FSMonitor: Scalable File System Monitoring for Arbitrary Storage Systems

Arnab K. Paul*, Ryan Chard†, Kyle Chard‡, Steven Tuecke‡,
Ali R. Butt*, Ian Foster†,‡
*Virginia Tech, †Argonne National Laboratory, ‡University of Chicago
{akpaul, butta}@vt.edu, {rchard, foster}@anl.gov, {chard, tuecke}@uchicago.edu

***Abstract*—Data automation, monitoring, and management tools are reliant on being able to detect, report, and respond to file system events. Various data event reporting tools exist for specific operating systems and storage devices, such as `inotify` for Linux, `kqueue` for BSD, and `FSEvents` for macOS. However, these tools are not designed to monitor distributed file systems. Indeed, many cannot scale to monitor many thousands of directories, or simply cannot be applied to distributed file systems. Moreover, each tool implements a custom API and event representation, making the development of generalized and portable event-based applications challenging. As file systems grow in size and become increasingly diverse, there is a need for scalable monitoring solutions that can be applied to a wide range of both distributed and local systems. We present here a generic and scalable file system monitor and event reporting tool, `FSMonitor`, that provides a file-system-independent event representation and event capture interface. `FSMonitor` uses a modular Data Storage Interface (DSI) architecture to enable the selection and application of appropriate event monitoring tools to detect and report events from a target file system, and implements efficient and fault-tolerant mechanisms that can detect and report events even on large file systems. We describe and evaluate DSIs for common UNIX, macOS, and Windows storage systems, and for the Lustre distributed file system. Our experiments on a 897 TB Lustre file system show that `FSMonitor` can capture and process almost 38 000 events per second.**

I. Introduction

Big Data science relies on sophisticated workflows that encompass a wide variety of storage locations as data flows from instruments to processing resources and archival storage. Each storage location may be completely independent of one another, managed in separate administrative domains and providing heterogeneous storage interfaces such as hierarchical POSIX stores, high-performance distributed storage, persistent tape storage, and cloud-based object stores. Data may be stored for short to long periods on each, be accessible to dynamic groups of collaborators, and be acted upon by various actors. As data generation volumes and velocities continue to increase, the rate at which files are created, modified, deleted, and acted upon (e.g., permission changes) make manual oversight and management infeasible.

While research automation [6], [11] offers a potential solution to these problems it is first necessary to be able to capture events in real-time and at scale across a range of storage systems and via heterogeneous interfaces. Such events may then be used to automate the data lifecycle (performing backups, purging stale data, etc.), report usage and enforce restrictions, enable programmatic management, and even autonomously manage the health of the system. Enabling scalable, reliable, and standardized event detection and reporting will also be of value to a range of infrastructures and tools, such as Software Defined CyberInfrastructure (SDCI) [14], auditing [9], and automating analytical pipelines [11]. Such systems enable automation by allowing programs to respond to file events and initiate tasks.

Most storage systems provide mechanisms to detect and report data events, such as file creation, modification, and deletion. Tools such as inotify [20], kqueue [18], and FileSystemWatcher [21] enable developers and applications to respond to data events. However, these monitoring tools do not support distributed storage systems, such as Lustre [4] and IBM's Spectrum Scale [26] (formally known as GPFS). Instead, distributed file systems often maintain an internal metadata collection, such as Lustre's Changelog, which enables developers to query data events. However, each of these tools provides a unique API and event description language. For example, a file creation action may generate an *IN_CREATE* event with inotify, a *Created* event with FileSystemWatcher, and a *01CREAT* event in Lustre's Changelog. Further, many monitoring tools are designed to monitor specific directories, not the entire file system. For example, inotify's default configuration can monitor approximately 512 000 directories concurrently. While this number can be increased on machines with considerable memory resources, the inability to recursively monitor directories restricts its suitability for the largest storage systems.

In this paper, we present a generalized, scalable, storage system monitor, called `FSMonitor`. `FSMonitor` provides a common API and event representation for detecting and managing data events from different storage systems. We have architected `FSMonitor` with a modular Data Storage Interface (DSI) layer to plug-in and use custom monitoring tools and services. To support the largest storage systems, such as those deployed at supercomputing centers, we have developed a scalable monitor architecture that is capable of detecting, resolving, and reporting events from Lustre's Changelog. We chose Lustre as our implementation platform because Lustre is used by 60% of the top 10 supercomputers [5]. While our scalable monitor has so far only been applied to Lustre stores, its design makes it applicable to other distributed data stores with metadata catalogs, such as IBM's Spectrum Scale.

978-1-7281-4734-5/19/$31.00 ©2019 IEEE

We present both `FSMonitor` and our scalable monitor. We discuss `FSMonitor`'s architecture and demonstrate its ability to reliably detect, process, and report events. We describe the scalable monitor and its optimizations, such as caching mechanisms, and show that it can capture site-wide events on leadership-scale storage devices. Finally, we evaluate both `FSMonitor` and the scalable monitor, and demonstrate its capabilities by using it to drive a research automation system. Our experiments show that `FSMonitor` scales well on a 897 TB Lustre store, where it is able to process and report 37 948 events per second, while providing a standard event representation on various storage systems.

II. Background and Related Work

We introduce and describe several tools commonly used to monitor and detect data events. We also describe a distributed file system, Lustre, and how the Changelog metadata catalog can be used to detect and report events.

A. Data Event Detection

Here we describe and evaluate `inotify`, `kqueue`, `FSEvents`, and `FileSystemWatcher`.

inotify [20] is a Linux kernel feature that provides a mechanism to monitor file system events. It is used to monitor individual files as well as directories. The inotify API provides the following three system calls.

- **inotify_init()** creates an instance of inotify and returns a file descriptor.
- **inotify_add_watch()** adds a watch associated with an inotify instance. A watch specifies the path of a file or a directory to be monitored along with the set of events which the kernel should monitor on that path.
- **inotify_rm_watch()** removes a watch from the inotify watch list.

A set of command line tools, called inotify-tools [19], provide an interface to simplify use of inotify. It consists of inotifywait and inotifywatch. inotifywait efficiently waits for changes to files and outputs events as they occur. inotifywatch listens for file system events and outputs a table summarizing event information for the given path over a period of time. A key limitation of inotify is that it does not support recursive monitoring, requiring a unique watcher to be placed on each directory of interest. This is restrictive as each watcher requires 1KB of memory. In addition, deployment of many watchers is costly as the process must recursively crawl the file system. Another drawback of inotify monitor is that it may suffer a queue overflow error if events are generated faster than they are read from the queue.

Opening, creating, and modifying a file in a watched directory causes *IN_OPEN, IN_CREATE*, *IN_MODIFY*, and *IN_CLOSE* events to be raised, respectively.

kqueue [18] is a scalable event notification interface in FreeBSD and also supported in NetBSD, OpenBSD, DragonflyBSD, and macOS. Its API includes the following functions.

- **kqueue()** creates a new kernel event queue and returns a descriptor.
- **kevent()** is a system call used to register events with the queue, and returns the number of events placed in the eventlist.
- **EV_SET()** is a macro provided for ease of initializing a kevent structure.

The kqueue monitor requires a file descriptor to be opened for every file being watched, restricting its application to very large file systems. Opening, creating, and modifying a file results in *NOTE_OPEN*, *NOTE_EXTEND*, *NOTE_WRITE*, and *NOTE_CLOSE* events, respectively.

FSEvents [7] is another event notification API for macOS. The kernel passes raw event notifications to the userspace via a device file called */dev/fsevents*, where a daemon filters the event stream and advertises notifications. The FSEvents API is available on macOS versions 10.7 and greater. The FSEvents monitor is not limited by requiring unique watchers and thus scales well with the number of directories observed. Creating and modifying a file will result in *ItemCreated* and *ItemModified* events.

The **FileSystemWatcher** class [21] in the *System.IO* namespace listens to file system changes and raises events for Windows. A new FileSystemWatcher instance is created with arguments to specify the directory and type of files to monitor, and the buffer into which file change reports are to be written. The operating system then reports file changes by writing to that buffer. The buffer can overflow when many file system changes occur in a short period of time, causing event loss. The monitor can only establish a watch to monitor directories, not files. To monitor a file, its parent directory must be watched in order to receive change events for all of the directory's children. Four event types are reported: *Changed*, *Created*, *Deleted*, and *Renamed*.

Discussion: The tools to monitor different file systems are designed with various APIs, event representations, and degrees of reliability. The vast difference in the different tools' event definitions suggests a need for a file system monitoring tool that standardizes the event representation.

Libraries have been created to simplify programming against the wide range of monitoring tools. For example, the Python Watchdog module [25] simplifies the use of several commonly used monitoring tools, including inotify, FSEvents, kqueue, and (for Windows) ReadDirectoryChangesW with I/O completion ports and ReadDirectoryChangesW with worker threads. Watchdog provides a standard API for developers to select and deploy a monitor. Facebook's Watchman [13], FSWatch, and FSMon [29] are similar to Python's Watchdog module in that they also provide a common interface for initiating different monitors. However, each of these tools relies on operating system facilities for file system event notification and cannot be easily extended with custom monitoring tools. In addition, while these aggregate libraries provide a standard interface for deploying monitors, they do not standardize the event representation.

`FSMonitor` uses the Watchdog library to monitor many storage systems, and then layers on additional capabilities to standardize representations and increase reliability.

B. Distributed File System

A distributed file system is designed to distribute file data across multiple servers so that multiple clients can access a file system simultaneously. Typically, it consists of *clients* that read or write data to the file system, *data servers* where data is stored, *metadata servers* that manage the metadata and placement of data on the data servers, and networks to connect these components. Data may be distributed (divided into stripes) across multiple data servers to enable parallel reads and writes. This level of parallelism is transparent to the clients, for whom it seems as though they are accessing a local file system. Therefore, important functions of a distributed file system include avoiding potential conflict among multiple clients, and ensuring data integrity and system redundancy.

Monitoring a distributed file system is challenging as events may be generated across various components in the system and they then pass through one of several metadata servers. Generally, the larger the data store, the more metadata servers required to manage the cluster. Further, as a distributed file system aims to deliver transparent parallelism, it is crucial that the event monitoring mechanisms look to users as if they are the same as a local file system.

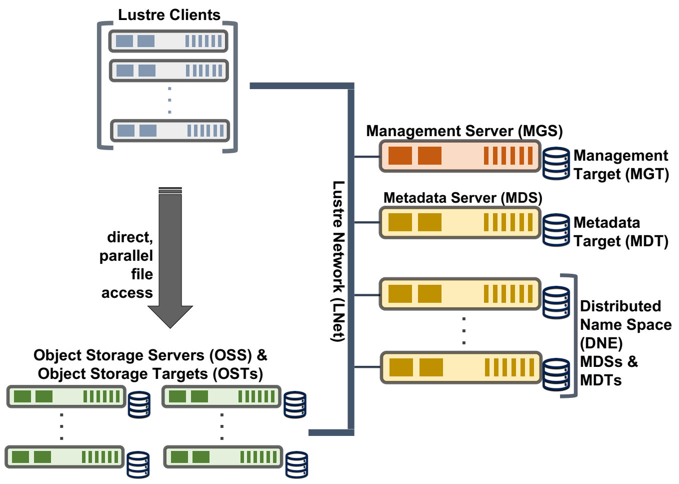
Lustre Clients
direct, parallel file access
Object Storage Servers (OSS) & Object Storage Targets (OSTs)
Lustre Network (LNet)
Management Server (MGS)
Management Target (MGT)
Metadata Server (MDS)
Metadata Target (MDT)
Distributed Name Space (DNE) MDSs & MDTs

Fig. 1: An overview of Lustre architecture.

1) Lustre Distributed File System: Figure 1 depicts the architecture of the Lustre file system. Lustre is designed for high performance, scalability, and high availability, and employs a client-server network architecture. A Lustre depoyment is comprised of one or more Object Storage Servers (OSSs) that store file contents; one or more Metadata Servers (MDSs) that provide metadata services for the file system and manage the Metadata Target (MDT) that stores the file metadata; and a single Management Server (MGS) that manages the system.

The *Management Server* (MGS) is responsible for storing the configuration information for the entire Lustre file system. This persistent information is stored on the *Management Target* (MGT).

Metadata Servers (MDS) manages namespace operations and are responsible for one *Metadata Target* (MDT). Namespace metadata, such as directories, filenames, file layouts, and access permissions are stored in the associated MDT. Every Lustre file system must have at least one MDT. A large file system may require more than one MDT to store its metadata, and therefore Lustre versions after 2.4 support a distributed namespace. A distributed namespace means metadata can be spread across multiple MDTs. MDS_0 and MDT_0 act as the root of the namespace, and all other MDTs and MDSs act as their children.

Lustre provides a *Changelog* to query the metadata stored in the MDT. Developers can create a Changelog listener and subscribe to a specific MDT, allowing them to retrieve and purge metadata records.

Object Storage Servers (OSS) store file contents. Each file is stored on one or more *Object Storage Targets* (OST) mounted on an OSS. Applications access file system data via Lustre clients which interact with OSSs directly for parallel file accesses. The internal high-speed data networking protocol for Lustre file system is abstracted and is managed by the Lustre Network layer.

2) Monitoring Distributed File Systems: Monitoring Lustre file system requires the use of specialized tools [22], of which Robinhood [17] is the most widely used. The Robinhood Policy Engine is capable of collecting events from Lustre file systems and using them to drive a policy engine to automate data management tasks, such as purging old files. Its architecture is shown in Figure 2.

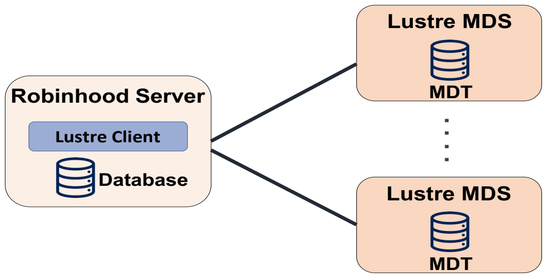

Fig. 2: Robinhood Architecture.

Robinhood uses an iterative approach to collect event data from metadata servers. A Robinhood server runs on the Lustre client and queries each MDS for events by querying the Changelogs. It then saves the events in a database on the Lustre client. For multiple MDSs, Robinhood polls all MDSs one at a time in a round robin fashion. Our approach in `FSMonitor` differs to this and employs a high performance message passing queue to concurrently collect, report, and aggregate events from each MDS. With this approach, `FSMonitor` is capable of far exceeding the performance of Robinhood for event collection.

Lustre is representative of many other distributed file systems. For example, IBM Spectrum Scale has a file audit logging capability from version 5.0. Spectrum Scale File Audit Logging takes locally generated file system events and puts them on a multi-node message queue from which they are consumed and written to a retention enabled fileset. Therefore, `FSMonitor` can be extended to build a scalable monitoring solution for Spectrum Scale in addition to Lustre and other file systems.

Summary: We need a file system event monitoring solution that can be applied to both local file systems (Linux, macOS, Windows) as well as distributed file systems (Lustre). The monitor needs to have a standard event representation. In `FSMonitor`, we standardize all event representations to the inotify format as this is the most widely used in industries [2]. The monitor also needs to be scalable so as to be able to capture all the events in a distributed file system. We implement our scalable solution for Lustre file system.

III. FSMonitor System Design

Designing a data event monitoring solution that can be applied to both local and distributed file systems is non-trivial as existing monitoring solutions cannot be generally applied. `FSMonitor` provides a standard event detection interface across file systems, implements capabilities to capture events from distributed file systems, and extends the reliability of the underlying event detection system by providing additional resiliency capabilities. `FSMonitor` is designed to be applied to arbitrary storage systems through a modular data storage interface which can be implemented to connect to arbitrary event interfaces. It provides a standardized API to collect and process file system events, and is a scalable and high-performance system that can be applied to large distributed file systems that generate many thousands of events per second.

A. Architecture

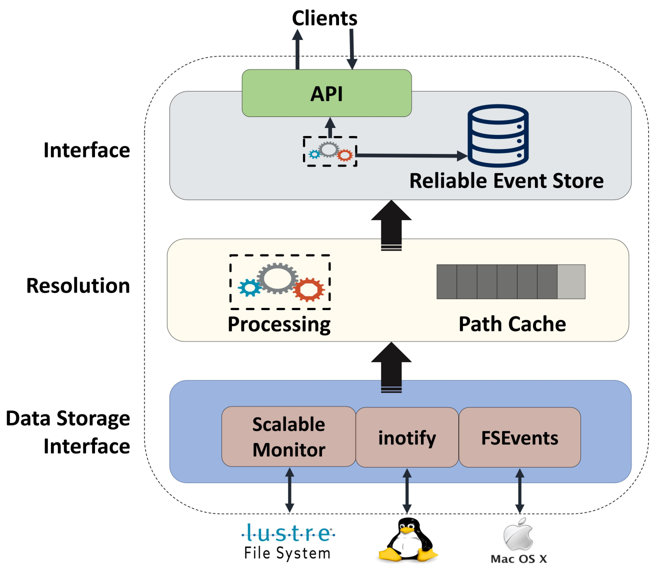

Fig. 3: An overview of `FSMonitor`'s architecture depicting the interface, resolution, and DSI layers.

Figure 3 illustrates the three-layer architecture of `FSMonitor`. The *Data Storage Interface* (DSI) layer provides an abstraction for interfacing with different storage systems. The *resolution* layer is used to process events and standardize their format, and the *interface* layer enables client users and programs to communicate with `FSMonitor`. Here we describe each of these layers in detail, explaining their design and capabilities.

1) Data Storage Interface: The lowest level of `FSMonitor` is responsible for interfacing with the underlying file system to capture events and report them to the resolution layer for processing. We employ a modular architecture via which arbitrary monitoring interfaces can be integrated into `FSMonitor`. We provide a set of DSIs to interface with the most common file systems. These DSIs implement our abstract interface and facilitate event extraction from `kqueue`, `FSEvents`, `inotify`, and `Windows FileSystemWatcher`, as well as a custom, scalable event monitoring solution that can be applied to the high performance distributed file systems, Lustre. Our DSI layer is also responsible for selecting the appropriate monitoring tool for the given storage device. We implement many of the local monitoring DSI's using the Python Watchdog module [25]. After the events have been collected, they are propagated to the resolution layer.

2) Resolution: The resolution layer provides a multi-faceted approach to reliably recording and aggregating events from the DSIs and then reporting them to the interface layer. This layer includes a queue to receive and manage events until they are processed. As events are received from a DSI plug-in they are immediately placed in the processing queue. The events are then processed to resolve and dereference paths such that events can be transformed into various representations. Rather than defining yet another event representation, we instead support transformation into any of the commonly defined formats (inotify, kqueue, FSEvents) by populating the appropriate event template. Thus, tools that are designed to respond to events in one of these different formats can be trivially used with `FSMonitor`. The resolution layer also provides optimizations that improve the overall event processing performance, such as caching and batching capabilities.

3) Interface: The topmost layer provides an interface for users and programs to interact with `FSMonitor`. This layer is responsible for reporting events and replying to requests. Programs can use the interface to capture events as they happen. The interface layer also provides batching capabilities to report events in groups. If users provide an event identifier, `FSMonitor` will only report events that have happened since that event. This layer is also responsible for providing fault-tolerance by storing all events received from the *resolution* layer into an event store (database). Once events have been retrieved from `FSMonitor`, they are flagged as having been reported and can be removed from the database. The size of this database is configurable and can be adjusted depending on the resources available to `FSMonitor`.

IV. Scalable Event Monitoring

We provide DSIs for many common event monitoring tools, including Linux, BSD, macOS, and Windows. However, Lustre, like many other distributed file systems, does not support these tools. Here we describe the design and implementation of the FSMonitor's scalable DSI for distributed file systems.

1) Lustre Changelog: TABLE I shows sample records in Lustre's Changelog. We run a script and see the events recorded in the Changelog. The script first creates a file, *hello.txt*, then the file is modified. The file is then renamed to *hi.txt*. A directory named *okdir* is then created. Finally, we delete the file.

Each tuple in Table I represents each of the file system events. Every row in the Changelog will have an *EventID* – the record number of the Changelog, *Type* – the file system event occurred, *Timestamp, Datestamp* – the date time of the event occurrence, *Flags* – masking for the event, *Target FID* – file identifier of the target file/directory on which the event occurred, *Parent FID* – file identifier of the parent directory of the target file/directory, and the *Target Name* – the file/directory name which triggered the event. It is evident that the Parent and Target FIDs need to be resolved to their original names before they can be sent to the client. The following events are recorded in the Changelog:

- **CREAT**: Creation of a regular file.
- **MKDIR**: Creation of a directory.
- **HLINK**: Hard link.
- **SLINK**: Soft link.
- **MKNOD**: Creation of a device file.
- **MTIME**: Modification of a regular file.
- **UNLNK**: Deletion of a regular file.
- **RMDIR**: Deletion of a directory.
- **RENME**: Rename a file or directory from.
- **RNMTO**: Rename a file or directory to.
- **IOCTL**: Input-output control on a file or directory.
- **TRUNC**: Truncate a regular file.
- **SATTR**: Attribute change.
- **XATTR**: Extended attribute change.

Note in Table I that Target FIDs are enclosed within *t = []*, and parent FIDs within *p = []*. *MTIME* event does not have a parent FID. *RENME* event has additional FIDs, *s = []* denoting a new file identifier to which the file has been renamed, and *sp = []* gives the file identifier for the original file. These features are important when resolving FIDs.

2) Design: Figure 4 shows an overview of our scalable DSI. Our monitor uses a publisher-subscriber model, where the metadata are acquired from each MDS's Changelog via a *collector*, processed and cached, then published to the *aggregator*. The aggregator gathers events from each collector for later consumption. The publisher-subscriber model provides scalability in the event collection process, which has been shown when monitoring Lustre server statistics [23], [24], [28]. We divide the overall design for the scalable monitor into four steps: *Detection, Processing, Aggregation* and *Consumption*. Below we describe each component of the scalable monitoring solution in greater detail.

Detection: Each collector service is responsible for extracting file system events from one MDS Changelog. Deploying collectors on individual MDSs enables every MDS to be

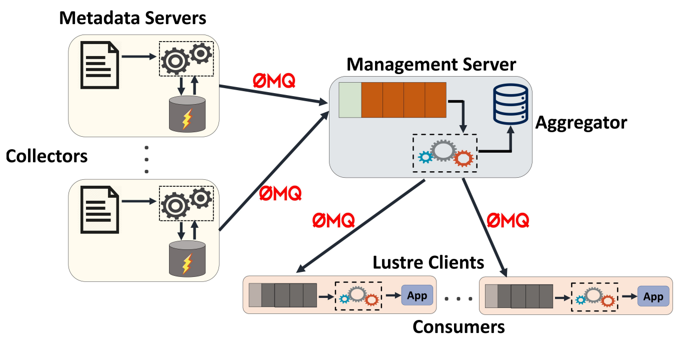

Fig. 4: The scalable Lustre monitor used to collect, aggregate, and publish events.

monitored in parallel. Every event that is extracted from the local Changelog needs to be processed before it can be published to the aggregator.

Processing: As seen in Section IV-1, every event recorded in the Changelog is associated with either a target or a parent file identifier (FID), or both. However, these FIDs are not necessarily interpretable by external applications, and thus must be processed and resolved to absolute path names.

The Lustre *fid2path* tool resolves FIDs to absolute path names. Whenever a new file system event is detected by a collector, we use these tool to convert raw event tuples into application-friendly file paths when the event is being published. However, the *fid2path* tool is slow and can delay the reporting of events. For example, in Section V-D2 we show that this delay can cause a decrease of 14.9% in the event reporting rate. To minimize this overhead, we implement the aggregator with a Least Recently Used (LRU) Cache to store mappings of FIDs to source paths.

Algorithm 47 shows the aggregator's processing steps. Events are processed in batches. The cache is used to resolve FIDs to absolute paths. Whenever an entry is not found in the cache, we invoke the *fid2path* tool to resolve the FID and then store the mapping *(fid – path)* into the LRU cache. In the case of UNLNK and RMDIR events, resolving target FIDs will give an error because that FID has already been deleted by the file system. Therefore, parent FIDs need to be resolved. If resolving parent FID raises an error, it means the parent directory has been deleted as well. FSMonitor resolves this and gives the event as 'ParentDirectoryRemoved'. As seen in Section IV-1, RENME events are provided with the old path (*sp = []*) and new path FIDs (*s = []*). Therefore, for RENME events, instead of target FID, old path and new path FIDs need to be resolved. Finally, events are published to the aggregator.

After processing a batch of file system events from the Changelog, a collector will purge the Changelogs. A pointer is maintained to the most recently processed event tuple and all previous events are cleared from the Changelog. This helps reduce the overburdening of the Changelog with stale events.

Aggregation: Collectors use a publisher-subscriber message queue (implemented with ZeroMQ [16]) to report events to an aggregator. When an event arrives to the aggregator it is

TABLE I: A sample ChangeLog record showing *Create File*, *Modify*, *Rename*, *Create Directory*, and *Delete File* events.

Event ID	Type	Timestamp	Datestamp	Flags	Target FID	Parent FID	Target Name
11332885	01CREAT	22:27:47.308560896	2019.03.08	0x0	t=[0x300005716:0x626c:0x0]	p=[0x300005716:0xe7:0x0]	hello.txt
11332886	17MTIME	22:27:47.327910351	2019.03.08	0x7	t=[0x300005716:0x626c:0x0]		hello.txt
11332887	08RENME	22:27:47.416587265	2019.03.08	0x1	t=[0x300005716:0x17a:0x0]	p=[0x300005716:0xe7:0x0]	hello.txt
						s=[0x300005716:0x626b:0x0]	
						sp=[0x300005716:0x626c:0x0]	hi.txt
11332888	02MKDIR	22:27:47.421587284	2019.03.08	0x0	t=[0x300005716:0x626d:0x0]	p=[0x300005716:0xe7:0x0]	okdir
11332889	06UNLNK	22:27:47.438587347	2019.03.08	0x0	t=[0x300005716:0x626b:0x0]	p=[0x300005716:0xe7:0x0]	hi.txt

```
Input: Lustre path $lpath$, Cache $cache$, MDT ID $mdt$
Output: $EventList$
while true do
    $events$ = read events from $mdt$ Changelog
    for event $e$ in $events$ do
        $resolvedEvent$ = processEvent($e$)
        $EventList$.add($resolvedEvent$)
    end
    Clear Changelog in $mdt$
    return ($EventList$)
end
Function processEvent
    Input: Event $e$
    Output: $resolvedEvent$
    Extract $event\_type, time, date$ from $e$
    try:
        $path$ = $cache$.get($targetFID$)
        if $targetFID$ not found in cache then
            $path$ = fid2path($targetFID$)
            $cache$.set($targetFID$, $path$)
        end
    catch $fid2pathError$:
        try:
            if $event\_type$ is UNLNK or RMDIR then
                $path$ = $cache$.get($parentFID$)
                if $parentFID$ not found in cache then
                    $path$ = fid2path($parentFID$)
                    $cache$.set($parentFID$, $path$)
                end
            end
            else if $event\_type$ is RENME then
                $oldpath$ = $cache$.get($oldFID$)
                $newpath$ = $cache$.get($newFID$)
                if $oldFID$ not found in cache then
                    $oldpath$ = fid2path($oldFID$)
                    $cache$.set($oldFID$, $oldpath$)
                end
                if $newFID$ not found in cache then
                    $newpath$ = fid2path($newFID$)
                    $cache$.set($newFID$, $newpath$)
                end
            end
        catch $fid2pathError$:
            if $event\_type$ is UNLNK or RMDIR then
                $path = ParentDirectoryRemoved$
            end
        end
    end
    $resolvedEvent$.add($event\_type, time, date,$
                        $path/oldpath\&newpath$)
    return ($resolvedEvent$)
```

Algorithm 1: Processing Changelog events.

placed in a processing queue. The aggregator service is multi-threaded, where one thread is responsible for publishing the aggregated file system events to the subscribed consumers, and the other thread stores the events into a local database to enable fault tolerance. This ensures minimal overhead on the system. An API is provided to the consumers to retrieve historic events from the database whenever a fault occurs. For our implementation, we use MySQL as the reliable event store.

Consumption: Consumers act as subscribers to the aggregator which publishes the events. Therefore, whenever a new event arrives to the consumer it filters the events and only passes on events related to those files and directories requested by the application. This filtering of events is not done at the aggregator in order to alleviate potential overheads if a large number of consumers were to ask to monitor different files and directories. The consumer service is also responsible for retrieving the historic events from a particular time stamp, from the reliable event store in the situation that a consumer has failed. Once events have been retrieved they are flagged as having been reported and can be removed from the data store when next data purge cycle is initiated.

V. Evaluation

To evaluate the performance and overhead of `FSMonitor`, we have deployed `FSMonitor` on multiple platforms including both Lustre distributed file system as well as macOS and Linux standalone environments. In this section, we first describe our testbeds, then we explain the workloads that we use for evaluating `FSMonitor` before evaluating `FSMonitor`'s performance.

A. Experimental Setup

We employ two types of testbed to evaluate `FSMonitor` on both local and Lustre file systems.

1) Local File Systems: We test `FSMonitor` on three systems: one for macOS, and two for Linux distributions.

- *macOS:* a MACBOOK Pro 2017 running macOS version 10.13.3, with 2.5 GHZ Intel Core i7, 16 GB 2133 MHz DDR3 memory, and 500 GB SSD storage.
- *Ubuntu:* a machine with Ubuntu 16.04 LTS, a 32 core 2 GHz AMD Opteron Processor 6370P, 64 GB memory, and 1 TB HDD storage.
- *CentOS:* a machine with CentOS 7.4, an 8-core AMD processor, 16 GB memory, and 512 GB HDD storage.

In addition, we employ two distinct workloads for macOS and Linux, one to test the uniformity in event definitions of `FSMonitor` which should be consistent to inotify, and the other to evaluate the performance of `FSMonitor` on local file systems. Both are implemented as Python scripts.

2) Distributed File Systems: We evaluate `FSMonitor` on three testbeds running Lustre.

- *AWS:* a deployment of Lustre on five Amazon Web Service EC2 instances. We build a `20 GB Lustre file system` using Lustre Intel Cloud Edition v 1.4 on five t2.micro instances and an EBS volume. Our Lustre configuration for AWS includes one MDS, one MGS, one OSS with one OST and two compute nodes.
- *Thor:* a deployment at the Distributed Systems and Storage Laboratory (DSSL) at Virginia Tech. Each node in the setup has 8-core processor, 16 GB memory and 512 GB storage. We deploy Lustre version 2.10.3 with one MGS, one MDS, ten OSSs each having five OSTs and two Lustre clients. Every OST is a 10 GB volume, resulting in a total of `500 GB Lustre store`.
- *Iota:* a production Lustre deployment on a pre-exascale system at Argonne National Laboratory with `897 TB`. This deployment has the same configuration and performance as the 150 PB store to be deployed on the Aurora supercomputer [1] at Argonne National Laboratory. The configuration includes Lustre's DNE with four MDSs. Iota comprises 44 compute nodes, each with 72 cores and 128 GB memory.

B. Experiment Workloads

We use a range of workloads to evaluate performance on the various testbeds.

For both local and Lustre deployments we use the following workloads:

- *Evaluate_Output_Script* is used to evaluate the types of event definitions that `FSMonitor` outputs. The script first creates a file *hello.txt*, then modifies it. It then renames the file from *hello.txt* to *hi.txt*. After that, it creates a new directory called *okdir*. Next, it moves *hi.txt* to the newly created directory *okdir*. Finally, it deletes the directory *okdir* and its contents.
- *Evaluate_Performance_Script* repeatedly creates, modifies, and deletes a file *hello.txt*, in an infinite loop. This script thus tests `FSMonitor` efficiency over a period of time and can be used to evaluate `FSMonitor`'s resource utilization and event reporting rate. This script is also used as a baseline for the Lustre distributed file system.

In order to test the scalability and performance of `FSMonitor` on Lustre we use the following workloads:

- *IOR*: The *InterleavedOrRandom* (IOR) [3] benchmark provides a flexible way to measure distributed file system's I/O performance. It measures performance with different parameter configurations, including I/O interfaces ranging from traditional POSIX to advanced parallel I/O interfaces like MPI-I/O. It performs reads and writes to and from files on distributed file systems, and calculates the throughput rates. For our evaluation, IOR is executed with single shared file mode and 128 processes.
- *HACC-I/O*: The *Hardware Accelerated Cosmology Code* (HACC) [15] application uses N-body techniques to simulate the formation of structure in collision-less fluids under the influence of gravity in an expanding universe. HACC-I/O captures the I/O patterns and evaluates the performance for the HACC simulation code. It uses the MPI-I/O interface and differentiates between FPP and SSF parallel I/O modes. We run HACC-IO for 4 096 000 particles under file-per-process mode with 256 processes for our experiments on `FSMonitor`.
- *Filebench*: This file system benchmark [27] can be used to generate a wide variety of workloads. Workloads can be described from scratch with the Workload Model Language, or existing workloads, from web server to database server, can be used with minor modifications. We used Filebench to create 50 000 files with sizes following a gamma distribution (mean 16 384 bytes and gamma 1.5), a mean directory width of 20, and mean directory depth of 3.6. The total size of all files generated is 782.8 MB. The benchmark had 1 445 858 operations with 24 095.141 operations per second.

C. *`FSMonitor`* on Local File Systems

As seen in Section II, Linux provides *inotifywait* [20] and macOS provides *FSEvents* [7] to monitor file system events. On macOS we compare `FSMonitor` with *FSWatch* [12], which uses `FSEvents`. We first compare the event definitions from `FSMonitor` with *FSWatch* and *inotifywait* and then we evaluate the performance.

TABLE II: File system events of `FSMonitor`.

`FSMonitor` on macOS and Linux
/home/arnab/test ***CREATE*** */hello.txt*
/home/arnab/test ***MODIFY*** */hello.txt* */home/arnab/test* ***CLOSE*** */hello.txt*
/home/arnab/test ***MOVED_FROM*** */hello.txt* */home/arnab/test* ***MOVED_TO*** */hi.txt*
/home/arnab/test ***CREATE,ISDIR*** */okdir*
/home/arnab/test ***MOVED_FROM*** */hi.txt* */home/arnab/test* ***MOVED_TO*** */okdir/hi.txt* */home/arnab/test* ***MODIFY,ISDIR*** */okdir* */home/arnab/test* ***CLOSE,ISDIR*** */okdir*
/home/arnab/test ***DELETE*** */okdir/hi.txt* */home/arnab/test* ***MODIFY,ISDIR*** */okdir* */home/arnab/test* ***CLOSE,ISDIR*** */okdir* */home/arnab/test* ***DELETE,ISDIR*** */okdir*

1) Output Analysis: The event definitions from `FSMonitor` from running *Evaluate_Output_Script* are shown in Table II. `FSMonitor` gives the same event definitions on both macOS as well as Linux environments. The results from `FSMonitor` is similar to `inotifywait` because `FSMonitor` by default standardizes event representation based on *inotify*. One of the major differences between `FSMonitor` in Linux and *inotify* is that `FSMonitor` has an additional option to monitor file system events on a path recursively while `inotifywait` does not output events happening inside subdirectories in a directory to be monitored. Therefore, if the path to be monitored is */dir*, and there are directories inside */dir*; *inotify* will not give file system events happening inside those directories but FSMonitor will produce those events. By default, FSMonitor will not monitor events recursively. The difference between FSMonitor and *inotify* is that to monitor events recursively, *inotify* requires watchers for every sub-directory but FSMonitor will monitor events recursively by just modifying the filtering rule in the Interface layer and thus will be able to monitor events more efficiently.

2) Performance Analysis: Here, we evaluate FSMonitor with respect to the *events report rate* when compared to *FSWatch* and *inotifywait*, and *resource utilization*. All results are the average of three runs.

TABLE III: Events reporting rate of FSMonitor, FSWatch and inotify.

	Events generated per second	**Events reported per second**	
		FSMonitor	*Other*
macOS	4503	4467	3004
Ubuntu	4007	3985	3997
CentOS	3894	3875	3878

Event Reporting Rate: In order to evaluate FSMonitor, we compare its event reporting rate with *inotifywait* and *FSWatch*. *Evaluate_Performance_Script* is used to obtain the results. Table III shows the baseline event generating performance for all three platforms. Using the script, we were able to generate 4503, 4007, and 3894 events per second on macOS, Ubuntu, and CentOS respectively. This was the system limitation rate of generating these events with our script. The *Other* column in Table III signifies *FSWatch* for macOS and *inotifywait* for Ubuntu and CentOS. We see that FSWatch performs poorly in comparison to FSMonitor on macOS. FSWatch reports only 3004 events per second, as against 4467 events per second by FSMonitor. On Ubuntu and CentOS, *inotifywait* performs a little better than FSMonitor. This is because of the minimal delay caused in the *interface* layer of FSMonitor due to the parsing of the path to be monitored.

TABLE IV: CPU and Memory usage of FSMonitor, FSWatch and inotify.

	CPU%		**Memory%**	
	FSMonitor	*Other*	*FSMonitor*	*Other*
macOS	0.1	0.1	0.01	0.01
Ubuntu	0.4	0.3	0.01	0.01
CentOS	0.2	0.3	0.01	0.01

Resource Utilization: We ran *Evaluate_Performance_Script* on all the local file systems in order to measure FSMonitor CPU and memory utilization. Table IV shows the CPU and memory usage of the monitors on the macOS, Ubuntu, and CentOS platforms. The *Other* column is *FSWatch* for macOS and *inotifywait* for Ubuntu and CentOS. All results are the average of three runs. CPU and memory utilization are calculated as the average utilization while FSMonitor, *FSWatch*, or *inotifywait* were executing. It is evident that no monitor makes heavy use of machine resources. The performance differences are not sufficiently high to allow us to conclude that any monitor is worse than any other. FSMonitor performs at par with the other popular local file system monitors in terms of resource usage.

D. FSMonitor on Lustre Distributed File System

We now explore FSMonitor's performance when deployed on several Lustre clusters.

1) Event capture rate: As when evaluating on a local file system, we first form a baseline throughput for Lustre on each of the three testbeds, *AWS*, *Thor*, and *Iota*. We use *Evaluate_Performance_Script* and record the number of events generated per second. As shown in Table V, *AWS* performs the worst, as its Lustre testbed is formed from t2.micro instances. *Thor* performs better than *AWS* and, as expected, *Iota*'s performance is the best of the three.

TABLE V: Lustre Testbed Baseline Event Generation Rates.

	AWS	**Thor**	**Iota**
Storage Size	20 GB	250 GB	897 TB
Create events/sec	352	746	1389
Modify events/sec	534	1347	2538
Delete events/sec	832	2104	3442
Total events/sec	1366	4509	9593

As *AWS* and *Thor* have only one MDS, FSMonitor extracts their events from only one MDS Changelog, processes them, and then communicates them to the MGS for reporting to the client. On *Iota*, events are generated from all four MDSs and thus need to be collected from all the MDSs and then aggregated on the MGS before they are sent to the consumer.

TABLE VI: Lustre Testbed Baseline Event Reporting Rates.

	AWS	**Thor**	**Iota**
Generated events/sec	1366	4509	9593
Reported events/sec *without cache*	1053	3968	8162
Reported events/sec *with cache*	1348	4487	9487

2) Event Reporting Analysis: As seen in Table VI, *AWS* generates over 1300 events per second, *Thor* around 4500 events per second, and *Iota* over 9500 events per second. When FSMonitor is used to report these events without caching the *fid2path* resolutions, the *AWS*-based FSMonitor can collect, process, and report only 1053 events per second. On *Iota*, it reports only 8162 events per second: 14.9% lower than the generation rate of $\sim$9500 events per second. When we analyze the FSMonitor event reporting pipeline, we notice that the performance is limited primarily in the processing of events after extracting them from the Changelog in the MDS. *fid2path* is costly and executing it for every event reduces overall throughput. Therefore, caching of *fid2path* mappings and batching events are necessary in such deployments.

We use an in-memory LRU cache to store the *fid2path* mappings. The size of the cache (number of *fid2path* mappings) is selected to be 5000 (we explore different cache sizes in Section V-D4). With the use of an in-memory cache, *AWS*-based FSMonitor is able to report 1348 events per second, FSMonitor on *Thor* is able to process and report 4487 events per second and on *Iota*, 9487 events per second. The loss in event reporting rate is due to the minimal processing required in FSMonitor.

The above results all use one MDS. On Iota, when we use all four available MDSs, the overall event generation rate is 38 372 events per second. FSMonitor reports 37 948 events per second to the consumer. Note that besides processing and filtering events, there is no additional overhead in the collection, aggregation, and reporting of events by FSMonitor. Also, there is no overall loss of events; events are queued and simply processed at a lower rate than they are generated.

3) Resource Utilization: We evaluate the resource utilization of every component of FSMonitor. *Evaluate_Performance_Script* is used to perform this analysis. Table VII shows the peak CPU and memory utilization on each of our three Lustre testbeds. From the results, it is evident that the CPU and memory costs of operating FSMonitor are low.

TABLE VII: FSMonitor Resource Utilization.

	CPU%			**Memory (MB)**		
	AWS	**Thor**	**Iota**	**AWS**	**Thor**	**Iota**
Collector - No cache	9.3	7.8	6.67	8.2	33.7	81.6
Collector with cache	6.6	1.5	2.89	9.92	25.7	55.4
Aggregator	2.7	0.57	0.06	5.7	7.2	17.6
Consumer	1.5	0.23	0.02	0.05	0.2	2.8

We show resource collector utilization both with and without a cache. The reduction in CPU utilization when caching is enabled is due to the lower number of *fid2path* invocations.

Next, we modified *Evaluate_Performance_Script* to include continuous creation and deletion of files without modification. We noticed that the Collector service on *Iota* had a CPU usage of 3.3%: a 12.4% increase in CPU usage when *Evaluate_Performance_Script* was tested. This is because delete events caused the *fid2path* mapping in the cache to fail for most events and result in *fid2path* calls on the parent directory. The memory usage did not change significantly.

We also changed *Evaluate_Performance_Script* to include only creation and modification of files, without deletion. CPU usage in this case was 2.3%: a decrease of 21.5% from *Evaluate_Performance_Script* testing. This is because more frequent mappings in the cache were found. Even in this scenario, memory usage did not differ significantly. The resource utilization of the Aggregator and Consumer stays the same even when caching is enabled.

Therefore, deploying FSMonitor on MDS, MGS and Lustre clients would result in a negligible overload on the overall performance.

4) Caching: In order to calculate the optimum size for the in-memory LRU cache we ran FSMonitor on *Iota* with *Evaluate_Performance_Script* and varied the cache size before every run. The results are shown in Table VIII.

The total number of events generated per second on *Iota* for one MDS is 9593, see in Table V. As seen in Table VIII, we observe improved performance using the in-memory LRU cache with size greater than or equal to 1000. For sizes 200 and 500, memory utilization on collector is even worse than than in FSMonitor without cache on *Iota*. The lowest CPU and memory utilization for Collector on *Iota* is for cache size 5000. Also, the number of events reported per second on one MDS is best for cache size of 5000. Increasing the cache size further to 7500 results in worse performance. Therefore, for our evaluation, we choose a cache size of 5000.

TABLE VIII: FSMonitor performance vs. cache size.

Cache Size (#fid2path)	**CPU% on collector**	**Memory (MB) on collector**	**Events/sec reported by each collector**
200	4.8	88.7	8644
500	3.5	84.3	8997
1000	2.98	75.6	9401
2000	2.95	61.3	9453
5000	2.89	55.4	9487
7500	2.92	60.7	9481

5) Comparison with Robinhood: For the case of *Iota* with four MDSs, we implement Robinhood [17] by having a subscriber in the client that polls the four publishers on MDS one at a time in a round-robin fashion. There is no role for MGS in this implementation. In comparison, FSMonitor has an aggregator service on MGS that polls all MDSs concurrently and pushes all events in a single queue to the clients. We evaluate performance by comparing the number of events per second received and processed at the client side by Robinhood vs. the number of events collected by the client via FSMonitor where the processing takes place at the MDSs and aggregation at the MGS. Our results show that Robinhood on *Iota* processes an average 7486 events per second from each MDS vs. 9847 events per second by FSMonitor. Combining all four MDSs, Robinhood processes 32 459 events per second in comparison to 37 948 events per second with FSMonitor. Therefore, with the advent of exascale supercomputers, where multiple MDSs with Lustre DNE will be common, parallel monitoring is necessary.

6) Benchmarks & Real World Applications: We evaluate FSMonitor on *Thor* using three workloads *HACC-I/O*, *Filebench* and *IOR*. We run all workloads simultaneously on the Lustre clients on the *Thor* testbed. We use FSMonitor to monitor the file system events on one client. We monitor $\backslash mnt \backslash lustre$ path. Table IX shows the file system events that is monitored by FSMonitor. We did not notice any delay in the event reporting procedure by FSMonitor when the three applications were executing simultaneously.

As *IOR* was executed in single-shared-file mode, only one *Create* and *Delete* file events were generated from IOR. *HACC-I/O* on the other hand was run in file-per-process mode for 256 processes. Therefore, 256 files were *created* and *deleted*. These file system events were correctly reported by FSMonitor. *Filebench* was used to create 50 000 files and therefore FSMonitor reports 50 000 *creates*. As seen in Table IX, all *Create* events are reported first for three applications and the *Delete* events for *IOR* and *HACC-I/O* are reported by FSMonitor.

VI. Use Cases

FSMonitor enables the development of various event-driven applications. Here we describe two use cases that make

TABLE IX: `FSMonitor` events for IOR, HACC-IO and Filebench.

`FSMonitor` events
/mnt/lustre **CREATE** /hacc-io/FPP1-Part00000000-of-00000256.data
/mnt/lustre **CLOSE** /hacc-io/FPP1-Part00000000-of-00000256.data
/mnt/lustre **CREATE** /hacc-io/FPP1-Part00000255-of-00000256.data
/mnt/lustre **CLOSE** /hacc-io/FPP1-Par00000255of-00000256.data
/mnt/lustre **CREATE** /ior/src/testFileSSF
/mnt/lustre **CLOSE** /ior/src/testFileSSF
/mnt/lustre **CREATE** /hacc-io//bigfileset/00000001
/mnt/lustre **CLOSE** /bigfileset/00000001
/mnt/lustre **CREATE** /bigfileset/00000...
/mnt/lustre **CLOSE** /bigfileset/00000...
/mnt/lustre **DELETE** /hacc-io/FPP1-Part00000000-of-00000256.data
/mnt/lustre **CLOSE** /hacc-io/FPP1-Part00000000-of-00000256.data
/mnt/lustre **DELETE** /hacc-io/FPP1-Part00000255-of-00000256.data
/mnt/lustre **CLOSE** /hacc-io/FPP1-Part00000255-of-00000256.data
/mnt/lustre **DELETE** /ior/src/testFileSSF
/mnt/lustre **CLOSE** /ior/src/testFileSSF

use of `FSMonitor` to implement applications and enable autonomous science and online, responsive file catalogs.

A. Research Automation

Scientific data generation is quickly becoming unmanageable at the human level. Instead, new tools and techniques are required to automate research process to support researchers' activities. Examples include pre-processing and quality control, analysis, replication, managing access control, and cataloging. As research becomes increasingly dependent on compute- and data-intensive analyses, the role of data-oriented automation becomes ever more important.

Rule-based systems, such as Robinhood and Globus Automate [6], enable users to apply actions in response to data events. Globus Automate is a flexible platform that allows users to define data management and manipulation pipelines across distributed resources. A pipeline, or *flow*, is comprised of a series of steps, where each step represents an invocation of a remote Web service, such as Globus Transfer, a catalog service, or a remote execution service.

As previously discussed, the application of such systems to a wide variety of storage systems is non-trivial. As such, we have developed a client to enable Globus Automate flows to be initiated in response to data events. The client incorporates both `FSMonitor` and Globus Auth to detect data events and securely initiate flows in response. When a data event is captured by `FSMonitor`, our client constructs a JSON document of metadata, such as the file type, size, owner, and location and transmits the data to a pre-defined Globus Automate flow. The flow is then reliably executed. We have used Globus Automate to perform a wide range of scientific data management tasks, such as performing on-demand analysis from synchrotrons [10] and publishing materials science datasets.

B. Responsive Cataloging

Being able to search and find data within local storage is a crucial tool in the scientific process. However, as storage systems grow to manage hundreds of petabytes and even exabytes of data, the cost to crawl and index the data is likely to become increasingly prohibitive. New techniques are required to populate and maintain catalogs of data as hundreds of users concurrently create, modify, move, and delete data. Event-based cataloging is necessary to enable flexible management of these extreme-scale stores. Combining event detection with metadata extraction and cataloging services provides a new avenue to enabling search capabilities over research data.

Combining `FSMonitor` with a metadata extraction tool, such as Skluma [8], can enable the dynamic cataloging of large research data. Skluma provides a suite of metadata extraction tools that can be applied to data. Skluma uses a pipeline to dynamically infer file types and passes them through selected metadata extractors in order to extract and derive metadata, such as the types and ranges of tabular data, keywords from free text, and scale and content information from plots and figures. We have integrated `FSMonitor` as a trigger source for Skluma such that we can apply metadata extraction containers to data as they are created. In addition, we can capture data movement and deletion events to dynamically modify a Globus Search index and maintain a useful, up-to-date catalog.

VII. Conclusion

We have presented `FSMonitor`, a scalable file system monitor that can report events for both local (Linux, macOS, Windows) and distributed (Lustre) file systems. `FSMonitor` uses a three-layer approach to file system event monitoring. The lowest layer, *DSI*, interacts with a file system to detect events and sends them to the middle layer, *Resolution*, where events are resolved to their absolute path names and aggregated to be sent to the upper layer, *Interface*, where aggregated events are stored in an event store. `FSMonitor` offers an API that allows clients to retrieve events from the event store.

`FSMonitor` implements a standard event definition process for any file system, and works seamlessly for both local and distributed file systems. On local file systems, `FSMonitor` achieves better or at par event reporting rates with low CPU and memory overhead, when compared to other monitoring tools. We also evaluated `FSMonitor` on three testbeds for Lustre file system, and we found that on a 897 TB Lustre system, `FSMonitor` reported almost 38 000 events per second with low resource utilization. Also, compared to iterative monitoring methods used by the popular Robinhood system, `FSMonitor` gives 14.5% improved performance in event reporting rate for multiple MDSs.

Acknowledgments

This research used resources of the Argonne Leadership Computing Facility, a Department of Energy Office of Science User Facility supported under Contract DE-AC02-06CH11357. We acknowledge generous research credits provided by Amazon Web Services. This work is also sponsored in part by the National Science Foundation under grants CNS-1405697, CNS-1615411, CNS-1565314/1838271 and OAC 1835890.

References

[1] Aurora Supercomputer. https://aurora.alcf.anl.gov/. Accessed: August 26 2019.

[2] Ian Shields, IBM - Monitor Linux file system events with inotify. https://developer.ibm.com/tutorials/l-inotify/. Accessed: March 7 2019.

[3] LLNL - IOR Benchmark. https://asc.llnl.gov/sequoia/benchmarks/IORsummaryv1.0.pdf. Accessed: March 11 2019.

[4] OpenSFS and EOFS - Lustre file system. http://lustre.org/. Accessed: March 23 2019.

[5] Top 500 List - November 2018. https://www.top500.org/lists/2018/11/. Accessed: March 2019.

[6] R. Ananthakrishnan, B. Blaiszik, K. Chard, R. Chard, B. McCollam, J. Pruyne, S. Rosen, S. Tuecke, and I. Foster. Globus platform services for data publication. In *Practice and Experience on Advanced Research Computing*, page 14. ACM, 2018.

[7] Apple. File system events. https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/FSEvents_ProgGuide/UsingtheFSEventsFramework/UsingtheFSEventsFramework.html, 2012. Accessed: Sept, 2018.

[8] P. Beckman, T. J. Skluzacek, K. Chard, and I. Foster. Skluma: A statistical learning pipeline for taming unkempt data repositories. In *29th International Conference on Scientific and Statistical Database Management*, page 41. ACM, 2017.

[9] T. W. Bereiter. Software auditing mechanism for a distributed computer enterprise environment, May 19 1998. US Patent 5,754,763.

[10] B. Blaiszik, K. Chard, R. Chard, I. Foster, and L. Ward. Data automation at light sources. In *AIP Conference Proceedings*, volume 2054, page 020003. AIP Publishing, 2019.

[11] R. Chard, K. Chard, J. Alt, D. Y. Parkinson, S. Tuecke, and I. Foster. Ripple: Home automation for research data management. In *37th IEEE International Conference on Distributed Computing Systems*, 2017.

[12] E. M. Crisostomo. fswatch. https://github.com/emcrisostomo/fswatch, 2013. Accessed: Sept, 2018.

[13] Facebook. Watchman: A file watching service. https://facebook.github.io/watchman/, 2015. Accessed: Sept, 2018.

[14] I. Foster, B. Blaiszik, K. Chard, and R. Chard. Software Defined Cyberinfrastructure. In *The 37th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2017.

[15] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann. HACC: Extreme scaling and performance across diverse architectures. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, Nov 2013.

[16] P. Hintjens. *ZeroMQ: messaging for many applications*. O'Reilly Media, Inc., 2013.

[17] T. Leibovici. Taking back control of HPC file systems with Robinhood Policy Engine. *arXiv preprint arXiv:1505.01448*, 2015.

[18] J. Lemon. Kqueue – A generic and scalable event notification facility. In *USENIX Annual Technical Conference, FREENIX Track*, pages 141–153, 2001.

[19] Linux. inotify-tools. https://github.com/rvoicilas/inotify-tools/, 2010. Accessed: Sept, 2018.

[20] R. Love. Kernel korner: Intro to inotify. *Linux Journal*, 2005(139):8, 2005.

[21] Microsoft. FileSystemWatcher. https://docs.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher?redirectedfrom=MSDN&view=netframework-4.7.2, 2010. Accessed: Sept, 2018.

[22] R. Miller, J. Hill, D. A. Dillow, R. Gunasekaran, G. M. Shipman, and D. Maxwell. Monitoring tools for large scale systems. In *Cray User Group Conference*, 2010.

[23] A. K. Paul, A. Goyal, F. Wang, S. Oral, A. R. Butt, M. J. Brim, and S. B. Srinivasa. I/O load balancing for big data HPC applications. In *International Conference on Big Data*, pages 233–242. IEEE, 2017.

[24] A. K. Paul, S. Tuecke, R. Chard, A. R. Butt, K. Chard, and I. Foster. Toward scalable monitoring on large-scale storage for software defined cyberinfrastructure. In *2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*, pages 49–54. ACM, 2017.

[25] Python. Watchdog. https://pypi.org/project/watchdog/, 2010. Accessed: Sept, 2018.

[26] D. Quintero, L. Bolinches, P. Chaudhary, W. Davis, S. Duersch, C. H. Fachim, A. Socoliuc, O. Weiser, et al. *IBM Spectrum Scale (formerly GPFS)*. IBM Redbooks, 2017.

[27] V. Tarasov, E. Zadok, and S. Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[28] B. Wadhwa, A. K. Paul, S. Neuwirth, F. Wang, S. Oral, A. R. Butt, J. Bernard, and K. W. Cameron. iez: Resource contention aware load balancing for large-scale parallel file systems. In *IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2019.

[29] S. Àlvarez. Fsmon. https://github.com/nowsecure/fsmon, 2016. Accessed: Sept, 2018.