. Thus, the effect of different allocation strategies becomes very obvious on the HDD.

Ext-4 handles scenarios 3 and 4 well, where files are allocated in the same directory. Since files are accessed sequentially, and the data of files inside a directory is stored (if possible) sequentially, it can take advantage of readahead in these cases. However, if accessed files are contained in different directories, the partitioning of metadata and data, as well as the distribution of directories across block groups, lead to a performance problem.

XFS performs well as long as only small files are contained in the scenario, as this data can be stored near - or even inside of - the XFS metadata structures. The introduction of big files, however, degrades the performance significantly even though they are never accessed.

DLFS shows the most constant performance of all file systems. While it is outperformed by both XFS and Ext-4 in their respective best-cases, the extreme performance degradation observed for the other file systems outside their best case scenarios is never encountered. As explained in Section 4.2.1, DLFS can take advantage of hardware readaheads in many cases to speed up file access.

4.9 Summary

The implementation of the direct lookup approach for the local case focuses on maximizing absolute metadata performance. To achieve this goal, a kernel level file system implementation was chosen and the name resolution in the Virtual File System layer was modified via a linux kernel patch to support direct lookup behavior.

The DLFS file system implementation was benchmarked against the widely used Ext-4 and XFS file systems. If all directory metadata and data relevant to a workload is cached, there is no significant difference to the lookup approaches: Both hierarchical and direct lookup can serve requests with a single I/O operation in this scenario. Therefore the metadata evaluation focuses on workloads with randomized access patterns.

Cold cache scenarios show the expected impact of the direct lookup approach when directly accessing files: When the directory hierarchy is not cached, skipping the IO related to reading in directories improves performance linearly by a factor depending on the average number of path components (directories in the path that would otherwise have to be read in).

Result for hot cache results are more complex. Not having to cache directory data and metadata can - in constraint main memory scenarios, enable the DLFS file system to cache all required metadata for file accesses when this is not possible for Ext-4 and XFS. As a result of having all metadata in main memory, metadata performance can is improved by orders of magnitude in these scenarios. Even when this is not the case (as for the tested one

million hash bucket configuration) the one-access nature of the direct lookup approach lets the DLFS file system consistently outperform both Ext4 and XFS in regards to metadata performance.

Data performance is not impacted significantly by the lookup implementation, but different optimizations of the tested file systems for accessing very small files and for writing data non-sequentially show big performance impacts for specific scenarios without there being a clearly superior system.

Chapter 5

Distributed File System Evaluation

5.1 Introduction

Due to the hierarchical lookup process, the normal approach to handle metadata in a distributed file system is to concentrate it on as few dedicated nodes as possible (*metadata server*) in order to facilitate quick access to all metadata associated with individual path components.

Skipping component traversal during the lookup operation, however, avoids performance degradation with highly decentralized metadata: During the lookup process only the metadata of the final path component is accessed. It is no longer required for metadata of other path components to be co-located at the same node. It can instead be distributed among all nodes of a distributed system. If metadata operations can be fully distributed among all hardware resources of a distributed system this clearly increases scalability.

To take full advantage of the capability for metadata distribution inherent to the direct lookup approach, a storage backend consisting of many simple, independent object storage devices is proposed[1].

[1] Distributing metadata to all storage nodes to take is also proposed by [APG11] and [ADD+08], see Chapter 6 for a discussion of related work.

Seagate's Kinetic storage devices (see Section 5.2) offer direct network access using a native key-value protocol and are used for this evaluation to coordinate all metadata management as well as to store all data.

The flat namespace provided by the key-value devices can be used directly by the direct lookup operation by setting a file's path as the key to its metadata.

5.2 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. Replacement drives do not have to be physically co-located.

In many distributed storage system architectures, storage nodes provide functionality aside from storing data such as failure detection or managing replicas. However, any system built on top of these ethernet drives cannot

rely on custom software running on the Kinetic devices to provide such functionality.

Furthermore, there is some overhead added on a per-device basis by the key-value management and Kinetic drives currently only offer a reduced bandwidth compared to drives implementing the traditional block interface. Despite these per-device performance limitations, the robustness aspect and scalability potential make the technology highly interesting for large scale systems.

The Kinetic drive interface provides (besides basic put, get, and delete) three features which are essential to the 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 [DDW08]).
- `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.

The full kinetic protocol definition and the various client libraries are available at github
`http://www.github.com/Seagate`. Additional information about the kinetic ecosystem can be accessed at `https://developers.seagate.com`.

5.3 Data and Metadata Layout

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

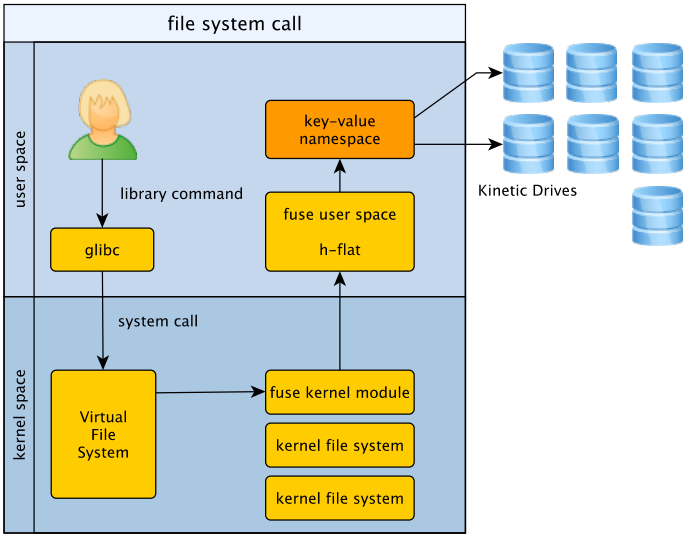


Fig. 5.1: IO-Path in Distributed System with Kinetic Drives

store all data and metadata (see Figure 5.1).

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. The value of the metadata key contains basic inode metadata, as well as the full path permissions for the file (compare Section 2.3.2.1).

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. When a client runs out of inode numbers it reserves a new block. The currently reserved range is stored on the kinetic backend in a normal key-value pair. Should a client need additional inode numbers, it reads the reserved_inodes key and uses an atomic put operation to increase the reserved inode range.

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 (the appropriate chunk number for a read or write request can easily be computed).

The primary reason to use the inode number to construct key-names of data keys is to decouple key-names from the file path: If the file should be moved or renamed later on this will not affect data-keys, as the inode number is not affected by this operation.

Additionally, when a file is deleted, data keys belonging to the file can be removed asynchronously and / or delayed in a background process. As the inode number will not be reused, the continued existence of these data keys cannot result in a name collision, even when the file name is immediately reused after deletion.

Both data and metadata keys are assigned to target storage devices using a hash function on the key name.

5.4 Hierarchical Functionality

The basic approach to support hierarchical file system functionality in a direct lookup environment using additional metadata has been introduced in Section 2.3.2.

A distributed multi-user environment introduces additional complexity. The global metadata has to be made available to all clients in a consistent manner. This could be achieved by multiple approaches:

- A *global service*, similar to a metadata server, could easily provide the required functionality.
- *Synchronous* updates (e.g. by performing the file system operation using a consensus based protocol) have the big advantage that outside of the operation itself no new complexity is added to the system and that all (valid) clients are guaranteed to have the same system wide metadata, and, therefore, the same view on the file system state. On

the other hand, performance and scalabilty issues are introduced when performing the operation itself, as clients have to know each other and cooperate.

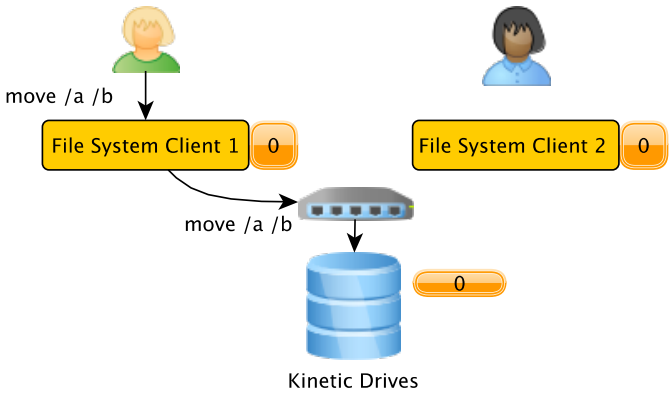
- *Asynchronous* updates are more complex. It cannot be guaranteed that all clients know about the same hierarchical operations that have been performed on the file system. The advantage is that the approach does not introduce any performance limitations.

In order not to limit scalability and re-introduce a central servers into the lookup process, the system-wide metadata has to be cached at every client. The asynchronous approach was chosen to enable an architecture of completely independently operating clients.

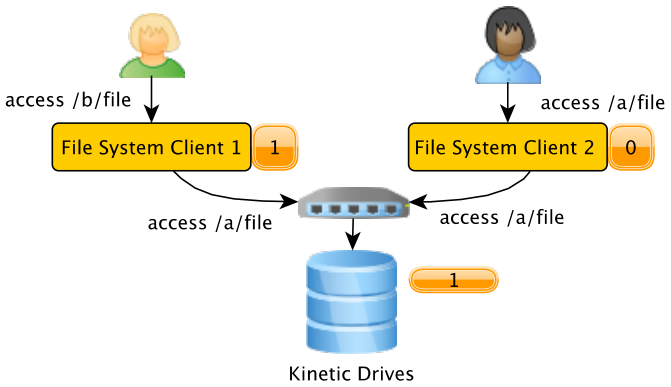
However, this leaves the consistency problematic unsolved. Situations where the local knowledge is not equivalent on all clients are at least temporarily unavoidable.

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. The resulting file system semantics will be referred to as *snapshot semantics* in the following.

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 5.2 using a directory move operation as an example. After the move has been performed in 5.2a by Client 1, it can (and must) use the new directory name when accessing files in the moved directory in



(a) Time Point A



(b) Time Point B

Fig. 5.2: Distributed Hierarchical Functionality: Snapshot Semantics

5.2b. Client 2 does not know anything about this move operation because its local snapshot containing hierarchical information has not been updated in 5.2b. 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.

5.4.1 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 location for path `/b` during the move operation. This key blocks the location so that no new file or directory can be created with that name and 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.

5.4.2 Implementation

Hierarchical file system operations are journaled as key-value pairs to the storage backend. Deleting a moved directory or symbolic link is equally journaled.

Key-names of the journaled events follow a naming scheme using a fixed prefix and a sequence number to provide concurrency resolution as well as a consistent sequence of events between multiple clients.

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 (compare implementation described in Section 4.3).

Periodically a complete snapshot is serialized and stored so that it can be chosen as a starting point by clients wishing to update their local snapshot. This limits the worst case time to obtain an up-to-date snapshot which is especially relevant for fresh clients joining the system.

Example Using a file system where no hierarchical operations have been performed yet, three clients concurrently attempt to do hierarchical operations:

- Client-1 wants to move directory /a to /b:
  {`journaled-op-1, move /a /b`}
- Client-2 wants to move directory /a to /c:

{journaled-op-1, move /a /c}

- Client-3 wants to create a symbolic link from /link to /a:
  {journaled-op-1, symlink /a /link}

Atomic put operations prevent the same key from being created multiple times. Following, only one client succeeds with the hierarchical operation. Let us assume Client-3 wins the race. The only journaled operation is thus
{journaled-op-1, symlink /a /link}
and the path-mapping table corresponding to the hierarchical snapshot for time-point 1 is accordingly:

/link	→	/a

Table 5.1: Path Mapping Table after first Hierarchical Operation

Client-1 and Client-2 fail to create the journal key for the hierarchical operation they want to perform. Following, both clients read in all journaled operations, update their internal path mappings to reflect the snapshot integrating the last known journaled operation (snapshot-1), verify that the operation they wish to perform is still valid and attempt to journal the operation again. Assume Client-1 wins the race this time. Journaled keys are now
{journaled-op-1, symlink /a /link},
{journaled-op-2, move /a /b}
and the path-mapping table corresponding to the hierarchical snapshot for time-point 2 is accordingly:

/link	→	/a
/b	→	/a
/a	→	/unique_name

Table 5.2: Path Mapping Table after second Hierarchical Operation

Client-2 fails again. It updates its snapshot to snapshot-2. It cannot verify that the attempted operation is legal at this point, as directory /a does no longer exist. Client-2 thus fails the operation with the corresponding error.

To continue the example, assume a user later on discovers that the symbolic link is leading to nowhere and decides to delete it. The corresponding list of journaled key-value pairs is now:
{journaled-op-1, symlink /a /link},
{journaled-op-2, move /a /b},
{journaled-op-3, delete /link}
and the path-mapping table corresponding to the hierarchical snapshot for time-point 3 is:

/b	→	/a
/a	→	/unique_name

Table 5.3: Path Mapping Table after third Hierarchical Operation

Any client can update its snapshot at any point by internally replaying the journaled operations in the supplied sequence.

5.4.3 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).

5.4.4 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.

One more interesting observation of the empirical data supplied in Chapter 3 is that rename and setattr operations almost always 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.

5.5 Direct Lookup Implementation

This evaluation focuses on system scalability and less on absolute performance. The advantages of a universally usable implementation that is not limited to a single custom Linux kernel, outweigh a potential slight loss of performance by the additional layer introduced by intercepting glibc library calls and manipulating paths. The current implementation works for all Linux kernels as well as for OSX.

Function call interception via library preloading has been introduced in Section 2.3.1. To re-iterate, it intercepts all path-based file system library function calls (such as open, stat, etc) and modifies the path to hide path-components below the file system mountpoint.

```
/a/mountpoint/path/to/file
```

$$\rightarrow$$

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

During the component-based lookup implemented in the Virtual File System layer[2] the lookup operation thus encounters the single path-component `path:to:file`, and calls the lookup operation of the file system mounted at `/a/mountpoint` for that path-component. Inside the file system, the substitution of the slash character is now undone and the full path is looked up.

Virtual Directories Hiding path components from the Virtual File System and making all paths appear as files located directly in the file system root introduces a performance bottleneck: The Virtual File System will serialize some operations as it believes every operation in the file system to take place in the same directory.

This additional serialization is not required and an unnecessary performance limitation as concurrency resolution is implemented on the file system level.

[2]Discussed in principle in Section 2.1 and in detail in Section 4.4.

To work around this behavior, *virtual directories* are introduced into the path below the mountpoint.

```
/a/mountpoint/path/to/file
```

$\rightarrow$

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

The virtual directory is removed by the file system before the file system lookup operation is executed.

File paths are hashed to one of many virtual directory names. The Virtual File System thus believes operations within the file system to execute inside hundreds of directories as opposed to a single directory, preventing unnecessary serialization from taking place.

Optional Hierarchical Lookup Library preloading is not hard-coded into the operating system kernel. It can be enabled and disabled at runtime. Additionally, as discussed in Section 2.3.1, it is theoretically possible for users to bypass the glibc file system library functions and use system calls directly. In this case no path manipulation would occur.

If the iointercept library is not preloaded or system calls are used by the user, the Virtual File System will employ standard hierarchical lookup. This means, that the file system will be queried for each path component separately. This is not problematic from a functional perspective, the file system remains completely usable in this scenario. All performance benefits linked to direct lookup functionality are however lost.

5.6 Metadata 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.

There are no metadata servers or other active services that could potentially take over a similar role in the proposed system architecture.

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: A read-only LRU cache with item expiration is used 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.

Write caching is completely avoided and versioned, atomic puts are used to prevent write conflicts as described in the following Section.

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.

5.7 Concurrency Control

Section 5.2 introduces kinetic capabilities. Versioned keys and atomic puts allow concurrency control without explicit locking in combination with the read-only caching Strategy.

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. The result on the second try could be completely different (the operation might even not be legal after the metadata update), but this process is entirely internal to the file system client.

The user only receives a result after the requested file system operation is completed. For most POSIX functionality this adequately solves concurrent access without using locks, as there is no defined order for two requests

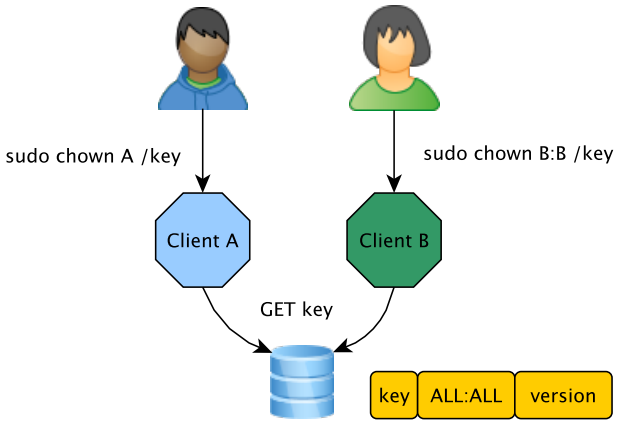


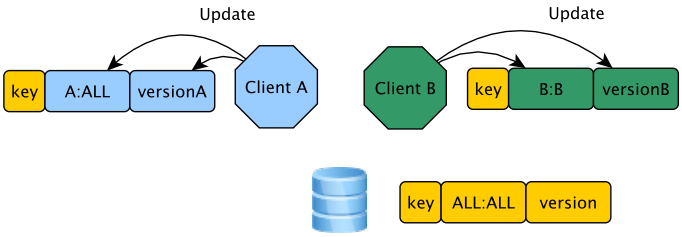
Fig. 5.3: Concurrency Control: Put - Starting Situation

spawning from independent file system clients. Serialization is done when necessary by creating temporary serialization keys.

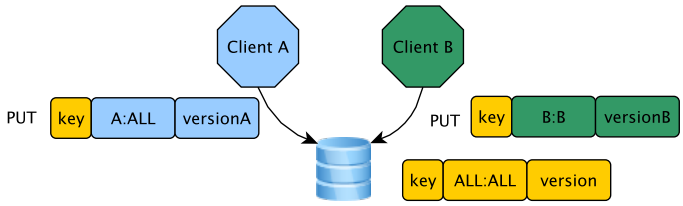
Example To illustrate the concurrency resolution process an example is discussed in detail in the following.

Figure 5.3 shows the initial situation: Both user A and user B concurrently attempt to assume ownership of the existing file */key* using the *chown* command. User A attempts to change solely the owner, while user B attempts to change both owner and group. In order to access the file in the first place, the file metadata stored in the value of the key *key* has to be read in from the device.

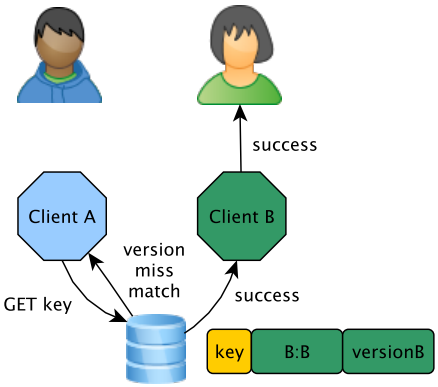
In Figure 5.4, the file system clients attempt to execute the operation requested by the users. Both clients detect that the requested operation is



(a) Clients locally update value and version



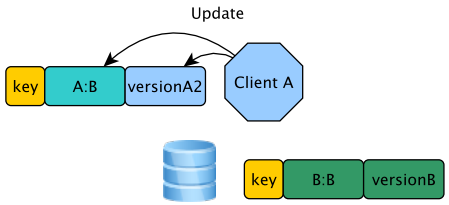
(b) Clients concurrently attempt to put updated key-value pair



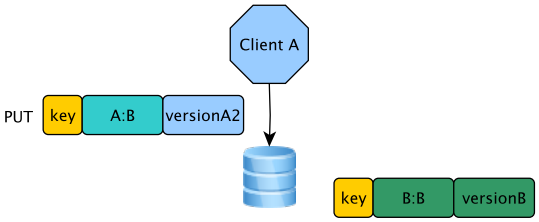
(c) Put B succeeds, Client A re-tries operation

Fig. 5.4: Concurrency Control: Put - Initial Operation Attempts

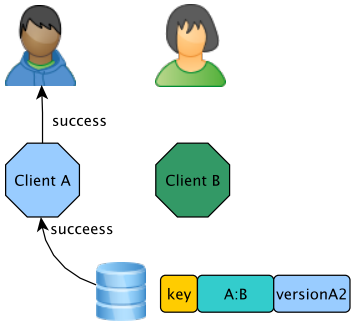
legal and accordingly modify the metadata value. Afterwards, a new unique version is generated for the new key-value pair and both clients attempt to write back the key-value pair using the put command.



(a) Local update of value and version



(b) Put operation



(c) Success

Fig. 5.5: Concurrency Control: Put - Resolution

The put command can only succeed for one client, as it executes atomically and requires a pre-defined key version to be current on the drive.

The example assumes that the put command of client B arrives first at the kinetic drive and thus succeeds. As client B has completed the requested

metadata operation, it returns success to the user.

The put command of client A failed and returned an invalid version error code. Client A knows that the metadata value it performed the requested operation on is outdated and it has to read in the current value from the kinetic device again.

Figure 5.5 shows how client A executes the metadata update and write-back process again. This time there is no version conflict, the put operation succeeds and the client returns success to the user.

As user A only attempted to change the user attribute and not the group attribute, only part of the metadata update performed by client B is overwritten in the final value.

Concurrency Resolution for Data Keys As mentioned in Section 5.6, multiple file system writes can be aggregated before a single flush operation is triggered for the associated data key. This behavior is necessary from a performance perspective.

In order to maintain the concurrency resolution behavior discussed for metadata, for every unflushed data key a list of updated bit-ranges is stored. If concurrency resolution becomes necessary, only the indicated bit ranges of up-to-date value are overwritten. This guarantees correct file system behavior if for example a concurrent compute job writes back fragmented data from many clients (e.g. in an array structure) and the writes are not aligned with the key size.

5.8 Directories

Directories are not accessed during direct lookup operations. The only file system operation that requires directory entries when using the direct-lookup approach is readdir / ls. Readdir functionality is not the most performance sensitive file system operation. It suggests itself to prioritize optimizing directory entry creation and removal over listing of directory entries.

Performance of creating files in a single directory[3] is usually limited by directory accesses: The directory is locked during file creation to ensure that a file with the same name is not created multiple times.

Locking directories becomes unnecessary with direct lookup: The key-value backend in combination with the fixed mapping between file path and key name automatically provides the required functionality. If multiple clients try to create the same key only one client can 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. Using the inode number for directory entry key-names allows directories to be renamed without affecting directory entries. It does not

[3]This is a common use case in Supercomputing. Users generate many files in a single directory concurrently using a large number compute nodes.

incur additional costs, as the directory metadata has to be accessed to verify access rights in any case.

Readdir functionality is supported by key-range requests (compare Section 5.2) for all key names starting with the directory's inode number. From a performance perspective, key-range requests return between 10 to 50 thousand key-names per second per first-generation Kinetic drive.

The number of Kinetic drives storing directory entry keys of a single directory is a configurable property. The ideal value depends on expected directory sizes. When using multiple drives, listing even large directories is reasonably fast. The generation of directory entries is not serialized by any central data structure and thus can scale linearly with the number of kinetic drives used.

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.

Summary Using dedicated directory entry keys in combination with relaxing directory time stamp updates allows files to be created without taking locks or writing to directory metadata. File creation performance in a single directory can thus reach the file-system-wide overall creation performance maximum.

5.9 Distributed System Reliability and Security

A distributed system introduces additional reliability and security challenges which are discussed in this Section.

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 the attacker could use the open Kinetic protocol to communicate with the drives directly (and get / put / delete keys), completely bypassing the file system client.

While security measures beyond standard Kinetic identification are not implemented in the file system client at this point, the following discusses some potential solutions.

Coarse Grained Enforcement Kinetic access rights can be granted for limited scopes of key-names. It is, for example, entirely possible to create a user John who solely can write keys that start with `/home/john`.

Since metadata key-names mirror path names in the 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.

Proxy Servers A simple way to keep Kinetic credentials completely out of clients hands is using proxy servers.

Clients 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. Clients could distribute calls among multiple proxy servers due to the minimalistic caching requirements (compare Section 5.6).

This approach also allows trusted clients (e.g. locally managed nodes vs remote nodes) to continue directly accessing the drives.

However, it introduces a level of indirection into the system for at least some clients an requires additional hardware.

Proxy Authentication 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.

Reliability

The proposed file system architecture consists of two components: File system clients and Kinetic drives.

Both drives and clients can individually crash or fail at any time. Additionally, environmental factors such as power outages or network failures can affect many drives and / or clients at the same time.

Client Crashes or Network Failures

File system clients are at the same time the only actors in the system and considered to be unreliable (they 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, metadata operations consisting of multiple steps are ordered so that a crash will always leave a dangling directory entry.

For example, a file creation will first put the directory entry key and only when the put operation succeeded put the metadata key. If the client should crash after creating the directory entry key but before creating the metadata key, this partial operation can be rolled back at any time by simply removing the dangling directory entry.

Move operations are a special case. If the target is a directory or a symbolic link it is a hierarchical operation and are handled as discussed in Section

5.4. If the target is a regular file, it is the by far the most complex metadata operation for the h-flat file system. An additional recovery-key becomes necessary to provide guaranteed rollback functionality for all crash scenarios.

The steps of the move operation are as follows:

1. Create new directory entry. This also serves as the first step for concurrency resolution: No two clients can move to / create the same entry.
2. Create recovery key containing both the path of both the origin and destination of the move operation.
3. Remove the old directory entry. This is the second synchronization step for concurrency resolution: No two clients can move from / delete the same entry.
4. Copy metadata key to new location delete it at old location.
5. Remove recovery key.

Without the recovery key it would be impossible to rollback correctly if the client crashes after step 3: There is no other way to relate the orphaned directory entry created in the destination directory to the deleted directory entry in the origin directory.

Device Failures

To handle device failures, a basic replication strategy has been implemented. Replication is often not the optimal strategy to provide redundancy for

large amounts of data. Other redundancy strategies such as erasure coding are applicable and have been implemented for Kinetic drives[4]. While the implementations might be integrated into the file system in the future they were not available at the point of time of the evaluation. For this reason, only the implementation of basic replication will be discussed.

Replication Replication is often handled using quorum based consensus algorithms, a method that is not applicable to Kinetic drives as they do not have the capability to be active participants in a voting based scheme.

The file system implements write-all, read-any replication primarily due to its simplicity in failure scenarios.

The following briefly discusses how Kinetic drive features (compare Section 5.2) are used to coordinate replicated writes as well as to handle drive failures in the context of independent, unreliable file system clients.

Even though updating a single key on a drive is atomic, **partial writes** can happen when replicating keys. This is most commonly the case when two or more clients try to update the same key at the same time and their update requests arrive in different orders at the different drives (keys are pushed in parallel); the situation can also arise if a client crashes during a replicated put operation.

Each replication group has a fixed priority order of drives that is used to resolve partial writes: Even if there are multiple independent clients involved, they will follow the priority order for competing updates and each client

[4]https://github.com/cern-eos/kineticio

detecting a partial write will ensure that the *correct* key is replicated to every drive of the replication group.

When a put operation fails due to a drive not being reachable (or returning unexpected errors), the drive is failed by the client encountering this situation. An updated description of the replication group is written to the remaining drives of the group and the *cluster version* of all drives of the group is increased.

The cluster version prevents any client with an outdated view on the replication group to perform operations on the remaining drives. It will instead encounter an invalid cluster version error for any operation and be forced to update its view of the replication group.

Due to the cluster version being linked to the replication group description, multiple clients encountering the same error at the same time is unproblematic.

If a failed drive of a replication group is replaced or reconnected to the network, it has to be reintegrated into its replication group using a namespace check utility that is supplied, as there is no advanced automatic cluster health monitoring features implemented yet. During the re-integration a drive is immediately used as a write target, but will only be used for reads after the process is complete and it is verified that the drive is up-to-date for all keys.

The file system can journal updates to keys in a replication group with a failed drive to drastically speed up this recovery process.

5.10 POSIX Conformity

Library preloading effectively limits maximum path length below POSIX standards. The whole path appears as a single path component to the Virtual File System as discussed in Section 5.5. This causes the Virtual File System layer to enforce the maximum size allowed for a single path component, which is significantly smaller than the size allowed for the entire path.

The second limitation is that if the file system is mounted in relaxed POSIX mode as described in 5.8 in order to increase file create performance, directory time stamps are not updated correctly.

Except for these points the direct lookup does not introduce any POSIX conformity limitation in the file system.

However, it should be noted that the the simple client-side write caching employed for file data would need to be extended to be fully POSIX conform. In the current implementation, written data only becomes visible to other clients based on the chunk size of data keys. Continuous smaller write IO requests are aggregated on the client up to the data key chunk size and thus not immediately visible to other clients.

5.11 Benchmarking

Kinetic drives were not yet in mass production at the point of time this performance evaluation was undertaken. Only 64 prototype drives were available for the evaluation.

In order to evaluate scalability beyond a 64 drive cluster, Amazon EC2 was used to simulate a larger amount of drives.

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[5].

Two of these test cases were used to evaluate some base performance metrics of the implementation:

- hpcsio03 benchmarks multi-stream data throughput with large block I/O
- hpcsio04 benchmarks file creation rates in combination with small write performance. This is achieved writing a random amount of data (up to 64 KB) to each created file.

In addition the mdtest[6] HPC benchmark is used to examine pure file creation performance in a single directory, as direct lookup promises increased scalability for this special case (see Section 5.8).

5.11.1 Hard & Software

The following software versions have been used for the evaluation on both environments:

[5]http://hpcs-io.cray.com/
[6]http://mdtest.sourceforge.net

- 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 64 Kinetic drives running firmware 2.0.5. The drives were located in three enclosures, each attached with a dual 10 gigabit connection to the network.

Two physical servers were used as file system clients. Each server contained 64 GB of main memory, dual Intel Xeon E5-2630 CPUs and an Intel 82599EB dual port 10 gigabit Ethernet controller. Both servers were running Ubuntu Linux 12.04. Gcc version 4.7.3 was 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 were chosen from the very low-end (and cheap) tier.

Kinetic Drives: Simulated on m1.small (1 virtual CPU, 1.7 GiB main memory) instances running Amazon Linux with a 3.10 kernel and OpenJDK 1.7.0_65. For the simulation, the official java Kinetic simulator (snapshot 0.6.0.2) was used.

File System CLients: m3.medium (1 virtual CPU, 3.75 GiB main memory) instances with Ubuntu 14.04 and gcc 4.8.2.

Benchmark Configuration Benchmarks are started with library preloading to enable direct lookup (Section 2.1) and use relaxed posix mode for directory timestamp updates (Section 5.8).

All benchmarks use mpi to scale beyond single process / single client machines. POSIX functionality is typically serialized per thread (e.g., file creation), so mpi processes were oversubscribed 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.

Unless explicitly stated otherwise, all benchmarks use a cache configuration with one second item expiration. Workloads are scaled on a per-drive basis.

- 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). Cache size is limited to 1GiB on EC2 clients. The O_DIRECT option is specified to minimize context switches.

- HPCSIO 04: Standard options are used (50 second runtime, 10 directories per process). The O_DIRECT option is specified to minimize context switches. Cache size is limited to 1GiB on EC2 clients.

- 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.

Impact of Replication The simple replication technique implemented for the system assigns drives to fixed size replication groups. A replication group size of one (effectively disabling replication) was used for the benchmarks 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.

To obtain an idea how absolute performance would compare in a system using triple replication groups as write targets instead of no replication, performance for the base DARPA hpcsio benchmarks are compared in the following.

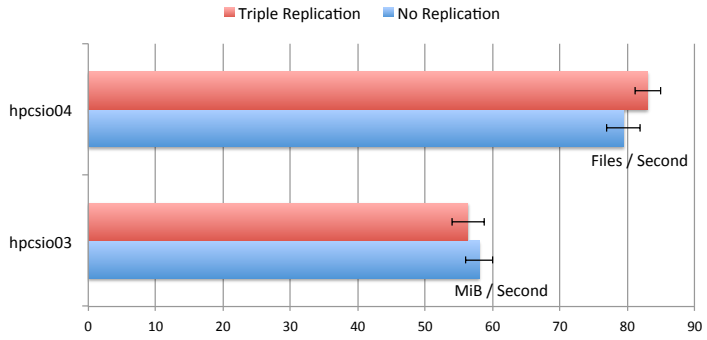


Fig. 5.6: Performance Impact of Replication

Figure 5.6 shows performance for a single triple replication group compared to a single drive without replication. Both values were obtained using a single file system client in the physical Kinetic cluster.

Intuitively for a write-all - read-any replication there is no performance gain when writing data. Instead, write performance is dictated by the slowest

drive in the group, as is reflected by the hpcsio03 write performance.

Maximum read performance on the other hand is multiplied by the number of replicas.

The hpcsio04 benchmark does not have an active read component. Due to the configured cache item expiration of one second, the metadata of every directory is verified every second during the benchmark run. Distributing the required drive accesses in the replication group has an overall positive performance impact, even though the write process is penalized.

Overall the performance impact, positive as well as negative, is quite small for the evaluated benchmarks and certainly can not change the overall scalability picture.

5.11.2 Results

Benchmarks are named based on the following scheme:

`location-#clients-#processes`

where location is either Longmont (`LM`), referring to the physical location of the Kinetic cluster, or the Amazon Cloud (`AWS`).

For up to 64 drives the results for the physical cluster can be compared directly with simulation results.

Additionally, 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

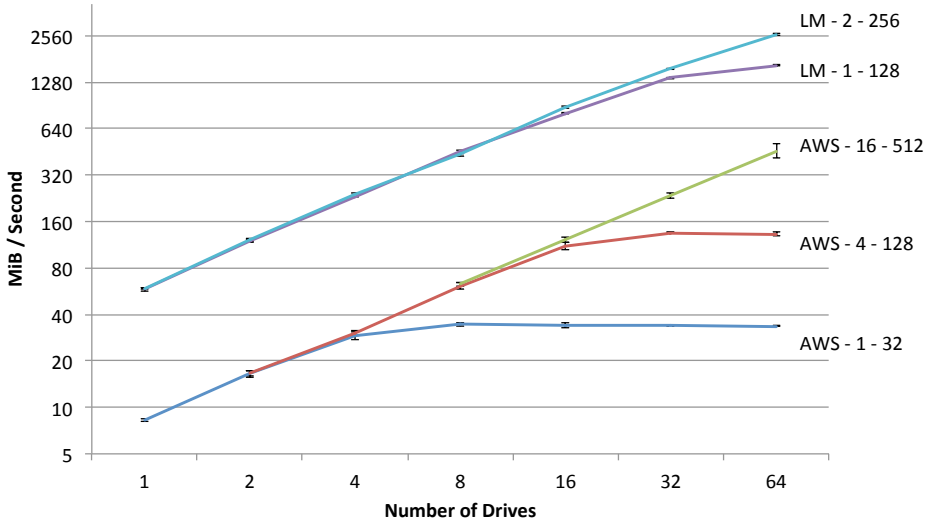


Fig. 5.7: Performance: Multi-Stream Data Throughput with Large Block I/O (HPCSIO 03)

Figure 5.7 displays observed write throughput for the HPCSIO 03 scenario. Throughput is shown for one to 64 drives, comparing system performance for both physical and simulated environments and a varying number of file system clients.

Absolute performance values differ quite drastically between the physical and simulated systems: The achieved per-drive write performance for physical drives is roughly 55 MB/s while the per-drive performance for simulated drives in the Amazon cloud is only 7.5 MB/s. Further, a single physical client can saturate 4 times as many drives as a single EC2 client.

Despite these large discrepancies in absolute performance, the physical and

simulated performance display very similar characteristics regarding the scaling properties of the system: 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 physical clients the observed performance 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 by 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 5.8) distributes creates across many directories and performs small data writes on freshly created files. The mdtest benchmark (Figure 5.9) in contrast is used to create files in a single directory without any data writes.

Differently from the previously considered data throughput, the observed per-drive performance for these IOPs based workloads is a lot closer when comparing the physical drive with the simulator.

The 16 client `AWS` scenario outperforms the 2 client `LM` scenario.

From a scaling perspective similar behavior to data throughput can be observed: Performance scales linearly until client performance becomes a

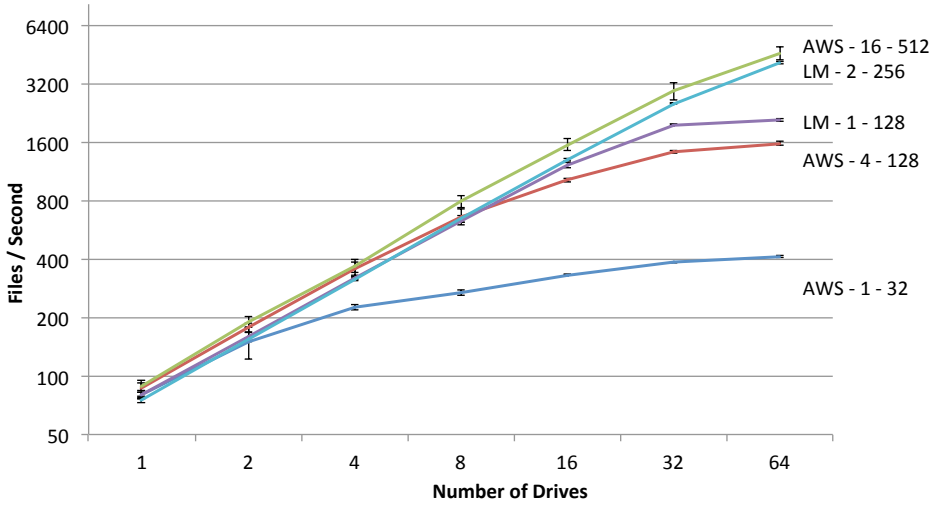


Fig. 5.8: Performance: File Creation and Small Writes (HPCSIO 04)

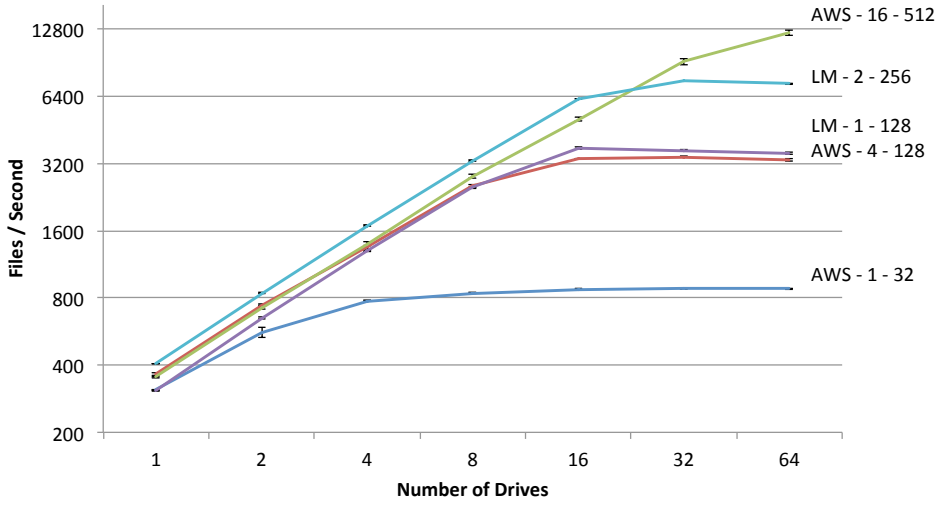


Fig. 5.9: Performance: File Creation in Single Directory (mdtest)

bottleneck.

In the HPCSIO 04 scenarios (Figure 5.8) 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 5.9), which will be discussed in the following section supported by additional data from larger-scale simulations.

	hpcsio03	hpcsio04	mdtest
64 Clients / 256 Drives	1719.00 MiB/s	16313.31 files/s	32762.30 files/s
95% Confidence Interval	61.86 MiB/s	672.25 files/s	1879.29 files/s
128 Clients / 512 Drives	3389.07 MiB/s	30519.90 files/s	41463.57 files/s
95% Confidence Interval	66.66 MiB/s	779.01 files/s	483.85 files/s

Table 5.4: Scalability: Performance in Large Scale Simulations - Averages and Confidence Intervals

Large Scale Simulations

This Section introduces larger simulations to help evaluating system scalability. Performance is evaluated for 256 and 512 drives in addition to the previously discussed results for up to 64 drives.

In order to evaluate system scalability without focusing on bottlenecks such as limited client performance, this part of the evaluation is focused on average per drive performance with a fixed client-to-drive ratio (four drives for every client).

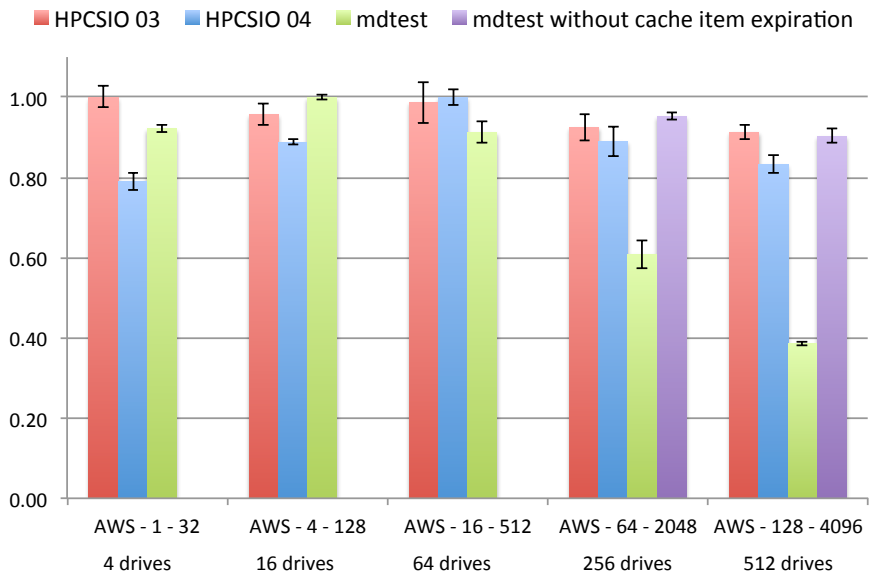


Fig. 5.10: Scalability: Normalized Per-Drive Performance for Benchmarks and Simulations

Figure 5.10 shows normalized per-drive performance for the benchmarks (see Table 5.4 for absolute values).

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 5.10, this performance degradation is not present when extending or disabling the item expiration time as described in Section 5.6. The corresponding loss in system agility, however, might not be acceptable in some use cases.

5.12 Summary

This chapter introduces a system architecture with fully decentralized metadata distributed equally among all storage nodes. Due to the direct lookup approach there is no need to centralize metadata, as no directories in the file path are accessed and a single storage node can resolve a lookup operation. The implementation takes full advantage of this property to maximize scalability by not introducing any centralized components and to simplify the overall system by only using low-level key-value storage nodes as a storage backend.

Both regular metadata and metadata containing hierarchical information is distributed in the storage backend using a hash-based placement approach.

As the focus of the implementation is showing scalability properties and not raw performance, the more flexible library preloading technique is used over a kernel-patch to provide hierarchical lookup support.

A relaxations of POSIX semantics is proposed that allows some hierarchical operations to be performed with eventual consistency. Individual file system clients may have a different snapshot of hierarchical metadata, giving them different views of the hierarchical namespace. This change allows supporting hierarchical functionality with fully independent, asynchronously updated file system clients that do not impose scalability limits to the system.

Evaluation of system scalability shows metadata performance scaling horizontally with available hardware resources in most cases. The only limiting factor to scalability that could be identified are concurrent file creations in a single directory. This can lead to overloading the storage node that holds the directory metadata when the cache-expiration is low and many clients keep re-verifying directory metadata.

Chapter 6

Related Work

6.1 Skipping Component Traversal

For both local and distributed scenarios, a number of systems have experimented with skipping component traversal during the lookup approach.

Brandt et al [BMLX03] use a hashing based scheme to select one of several metadata servers for a given file path, supporting distributed direct lookup functionality. The authors chose a lazy metadata update strategy to handle hierarchical functionality. This is similarly done in this thesis for directory permissions changes: Outdated permissions are verified recursively against their parent directory until they can be updated. In contrast to this thesis this approach is employed for all hierarchical functionality. When a lookup operation fails due to the requested metadata not being available at the expected location for any reason (e.g. a parent directory has been moved changing the file path), parent directories are recursively looked up on their respective metadata servers until the client request can be satisfied. This requires less client-side metadata compared to the approach taken in this thesis, but incurs a significant client-visible performance overhead after a directory move or when looking up paths to files that do not yet exist. It

also removes the flexibility of differentiating between large and small scale directory move operations: The approach forces affected metadata to be migrated in the same manner no matter if a directory move only changes the path to a dozen files or to billions of files. To avoid the recursive overhead when using symbolic links, the system lazily creates indirection entries on access. If a symbolic path to a file is reused it can be followed with just one additional access. However, this comes at the cost of creating additional metadata for every symbolic path used. This metadata will have to be kept up-to-date and, depending on the usage of symbolic links in the system, can become more numerous than actual file metadata.

Gonzalez et al [APG11] largely follow the approach of Brandt et al but, similarly to this thesis, employ object storage devices as a storage backend for both data and metadata. Differently from the approach taken in this thesis, the proposed object storage devices (*called OSD+*) are not simple key value stores but have capabilities similar to traditional metadata servers and fully implement the lazy metadata update strategy proposed by Brandt et al. The approach further differs by storing file metadata in directories and providing direct lookup only on a directory level, foregoing some of the performance and decentralization advantages of direct lookup in favor of reduced metadata migration requirements in case of a directory move.

The Vesta Parallel File System [CF96] places metadata at I/O nodes and uses path based hashing to select a master node responsible for the file metadata. It does not handle path permissions, hard links or supply a solution to directory moves beyond brute-force.

To avoid the directory-rename problematic for hash based metadata placement entirely, some approaches [WFWL09, LZ07] 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. This id-based metadata placement approach has also been used while keeping normal component traversal for the lookup operation [FXZ08].

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.

6.2 Placement Strategies

This thesis does not focus on placement strategies and the introduced file systems use a simple hash function for the purpose. For the distributed approach this would not be flexible enough in a production setting (it does not account for dynamically adding and removing nodes to/from the system). However, there is a huge set of literature on key placement in a dynamic, multi-target key-value environment, e.g. [SM98, WBMM06, WLBM07, BEMS07, MEK+11].

Another strategy for metadata placement besides pseudo-random hash-based placement is the use of bloom filters [ZJWX08, HZJ+08] to map file names to metadata locations.

6.3 Key-Value and Object Storage Devices

There have been different approaches to re-integrate file system support on top of simple key-value stores and file systems based on key-value stores have already been implemented [WBM+06, Sch03, WUA+08]. All of the above approaches maintain the traditional approach of providing metadata functionality using a separate metadata server (or server cluster).

Using the capabilities of a osd based storage backend to get rid of explicit metadata servers for file systems is an idea that is also proposed by [ADD+08]. The authors use object attributes to support directory functionality while keeping the overall file system architecture conventional.

Not taking into account skipping component traversal, h-flat shares a lot of similarities with the Ceph file system [WBM+06]. Both file systems build on a global object namespace, which is provided in Ceph's case by the RADOS [WLBM07] 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.

Similar to the Kinetic protocol, the ANSI T10 OSD standard[1] specifies a command set for object storage devices. In comparison, the T10 standard is broader and offers additional features (e.g. big objects, object attributes), but does not specify the low-level features (e.g. versioned atomic puts) the Kinetic protocol offers to handle concurrency on the device side. The concept

[1] http://www.t10.org/drafts.htm#OSD_Family

of simple, independent object storage devices was already well established [ADF+03, NFI+08] before Kinetic, but a hardware implementation at a drive level had not been available.

For pure object-storage systems, metadata management operations can be handled by individual object storage devices to remove pressure from (or, ideally, any need for) centralized metadata servers [NEF+12]. Another approach to overcome restrictions of centralized metadata servers is introducing programmability into the object storage devices [GLK+10].

6.4 Performance and Optimization

There are approaches to optimize the performance of the traditional component-based lookup: The embedded inode technique, for example, stores inodes inside the directory data [GK97] and can thus aggregate two separate steps of the lookup process into one disk access.

There exist a number of optimizations concerning access to the data of small files. Conceptually, almost all approaches follow the same idea: Storing the file data as close as possible to the metadata. In the best case scenario, file data can be stored directly inside the inodes [ZG07], using the space normally reserved for block pointers. Another approach is to cluster the data of multiple small (sub-blocksize) files into a single file system block [GK97]. Assuming that the clustered files are often accessed together, this can lead to high cache-hit rates.

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 [BKL+10] and the Hierachical Data Format [2].

A completely reverse view of metadata performance is taken by log-structured file systems [RO92, PCG02]. While metadata access is heavily penalized due to non-deterministic inode placement, the update-out-of-place technique enables metadata creation limited only by the sequential bandwidth of the storage device.

There are suggestions for both the distributed [HJZ+09] and the local case [SM09] to completely abandon the hierarchical namespace concept in favor of a search based, semantic one.

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

Chapter 7

Summary and Conclusion

This thesis is primarily based on three of the author's publications: [LMB10] describes a file system prototype implementing direct lookup on top of Ext-2 without supporting hierarchical functionality, [LCB13] extends this work to a general purpose file system and [LHCB16] introduces the distributed implementation of the approach.

The key characteristic of the direct lookup approach is that it decouples system performance from the existing directory hierarchy: The cost of a lookup operation no longer depends on the number of directories in a file's path, the cost of creating a file no longer depends on the number of files already existing in a directory and multiple processes can concurrently create files in the same directory without a performance penalty.

These performance characteristics become more beneficial the larger a file system grows.

The downside of the direct lookup approach compared to the traditional component-based lookup approach is that operations that affect the directory hierarchy, such as moving a directory, cannot be supported in a straightforward manner.

Chapter 2 introduces a variety of techniques that rely on additional metadata information being available at - or before - the lookup operation to make (almost) POSIX compliant file system functionality available while using the direct lookup approach.

While caching this additional metadata permanently is only possible in limited quantities, the analysis of real-world file systems in Chapter 3 shows that hierarchical operations that require metadata to be stored (and cached) with the proposed approach are extremely rare and, in addition, frequently of a type that necessitates only temporary additional metadata.

The kernel level local file system implementation discussed in Chapter 4 provides a straightforward comparison of metadata performance for the different lookup approaches. In cold cache scenarios, the hierarchical lookup employed by the ext4 and XFS file systems requires I/O for every path component and is multiple times slower compared to direct lookup. Hot cache scenarios showcase another layer of performance characteristics: Because directory metadata and data does not have to be cached for direct lookup, caching requirements can be significantly lower for certain scenarios, leading to performance improvements of several orders of magnitude.

The distributed file system introduced in Chapter 5 is based on hard drives that integrate a key-value interface and network access in the actual drive hardware (Kinetic storage platform). Taking advantage of higher-level drive functionality to handle metadata and concurrency resolution on the drives themselves, the system does not require any central servers (e.g. metadata, locking). Due to skipping component traversal, metadata operations can be

fully distributed among all hardware resources of the system, resulting in horizontally scaling metadata performance with available hardware resources.

Together, the local and distributed implementations show the desirability of skipping component traversal both from a single user performance perspective and from a system scalability standpoint.

To summarize, the key contributions of this thesis are:

1. Propose an approach to enable direct lookup operations within a hierarchical namespace.

2. Provide empirical data about previously unstudied characteristics of hierarchical file system operations.

3. Analyze scaling and performance implications of using the direct lookup technique in a POSIX environment by means of file system implementations for local and distributed scenarios.

Bibliography

[AADAD09] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Generating realistic *impressions* for file-system benchmarking. *ACM Transactions on Storage*, 2009. 57

[ABCd96] Virgilio Almeida, Azer Bestavros, Mark Crovella, and Adriana deOliveira. Characterizing reference locality in the www. Technical report, Boston University, Boston, MA, USA, 1996. 2, 57

[ABDL07] N. Agrawal, W.J. Bolosky, J.R. Douceur, and J.R. Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage*, 2007. 2

[ADD+08] Nawab Ali, Ananth Devulapalli, Dennis Dalessandro, Pete 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. 79, 126

[ADF+03] Alain Azagury, Vladimir Dreizin, Michael Factor, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, Julian Satran, Ami Tavory, and Lena Yerushalmi. Towards an object stor. In

Proceedings of the 20th IEEE Conference on Mass Storage Systems and Technologies (MSST), page 165, 2003. 127

[APG11] Ana Aviles-González, Juan Piernas, and Pilar González-Férez. A metadata cluster based on OSD+ devices. In *23rd International Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2011, Vitória, Espírito Santo, Brazil, October 26-29, 2011*, pages 64–71, 2011. 27, 79, 124

[BCF+99] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. *IEEE International Conference on Computer Communications*, 1999. 2, 57

[BEMS07] 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. 125

[BKL+10] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter 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. 128

[BMLX03] Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan 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. 123

[CF96] Peter F. Corbett and Dror G. Feitelson. The vesta parallel file system. *ACM Trans. Comput. Syst.*, 14(3):225–264, 1996. 124

[DDW08] Ananth Devulapalli, Dennis Dalessandro, and Pete Wyckoff. Data structure consistency using atomic operations in storage devices. In *Proceedings of the* 5^{th} *International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI)*, 2008. 81

[FXZ08] Yinjin Fu, Nong Xiao, and Enqiang Zhou. A novel dynamic metadata management scheme for large distributed storage systems. In *Proceedings of the* 10^{th} *IEEE International Conference on High Performance Computing and Communications HPCC*, pages 987–992, 2008. 125

[GK97] G. Ganger and M. F. Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. *USENIX Annual Technical Conference (ATC)*, 1997. 127

[GLK+10] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: An active distributed key-value store. In *Proceedings of the* 9^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 323–336, 2010. 127

[GML98] T. J. Gibson, E. L. Miller, and D. D. E. Long. Long-term file system activity and inter-reference periods. *International Computer Measurement Group Conference*, 1998. 2

[HJZ+09] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian. Smartstore: A new metadata organization paradigm with semantic-awareness for nextgeneration file systems. *High Performance Computing Networking, Storage and Analysis*, 2009. 128

[HZJ+08] Yu Hua, Yifeng Zhu, Hong Jiang, Dan Feng, and Lei 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. 125

[LCB13] Paul Hermann Lensing, Toni Cortes, and André 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. 129

[LHCB16] Paul Hermann Lensing, James Hughes, Toni Cortes, and André Brinkmann. File system scalability with highly decentralized metadata on independent storage devices. In *Proceedings of the 16th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, 2016. 129

[LMB10] Paul Lensing, Dirk Meister, and Andre Brinkmann. hashfs: Applying hashing to optimize file systems for small file reads.

International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI), 2010. 129

[LZ07] Zhong Liu and Xing-Ming Zhou. A metadata management method based on directory path. *Ruan Jian Xue Bao(Journal of Software)*, 18(2):236–245, 2007. 125

[MB11] D.T. Meyer and W.J. Bolosky. A study of practical deduplication. *USENIX Conference on File and Storage Technologies (FAST)*, 2011. 2

[MEK+11] Alberto Miranda, Sascha Effert, Yangwook Kang, Ethan L. Miller, André Brinkmann, and Toni 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. 125

[NEF+12] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen S. Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the* 10th *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–15, 2012. 127

[NFI+08] David Nagle, Michael Factor, Sami Iren, Dalit Naor, Erik Riedel, Ohad Rodeh, and Julian Satran. The ANSI T10 object-based storage standard and current implementations. *IBM Journal of Research and Development*, 52(4-5):401–412, 2008. 127

[PCG02] J. Piernas, T. Cortes, and J.M. García. Dualfs: a new journaling file system without meta-data duplication. *International Conference on Supercomputing*, 2002. 128

[RLA00] Drew S. Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 41–54, 2000. 2

[RO92] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 1992. 128

[Sat81] M. Satyanarayanan. A study of file sizes and functional lifetimes. *ACM Symposium on Operating Systems Principles (SOSP)*, 1981. 2

[Sch03] Philip Schwan. Lustre: Building a file system for 1,000-node clusters. In *Proceedings of the Linux Symposium*, page 9, 2003. 126

[SM98] Jose Renato Santos and Richard R. Muntz. Performance analysis of the RIO multimedia storage system with heterogeneous disk configurations. In *Proceedings of the 6^{th} ACM International Conference on Multimedia*, pages 303–308, 1998. 125

[SM09] M. Seltzer and N. Murphy. Hierarchical file systems are dead. *Workshop on Hot Topics in Operating Systems (HotOS)*, 2009. 128

[WBM+06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the* 7^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 307–320, 2006. 7, 126

[WBMM06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos 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. 125

[WFWL09] Juan Wang, Dan Feng, Fang Wang, and Chengtao Lu. MHS: A distributed metadata management strategy. *Journal of Systems and Software*, 82(12):2004–2011, 2009. 125

[WLBM07] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the* 2^{nd} *Parallel Data Storage Workshop (PDSW)*, pages 35–44, 2007. 125, 126

[WUA+08] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A. Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable performance of the panasas parallel file system. In *Proceedings of the* 6^{th} *USENIX Conference on File and Storage Technologies (FAST)*, pages 17–33, 2008. 126

[WXH+04] Feng Wang, Qin Xin, Bo Hong, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Tyce 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. 2

[ZG07] Z. Zhang and K. Ghose. hfs: a hybrid file system prototype for improving small file and metadata performance. *ACM SIGOPS/EuroSys*, 2007. 127

[Zip29] G K Zipf. Relative frequency as a determinant of phonetic change. *Harvard Studies in Classical Philology 15*, 1929. 57

[ZJWX08] Yifeng Zhu, Hong Jiang, Jun Wang, and Feng Xian. HBA: distributed metadata management for large cluster-based storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 19(6):750–763, 2008. 125