Efficient, Modular Metadata Management with Loris

Richard van Heuven van Staereling, Raja Appuswamy, David C. van Moolenbroek, Andrew S. Tanenbaum
Department of Computer Science,
Vrije Universiteit,
Amsterdam, Netherlands
{richard, raja, dcvmoole, ast}@cs.vu.nl

***Abstract*—With the amount of data increasing at an alarming rate, domain-specific user-level metadata management systems have emerged in several application areas to compensate for the shortcomings of file systems. Such systems provide domain-specific storage formats for performance-optimized metadata storage, search-based access interfaces for enabling declarative queries, and type-specific indexing structures for performing scalable search over metadata. In this paper, we highlight several issues that plague these user-level systems. We then show how integrating metadata management into the Loris stack solves all these problems by design. In doing so, we show how the Loris stack provides a modular framework for implementing domain-specific solutions by presenting the design of our own Loris-based metadata management system that provides 1) LSM-tree-based metadata storage, 2) an indexing infrastructure that uses LSM-trees for maintaining real-time attribute indices, and 3) scalable metadata querying using an attribute-based query language.**

I. Introduction

For over four decades, file systems have treated files as a set of attributes associated with an opaque sequence of bytes, and have provided a simple hierarchical structure for organizing the files. By providing a thin veneer over devices, and by not imposing any structure on the data they store, file systems have found widespread adoption in many application areas as preferred lightweight data stores. However, this very same generality has also led to the emergence of domain-specific, user-level metadata management systems in each application area to offset several shortcomings of file systems.

In the personal computing front, file systems have been used as document stores for housing a heterogeneous mix of data ranging from small text files to large multimedia files like photos, music and videos. With the amount of data stored by users increasing at an alarming rate, hierarchy-based file access and organization has lost ground to content-based access mechanisms. Most users have resorted to using attribute-based or tag-based naming schemes offered by multimedia and desktop search applications for managing and searching their data. These applications essentially build a user-level metadata management system that crawls the file system periodically to extract metadata, maintains indices on the extracted metadata, and offers application-specific search interfaces to query over metadata.

Modern-day enterprise storage systems house millions of files, and as each file has at least a dozen attributes (POSIX and extended attributes), these systems store an enormous amount of metadata. In addition, storage retention requirements for meeting regulatory compliance standards further fuels metadata growth. Administrators of such systems are constantly faced with the necessity to answer questions about file properties to make policy decisions like "which files can be moved to secondary storage?" or "which are the top *N* largest files?." Answering these questions require searching for relevant attributes over massive amounts of metadata. As using primitive utilities like the UNIX *find* utility at such large scales is not an option, administrators resort to using enterprise search tools. These applications build a user-level metadata management subsystem that gathers metadata periodically, maintains elaborate indices to speed up metadata queries, and offers administrator-friendly search interfaces.

In high-performance scientific computing, local file systems have been used as data stores in local nodes for multi-node, POSIX-compliant cluster/parallel file systems. Similar to enterprise systems, these systems also suffer from problems of scale. In addition, data provenance has emerged to be an extremely important technique in scientific computing for assessing the accuracy and currency of data. Several systems have proposed integrating provenance with parallel file systems to ensure complete, automatic provenance collection [1]. Such systems provide a metadata management system on top of local file systems that collect and store provenance using optimized storage formats, index provenance records, and support specialized query languages for querying provenance data.

The data-intensive scalable computing front has witnessed the growth of domain-specific distributed file systems [2]. These systems maintain separate data and metadata paths so that after a single-step authentication at the metadata server, clients can retrieve data directly from data nodes, thus preventing any single data server from becoming bottlenecked. While such architectures have scaled the data wall, they continue to remain bottlenecked when it comes to metadata scalability. Scaling directory operations to support millions of mutations and lookups per second is an ongoing topic of research. Recent research has shown how indexing algorithms employed by local file systems can have a profound impact on performance of distributed directory partitioning and indexing schemes [3]. Some researchers have even proposed using custom-built databases that use sophisticated indexing structures to optimize metadata storage and retrieval, as storage back ends for metadata servers [4]. These databases act as dedicated metadata management systems that obviate the need for using local file systems to store directory and other file metadata.

Thus, domain-specific metadata management systems have emerged as the "least common denominator" functionality across these application areas. However, such systems suffer from several problems that are well known [5]. First, since they are situated outside the mainline metadata modification path, they do not maintain indices in real time, which can result in stale query results due to inconsistent metadata. While this situation can be averted by updating indices frequently, user-level systems avoid this to reduce the performance impact caused by file system crawling. Unoptimized metadata placement makes crawling for metadata gathering an extremely slow, resource intensive operation. To avoid crawling the entire file system, some user-level systems [6] leverage new file system functionality like snapshotting and perform an incremental scan of only data modified since the last snapshot. While this in combination with other notification-based techniques can reduce the performance impact of crawling, it hardly helps to remedy the storage inefficiency inherent to user-level metadata systems. This inefficiency arises because metadata is stored twice, once in the file system itself, and a secondary copy in the elaborate indices maintained by user-level metadata systems. As metadata can consume a significant percentage of the storage capacity in large installations [5], this metadata duplication results in inefficiency that is unwarranted since the duplicated metadata is always inconsistent.

Thus, once the purview of local file systems, metadata management is now being performed by domain-specific, user-level systems that suffer from several shortcomings. Rather than building custom databases for storing metadata, we believe the the right solution to these problems is integrating support for metadata management into local file systems. First, since file systems are in the mainline path of all metadata operations, no separate polling or notification mechanism is needed for collecting changes. Second, since file systems are in charge of storing metadata, they can employ sophisticated, search-friendly storage formats to optimize metadata performance. Third, file systems can unify metadata storage and indexing using a single storage format, thus avoiding unnecessary duplication. Fourth, with metadata management being the least common denominator functionality, it is obvious that an integrated system can be used as a customizable framework for deploying domain-specific solutions. Such a customizable framework should possess two salient properties. First, it should be efficient; the integration of metadata management should not cause performance deterioration. Second, it should be modular; the metadata management functionality should be independent of other functionalities. Unfortunately, traditional file systems lack the latter property.

Traditional file systems use customized data structures for storing metadata and such data structures form an integral part of the file system's on-disk layout. Further, file systems handle a range of tasks from providing device-specific layout algorithms to implementing POSIX-style file and directory naming. As a result, any metadata management system that is integrated with one file system is inherently non-portable to other file systems, and a single implementation of metadata management cannot be used across multiple file systems. Thus, metadata management would have to be implemented on a case-by-case basis due to the lack of modularity of traditional file systems.

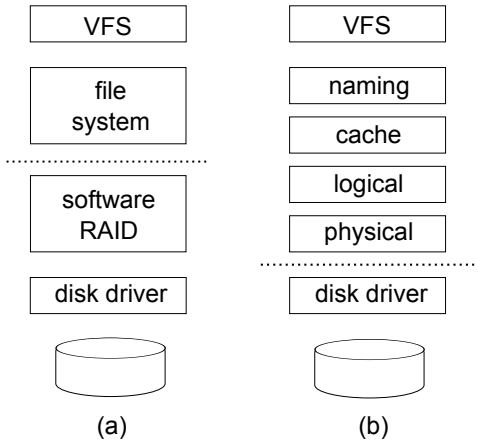

Fig. 1: The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file aware; the layers below are not.

In prior work [7], we presented Loris, a complete redesign of the traditional storage stack that solves several reliability, flexibility, and heterogeneity issues by design. In this paper, we show how Loris can be used as a efficient, modular metadata management framework. In doing so, we present our Loris-based metadata management system that provides 1) LSM-tree-based metadata storage, 2) an indexing infrastructure that uses LSM-trees for maintaining real-time attribute indices, and 3) scalable metadata querying using an attribute-based query language.

The rest of the paper is organized as follows. In Section 2, we present an overview of Loris and show how the Loris stack provides a convenient framework for integrating metadata management. We present the design of our metadata management system in detail in Section 3, following which we present an evaluation of our prototype using several micro and macrobenchmarks in Section 4. We then compare our approach with related work in Section 5. Finally, we present future work in Section 6, and conclude in Section 7.

II. Background: The Loris Storage Stack

In prior work [8], we highlighted a number of fundamental reliability, flexibility, and heterogeneity problems that plague the traditional storage stack and presented Loris, a fresh redesign of the stack, showing how the right division of labor among layers solves all problems by design [7]. In this section, we will briefly describe Loris' layered architecture.

Loris is made up of four layers, as shown in Figure 1. All the layers in Loris are file-aware, in contrast to the traditional stack, and the interface between the layers is file-centric, consisting of operations such as *create*, *delete*, *read*, *write*, and *truncate*. Files are identified throughout the stack using a unique *file identifier* (file ID). Each Loris file is also associated with several *attributes*, and the interface supports two attribute

manipulation operations—*getattribute* and *setattribute*. We will now briefly describe the responsibilities of each layer in a bottom-up fashion.

A. Physical layer

The physical layer exports a "physical file" abstraction to the logical layer. The logical layer stores data from both end-user applications, and other Loris layers in physical files. The physical layer is tasked with two primary responsibilities. The first responsibility of the physical layer is providing persistent storage of files and their attributes using device-specific layout schemes. Each storage device is managed by a separate instance of the physical layer, and we call each instance a *physical module*. Each device, and hence its physical module, is uniquely identified using a *physical module identifier*. Our prototype physical layer was based on the traditional UNIX layout scheme. The physical file abstraction is realized using inodes. The logical layer uses an inode number to refer to the target physical file in any file operation.

The second responsibility of this layer is providing data verification. Each physical layer implementation must support some comprehensive corruption detection technique, and use it to verify both data and metadata. As the physical layer is the lowest layer in the stack, it verifies requests from both applications and other Loris layers alike, thereby acting as a single point of data verification. Our prototype physical layer supports parental checksumming [9] and uses it to provide end-to-end data integrity.

B. Logical layer

The logical layer is responsible for providing per-file RAID services using physical files. The logical layer multiplexes data across multiple physical files and provides a virtualized *logical file* abstraction. A logical file appears to be a single, flat file to the cache layer above it. The logical layer abstracts away details like the physical files that constitute a logical file, the RAID level used by a file, etc. by using the logical file abstraction.

The central data structure in the logical layer that enables multiplexing is the *mapping file*. This file contains an array of *configuration information* entries, one per logical file. Each configuration information contains: (1) the RAID level used for this file, (2) the stripe size if applicable and (3) <physical module identifier, inode number> pairs that identify the physical files that make up this logical file. Since the mapping file is an extremely crucial piece of metadata, it is mirrored on all physical modules. A physical file with a fixed inode number is reserved in each physical module and is used to store the mapping file's data blocks.

Let us consider a logical file with file ID F1, that is stored using a RAID 0 configuration backed by physical files with inodes I1, I2 on physical modules P1, P2 respectively. Such a file would have F1=<raidlevel=0, stripesize=4096, <PF1=<P1:I1>, PF2=<P2:I2>> as its configuration information in its mapping entry. We will explain the entry creation process later while describing the naming layer. For now, let us consider a read request for this logical file. When the logical layer receives a request to read, say, 8192 bytes, from offset 0, it determines that the logical byte ranges 0-4095 maps onto the byte range 0-4095 in physical file PF1 and the logical byte range 4095-8191 maps onto the range 0-4095 in physical file PF2. Having determined this, the logical layer forwards a request to read 4096 bytes, at offset 0, from files PF1 and PF2 to physical modules P1 and P2 respectively.

C. Cache layer

The cache layer provides data caching on a per-file basis. As it is file aware, it can deploy different data staging and eviction policies for different files depending on their types and access patterns. Our prototype cache layer provides a simple fixed-block read ahead and LRU-based eviction for all files.

D. Naming layer

The naming layer acts as the interface layer. Our original prototype naming layer provided POSIX-style file/directory naming and attribute handling. The naming layer is also responsible for assigning a unique file ID to each Loris file. It processes a file create request by picking a unique file ID and forwarding the create call to the cache layer, which, in turn, forwards it to the logical layer. The logical layer picks physical modules for this logical file, and forwards a create call to those modules. The physical modules service the create call by allocating physical files and returning back their inode numbers. The logical layer records the <physical module identifier, inode number> pairs, in addition to other details, in the mapping file's configuration entry.

Directories are implemented as files containing a list of records that map file names to Loris file IDs. Only the naming layer is aware of this structure; below the naming layer, a directory is simply considered another opaque file. Loris attributes are used by the naming layer to store per-file POSIX attributes like modification time and access permissions. These attributes are passed from the naming layer to corresponding physical modules using the setattribute call. Our physical layer implementation stores attributes in the corresponding physical file's inode. The getattribute call is used by the naming layer to retrieve the stored attributes when necessary. Loris attributes are also used to exchange policy information between layers. An example of this usage is how we enable selective mirroring of directories on all physical modules to improve reliability and availability. When the naming layer issues a create call for creating a directory, it informs the logical layer that the corresponding logical file must be mirrored by passing the RAID level (RAID 1 on all physical modules) as an attribute. This attribute is not passed down for normal files. Thus, the attribute infrastructure in Loris serves dual purpose.

III. Efficient, Modular Metadata management with Loris

With the modular division of labor between layers in the Loris stack, the naming layer in Loris provides an ideal place for integrating metadata management. Since all layers in the

Loris stack are file aware, the cache, logical and physical layers can be conceptually seen as providing a file store for the naming layer. The naming layer could thus implement custom storage schemes that pack domain-specific metadata into performance-optimized file formats that would be stored by the lower layers as plain Loris files. By isolating metadata management in the naming layer, Loris provides a modular framework where naming implementations can be changed without affecting algorithms in other layers.

In this section, we will detail the design of our new naming layer that provides metadata management. It is made up of two sublayers, namely, the *storage management sublayer* and the *interface management sublayer*.

A. Storage management sublayer

The storage management sublayer is the lower layer and is responsible for providing domain-specific storage formats that optimize storage and retrieval of metadata. It provides a simple key-value interface to the interface management sublayer, and performs space-efficient packing of such key-value pairs in Loris files.

Our storage management sublayer uses write-optimized log-structured merge (LSM) trees [10] for storing key-value pairs. LSM-trees are multi-component data structures that consist of a number of in-memory and on-disk tree-like components. The fundamental idea behind maintaining two different types of components is to buffer updates temporarily in the in-memory component and periodically flush out batched updates as new on-disk components. These on-disk components are write-optimized tree structures that provide space-efficient packing of key-value pairs by filling up tree nodes completely. They are immutable, and thus, once created, they can either be deleted as a whole, or be used for key lookups, but can never be updated in place. By buffering updates in memory and writing them out in batches to a new on-disk component, LSM-trees avoid directly updating on-disk components, and thus eliminate expensive seek operations.

A lookup operation in an LSM-tree first checks the in-memory tree for the target key. A failure to locate the key results in searching the on-disk components chronologically. By using components that are tree structured, LSM-trees provide efficient indexing of data in both in-memory and on-disk components. However, the number of on-disk components that must be searched plays a crucial role in determining the overhead of lookup operations. Minimizing this overhead requires periodic merging of on-disk components to form a single densely-packed index. Because on-disk components are immutable, such a merge operation can happen asynchronously, in the context of a background thread without affecting foreground traffic.

There are two special boundary cases that arise when one uses an LSM-tree-based key-value store. The first one concerns updates to existing records. Updating an existing record is performed by adding a new record to the tree with the same key and the updated value. Lookups are executed by chronologically searching all components and returning as soon as there is a match, so new records implicitly obsolete any other records that exist in the tree with the same key. The second boundary case concerns record deletion. Delete operations are performed by inserting "tombstone" markers—records whose value denotes that the key-value pair has been deleted. If a key lookup operations ends up at a tombstone record, the lookup fails with an error that notifies the caller that the record being looked up has been deleted. In both boundary cases, the old, outdated records consume unnecessary space and are cleaned periodically during merging to improve space utilization.

We will explain how the LSM-tree-based key-value store is used to house POSIX metadata in Sec. III-B, but for now, we would like to emphasize that LSM-trees are ideally suited for storing metadata due to several reasons. First, metadata updates are rarely sequential. Most modern file systems use B-tree variants for storing metadata. It is well known that B-trees (and their variants) require random writes for random updates and may get fragmented over time [4]. Further, almost all existing storage technologies are ill suited for such random write workloads. Despite tremendous growth in capacity and bandwidth of disk drives the performance of small, random workloads continues to suffer from seek-imposed access latencies. RAID installations using parity-based redundancy schemes have known issues with small, random-write workloads [11]. Even modern SSDs suffer under a random-write workload and it has been shown that random writes can significantly reduce both the performance and lifetime of SSDs [12]. By using write-optimized LSM-trees, we convert slow, small, random metadata updates into fast, large, sequential write operations without sacrificing lookup performance.

Second, metadata lookups exhibit significant locality. A directory listing operation for instance looks up POSIX attributes of all files in a directory. A backup application might scan through extended attributes of each file in the file system to identify flagged files that must be incrementally backed up. Most file systems fail to exploit such locality as metadata is scattered all over the device. For instance, while file names are stored in directory data blocks, POSIX attributes are stored in inodes, and extended attributes are stored in blocks pointed to by inodes. As a result, metadata lookups result in expensive seek operations when disk drives are used, significantly crippling performance. In contrast, with LSM-trees, locality can be controlled with the choice of key format. As records in the leaves of both the in-memory tree and on-disk trees are sorted in key-order, iterating over records with the same key prefix is very efficient as they are more likely to be stored together on disk, and thus more likely to reside in the (block-level) cache. We will see later how we take advantage of this to achieve good directory listing performance.

Third, a significant number of files in several workloads exhibit very short lifetime. For instance, it has been shown that about 50% of files are deleted within 5 minutes, with 20% existing for less than half a minute in certain local file system workloads [13]. Development workloads also create

a large number of empty lock files, and short-lived compiler temporaries. As we will see later, supporting native searching in our naming layer requires the capability to create a large number of links to existing files, which maybe be temporary when queries are used for a one-time, dynamic search over metadata. Traditional file systems typically require additional implementation tricks to optimize for such short-lived metadata. As metadata updates are first handled in memory, LSM-trees handle such temporary files efficiently.

Our implementation employs AVL-trees for the in-memory component (but any search-efficient data structure will suffice), and densely-packed, two-level B^+-trees for storing on-disk components. As the on-disk components are immutable, nodes are packed full for optimal space efficiency. To limit the number of disk seeks to one per disk component, their root nodes are always kept in memory. Merge parameters like component size thresholds, maximum number of disk components, etc. are configurable to allow system-specific optimization.

In addition to the record-based lookup/insert operations, our LSM-storage sublayer's interface also exposes a *prefix lookup* operation that returns an iterator over all records whose key starts with the supplied prefix. We mentioned earlier that records with the same key prefix are likely to be stored together on disk; enumerating them can therefore be performed efficiently. Thus, while the choice of key format is used to control locality, the prefix lookup can be used as a means to exploit it.

B. Interface management sublayer

The interface management sublayer is responsible for translating domain-specific interface operations to key-value insertion or lookup operations on the underlying storage management sublayer. We will first explain how this sublayer maps well-known POSIX abstractions to key-value pairs, thereby providing POSIX-compatible naming. Then, we will describe extensions that provide scalable attribute-based metadata search using LSM-tree-based attribute indices.

1) POSIX interface: The interface management subsystem provides the POSIX environment to applications by mapping familiar file system primitives to key-value pairs. Thus, while the storage management subsystem stores key-value pairs, the interface subsystem controls the semantics of keys and values. We will now describe how we map per-file POSIX attributes and directories to key-value pairs.

As we mentioned earlier, each file in Loris is identified using a unique file ID. As each file in POSIX is associated with a set of POSIX attributes, a straightforward way to map these attributes would be to use the unique file ID as the key, and store all attributes as the value. It is important to note here that only file attributes are stored by the storage subsystem, not file data. Thus, a file create request would result in the interface management subsystem storing a new <file ID, <POSIX attributes>> pair in the LSM-tree following which, the create call would be forwarded to the lower layers. While subsequent metadata updates would result in the LSM-tree being updated, data updates would be immediately directed to the lower layers.

Supporting directories is a bit tricky. With our old naming layer, directories were files containing an array of <name, file ID> pairs, one per file stored in that directory. Since each directory is also a file, it also has its own unique file ID. Thus, one possible implementation would be to use the directory's file ID as the key and store this array of entries as the value. However, directories also have POSIX attributes associated with them, and hence, each directory could then be represented using two key-value pairs, one containing an array of entries, and the other containing the POSIX attributes of the directory itself. While this approach is simple to implement, it however suffers from the disadvantage that a lookup operation has to perform a linear scan through file names.

Avoiding such a linear scan requires using LSM-trees to index directory entries. Such an index would use file names as keys to lookup file identifiers. However, the key structure of such a tree is incompatible with the key structure for storing file attributes we mentioned earlier. Thus, using this approach requires maintaining two LSM-trees, one mapping file names to file identifiers, and the other mapping file identifiers to file attributes. The level of indirection also means that a directory listing operation would need to perform two lookups, one per tree. While the resulting implementation would be better than existing file systems due to the use of LSM-trees to store POSIX attributes, we wanted to eliminate this additional indirection to maximize performance gains.

Eliminating this level of indirection requires using a single tree that uses a unified key structure. Thus, we adopted an approach similar to the one used by BabuDB [4] for mapping both directories and POSIX attributes to key-value pairs. In this approach, we use the triplet <parent directory's file ID, file name, metadata type> as the key to store POSIX attributes for each file. The reasoning behind using this triplet is to speed up lookup operations. Each lookup operation attempts to resolve a filename in the context of a directory to retrieve the target file's identifier or file attributes. Since each directory in Loris is a file, and hence has a file ID, using the directory's file ID in combination with the file name can be used to search through the index for the relevant file's attributes.

The metadata type field in the key is an optimization to reduce the amount of metadata updates. It essentially classifies metadata into two categories: frequently-updated metadata (like access time and size), and rarely-updated metadata (such as modes, ACLs). By making this classification, updates to the LSM-tree are kept small due to the fact that a change to the frequently-updated metadata does not involve writing out all POSIX metadata. Thus, each file is associated with two key-value pairs, one per metadata type, and both these pairs can be located using the key-prefix <parent directory's file ID, file name>.

With this metadata scheme, we achieve several advantages when compared to our old naming layer. First, by linking files and directories in reverse with the parent's file ID, directories no longer need to store <file name, file ID> pairs in data

blocks. As a result, directories are empty files represented in the LSM-tree using two key-value pairs that record the directory's POSIX attributes. Thus, unlike our old naming layer, directory create requests need not propagate down to the lower layers, thus improving performance, as it eliminates further processing of create requests by these lower layers. For similar reasons, metadata updates are also significantly faster. As the old naming layer used Loris' attribute infrastructure to store POSIX attributes, for each change in any POSIX attribute, the naming layer had to use a setattribute call that percolated down the stack resulting in unnecessary overhead. The new naming layer completely avoids this as metadata updates are restricted to the LSM-tree.

Second, lookup operations are much more scalable as linear lookups are avoided in both in-memory and on-disk trees. Third, by using parent ID as a part of the key, lookups can benefit from significant locality. This is due to the fact that records in the leaf of the tree are sorted based on their parent ID and then by their file name. Thus, all file entries belonging to a directory are tightly packed, thereby speeding up operations like directory listing. Fourth, using LSM-based storage results in random metadata updates being converted into sequential write operations at the storage level, improving performance significantly. Our old naming, in contrast, incurred expensive disk seeks for each metadata update as POSIX attributes were stored in the inodes. Table I illustrates the mapping of metadata to key-value pairs using an example configuration.

Key	Value
<0, /, f> <0, /, r>	atime=2011-01-01 . . . id=1 links=4 mode=drwxr-xr-x . . .
<1, etc, f> <1, etc, r>	atime=2011-01-02 . . . id=5 links=2 mode=drwxr-xr-x . . .
<1, tmp, f> <1, tmp, r>	atime=2011-01-03 . . . id=3 links=2 mode=drwxr-xr-x . . .
<3, prog.c, f> <3, prog.c, r>	atime=2011-01-01 . . . size=2000 id=10 links=1 mode=-rw-r–r– . . .
<3, test.txt, f> <3, test.txt, r>	atime=2011-01-03 . . . size=100 id=13 links=1 mode=-rw——- . . .
<5, passwd, f> <5, passwd, r>	atime=2011-01-02 . . . size=1024 id=20 links=1 mode=-rwx—— . . .

TABLE I: Example mapping of POSIX metadata to key-value records. This example illustrates a small file tree containing the root directory /, the directories /etc and /tmp, and the files /etc/passwd, /tmp/test.txt and /tmp/prog.c. Keys are in the format <parent ID, filename, type>, causing metadata records of files in the same directory to be stored adjacently. Record values contain either frequently-updated metadata when the type value is 'f' and rarely-updated metadata when the type value 'r'. Metadata is shown here in human-readable format but is physically stored in native format using positional notation.

Linking files to their parent however does complicate the implementation of hard links, as a file's metadata is mapped to one unique <parent ID, filename> pair. Hard links require a file to be accessed from multiple names—this requires either storing the metadata redundantly for each name, or letting each name point to one central entry. The first approach will not scale as the number of links to a file increases, while locality advantages are lost in the second approach. In the latter case, this means one extra index traversal for each hard linked file. As hard links are not very common, we believe such an overhead is acceptable, and we do not duplicate metadata.

Our implementation stores the metadata of hard linked files under the different key <hardlink ID, file ID, type> (where hardlink ID is simply a reserved parent ID). Thus, metadata is retrieved in the same way, except by file ID rather than parent ID and filename. The original <parent ID, filename, type> record is still used, but only includes the file ID and a flag denoting that the name represents a hard linked file. As a result, enumerating the names in a directory can still be done with a single prefix lookup, but stat lookups require extra index traversals.

2) Efficient metadata search: Having explained how the conventional POSIX interface can be realized using our infrastructure, we will now describe extensions to the interface management subsystem that enables metadata search. Since metadata is well structured and can be logically considered to be a collection of attribute-value pairs associated with files, most user-level metadata management systems have adopted an attribute-based scheme for naming and searching. We will now show how our interface management sublayer can be extended to support such an attribute-based naming scheme. While we use attribute-based naming as an example, we would like to point out here that the infrastructure is flexible enough to accomodate other naming schemes, like tag- or keyword-based schemes, as well.

a) Real-time indexing: The first requirement for enabling efficient searching is attribute indexing. Consider a name lookup operation. Such an operation can be considered to be a search over POSIX metadata for the name attribute. In the absence of indexing, one would have to resort to a linear lookup over all file names, similar to our earlier prototype. As we mentioned earlier, we solved this one specific problem by indexing file names. However, such an index is not usable for searching over any other attribute. For instance, a search for files with size greater than 1 GB can be done only by performing a linear scan over each file's metadata. Though the locality-friendly, densely-packed LSM-tree design makes high-speed sequential scans possible, this approach does not scale as large installations can have millions of files.

To avoid linear lookup, we index attributes in an auxiliary LSM-tree that reverse maps attribute values to file ID. This tree contains records with the <attribute ID, attribute value, file ID> triplet as the key, and an empty value field. The file ID needs to be part of the key in order to guarantee its uniqueness. Table II shows a sample index tree for the example configuration in Table I.

Choosing which attributes to index is a policy decision that can be tuned for each installation and the ones that are indexed are identified in the tree using the attribute ID. Because LSM-trees are update-optimized, maintaining these indices is cheap. We further minimize their overhead by separating metadata trees and index trees. This allows us to adopt different merge parameters for index and metadata trees. For instance, since metadata search is relatively less frequent compared to index updates, we could trade off query performance for improving

Key	Value
<atime, 2011-01-01, 20>	
<atime, 2011-01-02, 13>	
<atime, 2011-01-03, 10>	
⋮	⋮
<size, 100, 13>	
<size, 1024, 20>	
<size, 2000, 10>	
⋮	⋮

TABLE II: Attribute index belonging to the example in Table I. Only files are indexed. This shows the subset of the index that covers the *atime* and *size* attributes. The value fields are unused.

Key	Value
1	<0, />
3	<1, tmp>
5	<1, etc>
10	<3, prog.c>
13	<3, test.txt>
20	<5, passwd>

TABLE III: Name index belonging to the example in Table I. As an example, the fully qualified path name to file ID 20 is retrieved as follows: first, key "20" is looked up to get parent ID 5 and name passwd. Next, key "5" is looked up to get parent ID 1 and name etc. At this point, the path is etc/passwd. Next, 1 is looked up, and the root is reached. Thus, the full path is /etc/passwd.

indexing performance by maintaining a higher number of on-disk components to store indexing entries, thereby delaying expensive merge operations.

b) Attribute-based search interface: Having described the indexing mechanism, we will now detail attribute-based query processing. Exposing new interfaces and functionalities can generally be done in two ways: 1) by extending APIs and system calls by introducing new function calls that applications can directly invoke, or 2) by overloading the semantics of objects in an existing interface. The latter is preferable for integration and compatibility reasons [14]. In the case of file systems, the POSIX-based interface is ubiquitous, and our goal is to preserve backward compatibility as much as possible. Therefore, we overload POSIX semantics without changing the VFS interface.

In Loris, we generalized the concept of a virtual directory [14] to provide *typed virtual directories*. Just like the Semantic file system [14], directories are considered virtual when their file entries are created on the fly. Each virtual directory is associated with a type that determines the mechanism that populates the file entries. For example, a search virtual directory is a virtual directory whose directory listing implementation provides query resolution. A version virtual directory [15], however, has a directory listing implementation that enumerates all versions belonging to a particular file. In this paper, we will describe only our search virtual directory's mechanism.

In Loris, query results are exposed through search virtual directories that are instantiated dynamically using a *well-formed query term* specified by the user. A query term is a boolean combination of attributes and associated conditions that must be met for a file to be a part of a search virtual directory. Such a query starts with '[' and ends with ']'. Our current prototype supports equality conditions that can be used to perform a point search over specific attributes. For instance performing a directory listing of the search directory with the name "[uid=100]" results in all files owned by the user with uid 100 being listed. Similar to Semantic FS [14], search directories can also be used to map conjunctive queries into tree-structured path names. For example, performing a directory listing of the search directory "[uid=100]/[size>1048576]" results in all files owned by the user with uid 100 and having size over 1 MB being listed.

Our search directory implementation performs query resolution in two steps. In the first step, the attribute index is referenced to determine which file IDs match the given query. For instance, the query "[uid=100]" results in our implementation performing a prefix lookup with the key <METADATA_ID(uid), 100> on the index LSM-tree. Thus, using a single range lookup, we can identify the file IDs of all the files that match a query. If the query is the conjunction of two subqueries, each subquery is performed separately and the intersection of file IDs is used.

In the second step, we need to derive the fully qualified path name of each file armed with the file ID. This is required because our mapping of POSIX attributes to key-value pairs uses <parent directory's file ID, file name, metadata type> as the key. Thus, it is not possible to use just the file ID to retrieve file metadata. To solve this problem, we added a new *name index* that maps file ID to <parent directory's file ID, file name>. Thus, each path can be generated by looking up the parent ID and file name corresponding to each file ID in the result and traversing the parent ID chain all the way to the root directory. Table III shows the name index belonging with the example configuration from Table I and illustrates how it can be used to form a full pathname from a file ID.

We are currently experimenting with different methods for exposing the results of a query. One such method is creating a symbolic link entry for each file, with the link name being the file name, plus a suffix if it is not unique. We currently perform the query on the fly during a lookup request, but a better approach would be to cache the results in a separate LSM-tree. Not only is caching better for performance, it would also simplify the implementation of search directories since a lot of code can be reused.

IV. Evaluation

In this section, we will evaluate several performance aspects of our naming layer. We implemented the new prototype as a part of the Loris stack running on the MINIX 3 multiserver operating system [16]. We will first present our microbenchmark-based evaluation that compares the scalability of our implementation with the original Loris naming layer. Following this, we will present a comparison of metadata storage/retrieval performance the two naming layers using *Postmark* and *Applevel* macrobenchmarks, and a comparison of query performance using native Loris queries vs. using

the *find* utility. Finally, we will present an evaluation of our attribute indexing implementation.

A. Test setup

All tests were conducted on an Intel Core 2 Duo E8600 PC, with 4 GB RAM, and four 500 GB 7200RPM Western Digital Caviar Blue SATA hard disk (WD5000AAKS). We ran all tests on 8 GB test partitions at the beginning of the disks.

B. Microbenchmarks

We used two microbenchmarks to evaluate the update and lookup performance of the LSM-tree-based naming layer. Our first microbenchmark created 100,000 files spread across 4, 10 and 100 directories, following which, our second microbenchmark performed the equivalent of "find | xargs stat" at the root directory. Table IV outlines the running times of these microbenchmarks with the old and the new naming layers. The results show that the old naming layer does not scale as the directory size increases, while the new naming layer's performance remains consistent. Performance gains achieved by the new naming layer under the create benchmark can be attributed primarily to the indexed lookup of file names. The find benchmark on the other hand also benefits from two other factors. First, as our POSIX-mapping associates file metadata directly with file names, we avoid the additional level of indirection inherent to the old naming layer (name to inode number, and inode number to attributes). Second, tight packing of metadata in our on-disk trees resulting in increased cache hits due to the locality inherent in metadata requests.

MINIX 3 does not have file systems that support tree-based directory indexing. As we explained earlier, a significant portion of the performance improvement in the last two microbenchmarks can be attributed to the indexed lookup of file names rather than the write-optimized LSM infrastructure. Since we wanted to isolate the performance gains of using the write-optimized LSM-tree, we built a random metadata update microbenchmark. In this benchmark, we perform 200,000 metadata operations (chmod, chown, utime) on randomly-chosen files in a three-level directory tree of 200,000 files. The number of files per directory was deliberately restricted to 64 (the number of entries per directory block in the old naming layer) to avoid the lookup bottleneck of the old naming layer. Thus, each lookup operation in the old naming layer has to retrieve only one directory block, and perform a linear scan over 64 entries. At such a small scale, linear lookups provide performance comparable to indexed lookups. As shown in table IV, this benchmark shows an improvement of about 34% which can be attributed to the batched flushing of metadata performed by the write-optimized LSM-tree.

Microbenchmark	Loris (MFS)	Loris (new)
create		
tree A (4 dirs × 25,000 files)	153,5 (1,0)	43,58 (0,28)
tree B (10 dirs × 10,000 files)	76,23 (1,0)	45,88 (0,60)
tree C (100 dirs × 1000 files)	47,98 (1,0)	42,76 (0,89)
find and stat		
tree A	79,3 (1,0)	6,86 (0,09)
tree B	27,5 (1,0)	6,00 (0,17)
tree C	9,21 (1,0)	5,71 (0,62)
random update	502,4 (1,0)	330,1 (0,66)

TABLE IV: Wall clock time for several microbenchmarks. All times are in seconds. Table shows both absolute and relative performance numbers, comparing our presented naming layer with our MFS-based naming layer.

Macrobenchmark	Loris (MFS)	Loris (new)
PostMark	744 (1,0)	511 (0,69)
Applevel		
copy	46,4 (1,0)	40,9 (0,88)
build	95,0 (1,0)	92,0 (0,97)
find	22,4 (1,0)	10,7 (0,48)
delete	32,6 (1,0)	29,1 (0,89)

TABLE V: Transaction time for PostMark and wall clock time for Applevel benchmarks. All times are in seconds. Table shows both absolute and relative performance numbers, comparing our new naming layer with the original naming layer.

C. Macrobenchmarks

1) Metadata performance: We used two macrobenchmarks, namely Postmark and Applevel, to evaluate the overall performance of the new naming layer. We configured PostMark to perform 20,000 transactions on 10,000 files, spread over 100 subdirectories, with file sizes ranging from 200 to 400 KB, and read/write granularities of 4 KB. Our application-level benchmark consists of a set of very common file system operations, including copying, compiling, searching, and deleting. The copy-phase involves copying over 75,000 files, including the MINIX 3 source tree. This source tree is compiled in the build-phase. The find-phase traverses the resulting directory tree and stats each file. Finally, the delete-phase removes all files. The results are listed in Table V.

The PostMark numbers show a performance increase of roughly 31%, demonstrating the effects of better metadata management when working with many small files. The application-level benchmark is much more data-oriented than the previous benchmarks. Consequently the results only show a moderate increase in performance, except in the find-component of the benchmark, which is completely metadata-oriented and twice as fast.

2) Query performance: We will now present our evaluation of the metadata search functionality in Loris. We evaluated the total time taken to resolve two typical administrative queries: 1) find all files owned by uid 100 with size > 1 MB, and 2) find all files modified in the last hour. The queries were run over a randomly-generated, three-level hierarchy containing 200,000 files. The query results encompass 1% of the total amount of files.

Query	Indexed search	Find
query 1	0,33 (1,0)	8,24 (25,0)
query 2	0,30 (1,0)	7,90 (26,3)

TABLE VI: Time for attribute-based searches to complete, in seconds, comparing Loris indexed search with the same query done using the Unix *find* utility.

We are currently experimenting with different interface setups and do not yet have a fully working query resolver. Instead, we simulated queries by hardcoding index lookups. Thus, the performance numbers present here reflect only index lookup time and does not include the overhead of other aspects of query processing like parsing for instance. The running times are listed in Table VI. For comparison, we ran the same query with *find*. We see that our indexing scheme achieves excellent performance—both the range scans for getting the matching file IDs and the pathname generation step are performed almost instantly.

D. Attribute indexing overhead

Finally, we reran the previous macrobenchmarks (PostMark, and Applevel without delete) with attribute indexing on for the following POSIX attributes: *size*, *uid*, *gid*, *atime*, *mtime* and *ctime*. Only files are indexed, not directories. We relaxed the merging parameters of the LSM-tree used for indexing purposes and included the running times in Table VII.

The results show that the overhead of indexing hovers between 4–19% for our tests. File creates are the most expensive, as exemplified by the copy-phase of the Applevel benchmark. This is because each new file adds an index entry for each indexed attribute plus an index entry in the name index. We believe this overhead is acceptable as it is possible to perform selective indexing of both attributes and files easily to reduce the overhead.

V. Related Work

In this section, we will present related work and compare our Loris-based metadata management infrastructure with other approaches. We classify related work into three categories based on whether it deals with metadata storage management, metadata interface management, or both.

A. Storage management

Today, most file systems use B-trees or their variants for indexing directory entries. Unfortunately, while these do provide efficient keyed lookup, it is well known that they are slow for high-entropy inserts due to their in-place updating of records which requires one disk seek per tree level in the worst case [4].

Spyglass is a user-level search application that proposed using multi-dimensional indexing structures in combination with hierarchical partitioning to provide scalable metadata search. In effect, Spyglass builds a user-level metadata management subsystem and as a result suffers from problems inherent to such systems. We on the other hand propose a modular integration of such functionalities within the Loris storage stack. We would like to point out here that even though we used LSM-tree based indexing, our framework is flexible enough to support other types of indices. As we will discuss later, we are working on modifying our prototype to support and evaluate different partitioning strategies to provide scalable searching.

Benchmark	Indexing disabled	Indexing enabled
PostMark	511 (1,0)	572,0 (1,12)
Applevel		
copy	40,9 (1,0)	48,7 (1,19)
build	92,0 (1,0)	105,2 (1,14)
find	10,7 (1,0)	11,1 (1,04)

TABLE VII: Attribute indexing overhead measurements. Time for PostMark and Applevel macrobenchmarks to complete, in seconds, with attribute indexing enabled/disabled.

BabuDB is a custom-built database back end intended to be used as metadata server back end for any distributed file system. BabuDB uses LSM-trees for storing file system metadata and their mapping of hierarchical file systems to database records is similar to our approach. They also exploit the snapshotting capabilities of the LSM-tree to support database snapshots. As BabuDB targets only the storage management aspect of metadata management systems, it does not provide real-time attribute indexing, or search-friendly interfaces for querying metadata.

B. Interface management

In order to cope with the hierarchical model's restrictions, file systems have been fitted with hard and symbolic links, allowing files to be referenced from multiple names. Unfortunately, such links are unidirectional. A problem, for example, is that when you delete the symbolic link target, you end up with a dead link. Gifford et al. introduced the concept of semantic file systems, providing associative access to files [14]. Virtual directories are used to list files matching a specified attribute-value pair; file type-specific transducers are used to extract this metadata from file contents. Although the path-based queries are syntactically POSIX-compatible, only conjunctive queries are possible. Others have extended upon these concepts [17] [18] [19].

HAC [20] exposes similar functionality through semantic directories: persistent directories associated with a query that are updated periodically. Unlike SFSs read-only virtual directories, semantic directories are tightly integrated into the hierarchy, and HAC allows adding and removing files from them. HAC has been architected such that it is possible to choose different mechanisms for associative access. For example, rather than the default full-text retrieval scheme, more sophisticated schemes can be used. All these systems focus primarily on the interface specification to enable content-based access and do not consider storage management issues.

C. End-to-end metadata management

Since databases have optimized storage formats for storing structured data, Inversion proposed using a relational database as the core file system structure [21]. By using several PostgreSQL tables to store file metadata and data, Inversion was able to extend transactional semantics of databases to file systems. Further, such a database could also be queried declaratively to search over metadata. However, it has been shown that such a database-based metadata back end suffers from significant performance limitations, making it unsuitable for performance-critical installations [5].

Magellan [5] is an on-disk file system that supports scalable searching of metadata. It integrates the search-optimized data structures used by Spyglass within a file system and provides a custom-built search interface, thus providing an end-to-end metadata management framework. However, the approach of integrating metadata management with on-disk file systems lacks modularity.

VI. Future Work

The new Loris-based metadata management framework opens up several possible avenues for future work. We will outline some possible directions in this section.

A. Partitioning

Spyglass and Magellan showed how partitioning could be used to achieve scalable search performance in large installations. We are working on implementing a simple *file volume-based* partitioning of the LSM-tree to improve scalability. The fundamental idea behind this partitioning strategy is to maintain a set of LSM-trees, both data and index trees, on a per-volume basis. Since indices are separated on a volume basis, and since users search for files in their own volumes most of the time, query evaluation can be sped up considerably as only the target volume's index is used to find matching files. Other partitioning techniques like hierarchical partitioning can also be implemented easily using our infrastructure.

B. Exploiting heterogeneity

As we explained earlier, the Loris stack can use the attribute infrastructure to exchange policy information between layers. For instance, the naming layer uses this infrastructure to inform the logical layer to mirror directories on all local devices. We plan to use the same infrastructure to assign different files to different types of storage devices. For instance, the Loris files used by the LSM-tree that contains key-value entries representing file system metadata can be stored on an SSD while the secondary indices could be stored on disk drives. Such an approach trades off query performance for space-efficient usage of SSD, as the SSD is used only for serving metadata requests.

VII. Conclusion

Application-level metadata management subsystems have evolved as a common solution in several application areas to provide scalable metadata indexing and search functionalities lacking in file systems. In this paper, we showed how Loris acts as a modular framework for integrating efficient metadata management subsystems with the storage stack. We presented the design of our Loris-based metadata subsystem and showed how it provides significant performance speedups for metadata intensive workloads. We also showed how our LSM-tree based indices and attribute-based search interface enables scalable, efficient metadata search.

Acknowledgment

This work has been supported by the European Research Council Advanced Grant 227874.

References

[1] S. Brandt, C. Maltzahn, N. Polyzotis, and W. Tan, "Fusing data management services with file systems," in *Proc. of the 4th Annual Workshop on Petascale Data Storage*. ACM, 2009, pp. 42–46.

[2] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[3] S. Patil and G. Gibson, "Scale and concurrency of giga+: file system directories with millions of files," in *Proc. of the Ninth USENIX Conf. on File and Storage Tech.*, ser. FAST'11, 2011.

[4] J. Stender, B. Kolbeck, M. Ho andgqvist, and F. Hupfeld, "Babudb: fast and efficient file system metadata storage," in *Proc. of the Sixth Intl. Workshop on Storage Network Architecture and Parallel I/Os*, may 2010, pp. 51 –58.

[5] A. Leung, I. Adams, and E. Miller, "Magellan: A searchable metadata architecture for large-scale file systems," University of California, Santa Cruz, Tech. Rep. UCSC-SSRC-09-07, Nov. 2009.

[6] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. Miller, "Spyglass: fast, scalable metadata search for large-scale storage systems," in *Proc. of the 7th USENIX Conf. on File and Storage Technologies*, ser. FAST '09. USENIX Association, Feb. 2009.

[7] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Loris - a dependable, modular file-based storage stack," in *Proc. of the The 16th IEEE Pacific Rim Intl. Symp. on Dependable Computing*, 2010.

[8] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Block-level raid is dead," in *Proc. of the Second USENIX Workshop on Hot topics in Storage and File systems*. USENIX Association, 2010.

[9] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea, "Parity lost and parity regained," in *Proc. of the Sixth USENIX Conf. on File and Storage Tech.* USENIX Association, 2008, pp. 1–15.

[10] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[11] D. Stodolsky, G. Gibson, and M. Holland, "Parity logging overcoming the small write problem in redundant disk arrays," in *Proc. of the 20th annual Intl. Symp. on computer architecture*, 1993, pp. 64–75.

[12] M. Polte, J. Simsa, and G. Gibson, "Enabling enterprise solid state disks performance," in *Proc. of the First Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, 2009.

[13] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A trace-driven analysis of the unix 4.2 bsd file system," in *Proc. of the Tenth ACM Symp. on Oper. Syst. Prin.*, 1985, pp. 15–24.

[14] D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole, Jr., "Semantic file systems," in *Proc. of the 13th ACM Symp. on Oper. Syst. Prin.*, ser. SOSP '91. New York, NY, USA: ACM, 1991, pp. 16–25.

[15] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Flexible, modular file volume virtualization with loris," in *Proc. of the 27th IEEE Symp. on Massive Storage Systems and Technologies*, ser. MSST2011, 2011.

[16] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a highly dependable operating system," in *Proc. of the Sixth European Dependable Computing Conf.* IEEE Computer Society, 2006, pp. 3–12.

[17] S. Bloehdorn, O. Görlitz, S. Schenk, and M. Völkel, "TagFS – tag semantics for hierarchical file systems," in *Proc. of the Sixth Intl. Conf. on Know. Mgmt.*, ser. I-KNOW '06, Graz, Austria, 2006.

[18] S. Sechrest and M. McClennen, "Blending hierarchical and attribute-based file naming," in *Proc. of the 12th Intl. Conf. on Distributed Computing Systems*, ser. ICDCS '92, 1992, pp. 572–580.

[19] H. Tada, O. Honda, and M. Higuchi, "A File Naming Scheme using Hierarchical-Keywords," in *Proc. of the 26th Annual Intl. Computer Software and Applications Conf.*, ser. COMPSAC '02, 2002, pp. 799–804.

[20] B. Gopal and U. Manber, "Integrating content-based access mechanisms with fierarchical file systems," in *Proc. of the 3rd Symp. on Operating Systems Design and Implementation*, ser. OSDI '99, Feb. 1999.

[21] M. Olson, "The design and implementation of the Inversion file system," in *Proc. of the Winter 1993 USENIX Technical Conference*, 1993, pp. 205–217.