Direct Lookup and Hash-Based Metadata Placement for Local File Systems

Paul Hermann Lensing
Barcelona Supercomputing Center
Carrer Jordi Girona 29
Barcelona, Spain
paul.lensing@bsc.es

Toni Cortes
Barcelona Supercomputing Center
Carrer Jordi Girona 29
Universitat Politècnica de Catalunya
Carrer Jordi Girona 31
Barcleona, Spain
toni.cortes@bsc.es

André Brinkmann
Johannes Gutenberg-Universität Mainz
Saarstraße 21
Mainz, Germany
brinkman@uni-mainz.de

ABSTRACT

New challenges to file systems' metadata performance are imposed by the continuously growing number of files existing in file systems. The total amount of metadata can become too big to be cached, potentially leading to multiple storage device accesses for a single metadata lookup operation. This paper takes a look at the limitations of traditional file system designs and discusses an alternative metadata handling approach, using hash-based concepts already established for metadata and data placement in distributed storage systems. Furthermore, a POSIX compliant prototype implementation based on these concepts is introduced and benchmarked. A variety of file system metadata and data operations as well as the influence of different storage technologies are taken into account and performance is compared with traditional file systems.

Categories and Subject Descriptors

D.4.3 [**OPERATING SYSTEMS**]: File Systems Management

General Terms

Design, Performance, Measurement

Keywords

File system design, metadata placement, randomization, hashing, direct lookup, metadata performance

1. INTRODUCTION

Storage capacity of hard drives has increased from 5 MB in the 1980's to today's 4 TB drives: An increase of almost six orders of magnitude. While file system studies [2, 10, 14, 19] show that the average file size also has increased, the increase is much less significant at roughly one order of magnitude.

SYSTOR '13, June 30 - July 02 2013, Haifa, Israel

ACM 978-1-4503-2116-7/13/06 ...$15.00

The number of files and directories existing in file systems, as well as the depth and complexity of the resulting hierarchical namespace, have thus grown continuously over decades and there is no indication of an end to this development.

While file systems have improved drastically in this time frame, some conceptual design decisions have remained constant. The component based lookup process used by today's file systems, for example, has remained conceptually unchanged. In order to read a file, every directory in the path has to be read as well. If the required metadata is not already cached, this leads to I/O for every single path component.

The high locality of accesses in most single user scenarios conceals this conceptual drawback in many cases, 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 such as web-servers [3, 6], the amount of main memory required to cache all in-use metadata can become infeasible and metadata operations can become the major single influence on file system performance [16, 22].

It is our opinion that the traditional lookup approach needs to be re-examined. An alternative approach to component based lookup is the direct lookup approach. Instead of discovering the location of a file's metadata by accessing its directory it relies on computing the location. This can be achieved by defining a number of hash buckets on the storage device and then hashing a file to one of these buckets based on its path. During a lookup operation the hash bucket is read in and the metadata of the file can be obtained from it. Because this process does not rely on the information normally gained through component traversal, no directories have to be accessed during the lookup operation. This advantage grows with the size of the file system, as the number of directories which would otherwise have to be accessed - potentially leading to I/O - increases. On the other hand, additional complexity is introduced to handle hash collisions, to achieve a non-fragmented data layout, to provide POSIX conform access permissions and to handle changes in the file's path due to a directory move operation.

In [13] we experimented with metadata placement and direct

lookup by implementing a limited prototype on top of Ext-2. Due to the conceptual differences of the metadata handling and the correspondingly required changes to file system architecture it couldn't provide much functionality outside of cold-cache lookups (e.g. no caching / links / moves / access restrictions). As results nevertheless looked quite promising we felt a more in-depth look at the approach to be warranted. In order to properly evaluate the performance implications of the proposed approach, a new, POSIX compliant file system was designed and implemented from scratch.

2. FILE SYSTEM DESIGN

As mentioned in the introduction, the main aim of the file system is to achieve direct lookup functionality using a hash based metadata placement strategy. In Linux, the different file system functions are called from the Virtual File System, which uses path components instead of the full path during the lookup process. Therefore, some slight modifications to the name resolution of the VFS Layer are necessary that cause it to use full paths exclusively if the underlying file system requests it (which poses no problem, as the current directory is always known during name resolution).

In the following the 'Direct Lookup File System' (DLFS for short) is described. The source code is available at `www.scalus.eu/projects/dlfs`.

2.1 On-Disk Layout

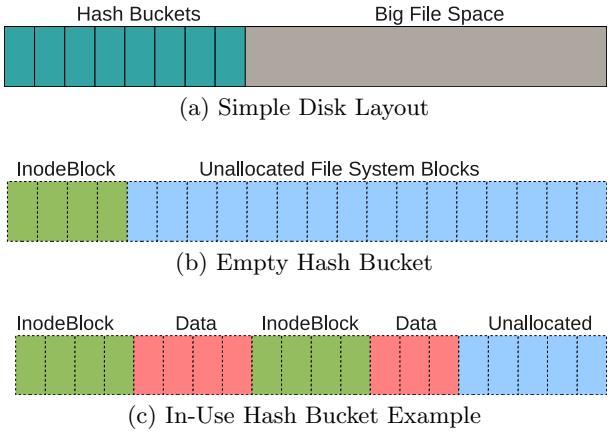
There are two fundamentally different ways the storage area of the underlying block device is used by the DLFS file system, which is reflected in the on-disk layout. *Hash buckets* store metadata and (optionally) small file data while *big file spaces* store the data of big files. This partitioning of the storage space 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. The functionalities required of a layout implementation are:

- unambiguously assign file paths to buckets (and return the bucket starting location)
- arbitrarily assign a big file space to an inode (and return the big file space starting location)

In the scope of this paper, including the evaluation part, we will use the most basic layout possible: a number of sequential hash buckets of homogeneous size followed by a single big file space as shown in Figure 1a. 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).

Inodes in Hash Buckets. 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 2.2.

While a file is assigned to a hash bucket, the actual inode representing the file metadata is stored in a four kilobyte sized *inode block* structure contained in the hash bucket. Figure 1b shows the contents of an example hash bucket containing a single inode block (the diagram shows 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 (all inode blocks allocated for a bucket build a linked list). This can impact performance: If multiple inode blocks have to be read in to find a specific inode, multiple drive accesses are required.



(a) Simple Disk Layout

(b) Empty Hash Bucket

(c) In-Use Hash Bucket Example

Figure 1: On Disk Layout

Data in Hash Buckets. 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 hash bucket, which 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 1c shows an in-use 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. Already allocated data blocks are migrated to a big file space if the corresponding file grows beyond a defined threshold value or when space is required to allocate additional inode blocks. 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.

Data in Big File Spaces. 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 will never contain 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 paper, 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.

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.

2.2 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

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

where n is the number of possible hash values and m the number of inodes in the bucket. The current file system implementation uses 64 bit hash values for the identification hash. Assigning 50 inodes (a single full inode block) to a hash bucket thus leads to at least one hash collision with a probability of 6.6×10^{-17} . It is obvious, however, that considering a single, isolated hash bucket does not make much sense; the whole file system has to be considered. The accumulated probability of at least one hash collision for h hash buckets can be computed as follows.

$$p(h,m,n) = 1 - \left(\prod_{k=1}^{m-1}\left(1 - \frac{k}{n}\right)\right)^{h} \quad (2)$$

Allocating 15 million files on a file system containing three hundred thousand buckets as an example leads to at least one hash collision occurring with a probability of 1.99×10^{-11}. If this probability is too high for a specific scenario, it is straightforward to arbitrarily lower it by increasing the size of the identification hash.

2.3 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 directly use the page cache for metadata caching. 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 only causes a flush of the dirty inode block. If the corresponding inode block were not cached, a read-modify-write cycle would be required. There also is a positive effect for all other metadata operations, because 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 available main memory. In-memory inodes are managed as usual by the Virtual File System; inode numbers are computed based on the location of the on-disk inode (instead of being assigned randomly) to prevent VFS cache inconsistencies.

2.4 Retaining Access Permissions

Traditional file systems implement POSIX compliant access restrictions by evaluating the access permissions for every component of a file path independently. Thus, by the time the permissions of a file are checked, it is already verified that the current user has the required access permissions for every directory in the path. Due to the direct lookup approach, however, another mechanism is required in order to verify access permissions of previous path components during the permission-check of the file inode itself.

For this purpose we introduce the concept of reachability sets. A reachability set contains the set of all access restrictions encountered in the path to an inode. There is a very limited number of different restrictions which can be applied for a single component of a path using POSIX security attributes: A specific user and / or group ID can either be required or excluded. Every inode inherits the reachability set of its parent directory when it is created. Whenever a directory is created or changes access permission the inherited reachability set is extended by the additional restrictions if necessary. Verifying that a user satisfies all entries of a reachability set is equivalent to verifying the access permissions of each directory independently and can be executed during the normal inode permission check.

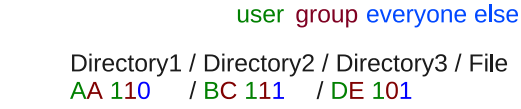


Figure 2: Path Example

Figure 2 shows an example path including the POSIX directory execute permissions specifying the access permissions of the path. The reachability sets for the path components follow:

- Directory1: Only user A and group X have access to any child of this directory. The corresponding reachability set thus contains the restriction: user A $\vee$ group X
- Directory2: As everyone is allowed access, no new restrictions apply. The reachability set can be directly inherited.
- Directory3: Group Z is excluded from access. The new reachability set for all children of Directory3 thus is: (user A $\vee$ group X) $\wedge \neg$ group Z

A single reachability set can be shared by many inodes. In the trivial case, only a single set exists in the whole

	File System A	File System B
files	10,770,676	6,419,483
directories	1,271,791	618,686
users-d	452	310
groups-d	115	84
reachability sets	638	391
reachability entries	821	489

Table 1: Analyzed File System Characteristics

file system. To take advantage of shared reachability sets, multiple sets are stored in a reachability block structure instead of storing them independently for every inode. Each inode stores a reference to its reachability block and an index to its reachability set.

A problem would occur if a very large number of reachability sets exist in a file system: If the reachability block containing an inode's reachability set is not already cached, an access to the storage device is necessary in order to check permissions. In order to examine the number of reachability sets which can be expected in a file system, two multi-user file systems at the Universitat Politècnica de Catalunya have been analyzed. The number of users and groups given in Table 1 refer to unique users / groups encountered for directories in the two file systems. As explained previously, file access rights are irrelevant with respect to reachability sets. It is noticeable that the average size of the reachability sets required to express all existing access restrictions is very small: 1.29 entries per reachability set for file system A and 1.25 for file system B. Additionally, the total number of reachability sets is quite limited. All reachability sets of file system A could be stored in three pages (the size of a single reachability entry is 10 bytes), and the ones of file system B in two pages. Caching such a small amount of data is no problem, even for a theoretical file system containing two orders of magnitude more reachability sets than the observed systems. It can therefore safely be assumed, that the required reachability check does not lead to an additional access to the storage device.

2.5 Directories

Hash based metadata placement and direct lookup change how metadata functionality is affected by directories. 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. As a consequence, however, 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. It is, however, required in order to add or remove the file name to the directory (and ensure that the directory actually exists), as well as inherit the correct reachability set in the case of creation. Therefore, on-demand lookup of the directory of a file has to be supported during file creation or deletion.

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. This allows the creation process to be optimized in DLFS: Simply appending the file name to the directory data structure can be done efficiently no matter the directory size, resulting in metadata creation independent of directory size. 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. All directory related overhead for metadata creation and removal in the file system is avoided in this case.

2.5.1 Special Cases

1. *Links to directories* All files contained in the directory subtree of the target directory are accessible through multiple paths. These paths naturally produce different location and identification hashes.

2. *Moving or renaming an existing directory* A rename or move operation changes the path of a directory as well as the path of all files in the directory sub-tree. As metadata placement depends on the path, this implies the move of all corresponding on-disk inodes.

3. *Changing the access permission of an existing directory* When access permissions of a directory change and this change alters the reachability set for the directory's children (which does not have to be the case), the childrens' reachability sets naturally have to reflect this change.

Due to the conceptual clash of these operations with the hash-based placement method (cost propotional to directory size), they need to be supported separately. We keep some additional information for affected paths in a key-value store to solve this problem:

If it is known at lookup-time that */symlink* is an alternative name for */a/b*, the real path can be substituted before the hashes are computed. With this substitution a call to */symlink/file* would be exactly the same to the lookup operation as a call to */a/b/file*. The same kind of path-substitution can be used to handle directory moves: Simply substituting the new pathname with the original pathname before executing the lookup operation will produce the correct behavior when using the new name. To achieve correct behavior for the original name (should not be valid any longer, also pathnames need to be re-usable) the original name should map to some non-existing file-system internal name. As an example the operation *mv a b* would lead to the two mappings: $b \longrightarrow a$ and $a \longrightarrow *a$.

If an existing reachability set changes due to a permission change of a directory, we store a timestamp for that path (using the same notation as above: $dir \longrightarrow timestamp$). If some file in the directory subtree is accessed, for example *dir/a/b/c/file*, the timestamp of when its reachability has last been validated can be compared to the timestamp of the most recent reachability change in its path to decide if it is up-to-date or needs to be re-validated. If required, the inode's reachability set is validated against its parent's reachbility set (recursively as necessary), propagating the reachability set changes down the path.

Costs & Implications. Additional memory and compute resources are needed to cache and search the key-value store used to keep the required additional information. The current implementation utilizes an uthash[1] hashtable as an

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

in-memory store for this purpose. The computational overhead equals the costs of the hash function used internally by uthash and is independent of the number of stored items. However, when checking whether the hashtable contains information for a specific path, each possible sub-path has to be checked separately (e.g. looking up path */a/b/file* will lead to checks of */a* and */a/b* to ensure that no additional information for directories is required). Therefore, the computational costs increase proportional to number of path components. In absolute numbers, however, the cost of an additional hash operation for each path component during a lookup operation is insignificant and does not impact overall file system performance (it only occurs when looking up a new inode inside DLFS, there is no impact on cache lookups inside the VFS). Required memory largely depends on the length of the strings required to store the paths of the entries. An average size of 150 bytes per entry is realistic, although this number can fluctuate in both directions depending on the actual file set and the predominating type of entries.

As the maximum practical size of the in-memory store is limited, it is prudent to actually apply the move and chmod / chown operations that have been executed using this store to the file system at some point. After the operation has been completely applied (e.g. reachability set of all children are updated) the corresponding entry can be removed. In this way, the total cost of the file system operations is not reduced at all by the approach, but can be deferred arbitrarily. The decision which operations should be applied thus depends on the directory size (a moved directory containing a billion files in its subtree probably should never lead to metadata migration), while the decision about when it should be applied depends on system load.

2.6 Limitations of the Implementation

POSIX Conformity. DLFS passes the POSIX test suite[2] with one notable exception: Since inode numbers are dependent on the location of the inode (see Section 2.3), 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. While the test suite complains about the change of inode number, the POSIX standard does not explicitly forbid it.

Reliability. The established method to provide reliability in local file systems is journaling combined with offline repair (fsck). We feel that this technique is adequately known and has been shown to solve file system reliability in the past and have omitted an implementation for DLFS. While some metadata operations (e.g. renaming directories) can be a much more time consuming in DLFS compared to traditional file systems, they remain completely deterministic and can therefore be journaled like any other metadata operation.

3. EVALUATION

Benchmarking Methodology: Metadata. Frequently used metadata benchmarks such as *metarates*[3] oversimplify accesses by constraining them to a few (or even only one) directory and creating all files used for the benchmark during the actual benchmark run. In contrast, our aim is to analyze more complicated access patterns on pre-existing file sets. The tool *Impressions* [1] 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). We take advantage of this opportunity and modify *metarates* 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

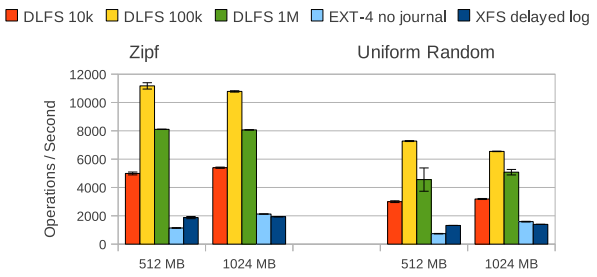
We call our modification *listrates*; along with a simple list-generator, its source code is available at `www.scalus.eu/projects/listrates`. The access patterns used for list generation are based on the Zipf distribution [27], commonly observed in multi user scenarios such as web servers [3, 6], as 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*[4] benchmark tool. We separately consider the performance of very small files in order to analyze the various optimizations of the individual file systems for this case.

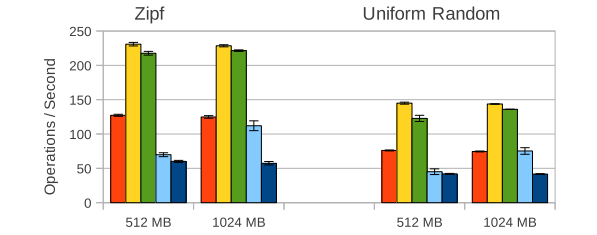
Hardware, Software and Configuration Specifics. One of the most critical hardware components for file system benchmarking is the storage device. The relative performance characteristics exhibited by hard drives (HDD) and solid state drives (SSD) are fundamentally different. While sequential access outperforms 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. We show benchmarks 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 benchmarks to be performed on both drives. 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 at the start of Chapter 2 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

[2] http://www.tuxera.com/community/posix-test-suite/
[3] http://www.cisl.ucar.edu/css/software/metarates/
[4] http://sourceforge.net/projects/ior-sio/

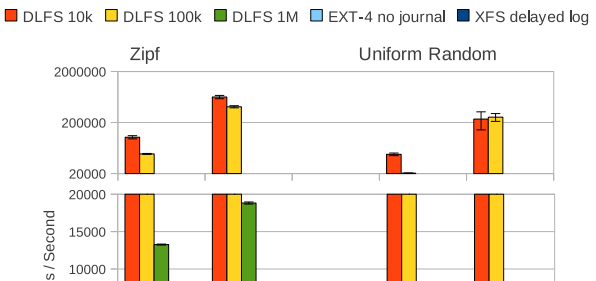


(a) SSD Cold Cache

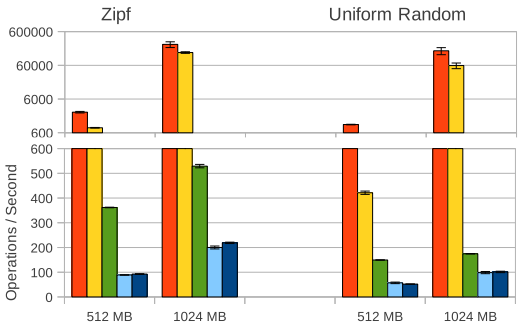


(b) HDD Cold Cache

Figure 3: Stat Rates, Cold Cache



(a) SSD Cold Cache



(b) HDD Cold Cache

Figure 4: Stat Rates, Hot Cache

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 Chapter 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, however, 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 reasonable given an initial space of 50 inodes in each bucket. In addition, we perform benchmarks with 10,000 buckets and 1 million buckets to show the impact of unsuitable layout configurations on file system behavior.

In order to limit the total number of benchmarks to a number that can be discussed in this paper, 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.

3.1 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. We show results 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 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 for other purposes).

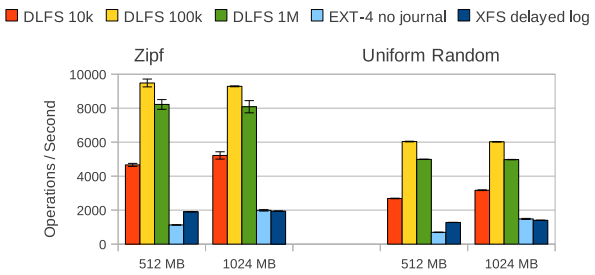
All results were obtained by performing 20,000 accesses using the given access distribution. In the hot cache scenarios, the cache was warmed up prior to the benchmark run by metadata operations of the respective type until no further performance improvement could be observed.

3.1.1 Metadata Access

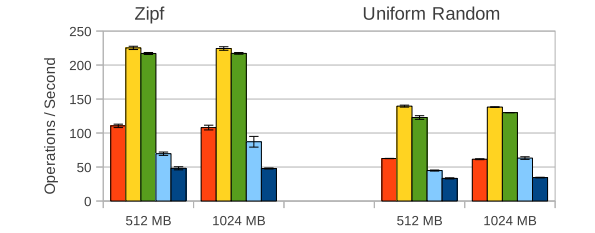
Measured performance is displayed in Figure 3 (cold cache) and Figure 4 (hot cache; note that the broken diagram uses logarithmic scale for the top half in order to display the widely differing results). The main factors influencing file system performance are a combination of cache size, storage technology, and access pattern. In the following, we discuss the influence of these factors independently.

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 2: Number of Inode Blocks after FileSystem creation and additional dynamic Inode Blocks created during allocation.



(a) SSD Cold Cache

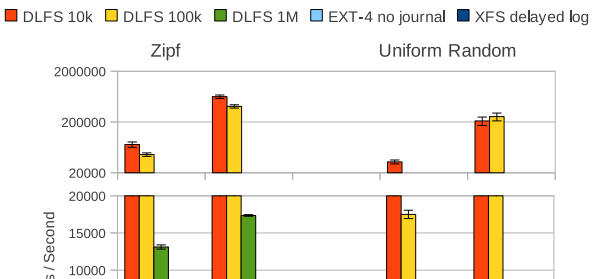


(b) HDD Cold Cache

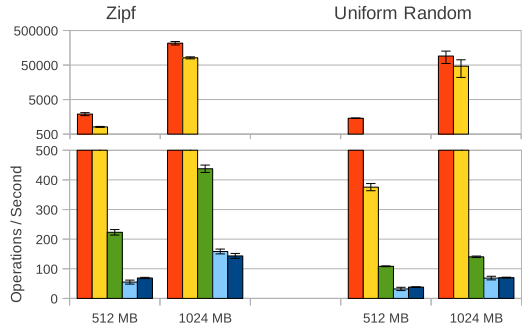
Figure 5: Utime Rates, Cold Cache

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 3), the total amount of encountered metadata for 20,000 accesses is quite limited. 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 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: With both configurations a one-access lookup can be expected, but since the scenario reads 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), 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.



(a) SSD Hot Cache



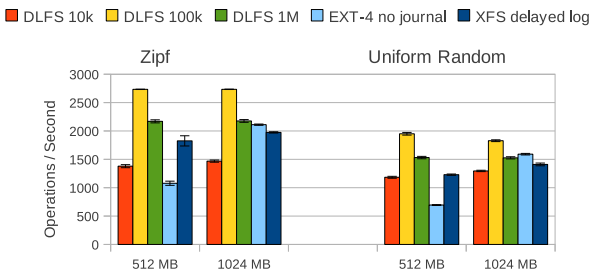
(b) HDD Hot Cache

Figure 6: Utime Rates, Hot Cache

Effect of Storage Technology. The relative performance advantage of DLFS in the cold cache scenario (Figure 3) 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), 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.

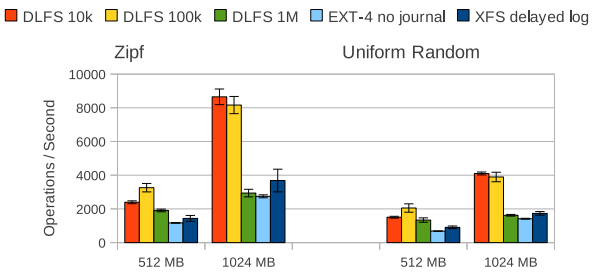
Effect of Access Pattern. Uniform random accesses are slower than Zipf distributed accesses for all file systems. As multiple accesses to the same file will not result in multiple disk accesses, this behavior is intuitive. The relative performance difference is similar across file systems and storage technologies.



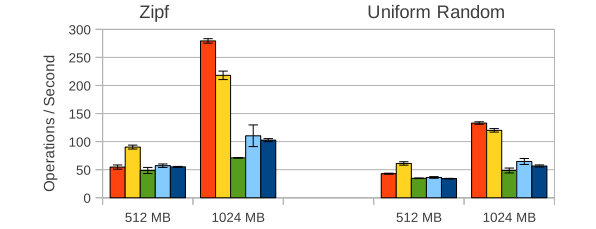
(a) SSD Cold Cache

(b) HDD Cold Cache

Figure 7: Create Rates, Cold Cache



(a) SSD Hot Cache



(b) HDD Hot Cache

Figure 8: Create Rates, Hot Cache

3.1.2 Metadata Update

The observed performance is displayed in Figure 5 (cold cache) and Figure 6 (hot cache). Overall, the results are quite similar to metadata accesses discussed in Section 3.1.1, showing that metadata lookup and not metadata writeback dominates performance. For the lookup part, the same effects for cache sizes, storage technologies and access patterns apply as discussed previously.

Cold cache results are almost identical to metadata access, with absolute performance numbers decreasing only slightly. The updated metadata is, for the biggest part, simply not yet written back to the secondary storage device at the end of the 20,000 metadata updates. The hot cache results, in contrast, show the performance impact of writing back the updated metadata. The expected loss of performance compared to metadata accesses can be observed for all file systems. Generally speaking, the update performance in the hot cache benchmarks lies between 60% and 80% of the metadata access performance.

3.1.3 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).

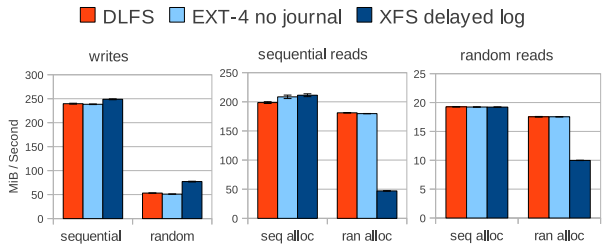
It should be noted that if directories are disabled as described in Section 2.5, metadata creation becomes equivalent to the metadata update operation from a performance point of view. If, however, full directory support is required DLFS loses some of its advantages when creating files instead of only 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 7 and Figure 8 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 7), the relative performance of the different 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.

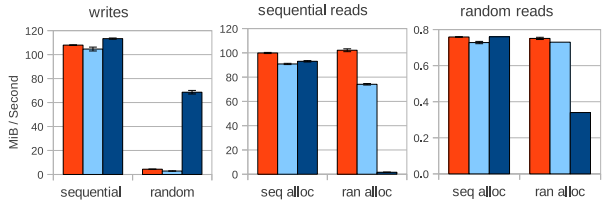
In the hot cache scenario (Figure 8), DLFS performance characteristics differ from previous observations. As directory metadata and data now have to be cached, the previously achieved high percentage of in-memory inode blocks is not reached. The drastic performance gain observed in previous hot cache 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. Apparently, 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.

3.2 Data Performance

Differently from the metadata benchmarks, the number of buckets of the DLFS file system layout cannot influence 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 2.1. In our small file benchmarks, we measure the completion time of a read request sent directly after a small file is opened successfully, in order to obtain small file performance independent of the lookup performance. As just a single hash bucket will be accessed for each measurement, the total number of buckets in the file system is irrelevant.

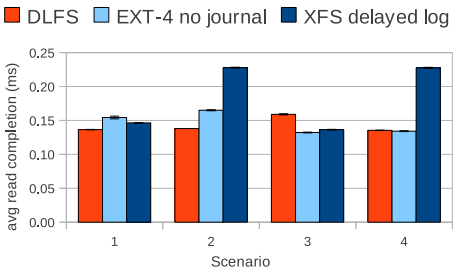


(a) SSD

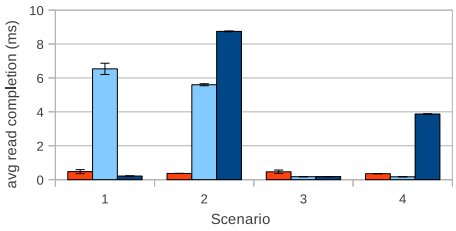


(b) HDD

Figure 9: Big File IOR Performance



(a) SSD



(b) HDD

Figure 10: Small File Results (lower bars are better)

3.2.1 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 9. It is important to note that we differentiate for read accesses between a sequentially or randomly allocated file, 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 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 to the randomly allocated file is almost equivalent to the sequentially allocated case. 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. 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.

3.2.2 Small File Performance

As each file will only be accessed once in this benchmark and cached metadata is not important for performance (measurements start after the inode is read in successfully), 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. 1,000 4 kilobyte files are used in all scenarios, which are accessed sequentially in the order they were allocated. Additional 4 megabyte files were used in some scenarios to show 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.

The results displayed in Figure 10 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 2.1, DLFS can take advantage of hardware readaheads in many cases to speed up file access.

4. RELATED WORK

Related work on metadata placement is primarily done in the context of metadata distribution across multiple metadata servers for parallel file systems. Although this is a higher-level approach, conceptually there are many similarities and a hash bucket of DLFS can be thought of as an extremely small metadata server for many comparison purposes. The *Lazy Hybrid* approach [5], which uses a hashing based metadata placement scheme and supports lookup independent of the directory path, shows the highest correlation with our work. Other work considering direct lookup approaches rely on other mechanisms for metadata lookup, such as bloom filters [26, 12] or additional directory information available in a distributed database [23]. All these approaches use a dual-entry access control list (ACL) structure to resolve access permissions, where the first ACL stores normal file permissions and the second ACL represents the path permissions by storing the intersection of all file permission ACLs of the path. Conceptually, if not from an implementation point of view, this is quite similar to reachability sets. The use of hash functions in file systems is researched more extensively for data distribution purposes in big storage networks [7, 18, 24].

Apart from our own work [13] mentioned in the introduction, no further work on direct lookup and randomized metadata placement in the context of local file systems is known to the authors. There are, however, approaches to optimize the performance of the traditional component-based lookup: The embedded inode technique, for example, stores inodes inside the directory data [9] 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 [25], 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 [9]. Assuming that the clustered files are often accessed together, this can lead high cache-hit rates.

A completely reverse view of metadata performance is taken by log-structured file systems [17, 15]. 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. Another approach to avoid file system related metadata problems is to drastically reduce the number of existing files by storing many (logical) files in one big file system file and move required metadata functionality from the file system into the file itself. Representatives of this approach are Facebook's photo store Haystack [4] and the Hierachical Data Format[5].Finally, there are suggestions for both the distributed [11] and the local case [21] to completely abandon the hierarchical namespace concept in favor of a search based, semantic one.

5. SUMMARY

We introduced a kernel level file system implementing hash-based metadata placement and the direct lookup approach, as well as an extension to the metarates metadata benchmark that can be used with pre-existing file-sets; both of which are available open source. Some unusual file-system internal functionality is required to provide POSIX conform hierarchical access permissions in combination with the direct lookup approach (Section 2.4) as well as providing the possibility of multiple (links) or changing (moves) file paths (Section 2.5.1). The evaluation has shown that the direct lookup approach has a profound impact on file system metadata performance. The different caching requirements (no need to cache directory metadata or data) lead to pure in-memory lookups for DLFS in many hot cache scenarios, while cold cache scenarios showcase the fileset independent one-access lookup property. Even with a highly disadvantageous file system layout, metadata access in the examined scenarios is consistently faster than in compared systems. As has to be expected, data performance is not significantly affected by the different approaches to metadata.

6. ACKNOWLEDGMENTS

This work was partially supported by the Spanish Ministry of Science and Technology under the TIN2012-34557 grant, the Catalan Government under the 2009-SGR-980 grant, and the EU Marie Curie Initial Training Network SCALUS under grant agreement no. 238808.

7. REFERENCES

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

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

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

[3] V. Almeida, A. Bestavros, M. Crovella, and A. deOliveira. Characterizing reference locality in the www. Technical report, Boston University, Boston, MA, USA, 1996.

[4] D. Beaver, S. Kumar, H. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: facebook's photo storage. *Symposium on Operating Systems Design and Implementation*, 2010.

[5] S. A. Brandt, E. L. Miller, D. D. E. Long, and L. Xue. Efficient metadata management in large distributed storage systems. *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies*, 2003.

[6] 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.

[7] A. Brinkmann, S. Effert, F. Meyer auf der Heide, and C. Scheideler. Dynamic and redundant data placement. *IEEE International Conference on Distributed Computing Systems*, 2007.

[8] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, adaptive placement schemes for non-uniform capacities. *ACM Symposium on Parallel Algorithms and Architectures*, 2002.

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

[10] 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.

[11] 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.

[12] Y. Hua, Y. Zhu, H. Jiang, D. Feng, and L. Tian. Scalable and adaptive metadata management in ultra large-scale file systems. *IEEE International Conference on Distributed Computing Systems*, 2008.

[13] P. Lensing, D. Meister, and A. Brinkmann. hashfs: Applying hashing to optimize file systems for small file reads. *International Workshop on Storage Network Architecture and Parallel I/Os*, 2010.

[14] D. Meyer and W. Bolosky. A study of practical deduplication. *USENIX Conference on File and Storage Technologies*, 2011.

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

[16] D. S. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. *USENIX Annual Technical Conference, General Track*, 2000.

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

[18] J. Santos and R. Muntz. Performance analysis of the rio multimedia storage system with heterogeneous disk configurations. *ACM International Conference on Multimedia*, 1998.

[19] M. Satyanarayanan. A study of file sizes and functional lifetimes. *ACM Symposium on Operating Systems Principles*, 1981.

[20] C. Schindelhauer and G. Schomaker. Weighted distributed hash tables. *ACM Symposium on Parallel Algorithms and Architectures*, 2005.

[21] M. Seltzer and N. Murphy. Hierarchical file systems are dead. *Hot Topics in Operating Systems Workshop*, 2009.

[22] F. Wang, Q. Xin, B. Hong, S. A. Brandt, E. L. Miller, D. D. E. Long, and T. T. McLarty. File system workload analysis for large scientific computing applications. *IEEE Conference on Mass Storage Systems and Technologies*, 2004.

[23] J. Wang, D. Feng, F. Wang, and C. Lu. Mhs: A distributed metadata management strategy. *Journal of Systems and Software*, 2009.

[24] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn. Rados: A fast, scalable, and reliable storage service for petabyte-scale storage clusters. *International Petascale Data Storage Workshop*, 2007.

[25] Z. Zhang and K. Ghose. hfs: a hybrid file system prototype for improving small file and metadata performance. *ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[26] Y. Zhu, H. Jiang, J. Wang, and F. Xian. Hba: Distributed metadata management for large cluster-based storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 2008.

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