Coupling Storage Systems and Self-Describing Data Formats for Global Metadata Management

Michael Kuhn* and Kira Duwe†
Faculty of Computer Science
Otto von Guericke University Magdeburg
Magdeburg, Germany
michael.kuhn@ovgu.de*, kira.duwe@ovgu.de†

***Abstract*—Traditional I/O stacks feature a strict separation of layers, which provides portability benefits but makes it impossible for storage systems to understand the structure of data. Coupling storage systems with self-describing data formats can offer benefits by making the storage system responsible for managing file metadata and allowing it to use structural information for selecting appropriate storage technologies.**

Our proposed storage architecture enables novel data management approaches and has the potential to provide significant performance improvements in the long term. By making use of established self-describing data formats, no modifications are necessary to run existing applications, which helps preserve past investments in software development.

Specifically, we have designed and implemented an HDF5 VOL plugin to map file data and metadata to object and key-value stores, respectively. Evaluations show that our coupled storage system offers competitive performance when compared with the native HDF5 data format. In some cases, performance could even be improved by up to a factor of 100.

***Index Terms*—file system, storage system, self-describing data format, metadata management**

I. Introduction and Motivation

Over the last decades, societies came to rely more than ever on technological progress in information technology. Especially in the area of scientific research, this does enable the possibility to solve increasingly complex problems, which nowadays require the computational power of supercomputers. The rising complexity of the processed problems as well as the growth of computation power leads to rapidly increasing data volumes. The globally produced data volume doubles approximately every two years, leading to an exponential data deluge.

Many HPC applications generate vast amounts of data and write them to parallel distributed file systems where they are kept for further processing. All production-level file systems currently in use offer a POSIX I/O interface that treats file data as an opaque byte stream. As it is not possible to reconstruct the data format from this byte stream without prior knowledge, self-describing data formats such as NetCDF (Network Common Data Format) and ADIOS (Adaptable IO System) are widely used to be able to easily exchange data with other researchers and annotate data with meaningful metadata. For example, the World Data Center for Climate (WDCC) alone holds multiple petabytes of data in such formats and provides access for other researchers. The data structure information is encoded in the files themselves, which makes it possible to group and annotate data. Moreover, data can be accessed and interpreted without having prior knowledge about its structure.

In a typical HPC I/O stack, applications only interface directly with a high-level I/O library such as NetCDF, which depends on HDF5 (Hierarchical Data Format) and so on. The coupling between the different layers is loose and mainly used for performance tuning. However, structural information about the data is lost as it is handed down through the layers.

Supercomputers usually make use of parallel distributed file systems to satisfy the demand for high capacity and throughput. Data and metadata are typically separated in such file systems for performance reasons because data and metadata produce different access patterns and thus require different optimizations. This separation leads to two different types of metadata: *file system metadata* (such as ownership and permissions) is stored on the file system's metadata servers, while *file metadata* (such as the type of physical quantities used in a simulation application) is stored within files on the file system's data servers. File system metadata is typically in the range of 5 % of the overall data volume, which leads to significant amounts of metadata in current multi-petabyte file systems. It is, therefore, necessary to develop efficient and scalable approaches to handle these increasing amounts of metadata.

Figure 1 shows the different kinds of metadata and data present when storing self-describing data in a file system. While the file system is responsible for managing folders and files, self-describing data formats typically offer similar concepts called *groups* and *datasets* in this example. The functionality offered is effectively that of a file system nested within the actual file system.

The goal of this paper is to address the weak treatment of different types of metadata. Opening a file and reading specific portions of data (for example, a single output dimension) first requires accessing the file system metadata to identify the servers holding file data. Afterwards, file metadata is read to determine which portions of the file have to be accessed. Finally, the actual data can be read from the file system. This

This work is partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 417705296. More information about the CoSEMoS (Coupled Storage System for Efficient Management of Self-Describing Data Formats) project can be found at https://cosemos.de.
*Corresponding author

978-1-7281-7624-6/20/$31.00 ©2020 IEEE
DOI 10.1109/CSCI51800.2020.00229

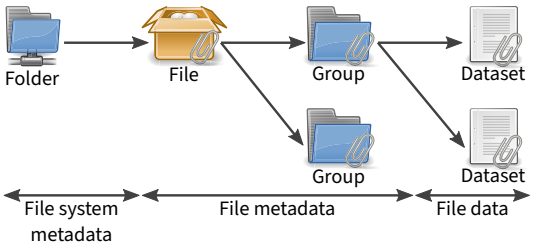

Figure 1. Exemplary self-describing data format stored in a file system. Different kinds of metadata are managed by the file system and the self-describing data format.

strict separation between file system metadata and file metadata leads to inefficient file access. Maintaining metadata requires coordinated access to a shared resource and therefore imposes significant synchronization overhead due to strict semantics mandated by POSIX. Moreover, file metadata is distributed across potentially many different data servers that have to be contacted, causing additional overhead. Data servers are commonly optimized for streaming I/O and therefore cannot deal well with small random accesses that are typical for metadata requests.

II. Global Metadata Management

To address the issues outlined above, we have adapted the I/O stack to handle both file system metadata and file metadata using the storage system's metadata backends, which enables efficient metadata management and improves metadata performance. To this end, the storage system and self-describing data formats have been closely coupled to benefit from the available structural information. This coupling can be used to tune the backends' hardware appropriately for their respective access patterns and to optimize access to both types of metadata using established database technologies. Moreover, this deep integration of metadata enables the storage system to handle different kinds of metadata optimally. For instance, while file system metadata might be best managed by a key-value store, file metadata could be stored in an SQL database. We expect this to have considerable long-term performance benefits as previous studies have shown that metadata performance can be improved significantly using appropriate database techniques [1]. Handling the file metadata within the storage system will be used for novel data management approaches such as a specialized data analysis interface.

Our objective is achievable without exposing additional complexity to the end-user, as will be shown in the following section. Coupling the storage system and self-describing data formats lays the foundation upon which future improvements such as adaptable I/O semantics and an intelligent storage selection can be built. The structural information made available to the storage system by global metadata management will allow intelligent decisions to be taken.

We build on JULEA, a flexible storage framework for HPC that can be used to rapidly prototype new approaches in research and teaching [2]. It provides basic storage building blocks that are powerful yet generic enough to support parallel and distributed storage use-cases. This kind of flexible functionality is achieved by providing well-defined and well-documented plugin interfaces with common requirements in mind. It is possible to run it without administrator or system-level access to enable easy large-scale experiments on supercomputers, where such access is typically not available. Making use of this framework allows us to focus on the objectives defined above instead of having to implement basic storage system functionality. We make use of this functionality to extend a selected self-describing data format to enable global metadata management. A detailed overview of our proposed architecture will be given in the next section.

Our proposed approach couples the storage system and self-describing data formats in a way that allows the storage system to handle both file system metadata as well as file metadata appropriately. The storage system's architecture is shown in Figure 2. Applications utilize a *self-describing data format (SDDF)* API such as HDF5. The modified SDDF library, in turn, transfers the I/O operations to JULEA via several clients that communicate with JULEA's backends to store data and metadata. There is currently support for object and key-value storage with a matching client for each backend type.

JULEA's architecture allows selecting from multiple different backends without requiring changes for applications and libraries. Both the client and backend interfaces have been designed with the requirements of the respective backend types in mind and are abstract enough to support a wide range of backend implementations. For instance, there are object backends for both RADOS and POSIX file systems. Moreover, key-value backends for LevelDB, LMDB, MongoDB and SQLite are available. The different types of metadata are separated and mapped to appropriate backends for maximum performance. In the future, the SDDF implementation will find a fitting storage backend and technology for a given set of requirements. We are also planning to add a data analysis interface (DAI) that will enable comfortable and efficient post-processing by offering a direct interface to query file system metadata as well as file metadata. In contrast to existing analysis tools that have to extract the metadata and store it in a database, this will allow more efficient access and thus reduce overhead as well as redundancies.

The highest level of our storage system are application-facing interfaces. For backward compatibility, we focus on existing SDDF interfaces such as HDF5. These interfaces are widely used in several scientific fields (for example, climate science and high-energy physics) and, therefore, allow us to support a wide range of existing applications. Moreover, HDF5 is used as a basis for other data formats such as NetCDF and NeXus. The SDDF API sits on top of JULEA, offering well-known interfaces for scientific applications. The specific SDDF implementation is responsible for separating the metadata and data, and for making use of the pertinent JULEA backends.

We have designed and implemented a proof-of-concept plugin for HDF5 that allows existing applications to make use

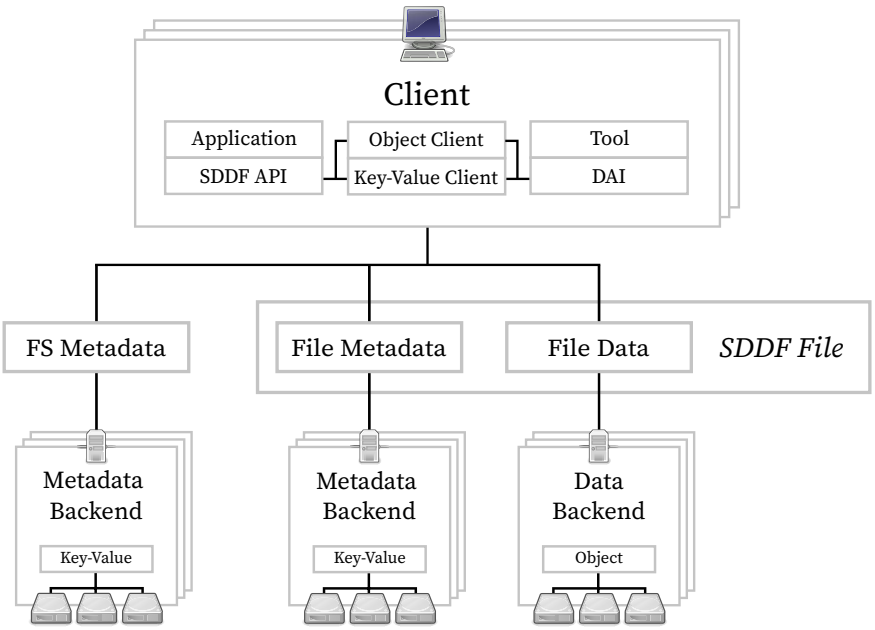

Figure 2. Overview of our proposed architecture. To realize backward compatibility, applications are using unmodified SDDF APIs. The SDDF library itself is adapted to make use of JULEA's clients to handle data and metadata appropriately.

of our storage system. Even though no application changes are required, this allows our storage system to handle file metadata in a way that is not possible with traditional file systems. It is important to note that our storage system approach is not limited to one specific SDDF and we are currently in the process of providing an ADIOS2 engine.

HDF5 comprises a set of file formats and libraries that allow storing and accessing self-describing collections of data and is widely used in scientific applications. It supports several types of data structures, two of the most important ones being *datasets* and *groups*. These two data structures are used analogously to files and directories, that is, datasets are used to store data, while groups are used to structure the namespace. Groups can contain several datasets as well as other groups, leading to a hierarchical layout. Datasets can store multi-dimensional arrays of a given data type, with each dimension being potentially unlimited. These data structures within an HDF5 file are then accessed using POSIX-like paths such as `/ocean/depth`.

Additionally, arbitrary metadata can be attached to groups and datasets in the form of user-defined, named *attributes*. These can be used to store information such as the allowed minimum and maximum values, or the unit of the physical quantities within a dataset together with the actual data. As HDF5 files are self-describing they allow accessing them without any prior knowledge about their structure or content. Moreover, *links* are used to establish relationships between groups and datasets. For instance, one dataset can be linked into several groups and accessed as one of their members.

HDF5 offers the The Virtual Object Layer (VOL) plugin interface that can be used to adapt its I/O behavior by enabling fine-grained control over the management of HDF5's different data structures. Specifically, it is possible to implement functions for creating, opening, reading, writing, deleting and closing attributes, datasets, datatypes, files, groups, links and objects. This allows separating file data and file metadata in a way that structural information can be used by the storage system for intelligent decisions. Additionally, file data and file metadata can be handled appropriately by storing file data in an object store, while file metadata is stored in a database.

Our HDF5 client uses the Virtual Object Layer. By using the VOL, it is possible to gain fine-grained access to file data and file metadata while still conforming to the HDF5 standard. This allows us to keep the native SDDF format. Therefore, this approach is transparent to existing applications because we do not introduce a separate generic storage format for self-describing data. HDF5's VOL allows defining functions for attributes, datasets, datatypes, files, groups, links and objects. For our implementation, we have focused on the most important areas of files, groups, datasets and attributes for now. This allows covering a wide range of HDF5 functionality without having to implement very specific niche features.

Notably, our current design does not consider links, which are normally used to establish relationships between groups and datasets, and allow one dataset to be referenced by multiple groups. Instead, we will assume that each dataset belongs to exactly one group for now. However, this is not an inherent limitation of our design and will be taken care of later. Furthermore, the use of the key-value backend is a first approach to store the file metadata and the file data separately. Our long-term goal is their management using a database backend. A detailed outline is given in Section V.

III. Evaluation

To be able to evaluate the viability of our approach, we have conducted several benchmarks and compared the native HDF5 data format against the one implemented by our HDF5 VOL plugin. More precisely, we have measured the most relevant operations for the data structures presented in the last section.

All benchmarks were executed on compute nodes of the Mistral supercomputer located at the German Climate Computing Center (DKRZ). Each compute node is equipped with two Intel Xeon E5-2680v3 CPUs with 12 cores each (2.5 GHz), 64–128 GB of main memory and connected via a 54 Gbit/s InfiniBand FDR network. Furthermore, all compute nodes are connected to a 54 PiB Lustre file system that has been used for the native HDF5 data format. While the file system offers a maximum throughput of roughly 500 GB/s, single-stream I/O performance is limited to approximately 1.1 GB/s. All measurements were conducted using three different configurations to be able to compare the native HDF5 functionality with our proposed approach:

Native: The native HDF5 data format was used such that the HDF5 library stored file data and file metadata on the Lustre file system. It has to be noted that POSIX file systems such as Lustre make heavy use of Linux's page cache. This means that small write operations are first aggregated in the local cache before being sent to the remote file system servers. Moreover, read operations do not have to communicate with the remote servers at all if they can be satisfied from the cache.

VOL, safe: Our HDF5 VOL plugin was used on top of JULEA using one compute node's local storage, provided by an ext4 file system on top of LVM on a 128 GB SSD. Even though storage is local, the SSD's throughput is considerably lower than that of the Lustre file system. Its sequential write and read performance was measured to be 180 MB/s and 460 MB/s, respectively. Key-value pairs were stored in JULEA's existing LevelDB backend, while objects were stored using the POSIX backend. Both backends were configured to store data in the `/tmp` directory located on the SSD.

VOL, unsafe: This configuration is the same as the second one but without requesting a reply for each network message whenever possible. It is labeled *unsafe* since the success of each operation cannot be checked without a reply. We have included this configuration because even though Lustre does check operations sent to its servers, not every operation is sent to the servers due to cache aggregation. Therefore, it is complicated to compare these different behaviors and we have opted to include both extremes for our VOL approach.

a) Files: Figure 3 shows the results of creating and opening an increasing number of empty HDF5 files. While there are other operations such as deleting, these are not considered in this evaluation.

As can be seen, performance for both operations is relatively stable for the native HDF5 data format, regardless of the number of files. While the performance of the create operations is in the range of 190–220 operations per second, opening files is considerably faster with 710–770 operations per second. However, since empty HDF5 files were created, these performance numbers are likely mainly determined and limited by Lustre's metadata performance. To limit the toll on the Lustre file system, which is a shared resource for all users, we have set a maximum of 20,000 files to be created and opened.

Both VOL configurations store information about HDF5 files in JULEA's key-value store and are thus not limited by the underlying file system's performance. Since key-value stores are optimized for access patterns as encountered during metadata management, performance is improved in comparison to the native HDF5 data format. The safe configuration reaches roughly 10,000–11,000 operations per second for both creating and opening files. Since opening files requires a reply to be sent, the open performance of the unsafe configuration does not differ from the safe configuration. The create performance, however, is drastically improved. To gather meaningful results, up to 1,000,000 files were used for the unsafe configuration, while 100,000 were enough for the safe one.

Being able to rapidly open a large number of HDF5 files can have significant advantages during processing of application results. For instance, different configurations might be compared, requiring many files to be accessed. HPC file systems commonly contain millions of files produced by dozens of different applications. The presented improvements allow inter-application analyses, such as ensemble comparisons found in climate science, to be performed more efficiently.

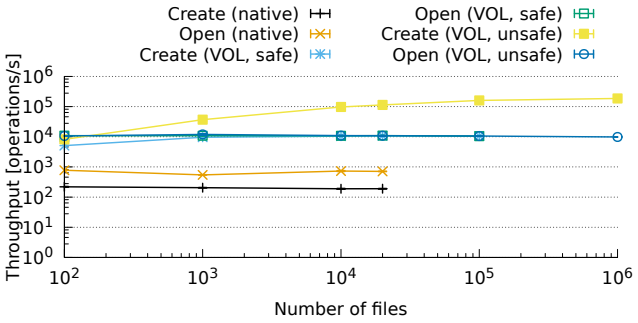

Figure 3. Throughput in operations per second for creating and opening HDF5 files (note the logarithmic axes).

b) Datasets: Since the purpose of datasets is to store data, their main operations are creating and opening, as well as writing and reading. The results for writing and reading datasets are shown in Figure 4. Each dataset had a size of 4 MiB and was accessed using a single write or read call.

When using the native HDF5 data format, writing starts out at 1,000–1,100 MiB/s and drops to 175 MiB/s. As mentioned previously, the file system's single-stream performance is roughly 1.1 GB/s, which HDF5 cannot saturate in this case. When reading, performance starts at 3,100 MiB/s for 100 to 1,000 datasets. Since this number is well above the single-stream performance maximum, Linux's page cache is likely involved. Since 1,000 datasets occupy only 4,000 MiB, they fit into the cache easily. As the total amount of data grows, read performance drops to 40 MiB/s. To keep the load on the shared

file system within reasonable levels, a maximum of 20,000 datasets were measured.

When using the safe configuration, writing starts out at 430 MiB/s and drops to 60 MiB/s. It is important to note that the datasets' objects were stored on the compute node's local storage, which offers significantly lower performance than the Lustre file system. For reading datasets, the performance starts at 600 MiB/s, regardless of the configuration, and then drops to 100 MiB/s. When using the unsafe configuration, the write performance is on par with that of the native HDF5 data format for 100 to 1,000 datasets but drops to 60 MiB/s for more. Due to the low amount of local storage, measurements were limited to 10,000 datasets, which equals 40,000 MiB.

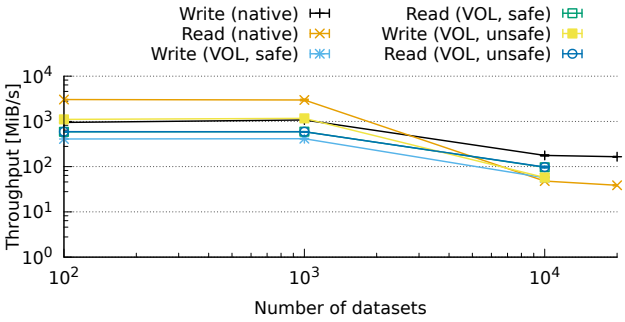

Figure 4. Throughput in MiB/s for writing and reading HDF5 datasets with a size of 4 MiB each (note the logarithmic axes).

c) Attributes: Figure 5 shows the results of writing and reading an increasing number of attributes with a size of 4 KiB within a single file. Since attributes are typically used to carry metadata about a file, group or dataset, and thus are seldom used empty, we have only performed measurements for writing and reading attributes. While writing includes the attribute's creation, reading includes its opening.

As can be seen, performance degrades dramatically for the native HDF5 data format when increasing the number of attributes. While writing starts with roughly 4,000 operations per second, it decreases to only 40. Reading drops from roughly 40,000 operations per second to approximately 3,000. Since such performance degradations do not occur with the other tested HDF5 data structures, the reason has to be an inefficiency specific to attributes. The exact cause is unknown and has to be investigated further. Due to the low performance, measurements were only performed up to a maximum of 20,000.

When using the safe configuration, writing attributes starts at 2,8000 operations per second and steadily increases to 5,000. Therefore, our VOL-based is able to handle even large numbers of attributes. For both the safe and unsafe configuration, read performance stays constant at 3,000 operations per second. When writing attributes using the unsafe configuration, performance starts at 6,000 operations per second, increases to its maximum of 24,000 for 1,000 attributes and then decreases to 6,000. Again, these numbers are significantly lower than for files but can be explained by the increased size of the key-value pairs and the fact that two key-value pairs are managed for each attribute.

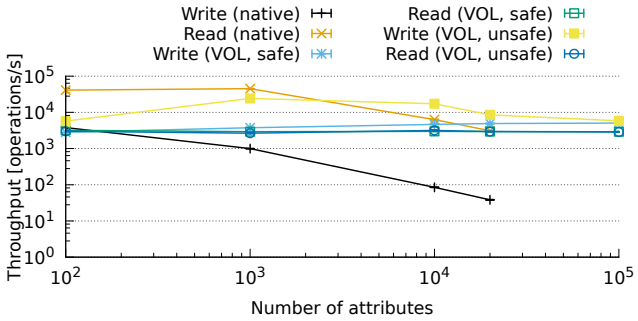

Figure 5. Throughput in operations per second for writing and reading HDF5 attributes with a size of 4 KiB each (note the logarithmic axes).

In conclusion, our VOL-based approach offers competitive performance when compared with the native HDF5 data format. For attributes, we could even improve performance by a factor of 100. However, when writing and reading datasets, further analyses and optimizations are still necessary.

A major improvement of our approach is the fact that all file metadata regarding HDF5's data structures is stored within the underlying key-value store, which can be queried with high performance. Moreover, it is not necessary to search individual HDF5 files for attributes or datasets anymore. Instead, all metadata of all HDF5 files can be queried at once. However, it is currently necessary to use the basic key-value interface provided by JULEA. In the future, a dedicated data analysis interface will provide convenient access to libraries and tools such as the Climate Data Operators.

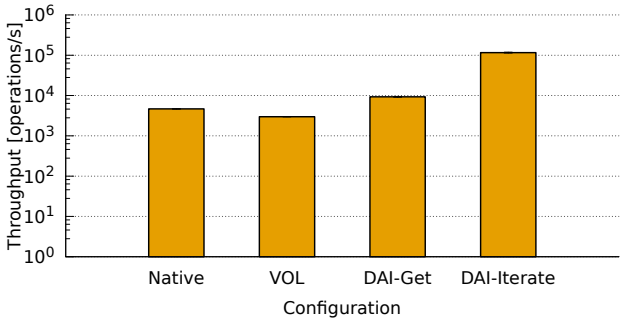

Figure 6. Throughput in operations per second for reading a total of 100,000 HDF5 attributes spread across 10 HDF5 files (note the logarithmic y-axis).

d) Data Analysis Interface: Next, we want to analyze the potential performance benefits of such a data analysis interface. For this measurement, 10 HDF5 files have been created with 10,000 attributes each. Afterwards, all attributes have been read from all files. This represents a typical use case where multiple files have been created by an application and results from multiple runs should be compared or aggregated. Four configurations have been used:

Native: Uses the native HDF5 file format as in the previous tests. File data and file metadata are stored in the Lustre file system. Again, it has to be noted that POSIX file systems make heavy use of Linux's page cache. Due to this, read performance is likely higher than what would be observed in a read-world application since data is usually not read directly after it has been written.

VOL: Uses JULEA's HDF5 VOL plugin, storing file data in the object backend and file metadata in the key-value backend. This provides no performance benefits apart from those enabled by JULEA's backends themselves, since all accesses still have to be performed via HDF5's interface and thus routed through the VOL plugin.

DAI-Get: Accesses the key-value store directly via JULEA's key-value client, with each key-value pair being requested separately. The value, a serialized version of the HDF5 attribute, is deserialized and returned to the application.

DAI-Iterate: Accesses the key-value store directly and uses JULEA's iterator interface to query all key-value pairs at once. Some filtering is performed to identify and process only those values belonging to attributes. Once a matching value is found, it is again deserialized and returned to the application.

Figure 6 shows the results of reading a total of 100,000 attributes spread over 10 files, with each attribute having a size of 4 KiB. Creating the files and writing the attributes has been excluded for this measurement.

As can be seen, performance of the native and VOL configurations is roughly in line with the results obtained previously: While the native HDF5 file format reaches almost 4,700 operations per second, the VOL plugin is slightly slower with almost 3,000 operations. It is important to note that all accesses have to pass the full HDF5 I/O stack for these configurations and therefore have to first open the files and then access the individual attributes. The DAI-Get and DAI-Iterate configurations, on the other hand, access JULEA's key-value store directly and achieve throughputs of almost 9,300 and 120,000 operations per second, respectively. Even accessing each attribute separately using JULEA's key-value get operation already provides twice the performance of the native HDF5 file format. Using the more advanced iterate operation allows achieving a speedup of almost 25. This improvement can be used to enable advanced features or enhance performance. Since all file metadata is stored in JULEA's key-value backends, external tools can always access up-to-date file metadata, requiring no further indexing.

It has to be kept in mind that JULEA's key-value interface provides only basic operations such as put, get and iterate. Specifically, searching for a particular attribute or all attributes matching a certain query requires iterating over all attributes and selecting all appropriate ones. The database client currently in development will be able to provide further benefits by allowing libraries to specify schemas and perform advanced queries akin to relational database management systems.

Moreover, while we chose JULEA's LevelDB key-value backend and POSIX object backend for our evaluation, many other combinations provide interesting opportunities. In particular, it is possible to combine server-side backends with client-side ones, which could be used to build hybrid configurations: While file metadata can be stored on a dedicated node using a key-value backend, a client-side POSIX object backend can make use of existing infrastructure such as an existing parallel file system. Moreover, the RADOS object backend could be used to store file data in the cloud while file metadata is kept in a key-value store for fast access.

IV. Related Work

As fundamental standards such as POSIX as well as MPI-IO pose extreme challenges especially regarding metadata management, numerous approaches to design new systems have been proposed. Also, several extensions and changes to the existing systems have been presented.

A very thorough and promising proposal is the *Fast Forward Storage and IO (FFSIO)* project [3]. Its goal is to provide an exascale storage system able to support both HPC and big data workloads called *DAOS (Distributed Asynchronous Object Storage)*. It is designed for future systems and requires a considerable amount of NVRAM (Non-Volatile RAM) and NVMe (Non-Volatile Memory Express) devices, which will not be suitable for all users such as small research labs due to the high acquisition costs. Evaluations show that the adaptation of existing applications to DAOS is simplified through the HDF5 API [4]. DAOS will also support other common interfaces as POSIX and MPI-IO. However, evaluations show that object stores present better scalability than POSIX with DAOS outperforming Ceph's RADOS and OpenStack's Swift [5].

Another approach to improved metadata management is *EMPRESS (Extensible Metadata PRovider for Extreme-scale Scientific Simulations)*. The user can highlight interesting areas of data before storing through the usage of custom metadata tags [6]. The insights of EMPRESS 1 have been used to develop EMPRESS 2, which includes extensive query functionality, fault tolerance as well as atomic operations [7]. The difference to our work is that we do not require application changes to incorporate new metadata tagging. We use the metadata that is already available inside the self-describing file.

Another object-oriented approach to metadata management is *SoMeta (Scalable Object-centric Metadata Management)*, which organizes the file system metadata in form of objects that are stored in a distributed hash table [8]. Similar to EMPRESS, developers have the possibility of tagging this metadata with additional information. These tags are stored as key-value pairs. SoMeta's metadata search is 16× faster than SciDB [9] and MongoDB. There have also been efforts from the *ADIOS (Adaptable IO System)* developers to enable querying of large scientific datasets [10]. While they also use a database approach, they store the whole self-describing file not only the metadata in the form of the attributes. This severely impacts the performance as the large number of blocks slows down the used indexing mechanisms considerably.

The *DAMASC (Data Management Services for Scientific Computing)* intend to change the file system interface by incorporating database mechanisms and passing the responsibility

for efficient access to the file system [11]. DAMASC was supposed to be integrated into Ceph but has not been realized yet. Another optimization of file system metadata management using a key-value store is *LocoMeta*, which flattens the directory content to improve access patterns and splits metadata into parts to lower write traffic [12].

There are a number of domain- or application-specific solutions such as the *ADDS (Atmospheric Data Discovery System)* [13] or the Sloan Digital Sky Survey (SDSS) *CAS (Catalog Archive System)* [14]. While they offer a variety of metadata queries, they do not make use of file metadata and are restricted to a specific area and thus not portable enough for wide-spread use. *ATLAS* has been developed for the LHC (Large Hadron Collider) [15] but has evolved from a very specific approach to a generic metadata framework [16]. The European Centre for Medium-Range Weather Forecasts (ECMWF) has developed its own storage and archive systems *ECFS (ECMWF File Storage System)* and *MARS (Meteorological Archival and Retrieval System)* [17]. MARS provides a database-like interface allowing customized queries to the object store. While we follow the same idea to manage metadata, our approach will not be domain- or system-specific.

In conclusion, all current solutions only analyze and manage the file system related metadata or offer additional tagging mechanisms but they do not use the file metadata inside data formats. However, file metadata is important both from data management and performance points of view due to self-describing data formats effectively being file systems within a file system. We will unify the handling of all metadata within the storage system and use it for its innovative optimization and management possibilities.

V. Conclusion and Future Work

Our coupled approach enables file system metadata as well as file metadata to be managed by the storage system. We have designed an HDF5 VOL plugin that achieves this goal without requiring application changes, lowering the barrier for adoption. Performance evaluations have shown promising results when compared to the native HDF5 data format. Moreover, managing file metadata within the storage system provides interesting opportunities for performing queries across all available metadata. Based on the promising results, we will further extend the data analysis interface that will allow external applications direct access to the file metadata for increased performance and novel data management opportunities. Applications will be able to specify additional metrics that should be kept in the metadata backends for efficient access later on.

In the future, we are planning to extend global metadata management for additional self-describing data formats and I/O interfaces. In particular, work has started on an ADIOS2 engine. ADIOS2 provides similar functionality to HDF5 and thus is a fitting candidate to further generalize our architecture and interfaces such that support for additional SDDFs can be added more easily in the future. We are also working on a more database-like backend that can be used for more sophisticated queries. This will allow improved performance and more complex requests to be formulated.

Moreover, we are planning to address more problems we have identified in current storage systems. Namely, performance problems caused by static I/O semantics and inefficient data placement can benefit greatly from our architecture. Since file metadata as well as file system metadata are now managed by the storage system itself, more intelligent decisions become possible. For instance, the storage system can make use of information derived from file metadata to place file data on appropriate tiers of a hierarchical storage landscape.

References

[1] K. Ren and G. A. Gibson, "TABLEFS: enhancing metadata efficiency in the local file system," in *USENIX Annual Technical Conference*. USENIX Association, 2013, pp. 145–156.

[2] M. Kuhn, "JULEA: A flexible storage framework for HPC," in *ISC Workshops*, ser. Lecture Notes in Computer Science, vol. 10524. Springer, 2017, pp. 712–723.

[3] J. F. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and friends: a proposal for an exascale storage system," in *SC*. IEEE Computer Society, 2016, pp. 585–596.

[4] M. S. Breitenfeld, N. Fortner, J. Henderson, J. Soumagne, M. Chaarawi, J. Lombardi, and Q. Koziol, "DAOS for extreme-scale systems in scientific applications," *CoRR*, vol. abs/1712.00423, 2017.

[5] J. Liu, Q. Koziol, G. F. Butler, N. Fortner, M. Chaarawi, H. Tang, S. Byna, G. K. Lockwood, R. Cheema, K. A. Kallback-Rose, D. Hazen, and Prabhat, "Evaluation of HPC application I/O on object storage systems," in *PDSW-DISCS@SC*. IEEE, 2018, pp. 24–34.

[6] M. Lawson, C. Ulmer, S. Mukherjee, G. Templet, J. F. Lofstead, S. Levy, P. M. Widener, and T. Kordenbrock, "Empress: extensible metadata provider for extreme-scale scientific simulations," in *PDSW-DISCS@SC*. ACM, 2017, pp. 19–24.

[7] M. Lawson and J. F. Lofstead, "Using a robust metadata management system to accelerate scientific discovery at extreme scales," in *PDSW-DISCS@SC*. IEEE, 2018, pp. 13–23.

[8] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "Someta: Scalable object-centric metadata management for high performance computing," in *CLUSTER*. IEEE Computer Society, 2017, pp. 359–369.

[9] P. G. Brown, "Overview of scidb: large scale array storage, processing and analysis," in *SIGMOD Conference*. ACM, 2010, pp. 963–968.

[10] J. Gu, S. Klasky, N. Podhorszki, J. Qiang, and K. Wu, "Querying large scientific data sets with adaptable IO system ADIOS," in *SCFA*, ser. Lecture Notes in Computer Science, vol. 10776. Springer, 2018, pp. 51–69.

[11] S. Brandt, C. Maltzahn, N. Polyzotis, and W.-C. Tan, "Fusing Data Management Services with File Systems," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, ser. PDSW '09, 2009.

[12] S. Li, F. Liu, J. Shu, Y. Lu, T. Li, and Y. Hu, "A flattened metadata service for distributed file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 12, pp. 2641–2657, 2018.

[13] S. L. Pallickara, S. Pallickara, and M. Zupanski, "Towards efficient data search and subsetting of large-scale atmospheric datasets," *Future Generation Comp. Syst.*, vol. 28, no. 1, pp. 112–118, 2012.

[14] A. R. Thakar, A. Szalay, G. Fekete, and J. Gray, "The catalog archive server database management system," *Computing in Science & Engineering*, 2008.

[15] S. Albrand, T. Doherty, J. Fulachier, and F. Lambert, "The atlas metadata interface," in *Journal of Physics: Conference Series*. IOP Publishing, 2008.

[16] J. Fulachier, O. Aidel, S. Albrand, F. Lambert, A. Collaboration *et al.*, "Looking back on 10 years of the atlas metadata interface. reflections on architecture, code design and development methods," in *Journal of Physics: Conference Series*, 2014.

[17] M. Grawinkel, L. Nagel, M. Mäsker, F. Padua, A. Brinkmann, and L. Sorth, "Analysis of the ECMWF storage landscape," in *FAST*. USENIX Association, 2015, pp. 15–27.