# DelveFS – An event-driven semantic file system for object stores

Marc-André Vef*, Rebecca Steiner*, Reza Salkhordeh*, Jörg Steinkamp*,
Florent Vennetier†, Jean-François Smigielski†, André Brinkmann*
* Johannes Gutenberg University Mainz, Mainz, Germany
† OpenIO, France

***Abstract*—Data-driven applications are becoming increasingly important in numerous industrial and scientific fields, growing the need for scalable data storage, such as object storage. Yet, many data-driven applications cannot use object interfaces directly and often have to rely on third-party file system connectors that support only a basic representation of objects as files in a flat namespace. With sometimes millions of objects per bucket, this simple organization is insufficient for users and applications who are usually only interested in a small subset of objects. These huge buckets are not only lacking basic semantic properties and structure, but they are also challenging to manage from a technical perspective as object store file systems cannot cope with such directory sizes.**

**DelveFS is the first object store file system that solves this challenge by offering the ability to compose a custom semantic file system that allows multiple unique *views* onto the object store. Through flexible filters, users can specify each view's content, tailored to their unique interests or an application's requirements. By processing object store events which describe changes in the object store, DelveFS is able to keep all views eventually consistent. DelveFS allows to operate concurrently through the object and file system interfaces on the same set of objects, delivering similar file system throughput compared to the native object store interfaces or other file system connectors.**

## I. Introduction

Data-driven applications are responsible for a huge fraction of today's innovations in research and industry. Their underlying technologies, such as big data processing or machine learning, depend on the availability of huge data sets. As a result, data is being generated and collected at an increasingly rapid pace. The capacity requirements to store sequenced genomes for the bioinformatics community, for instance, are growing faster than the storage density of hard drives [1], [2]. The European Centre for Medium-Range Weather Forecasts (ECMWF) reached a storage capacity of over 100 petabytes already in 2014 with a 45% annual growth rate [3]. The particle physics community stores hundreds of petabytes of data generated at the Large Hadron Collider (LHC) at CERN [4], while the Square Kilometre Array (SKA) telescope will require more than 100 petabytes of short term buffer storage for a 12-hour observation duration and exabytes of long-term storage [5].

Companies also started to manage their data in *data lakes*, where all information is stored in the raw format [6]. In data lakes, before accessing data, applications need to find the corresponding files among all the files in the lake, adding to their complexity. Object storage with its high scalability, low maintenance overhead, flat namespace hierarchy, and ease of use has made this storage solution a popular option for such scale-out use cases. Today, many commercial and academic object stores like Amazon S3 [7], BlobSeer [8], Ceph [9], Swift [10], OpenIO [11], or Ursa Minor [12] are available, tackling use cases ranging from archiving, media and entertainment, and scientific data to the Internet of Things.

New Internet scale applications like Dropbox or Netflix have been designed to directly use object storage systems through their native object API [13], [14]. Nevertheless, many data-driven applications are unable to use the new I/O interfaces offered by object stores as they rely on a file system which is semantically similar to POSIX. Adapting the I/O layer of these applications is typically very time-consuming and, especially for many proprietary applications, not feasible.

Consequently, several object stores have added a file system interface or rely on third-party file system connectors [15], [16], [17], [18], [19] to represent containers (or buckets) as directories and objects as files. These connectors allow applications to use object stores through a file system interface without needing to be modified. Yet, these file systems only offer a very basic representation of objects as files in a flat namespace. This is especially an issue when millions of objects are stored in single containers [20] because of the complete lack of basic semantic properties and structure.

Although file systems can offer a hierarchical structure to categorize the data, hierarchies are often insufficient when dealing with large data volumes. For that reason, the usefulness of the hierarchical file system model has been questioned several times over the past three decades [21], [22], [23]. Instead, a semantic model was proposed where data is dynamically searchable so that its organization fits the way it is accessed. In this regard, some object stores support search capabilities, e.g., basic filters for object name wildcards [24] or time ranges [25], but also more sophisticated metadata search capabilities [26], [27] that are, however, not available in file system connectors.

In addition to the missing structure and semantics of data, mapping an object store's flat namespace into a file system can also result in severe challenges when a large amount of data is presented [28]. This is because POSIX-like file systems are built for managing a hierarchical structure, utilizing many mechanisms to optimize for this model, such as the *dentry cache* which caches directory entries in memory. This is in contrast to object stores which may gain scalability benefits by using a flat namespace design [29].

In this paper, we present DelveFS – an object store file system in user space which offers the ability to compose a custom semantic file system, kept eventually consistent by processing *object store events* which describe changes in the object store, e.g., the creation of an object. Tailored to the user's needs, the file system presents unique *views* onto the object store that can be flexibly filtered by several fields, such as object names, size, access time, or object properties (similar to extended attributes in file systems). To support applications which depend on file locations within a directory tree, DelveFS allows a hierarchical structure on top of the user-defined view emulated within the object store's flat namespace. In addition, DelveFS supports distributed access from clients running on many machines, each having potentially different but possibly overlapping views. We achieve these features without restricting the use of other object store interfaces which can concurrently run beside DelveFS.

We demonstrate how we use events to map objects to files in a flat namespace and how they can be utilized to provide meaningful semantic subsets of data. Through events, DelveFS is able to provide low-latency updates of views via its *event handler* without heavily disrupting the object store by otherwise constantly contacting it for changes. Since the file system uses the object store as a black box, it provides eventual cache consistency with close-to-open semantics between its different interfaces and achieves similar throughput compared to the native object store interfaces.

DelveFS aims for several use cases, such as (1) (virtually) reducing the number of objects in large containers, (2) filtering for specific types of objects to ease access to target data of, e.g., data lake applications, and (3) automatically classifying new objects to the context of user-defined views. This can be useful, for example, in short read alignment tools in biology where large input sequences, classified by object metadata, are aligned to a reference genome. Instead of scanning potentially huge containers for input objects with specific metadata and putting file system connectors and object stores under high loads, applications can directly filter for desired metadata via DelveFS' views. This separates the logic for filtering files based on regulations and market decisions from applications.

In the following, Section II provides a background to describe some of today's object stores' core functionalities that are important for DelveFS. Section III describes DelveFS' goals, its central components, and main design considerations. In Section IV, we evaluate DelveFS w.r.t. event processing, event latencies, and metadata and data performance. Finally, Section V concludes and outlines our future work.

## II. Background and related work

This section gives an overview of today's object stores and their file system connectors, outlining their features, differences, benefits, and drawbacks.

*a)* ***Object storage:*** Object-based storage has several advantages over traditional storage methods like relational SQL databases or file systems. The complexity of containers and object stores, for example, does not increase when objects are added, and object stores, hence, tend to scale well [30]. The flat structure also enables low maintenance overhead.

Many Internet scale applications like Dropbox or Netflix use object-based storage systems for unstructured data. Nearly all cloud providers also offer or internally use object storage [31], [32], [33], while open-source and commercial offerings enable on-premise hosting of object stores [10], [9]. Several academic implementations and optimizations of object stores have fostered the development of object storage systems [12], [34], [35], [36], [37], [38].

We have built DelveFS on top of the OpenIO object storage architecture [11]. The only specific feature which DelveFS relies on is the availability of an event notification feature, independent of its implementation. OpenIO also offers a traditional file system connector (OIO FS) [39] based on *FUSE* [40] (Filesystem in Userspace) which directly maps objects to files. However, the two file system architectures are not immediately comparable since OIO FS is mounted as a network file system (NFS) to its FUSE instance while multiple DelveFS clients operate directly on FUSE.

*b)* ***Object properties:*** Object properties are similar to extended file attributes supported by several of today's file systems [41], [42], [43]. They are key-value pairs which allow users to add custom metadata to an object for a more detailed classification. Most object storage solutions support object properties, but they differ in their usage.

Amazon S3, for instance, allows two types of object properties: metadata and tags. Object metadata is tied to the immutability of the object and cannot be changed, whereas tags are separately managed. OpenStack Swift, on the other hand, only allows to upload a set of all properties, instead of modifying each property individually.

For DelveFS, flexible object property implementations, e.g., offered by OpenIO or Google Cloud Storage, work best since they allow updating key-value pairs similar to extended attributes in file systems. This is because DelveFS allows the modification of object properties and uses them not only as extended attributes but also to offer a hierarchical namespace on top of the object store's actual flat namespace.

*c)* ***Object storage file systems:*** Many data-driven applications cannot work with S3 or custom object storage interfaces and need a standard POSIX file system interface instead. Hence, numerous file system *connectors* have been developed [44], [19], [45], [15], [17], [18], [46], [47], [48], [39], allowing an object store to be used as a UNIX file system.

In general, file system connectors present object store containers (or buckets) as directories and objects as files. Yet, they differ considerably in functionality and how they operate. S3QL supports multiple object stores and offers on-the-fly encryption [44]. SCFS provides strong consistency over object stores [19]. BlueSky employs a log-structured design to hide the latency of accessing the backend object store [45]. CephFS [15] uses its own RADOS object store to offer a file system interface.

The above-listed file systems do not offer *dual-access*, and objects can only be accessed through the file system interface if consistency is required. In this regard, the authors in [17] discuss the importance and efficiency of dual-accessing the object store through multiple interfaces. Object store file systems allowing dual-access are S3FS [18], Goofys [46], SVFS [47], RioFS [48], and OIO-FS [39].

However, none of the above-mentioned file systems take containers with millions or even billions of objects into account that became increasingly common in today's object stores [20]. Instead, they only offer a basic representation of objects as files in a flat namespace, lacking any structure and semantics in addition to technical challenges due to metadata management [28]. Specifically, improving metadata management in distributed storage systems is an active research area [49], [50], [51], [52], [53], and metadata performance is still an issue in compute clusters [54], [55].

Nevertheless, even if a hierarchical structure would be used to retain some structure, it is still not enough when managing large data volumes. As a result, semantic file systems have been proposed several times over the past decades [21], [22], [23] where the data is dynamically searchable to achieve a representation that fits the way the data is accessed. The authors in [22], for example, use path suffixes to filter files by ownership and more. Some object stores allow basic filtering mechanisms, e.g., for object name wildcards [24] or time ranges [25], or more sophisticated search capabilities for object properties [26]. Similar functionality is not available in object store connectors.

DelveFS provides dual-access and offers considerably higher usability than above-listed file systems by providing semantic file system views based on a metadata filtering mechanism. Additionally, DelveFS overcomes existing restrictions in managing consistency by coupling the object store's view and the user-defined file system's view through the event interface of modern object stores. Due to the event interface, the file system achieves these features without disrupting the object store, providing fast namespace updates to clients at a low latency.

*d)* ***Event notifications:*** DelveFS requires notifications when an object is changed to ensure that both the object interface and the file system views can become eventually consistent. DelveFS, therefore, processes an object store's *events* and heavily uses object properties. Object store events or *notifications* refer to data which describe an operation within the object store, such as object write/upload operations, and can be used to notify an application when an object or container is updated. Events are available in object stores like OpenIO, Amazon S3, or in Google Cloud, but they are still missing in Swift or RADOS. Nonetheless, the event interfaces offered by various object stores differ significantly by how events are accessed, by the events' content, and by the general features the event interface offers, i.e., which operations produce events.

For DelveFS, events that describe changes of object properties or object sizes are particularly important as they are crucial to update a client's view on its defined namespace. This is because object metadata, e.g., properties and sizes, can be used when a client defines its view (see Section III-D). Object property events are, however, only supported by OpenIO and by Google Cloud Storage. DelveFS uses the OpenIO event interface as an example to leverage on such events.

Agni [17] and YAS3FS [56] use events to notify file system clients of changes in the object store. Such FUSE-based file systems, which are usually used in object storage connectors [17], [18], [44], [39], [19], [46], [47], [48], are, however, known to struggle with workloads that involve a large number of metadata operations [57] – also shown in Section IV-C2. To address this shortcoming, DelveFS processes events significantly further to allow user-defined semantic file system views, allowing fine-grained searches and reducing the overhead of listing otherwise potentially huge directories.

## III. Design

In this section, we present DelveFS' goals, its architecture and system components, and discuss how object store events are processed in the file system and how this processing influences its consistency model.

### A. Goals

DelveFS was designed to achieve the following objectives:

**Functionality** DelveFS should provide a file system interface to an existing object store through the Linux *virtual file system* (VFS). The file system should maintain its own persistent view of the object store via metadata information, but it should not store data outside of the object store except for caching.

**Dual-access** The object store's interfaces should be able to be used at the same time as DelveFS. In addition, multiple DelveFS clients should be able to operate concurrently and access the object store in parallel. However, we assume that a single object is only modified by one client, either DelveFS or through an object store interface, at a time.

**Object property filtering** Users should be able to define rules within DelveFS which only allow them to view objects in the file system that are of particular interest. Such rules then filter the object namespace for containers, objects, object sizes, object access time, or object properties. Filtering for container and object names should allow not only exact name searches but also more complex expressions, such as filtering for all objects with a characteristic prefix or suffix.

**Consistency model** DelveFS is not intended to strictly follow the POSIX I/O semantics. DelveFS should instead provide an eventual consistency model with *close-to-open* semantics [58]. These semantics are sufficient for our assumption that only a single client is modifying an object at a time.

A stronger consistency model, e.g., provided by several parallel file system implementations [59], [60], would also require a distributed locking mechanism, and, hence, full control of the object store. However, we use the object store as a black box so that various object store interfaces and DelveFS can be used at the same time. Further, aligning DelveFS more to NFS semantics allows us to significantly reduce the file system's complexity and avoid otherwise intricate and communication-intensive distributed locking mechanisms [61], [62]. Providing such file system semantics implies that changes applied to files within DelveFS may not be immediately visible in the object store and vice versa due to caching mechanisms and event processing, affecting both metadata and data operations.

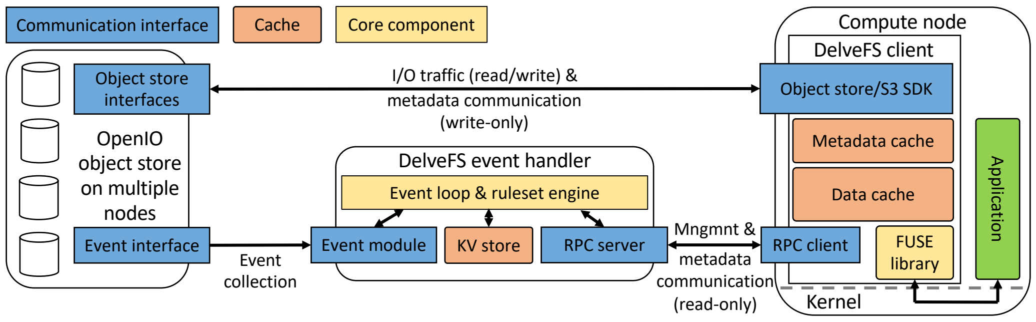


Fig. 1: DelveFS' architecture with its components and interaction with the object store.

## B. Architecture

Figure 1 shows DelveFS' architecture, its main components, and its interaction with an object store. The object storage backend is given (in this example) by an OpenIO object store [11] running on multiple nodes. The object store provides interfaces to communicate metadata and bulk data between the OpenIO object store and its clients. DelveFS includes the event handler and supports multiple FUSE file system clients.

The event handler is running as a single instance and is the core of DelveFS' ecosystem. It has two primary responsibilities: maintaining the state of the object store's namespace by processing its events and filtering the namespace for user-defined rules. DelveFS allows multiple concurrently running clients which can be added to and removed from the DelveFS ecosystem at any time. To significantly reduce the load on the single event handler instance and to increase file system performance, clients heavily utilize metadata and data caches.

In general, DelveFS' architecture focuses on managing the object store's metadata to provide a semantic file system *view* of objects of interest. So-called *rulesets* define such views, each containing a set of filters for relevant objects which are then placed as files into a directory, defined by the ruleset.

The life cycle of a file starts with its creation either via the object store interface or the file system interface. Afterward, the corresponding metadata is forwarded via the event interface to the event handler which updates all affected file system views of DelveFS clients. Clients do not get notified by these updates directly, but lazily ask the event handler themselves when required, for instance, when a user lists a directory. Note, since DelveFS' architecture allows dual-access, views may also be affected by objects that have been created or removed outside of DelveFS, e.g., through the object interface, whose produced events are then processed by the event handler.

To remain consistent, DelveFS requires at least three event types: an object creation event, an object removal event, and an object update event when its size or its properties are modified. Properties, i.e., key-value pairs similar to file system extended attributes, allow a fine-granular way of filtering the namespace.

The communication between DelveFS' components uses the *Mercury* RPC communication library [63]. DelveFS interfaces Mercury through the *Margo* library [64] which provides *Argobots*-aware wrappers [65] to Mercury's API.

In the following, we describe each component in detail.

## C. Event handler

The event handler stores the object store's current state, i.e., its containers and objects, for DelveFS clients' registered rulesets. Therefore, the event handler maintains and provides the metadata of objects of interest, including their directory structure. In principle, any user can launch the event handler, e.g., within the context of an HPC compute job. However, depending on the used object store, allowing the user access to the object store's event interface may require administrative support. In summary, the event handler consists of four components (see Figure 1): (1) A key-value store (KV store) to keep track of the object store's state, (2) an event loop which handles incoming *events*, describing a change within the object store, (3) a parallel rule-matching engine to process the content of each event, and (4) an RPC communication layer to exchange messages with DelveFS clients.

To store an object's metadata, e.g., an object's size, the event handler operates a local RocksDB KV store, providing a high-performance embedded database for key-value data [66]. When a client connects, the event handler queries the object store to list all objects of containers that are part of the client's rulesets, if not already available in RocksDB. Therefore, RocksDB does not necessarily store the full object store state depending on the rulesets and object store size. Lazily populating the KV store allows us to avoid a time-consuming startup time caused by scanning the entire object store.

Furthermore, DelveFS actively discards all events that are irrelevant to a ruleset. For low-latency access, some of the object's metadata that match a connected client's ruleset is kept in memory. Adapting to Google's object store would additionally allow only to listen for events of specified containers, which makes discarding events unnecessary.

In the OpenIO case, events are exposed by one *beanstalkd* [67] work queue per OpenIO node that is accessed by DelveFS' event handler. Each event in the beanstalkd work queue describes a single change in the object store, such as the creation of objects or object properties, and is removed from the work queue after it has been processed.

The *event loop* is responsible for pulling events from the beanstalkd work queues and processing them. At its core, it consists of multiple *progress threads* which access beanstalkd

```yaml
---
Gecko:
  account: "delvefs"
  container: "gecko_genome"
  object: "*.fasta"
  properties:
    color: "black"
    color: "green"
Leopard:
  [...]
---
```

Fig. 2: An example of the `Gecko` ruleset in YAML syntax and its defined rules to filter for objects in the `gecko_genome` container.

queues and so-called *worker threads* which process new events that have been accessed by the progress threads. The progress thread identifies the event type and passes the event's content to a worker thread, created from a dedicated thread pool. During the *event matching*, each worker thread initially checks if the event refers to a tracked container and if the described changes apply to any registered ruleset. In that case, the worker thread updates the KV store and ruleset (if applicable) accordingly. Note that the event handler does not notify a DelveFS client if an incoming event affects its rulesets. Instead, the client lazily requests this information when required.

In the case of an event handler failure, no client data is lost as the event handler is a read-only medium. It does not immediately affect clients, although them being unable to update their ruleset metadata information. Hence, an interruption does not break the eventual consistency guarantees. Further, because the event handler does not store additional information beyond what is available in the object store, there is no immediate need for an advanced crash consistency procedure to conserve events. Such a procedure is part of our future work and could decrease the necessary time when clients lazily ask for their data after the event handler is restarted.

### *D. File system client*

The file system client is directly accessed by an application and connects to the object store and the DelveFS event handler. It is based on the FUSE library, providing an interface to export a user space file system implementation to the kernel. More precisely, DelveFS is using FUSE's low-level API which allows a flexible file system design and avoids various overheads of the high-level API [57].

Once started, it is mounted and can be used like any other local file system. The client consists of four components: (1) A file system implementation based on FUSE's low-level API, (2) an in-memory caching module to store metadata as well as data for a short time frame, (3) an object store module to directly communicate with the object store, and (4) an RPC communication layer based on Mercury to exchange messages with the DelveFS event handler.

Next, we discuss how rulesets work and describe DelveFS' metadata and data management, including their cache policies.

*a) Rulesets:* When the client is started, it requires a mount path and a file containing a *ruleset*, which is a collection of rules in *YAML* syntax [68] that the object store should be filtered for. Each ruleset may consist of five rule categories with each rule further narrowing down the results: container name (mandatory), object name, object size, object access time, and object properties. All rulesets are initially sent to the event handler which evaluates them and responds with the results, i.e., the metadata about all matched objects.

Figure 2 presents an example YAML ruleset with the name `Gecko` (defined in Line 2), showcasing some of the rule categories. All objects that match all rules of the set are then placed into a single directory named after the ruleset. The ruleset matches all objects in the `gecko_genome` container with the properties black color **or** green color. Alongside exact matches, DelveFS also supports partial matches by using the $*$ character. In this case, the `object` keyword matches against all objects whose name ends with `"*.fasta"`, an extension for files storing nucleotide sequences or protein sequences in the FASTA format [69].

*b) Metadata management:* The client contacts the event handler via read-only RPC messages to retrieve the latest update of a directory's state, i.e., the ruleset's content. The state is represented as directory entries and inodes and is kept in in-memory caches, considered *recent* for a configurable amount of time. Therefore, changes in the object store may not be immediately visible to the client. Note that a file can disappear from a client's ruleset directory in specific circumstances. For example, the corresponding object could have been removed from the object store, or an object's properties have changed and no longer match all ruleset criteria.

File system operations which modify the namespace, i.e., creating and removing files or updating their extended attributes, directly communicate with the object store. This avoids potential consistency issues between DelveFS and the object store. In general, the object store is always treated as the primary source of truth and is updated before DelveFS. For example, if a client creates a file, a corresponding object is created as well. DelveFS' event handler is then unaware of this object until the corresponding create event is processed. Note, the event handler cannot differentiate whether the object store was modified by a DelveFS client or by another source for a given event.

*c) Data management:* Similar to metadata operations, an I/O operation does not notify another DelveFS node and communicates with the object store via its API directly. However, in terms of I/O operations, the file system interface is not immediately compatible with an object interface due to their data handling. For instance, a (local) file system assumes that data is available in mutable blocks and thus accessible and changeable at any offset. Object storage, on the other hand, introduced the concept of immutable objects which cannot be modified in-place after creation [10], [7].

For this reason, modifications of an object require downloading the object, modifying it locally, and then uploading a new object with the same name that replaces the old object. It is worth mentioning that object replacement is not available in all object stores and can depend on container configurations, in which case the object has to be removed first. FUSE-based file systems, on the other hand, split file system operations for reads and writes so that they use a maximum buffer size of 1 MiB and 128 KiB, respectively. In case of a write operation, downloading the full object and replacing at most 128 KiB of data before uploading it back to the object store is, therefore, inefficient and slow while causing unnecessary additional loads over the network.

To address this issue, DelveFS uses a write-back data cache. It supports both the OpenIO API and the S3 API, which translates (remote) object store objects to files (henceforward called *object files*) and stores the cached data at a user-defined local file system path outside of the FUSE environment. For write operations, the data cache allows DelveFS to work locally until a `close` or `fsync` command is issued which causes DelveFS to upload the object file to the object store. DelveFS' semantics assume that data is not changed outside this client until the data is uploaded back to the object store and, therefore, only allow a single writer to each file.

For read operations, DelveFS downloads the remote object on the first encountered read operation and places the resulting file in the data cache, allowing all following operations to run locally. The cached file is then considered recent for a configurable amount of time. To avoid downloading an object repeatedly after the caching time expires, DelveFS compares the checksum value of the object file and the remote object.

*d) Virtual directories:* Rulesets allow users to create a semantic file system view, but, due to the object store's flat namespace, ruleset directories would not have a further directory structure which might be required by applications and users. Thus, DelveFS can emulate a directory hierarchy within a ruleset, e.g., to rebuild the complete Linux operating system hierarchy. These directories are entirely virtual since object stores do not support recursive container structures. In other words, an object's directory association is stored as an object property resolved only at the client. As a result, the directory hierarchy of an object is fixed for all clients.

An alternative approach for such directories are dedicated directory objects, similar to directory blocks in file systems, only containing information of object names belonging to the directory. We decided against this approach because it would require a central entity (e.g., a global lock manager) that controls access to such an object so that file system clients do not overwrite the directory information of each other. In addition, directory objects are of no value to object store users and would be considered junk objects.

Nevertheless, in both approaches, virtual directories are tied to the object and fixed for all clients, although its content may differ based on the ruleset directory rules. If necessary, showing virtual directories can be disabled in the ruleset.

## IV. Evaluation

This section presents the evaluation of DelveFS concerning the performance of the event processing, the file system's metadata handling and (ruleset) directory capabilities, and its data accesses. Finally, we present an example of how rulesets can benefit an application's workflow.

### A. Experimental setup

In our experiments, we used the Intel Skylake partition of the MOGON II cluster at the Johannes Gutenberg University Mainz. MOGON II consists of 1.876 nodes in total, from which 1.046 nodes contain two Xeon Gold 6130 Intel Skylake processors with main memory capacities ranging from 64 GiB up to 1,536 GiB. The cluster nodes are interconnected by a 100 Gbit/s Intel Omni-Path network to establish a fat-tree, and it offers a Lustre parallel file system as storage backend.

The DelveFS event handler uses local-node SATA SSDs as backend storage for RocksDB. The DelveFS clients and the event handler use TCP/IP over Omni-Path as the network protocol. In all DelveFS experiments, the DelveFS clients and the applications under test are pinned to separate processor sockets to minimize possible interference. The FUSE client library runs in multithreaded mode and uses up to ten threads.

To demonstrate DelveFS' functionality and performance, we use an OpenIO object store that runs on ten compute nodes. Nine nodes serve as storage nodes for data and metadata, and one node serves as a management node and gateway for metadata requests. For data and metadata, OpenIO' storage nodes use data center SATA SSDs, offering approximately 1.8 TB of storage in total. The object store was used in isolation without any other user accessing it. Internally, OpenIO communicates using TCP/IP over Omni-Path. External nodes which use the object store use TCP/IP over Omni-Path as well to download and upload objects. In S3FS experiments, we use OpenIO's S3 interface. Finally, we disabled data replication in OpenIO and limited versioning to two versions per object.

### B. Event processing

DelveFS clients depend on the performance of the object store's event interface and the event handler's ability to process events quickly. In the case of the OpenIO object store, events are exposed by the beanstalkd work queue (see Section III-C) with each of the nine OpenIO nodes running a separate beanstalk daemon. Henceforward, we use the terms beanstalkd and work queue interchangeably.

The event handler allows DelveFS to store metadata outside the object store. Therefore, it avoids that read-heavy metadata requests, e.g., `lookup` or `readdir`, are issued to the object store directly. Additionally, the event handler allows concepts like rulesets which filter the namespace in a user-defined way. Depending on a central entity for this use case implies that the event handler must be able to process events with reasonable throughput. In addition, each event must be available for the client at a low latency if it matches any of its defined rulesets.

We designed three scenarios to evaluate the various components of the event handler and to measure its event processing throughput and event latencies. The first scenario (`RAW`) covers the event handler's raw throughput capabilities when events are not matching any client ruleset. Since events are only

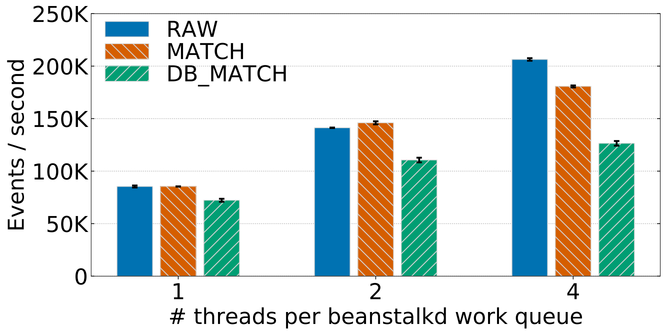


Fig. 3: DelveFS' event throughput in three scenarios.

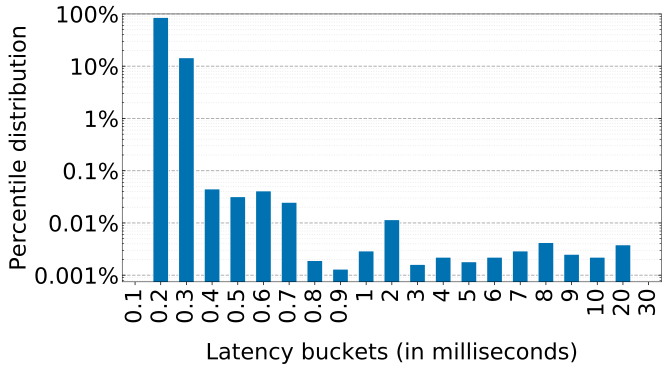


Fig. 4: DelveFS' latency distribution of the event handler's event processing in the MATCH scenario.

removed from the work queues and then discarded due to their inapplicability, this represents the best case as the least amount of functionality is triggered. The second scenario (MATCH) measures performance degradation by continuously matching events to rulesets. MATCH is only accessing a single object resulting in no updates to the KV store. It is, however, matching to 100 different rulesets. MATCH continuously accesses RocksDB to read and update metadata, while the limited number of objects allows RocksDB to (nearly) completely work in memory. The third scenario (DB_MATCH) also stresses the KV store. DB_MATCH events do not only match each event to 100 rulesets but also refer to enough objects in total (10,000 in this case) so that RocksDB must continuously read and write so-called *static sorted table files* (SST files) from and to the underlying storage. SST files contain parts of the KV store's entries to persist the KV store's state to disk. Hence, the DB_MATCH workload targets all event handler components. A file system client (on a separate node) set up all necessary rule sets while remaining idle during all experiments.

*a) Event throughput:* To measure the event throughput for the above-described scenarios, we artificially inserted two million events into each of the nine work queues of the OpenIO nodes. We used an insertion rate of ~390,000 events per second in total (about 1-minute runtime) as it exceeds the event handler's processing capabilities in the RAW scenario, allowing us to examine its sustained event throughput.

We did not trigger events by using OpenIO directly because, due to its limited installation size, OpenIO emitted less than 1,000 events per second. Instead, by using similar artificial events, we were able to explore the event handler's processing capabilities in an environment which generates a significantly higher number of events per second.

Figure 3 presents the event handler's throughput for all three use cases for up to four threads per work queue (x-axis). The y-axis depicts the event handler's total throughput in each case. In the RAW scenario, the event handler processed up to 200K events per second for four accessing threads per work queue.

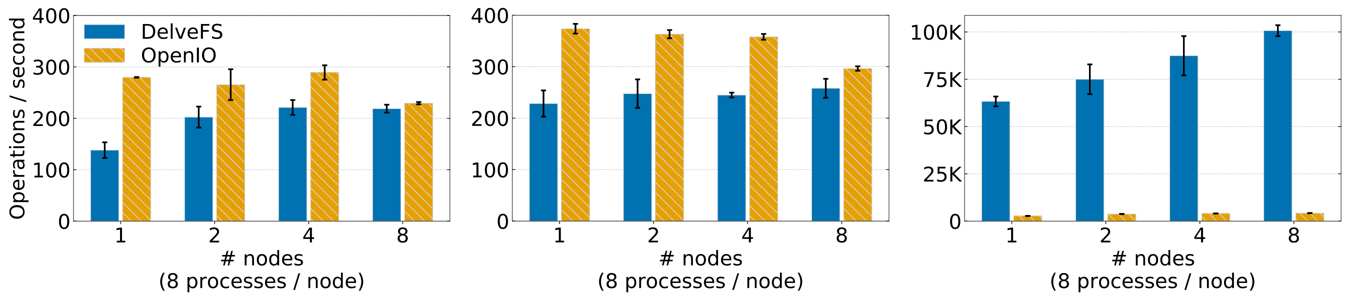
In the MATCH scenario, the processing throughput was reduced to 180K events per second, mainly attributed, however, to repeated KV store accesses as each event results in a put operation to update the object's metadata. Increasing the number of threads for each work queue did not yield further improvements in the RAW and MATCH scenarios.

In the DB_MATCH scenario, the event throughput reached up to 126K events per second for four work queue threads. Similar to the MATCH scenario, the performance degradation is not attributed to matching events to rulesets, but to the continuous access and modification of the objects' metadata information in RocksDB. However, the larger number of objects also caused the creation of SST files by the KV store on the backend SSD storage that is constantly written and read. This slightly delays access to each KV store entry, resulting in a ~30% reduced throughput compared to MATCH for four work queue threads.

*b) Event latency:* The time from when an event enters the work queue until its changes are available to the client is defined as *event latency*. Note that the event handler does not actively notify connected clients about any changes to a ruleset (see Section III-C). Therefore, an event's content is available to be accessed by clients (if they actively require it) as soon as the event handler has processed it.

We evaluated DelveFS' event latency by running the MATCH and DB_MATCH workloads. We have omitted RAW because it only processes events which would be irrelevant for all clients. For each of the two used workloads, we inserted eight million events in total to all work queues. The event insertion rate has been set to ~50,000 events per second so that the maximum latency of MATCH remained below 30 milliseconds. We also ensured that this insertion rate does not lead to a saturation in the DB_MATCH scenario since the event latencies of the more complex DB_MATCH scenario significantly differ from MATCH. Finally, to compute accurate latencies, the event handler and workload-inducing benchmark were run on the same compute node, each on a separate processor socket.

Figure 4 presents the latency distribution for MATCH of eight million events. In this scenario, each event is matched to 100 rulesets with continuous updates to an object's metadata saved in the KV store (see above for details). The x-axis depicts a non-linear range of latencies to capture all processed events' latencies, ranging from 100 microseconds to 30 milliseconds. The y-axis represents the percentage of events that fall into each latency bucket on a logarithmic scale. For instance, the



(a) Create throughput (b) Remove throughput (c) Stat throughput

Fig. 6: DelveFS' metadata throughput when compared to the direct usage of OpenIO' SDK.

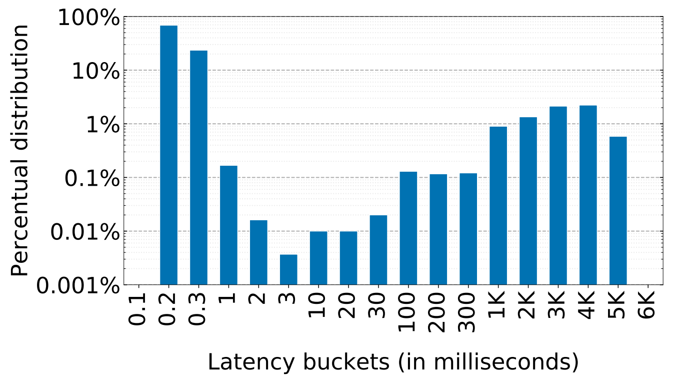


Fig. 5: DelveFS' latency distribution of the event handler's event processing in the `DB_MATCH` scenario.

latencies of all events in the 200 $\mu s$ bucket range from 101 to 200 $\mu s$. In this experiment, the event handler showed a low latency for over 99.8% of events which were processed in less than 400 $\mu s$. The longest latency accounted for 30 $ms$. We achieved these low latencies as both the rulesets and nearly all updates to RocksDB have been kept in memory.

Figure 5 shows the latency distribution for `DB_MATCH` for 8 million events where all events refer to a total of 10,000 objects in addition to each event being matched to 100 rulesets. Compared to the latency distribution of `MATCH`, this workload resulted in a significantly higher range of latencies, attributed to congestion at the KV store that can take several seconds. Nonetheless, over 92% of all events are processed in less than 300 $\mu s$ with an overall event throughput that is well above the load that our experimental setup can emit.

## C. Metadata performance

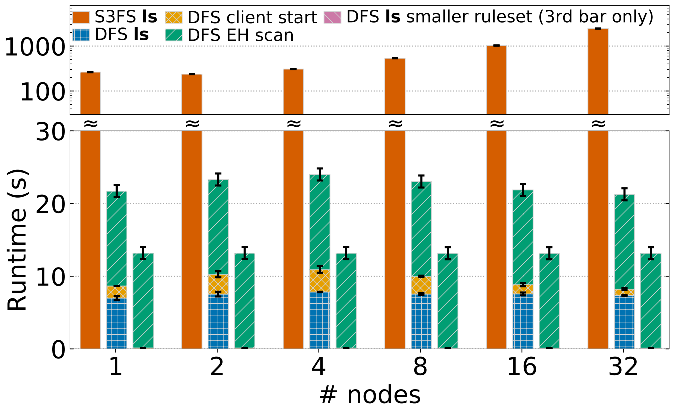
This section discusses DelveFS metadata performance, both for file metadata and directory metadata operations.

*1) Microbenchmarks:* We evaluated DelveFS' metadata performance using a parallel microbenchmark which *creates*, *removes*, and *stats* $1,000$ zero-byte files per process in a single container for 8 processes per node. We compared its metadata operation throughput with the equivalent operations when using OpenIO's SDK directly to explore the performance impacts of DelveFS' file system layer.

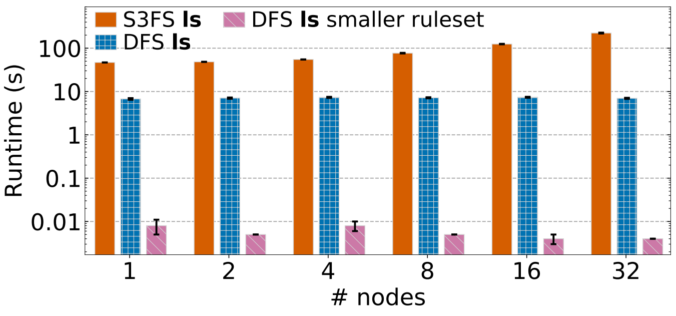
Figure 6 presents the results, each data point representing the mean of five iterations. The standard deviation has been calculated as the percentage of the mean. OpenIO's SDK throughput reached up to 290 creates per second, 370 removes per second, and 4,200 stats per second. OpenIO reaches the maximum throughput for create and remove operations already for a single client node, showing potential metadata congestion at the object store. DelveFS did not achieve the same throughput as each create and remove operation causes multiple consecutive internal FUSE function invocations, causing a number of overheads in FUSE's internal components [57]. Overall, the observed overhead was significantly more prevalent in metadata experiments than in the later discussed data experiments. These effects, however, become less severe for more than one node where we approached the create and remove throughput limit of the object store.

DelveFS' `stat` throughput was over 23× higher compared to OpenIO's SDK because the DelveFS client does not rely on the object interface when asking for an object's size and checksum and instead acquires it from the event handler (see Section III-D). The event handler can then serve the results from memory without additional lookups to RocksDB. Bigger systems might show a lower stat throughput when not all metadata can be served from memory.

*2) Directory operations:* We performed several experiments to measure the time required to list large containers with the commonly used `ls -l` command for both DelveFS and S3FS. We restricted the container size to 100K objects, as the `ls -l` command did not finish when using larger container sizes (e.g., 150K objects) for S3FS. S3FS was mounted to the appropriate container while DelveFS clients used two rulesets with the same mount point: (1) A ruleset which contains all 100K objects of the container, and (2) a ruleset which filters the container for a rule-defined object name prefix, reducing the number of files within the directory to 100. We used this configuration to compare DelveFS' directory listing capabilities with the established S3FS object storage connector and to highlight the performance benefits of rulesets if they operate on a large container. Figures 7a and 7b show the time required in seconds (y-axis) to execute the `ls -l` command for an increasing number of nodes (x-axis) for a cold and hot cache, respectively. Each node ran a single process, and each



(a) Cold cache



(b) Hot cache

Fig. 7: Listing a 100K object directory in DelveFS and S3FS compared to using a ruleset, reducing its size to 100 objects.

data point presents the mean of three iterations.

Figure 7a presents the results for the above-described scenarios on a cold cache. Operating system caches were flushed, and the file systems, including the event handler, were restarted after each iteration. DelveFS' cold cache `ls -l` must include the time for two additional operations because the client relies on the event handler's container scanning (`DFS EH scan`) and ruleset processing capabilities, triggered when the client is started (`DFS client start`). For a single node, S3FS required ~360 seconds to list the container contents compared to DelveFS' ~22 seconds in total. For increasing node numbers, the time for S3FS increased to ~2450 seconds at 32 nodes, while DelveFS' performance remained stable. We omit further node numbers as DelveFS' performance remained stable for up to 128 tested client nodes.

The reason for S3FS' 110× longer runtime at 32 nodes is that clients must share a connection to the object store. This bottleneck becomes visible at eight nodes, worsening for more nodes. DelveFS' event handler, on the other hand, reads the container's content only once for all DelveFS clients using `DFS EH scan`, while it then can simply use its event processing. The startup of DelveFS clients `DFS client start` for a 100K file ruleset required less than 4 seconds in all cases, including the communication with the event handler and its ruleset processing. In fact, the average client startup time decreased for more nodes because the event handler can reuse the same ruleset definitions of other clients while they were starting up at the same time. The `ls -l` command itself took less than eight seconds in all cases, mainly attributed to FUSE's internal `readdir()` function that can only serve ~100 entries at a time (depending on the file name length).

When the ruleset `DFS ls smaller ruleset` with 100 objects was used, the client startup time required less than 150 milliseconds, while `ls -l` took less than 20 milliseconds. The startup time and `ls -l` are, hence, not visible in the figure. Therefore, the ruleset reduced the initial working set of the large container and significantly improved the performance of directory operations.

Figure 7b presents the same cases on a logarithmic scale for a hot cache, that is, the directory has recently been listed. This allows each DelveFS client to operate locally. Therefore, the time required to read the container content and to start the client is not included in the figure, leaving less than 8 seconds to list the 100K file ruleset directory (mainly due to FUSE's overhead) and less than 10 milliseconds to list the 100 file ruleset directory. For a single node, S3FS required ~46 seconds to list the directory but again bottlenecked at around 8 nodes, indicating that the file system is not operating entirely local.

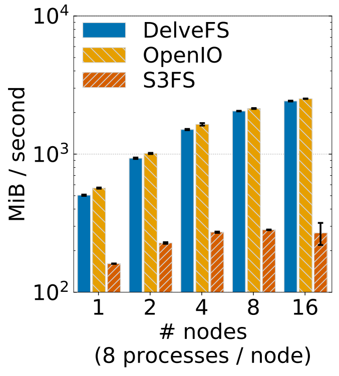
## D. *Data performance*

In this section, we investigate DelveFS' I/O performance and present an example of how rulesets can be used in a complex workflow by using the Bowtie [70] application.

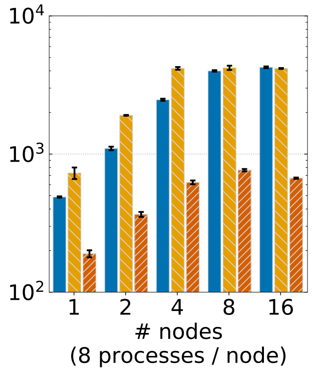
*1) Microbenchmarks:* We ran several experiments to evaluate DelveFS' I/O performance. For this task, we have compared DelveFS to directly using the OpenIO SDK (presenting the upper bound) and S3FS. In our experiments, each iteration accessed its own file/object and performed 4 GiB writes and reads per process in succession for up to 16 nodes. In the case of DelveFS, each experiment iteration can be broken down into four *core I/O operations*: (1) Writing test data to local storage, (2) uploading the data from local storage to the object store, (3) downloading the data from the object store to local storage, and (4) reading the test data from local storage.

To investigate the potential FUSE-induced overhead and to compare DelveFS directly with the equivalent object store interactions, we translated these four core I/O operations so that they are used in succession when using the OpenIO SDK. For the S3FS comparison, we disabled its multipart feature because it resulted in an up to 10% higher throughput. Moreover, since each core I/O step involves I/O to or from local storage, and because the network bandwidth to the OpenIO object store is significantly higher than the node-local SSDs bandwidth, the DelveFS client's and S3FS's data cache are placed on a RAM disk. However, to prevent read operations reading from the RAM disk, the file systems are restarted, and the caches are flushed after each write experiment. Finally, to avoid measuring the buffer cache instead of the sustained node-local SSDs performance in OpenIO, we limited the amount of available memory on the object store nodes.

Figure 8 shows the combined I/O throughput on a logarithmic y-axis for up to 16 client nodes. Each data point represents



(a) Write throughput



(b) Read throughput

Fig. 8: DelveFS' sequential data throughput when compared to the direct usage of OpenIO' SDK and S3FS.

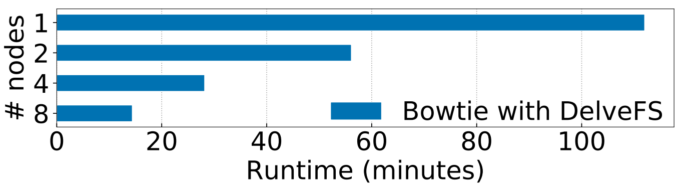


Fig. 9: Bowtie runtimes with DelveFS.

the mean of five iterations with the standard deviation as the percentage of the mean. For 16 nodes in the OpenIO SDK case, the upload (write) and download (read) throughput reached up to $2,500$ and $4,100$ MiB per second, respectively. The throughput is bound by the network for smaller node numbers, while OpenIO achieved to saturate the expected SSD I/O throughput starting at 4 clients for reads and 16 clients for writes. DelveFS was slightly slower for writes, while it was able also to saturate the object backend for reads starting at 16 clients. We omit further node numbers as the data throughputs did not increase for up to 128 tested client nodes.

S3FS achieved at most ~282 MiB and ~764 MiB of write and read throughput, respectively, reached at 8 nodes. Note, however, that S3FS was unable to complete the workload for 16 nodes, causing I/O errors upon closing the files, resulting in an incomplete upload to the object store. To allow a successful completion, we limited the number of processes to 4 per node for S3FS' 16 node experiment.

*2) Short read alignment:* As an example of how DelveFS' rulesets could be used, we ran a typical workload from the biology field where many *short reads*, i.e., short sequences of less than 200 nucleobases produced by *Next Generation Sequencing* (NGS), are aligned to a reference DNA sequence. For this task, we used the popular *Bowtie* [70] application to align a total of ~100 GiB of short reads (consisting of 16 *fastq* input files[1]) to a human reference sequence. During the alignment process ~11 GiB of output data was created.

[1]We downloaded short read sequences from the National Center for Biotechnology Information's (NCBI) Sequence Read Archive (SRA) [71]

In this case, we used rulesets to describe a *publish-subscribe* model where the publisher (the sequencer) addressed its subscribers (the alignment processes) based on their interests, defined as object properties in rulesets. We started DelveFS clients on up to eight nodes, each with its own unique ruleset. The publisher then placed the 16 fastq input objects in a container via the object interface. For each experiment, we defined the input objects' properties to address the rulesets of a set of DelveFS clients, distributing the workload.

A subscriber process on each client node checked the ruleset directory every minute for new input data and launched the Bowtie application for each input file in sequence. Figure 9 presents the Bowtie runtimes for these four experiments, showing the impact of implicit load balancing. Finally, the alignment results were written to an output ruleset directory which automatically assigned its defined properties to the corresponding output files.

This example shows that rulesets are not only able to improve overall performance, e.g., shown for the `DFS ls smaller ruleset` above, but also to simplify the development of complex workflows and of load balancing schemes.

## V. Conclusion

With DelveFS [2], we have presented a novel approach how object store events can be used to provide unique semantic file system views for each user. Because experimental data will likely significantly increase, tailoring object namespaces of billions of objects to the user's needs will become even more important in the future. We have shown how DelveFS processes up to 200,000 events per second, propagating object store changes to a client's view at a low latency. DelveFS allows dual-access, and we have demonstrated its similar metadata and data throughput, compared to native object storage interfaces. Finally, we have evaluated how DelveFS' rulesets can significantly improve directory operations and have shown how they could simplify complex workflows.

In the future, we plan to extend DelveFS into three directions. First, we plan to develop more efficient techniques to store rulesets in the event handler to be able to persist them to backend storage w.r.t. failure recovery and to group them based on their similarities. Second, we intend to explore the possibility of using a distributed event handler to increase DelveFS' event processing capabilities further and remove the single point of failure in updating clients. Third, we aim to support additional object store interfaces and study the benefits of other consistency concepts, such as uploading objects during the writing process in intervals.

## Acknowledgements

This work has been partially supported by the German Research Foundation (DFG) through the ADA-FS project the Priority Programme 1648 "Software for Exascale Computing". Further, the authors would like to thank our anonymous reviewers for their valuable comments. The authors gratefully acknowledge the computing time granted on Mogon II.

[2]https://gitlab.rlp.net/DelveFS/delvefs

## REFERENCES

[1] E. Ayday, E. D. Cristofaro, J. Hubaux, and G. Tsudik, "Whole genome sequencing: Revolutionary medicine or privacy nightmare?" *IEEE Computer*, vol. 48, no. 2, pp. 58–66, 2015.

[2] M. Baker, "Next-generation sequencing: adjusting to data overload," *Nature Methods*, vol. 7, pp. 495 – 499, 2010.

[3] M. Grawinkel, L. Nagel, M. Mäsker, F. Padua, A. Brinkmann, and L. Sorth, "Analysis of the ECMWF storage landscape," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST 2015, Santa Clara, CA, USA, February 16-19, 2015*, 2015, pp. 15–27.

[4] A. J. Peters, E. Sindrilaru, and G. Adde, "Eos as the present and future solution for data storage at cern," in *21st International Conference on Computing in High Energy and Nuclear Physics (CHEP)*, 2015.

[5] P. Quinn, T. Axelrod, I. Bird, R. Dodson, A. Szalay, and A. Wicenec, "Delivering SKA science," *CoRR*, vol. abs/1501.05367, 2015.

[6] R. Hai, S. Geisler, and C. Quix, "Constance: An intelligent data lake system," in *Proceedings of the International Conference on Management of Data (SIGMOD), San Francisco, CA, USA, June 26 - July 01*, 2016, pp. 2097–2100. [Online]. Available: https://doi.org/10.1145/2882903.2899389

[7] J. Murty, *Programming Amazon web services - S3, EC2, SQS, FPS, and SimpleDB: outsource your infrastructure*. O'Reilly, 2008.

[8] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "Blobseer: Next-generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, no. 2, pp. 169–184, 2011.

[9] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: a scalable, reliable storage service for petabyte-scale storage clusters," in *Proceedings of the 2nd International Petascale Data Storage Workshop (PDSW), November 11, Reno, Nevada, USA*, 2007, pp. 35–44.

[10] J. Arnold, *OpenStack Swift: Using, Administering, and Developing for Swift Object Storage*, 1st ed. O'Reilly and Associates, Oct. 2014.

[11] OpenIO, "Openio core solution description," Technical White Paper, https://www.openio.io/resources/, pp. 1 – 17, 2016, accessed on May 2, 2020.

[12] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie, "Ursa minor: Versatile cluster-based storage," in *Proceedings of the Conference on File and Storage Technologies (FAST), December 13-16, San Francisco, California, USA*, 2005.

[13] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z. Zhang, "Unreeling netflix: Understanding and improving multi-cdn movie delivery," in *Proceedings of the IEEE INFOCOM, Orlando, FL, USA, March 25-30*, 2012, pp. 1620–1628.

[14] I. Drago, M. Mellia, M. M. Munafò, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: understanding personal cloud storage services," in *Proceedings of the 12th ACM SIGCOMM Internet Measurement Conference (IMC), Boston, MA, USA, November 14-16*, 2012, pp. 481–494.

[15] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *7th Symposium on Operating Systems Design and Implementation (OSDI), November 6-8, Seattle, WA, USA*, 2006, pp. 307–320.

[16] D. Zhao, X. Yang, I. Sadooghi, G. Garzoglio, S. Timm, and I. Raicu, "High-performance storage support for scientific applications on the cloud," in *Proceedings of the 6th Workshop on Scientific Cloud Computing (ScienceCloud), Portland, Oregon, USA, June 16*, 2015, pp. 33–36.

[17] K. Lillaney, V. Tarasov, D. Pease, and R. C. Burns, "Agni: An efficient dual-access file system over object storage," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC), Santa Cruz, CA, USA, November 20-23*, 2019, pp. 390–402.

[18] R. Rizun, "S3fs-fuse," https://github.com/s3fs-fuse/s3fs-fuse, 2019, accessed on January 7, 2020.

[19] A. N. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin, and P. Veríssimo, "SCFS: A shared cloud-backed file system," in *USENIX Annual Technical Conference (ATC), Philadelphia, PA, USA, June 19-20*, 2014, pp. 169–180.

[20] J. Barr, "Amazon s3 batch operations," https://aws.amazon.com/blogs/aws/new-amazon-s3-batch-operations/, 2019, accessed on May 2, 2020.

[21] M. I. Seltzer and N. Murphy, "Hierarchical file systems are dead," in *Proceedings of HotOS'09: 12th Workshop on Hot Topics in Operating Systems, May 18-20, 2009, Monte Verità, Switzerland*, 2009.

[22] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. O'Toole, "Semantic file systems," in *Proceedings of the Thirteenth ACM Symposium on Operating System Principles, SOSP 1991, Asilomar Conference Center, Pacific Grove, California, USA, October 13-16, 1991*, 1991, pp. 16–25.

[23] Y. Hua, H. Jiang, Y. Zhu, D. Feng, and L. Tian, "Semantic-aware metadata organization paradigm in next-generation file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 2, pp. 337–344, 2012.

[24] Google, "Wildcard names," https://cloud.google.com/storage/docs/gsutil/addlhelp/WildcardNames, 2020, accessed on May 2, 2020.

[25] M. Deck, "Building and maintaining an amazon s3 metadata index without servers," https://aws.amazon.com/blogs/big-data/building-and-maintaining-an-amazon-s3-metadata-index-without-servers/, 2015, accessed on May 2, 2020.

[26] Y. Sadeh, "New in luminous: Rgw metadata search," https://ceph.io/rgw/new-luminous-rgw-metadata-search/, 2020, accessed on May 2, 2020.

[27] P. H. Lensing, T. Cortes, and A. Brinkmann, "Direct lookup and hash-based metadata placement for local file systems," in *6th Annual International Systems and Storage Conference, SYSTOR '13, Haifa, Israel - June 30 - July 02, 2013*, R. I. Kat, M. Baker, and S. Toledo, Eds. ACM, 2013, pp. 5:1–5:11.

[28] B. Welch and G. A. Gibson, "Managing scalability in object storage systems for HPC linux clusters," in *21st IEEE Conference on Mass Storage Systems and Technologies / 12th NASA Goddard Conference on Mass Storage Systems and Technologies, Greenbelt, Maryland, USA, April 13-16*, 2004, pp. 433–445.

[29] P. H. Lensing, T. Cortes, J. Hughes, and A. Brinkmann, "File system scalability with highly decentralized metadata on independent storage devices," in *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2016, Cartagena, Colombia, May 16-19, 2016*. IEEE Computer Society, 2016, pp. 366–375.

[30] M. P. Mesnier, G. R. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, vol. 41, no. 8, pp. 84–90, 2003.

[31] S. L. Garfinkel, "Commodity grid computing with amazon's S3 and EC2," *;login:*, vol. 32, no. 1, 2007.

[32] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP), Cascais, Portugal, October 23-26, 2011*, 2011, pp. 143–157.

[33] M. Subramanian, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, S. Viswanathan, L. Tang, and S. Kumar, "f4: Facebook's warm BLOB storage system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Broomfield, CO, USA, October 6-8*, 2014, pp. 383–398.

[34] G. Mathur, P. Desnoyers, D. Ganesan, and P. J. Shenoy, "Capsule: an energy-optimized object storage system for memory-constrained sensor devices," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys), Boulder, Colorado, USA*, 2006, pp. 195–208.

[35] J. Terrace and M. J. Freedman, "Object storage on CRAQ: high-throughput chain replication for read-mostly workloads," in *USENIX Annual Technical Conference (ATC), San Diego, CA, USA, June 14-19*, 2009.

[36] P. Matri, A. Costan, G. Antoniu, J. Montes, and M. S. Pérez, "Týr: blob storage meets built-in transactions," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Salt Lake City, UT, USA, November 13-18*, 2016, pp. 573–584.

[37] R. G. Tinedo, P. G. L. and Marc Sánchez Artigas, J. Sampé, Y. Moatti, E. Rom, D. N. and Ramon Nou, T. Cortes, W. Oppermann, and P. Michiardi, "Iostack: Software-defined object storage," *IEEE Internet Computing*, vol. 20, no. 3, pp. 10–18, 2016.

[38] R. Nou, A. Miranda, M. Siquier, and T. Cortes, "Improving openstack swift interaction with the I/O stack to enable software defined storage," in *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC$^2$), Kanazawa, Japan, November 22-25*, 2017, pp. 63–70.

[39] OpenIO, "Openio fs architecture," https://docs.openio.io/latest/source/arch-design/fs_overview.html#label-oiofs-architecture, 2020, accessed on January 7, 2020.

[40] M. Szeredi and N. Rath, "Fuse (filesystem in userspace)," https://github.com/libfuse/libfuse, 2020, accessed on May 2, 2020.

[41] C. Lee, D. Sim, J. Y. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proceedings of the 13th USENIX Conference on*

*File and Storage Technologies (FAST), Santa Clara, CA, USA, February 16-19*, 2015, pp. 273–286.

[42] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *7th USENIX Conference on File and Storage Technologies (FAST), February 24-27, San Francisco, CA, USA. Proceedings*, 2009, pp. 153–166.

[43] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivie, "The new ext4 filesystem: current status and future plans," in *Proceedings of the 2007 Linux Symposium, Volume Two, Ottawa, Ontario, Canada*, 2007, pp. 21 – 32.

[44] N. Rath, "S3ql – a full-featured file system for online data storage," https://github.com/s3ql/s3ql, 2019, accessed on May 2, 2020.

[45] M. Vrable, S. Savage, and G. M. Voelker, "Bluesky: a cloud-backed file system for the enterprise," in *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST), San Jose, CA, USA, February 14-17*, 2012, p. 19.

[46] K.-H. Cheung, "Goofys," https://github.com/kahing/goofys, 2019, accessed on May 2, 2020.

[47] X. Lucas, "Svfs," https://github.com/ovh/svfs, 2019, accessed on May 2, 2020.

[48] P. Jonkins, "Riofs," https://github.com/skoobe/riofs, 2018, accessed on January 7, 2020.

[49] S. Patil and G. A. Gibson, "Scale and concurrency of GIGA+: file system directories with millions of files," in *9th USENIX Conference on File and Storage Technologies, San Jose, CA, USA, February 15-17, 2011*, 2011, pp. 177–190.

[50] S. Patil, K. Ren, and G. Gibson, "A case for scaling HPC metadata performance through de-specialization," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012*, 2012, pp. 30–35.

[51] K. Ren, Q. Zheng, S. Patil, and G. A. Gibson, "Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, 2014, pp. 237–248.

[52] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu, "An ephemeral burst-buffer file system for scientific applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, 2016, pp. 807–818.

[53] J. Xing, J. Xiong, N. Sun, and J. Ma, "Adaptive and scalable metadata management to support a trillion files," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA*, 2009.

[54] A. Brinkmann, K. Mohror, W. Yu, P. H. Carns, T. Cortes, S. Klasky, A. Miranda, F. Pfreundt, R. B. Ross, and M. Vef, "Ad hoc file systems for high-performance computing," *J. Comput. Sci. Technol.*, vol. 35, no. 1, pp. 4–26, 2020.

[55] M. Vef, N. Moti, T. Süß, M. Tacke, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "Gekkofs - A temporary burst buffer file system for HPC applications," *J. Comput. Sci. Technol.*, vol. 35, no. 1, pp. 72–91, 2020.

[56] D. Poccia, "Yas3fs: Yet another s3-backed file system," https://github.com/danilop/yas3fs, 2019, accessed on May 2, 2020.

[57] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: performance of user-space file systems," in *15th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 27 - March 2*, 2017, pp. 59–72.

[58] T. Haynes, "Network file system (NFS) version 4 minor version 2 protocol," *RFC*, vol. 7862, 2016. [Online]. Available: https://doi.org/10.17487/RFC7862

[59] F. B. Schmuck and R. L. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the Conference on File and Storage Technologies (FAST), January 28-30, Monterey, California, USA*, 2002, pp. 231–244.

[60] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *6th USENIX Conference on File and Storage Technologies (FAST), February 26-29, San Jose, CA, USA*, 2008, pp. 17–33.

[61] M. Vef, V. Tarasov, D. Hildebrand, and A. Brinkmann, "Challenges and solutions for tracing storage systems: A case study with spectrum scale," *ACM Transactions on Storage (TOS)*, vol. 14, no. 2, pp. 18:1–18:24, 2018.

[62] S. A. Wright, S. D. Hammond, S. J. Pennycook, R. F. Bird, J. A. Herdman, I. Miller, A. Vadgama, A. H. Bhalerao, and S. A. Jarvis, "Parallel file system analysis through application I/O tracing," *Computer Journal*, vol. 56, no. 2, pp. 141–155, 2013.

[63] J. Soumagne, D. Kimpe, J. A. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. B. Ross, "Mercury: Enabling remote procedure call for high-performance computing," in *IEEE International Conference on Cluster Computing (CLUSTER), Indianapolis, IN, USA, September 23-27*, 2013, pp. 1–8.

[64] P. H. Carns, J. Jenkins, C. D. Cranor, S. Atchley, S. Seo, S. Snyder, and R. B. Ross, "Enabling NVM for data-intensive scientific services," in *4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, INFLOW@OSDI, Savannah, GA, USA, November 1*, 2016.

[65] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. H. Carns, A. Castelló, D. Genet, T. Hérault, S. Iwasaki, P. Jindal, L. V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. H. Beckman, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2018.

[66] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb," in *8th Biennial Conference on Innovative Data Systems Research (CIDR), Chaminade, CA, USA, January 8-11, Online Proceedings*, 2017.

[67] K. Rarick, "Beanstalk is a simple, fast work queue," https://github.com/beanstalkd/beanstalkd, 2020, accessed on May 2, 2020.

[68] O. Ben-Kiki, C. Evans, and B. Ingerson, "Yaml ain't markup language version 1.2, 3rd edition," https://yaml.org/spec/1.2/spec.html, 2009, accessed on May 2, 2020.

[69] W. R. Pearson, "Rapid and sensitive sequence comparison with fastp and fasta," *Methods in enzymology*, vol. 183, pp. 63–98, 1990.

[70] B. Langmead, "Aligning short sequencing reads with bowtie," *Current protocols in bioinformatics*, vol. 32, no. 1, pp. 11–7, 2010.

[71] "National center for biotechnology information: Sequence read archive," https://www.ncbi.nlm.nih.gov/sra, 2020, accessed on May 2, 2020.