

PoliMOR: A Policy Engine “Made-to-Order” for Automated and Scalable Data Management in Lustre

Anjus George

National Center for Computational
Sciences, Oak Ridge National
Laboratory
Oak Ridge, TN, USA
georgea@ornl.gov

Christopher Brumgard

National Center for Computational
Sciences, Oak Ridge National
Laboratory
Oak Ridge, TN, USA
brumgardcd@ornl.gov

Rick Mohr

National Center for Computational
Sciences, Oak Ridge National
Laboratory
Oak Ridge, TN, USA
mohrrf@ornl.gov

Ketan Maheshwari

National Center for Computational
Sciences, Oak Ridge National
Laboratory
Oak Ridge, TN, USA
maheshwarikc@ornl.gov

James Simmons

National Center for Computational
Sciences, Oak Ridge National
Laboratory
Oak Ridge, TN, USA
simmonsja@ornl.gov

Sarp Oral

National Center for Computational
Sciences, Oak Ridge National
Laboratory
Oak Ridge, TN, USA
oralhs@ornl.gov

Jesse Hanley

National Center for Computational
Sciences, Oak Ridge National
Laboratory
Oak Ridge, TN, USA
hanleyja@ornl.gov

ABSTRACT

Modern supercomputing systems are increasingly reliant on hierarchical, multi-tiered file and storage system architectures due to cost-performance-capacity trade-offs. Within such multi-tiered systems, data management services are required to maintain healthy utilization, performance, and capacity levels. We present PoliMOR, a pragmatic and reliable policy-driven data management framework. PoliMOR is composed of modular, single-purpose agents that gather file system metadata and enforce policies on storage systems. PoliMOR facilitates automated and scalable data management with customizable agents tailored to HPC facility-specific storage systems and policies. Our evaluations demonstrate the scalability and performance of PoliMOR both by its individual agents and as a collective entity. We believe PoliMOR is widely applicable across HPC facilities with large-scale data management challenges and will garner interest from the HPC community, given its flexible and open-source nature.

KEYWORDS

multi-tiered parallel file system, high performance computing, message queues, policy engine, storage and data management

ACM Reference Format:

Anjus George, Christopher Brumgard, Rick Mohr, Ketan Maheshwari, James Simmons, Sarp Oral, and Jesse Hanley. 2023. PoliMOR: A Policy Engine “Made-to-Order” for Automated and Scalable Data Management in Lustre. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3624062.3624190>

1 INTRODUCTION

The ever increasing I/O rate and quantity of data generated by HPC applications are driving parallel file system complexity as sites utilize heterogeneous storage tiers to improve performance while balancing cost and capacity. Recent HPC systems such as Frontier [21], LUMI [9], and El Capitan [14] illustrate the increased deployment of multi-tiered storage systems. The tremendous amount of data dispersed across multiple tiers makes data and capacity management challenging. Data management practices must minimize wastefulness of storage resources, promote fair usage, and satisfy application I/O needs (ideally in a systematic and automated fashion). Common tasks may include purging old files to recover disk space, staging data to a tier with higher performance, or migrating data to tape. Sites could potentially handle some of these

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

SC-W 2023, November 12–17, 2023, Denver, CO, USA

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<https://www.energy.gov/doe-public-access-plan>).

tasks using in-house scripts, but such tools often have limited flexibility, functionality, and/or scope. Software tools exist to handle file purging [17], data indexing [22], and data migration [28]. However, these tools only provide pieces of a complete data management solution with no integration amongst them, putting the burden on storage administrators to implement the software needed to tie these tools together.

An ideal solution would be a single policy engine to coordinate all aspects of data management. Commercial policy engines exist such as Cray ClusterStor Data Services (CDS) [3], DDN Strategem [19], and IBM Spectrum Scale ILM [6], but each of these is tied to a specific vendor product and not available for installation on an arbitrary storage system. Furthermore, these solutions come with a limited ability to customize the software and licensing fees that could be cost prohibitive for some sites. When it comes to open source alternatives, the options are even fewer. Robinhood [16] is the leading choice and is often deployed in conjunction with Lustre [11] file systems, utilizing Lustre’s Changelog and Hierarchical Storage Management (HSM) [7] features to monitor the file system and perform data movement. Unfortunately, this solution is subject to scalability and functionality limitations (described in §2) that may cause problems for very large storage systems. Between the vendor lock-in of commercial solutions and the lack of open source alternatives, many sites find it difficult (if not impossible) to deploy a single policy engine to orchestrate their data management tasks.

To provide an alternative to commercial policy engines and address shortcomings in existing options, we have introduced PoliMOR - a scalable and customizable policy engine framework for automated data management. Unlike existing policy engines, PoliMOR uses a micro-service design and is composed of independently scalable single-purpose agents communicating via a common message queue system. PoliMOR agents handle tasks such as gathering file metadata, making policy decisions, and executing actions based on policies. PoliMOR’s extensibility allows for the integration of platform-specific utilities/tools to its agents, making it suitable for other filesystems like GPFS [5] and BeeGFS [13]. Additionally, as a contribution to the HPC community, we provide an open-source implementation of PoliMOR, accessible on our GitHub¹ repository.

The rest of this paper is organized as follows. §2 identifies drawbacks of existing solutions and outlines features for a new policy engine. §3 describes the architecture, design, and implementation of PoliMOR. §4 presents evaluation results on PoliMOR’s scalability, performance, and resource utilization. §5 concludes the paper and discusses future work directions.

2 BACKGROUND AND MOTIVATION

Frontier, an exascale system at OLCF (Oak Ridge Leadership Computing Facility), provides computational resources for applications with a variety of I/O requirements. A multi-tiered Lustre file system named Orion was deployed to address these needs and accommodate up to 10 billion files. Orion [8] provides three storage tiers: a 10 PB NVMe-based metadata tier, a 11.5 PB NVMe-based performance tier, and a 679 PB HDD-based capacity tier. In addition to OLCF’s standard 90-day purge policy, an automated mechanism was needed to move 2-day old data from the performance tier to

the capacity tier. The use of multiple single-purpose tools was not desired because the lack of coordination could lead to conflicts and sub-optimal use of resources (e.g. - wasting time migrating a multi-petabyte file only to have it purged a few hours later). A single policy engine was needed, and the only potential solution was Robinhood combined with Lustre Changelogs and HSM².

Robinhood and Lustre. Robinhood subscribes to Lustre Changelogs, processes events, and updates its database to maintain a snapshot of the current state of the file system. Our initial evaluation of Robinhood identified several issues:

- **Lack of parallelism and fault tolerance:** There is no support for running multiple Robinhood instances or policy actions across multiple servers.
- **Database performance:** Robinhood operates best if the entire database is cached in memory, but this is only practical for millions of files, not billions. Even with SSDs and significant tuning, database performance is a bottleneck, as evidenced by past experiences at OLCF with smaller file systems. Additionally, missing changelog entries can lead to database entry discrepancies [4, 27].
- **No incremental policy enforcement:** Robinhood uses triggers (based on usage levels or scheduled periodically) to determine when policies run. Each policy run performs a full database query to find matching files. For certain policies, it would be more efficient to simply perform the check when updated file metadata arrives avoiding stale entries.
- **Extensive tunings and configurations:** Robinhood necessitates administrators to perform substantial parameter tunings at the database, file system, and kernel levels before deployment (see “Feeding the beast” section in [2]).

Based on this information, Robinhood was unsuitable for Orion data management, necessitating the need for an alternative solution.

New Policy Engine Design. We began conceptualizing a policy engine based on the following guidelines:

- **Act upon the current file metadata:** Any “copy” of the file system’s state can become outdated if file changes fail to be captured. Instead, throttle and use demand-based scanning of the files, and then use this current metadata for policy matching.
- **Do not store file metadata long-term:** As new file metadata arrives, determine if the file matches an existing policy and take action if needed, but do not retain the metadata.
- **Support modularity:** The policy engine should be constructed from modular components that can be customized based on a site’s individual needs.
- **Design for scalability:** Any component should be able to scale across one or more hosts. Likewise, performance should increase accordingly.

These guidelines formed the basis for the development of PoliMOR. While the initial implementation of PoliMOR focuses on supporting the needs of the OLCF, the architecture is general enough to be applied to any storage system.

¹<https://github.com/olcf/polimor>

²This was prior to the availability of the Cray CDS policy engine. However, subsequent testing with CDS v1.0 showed limitations migrating large numbers of small files which is unsuitable for use on Orion.

3 DESIGN AND IMPLEMENTATION

3.1 Architecture

As shown in Figure 1, PoliMOR at its core has three different types of components designated to perform 1. *File system Scan*, 2. *Policy Matching*, and 3. *File System Actions*. These core components of PoliMOR are called **agents** and are named ‘**Scan Agents**’, ‘**Policy Agents**’, and ‘**Action Agents**’. Action Agents assume specific names depending upon the actions performed (such as purge, migration, or archive). Agents rely on a messaging system to communicate with each other providing temporal decoupling of senders and receivers through a collection of underlying message queues.

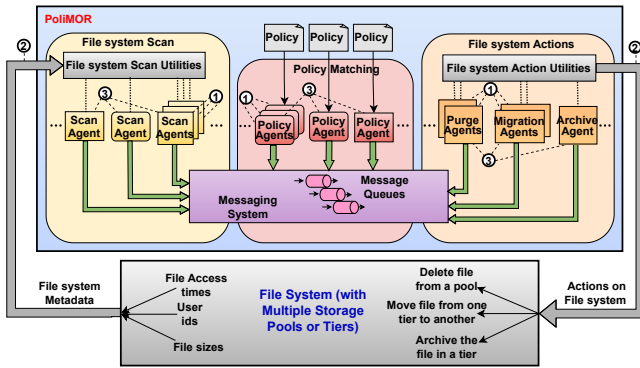


Figure 1: PoliMOR Architecture

In the following sections we describe the architecture and design abstractions of PoliMOR in detail.

3.2 PoliMOR Abstractions

3.2.1 Agents. PoliMOR agents are architected based on three primary design goals - ① *independent scalability*, ② *fully automated operation*, and ③ *customizability*. Agents are single-purpose and event-driven, and adhere to a micro-service-oriented design pattern to enable independent scalability (see ① in Figure 1). The scalability of agents allows facilities to optimize the number of agents to be deployed based on the size of their file system, site-specific policies, and policy-driven system load. The agents do not require administration intervention achieving its goal of fully automated operation. After PoliMOR is started, all agents continuously operate in parallel and pipelined fashion.

Scan Agents: Scan agent accepts administrator specified directory trees and scan interval for its periodic operation. They recursively collect file metadata required by the policies and concurrently broadcast it to the message queues.

Policy Agents: After receiving the metadata from the scan agents, the policy agents enforce policy driven actions by sending the action requests for the selected files to the designated queues. Each policy agent works independently without a shared scheduler, making decisions on an in-order-first-match basis.

Action Agents: Action agents are responsible for executing the decisions made by the policy agents (e.g., file purging, migration, or archival) on files/directories. The pipelined execution of all the agents ensures automated operation and propagation of immediate actions during the file system scan (see ② in Figure 1).

Additionally, facilities can customize agents by integrating site-gnostic tools/utilities. For example, a scan agent can have any utility that performs file system indexing. Similarly, purge and migration agents can integrate external tools/utilities for file removal and migration, independent of the file system’s built-in capabilities. Further, facilities can incorporate additional action agents (see ③ in Figure 1) such as monitoring/recording agents to collect intermediate data or logging.

3.2.2 Policies. An administrator provided policy comprises a collection of conditions and actions to be taken when a file meets those conditions. For instance, a file migration policy can have two different conditions such that if *file size > 1GB* and *file access time > 70 days*, then the file needs to be migrated off a particular storage pool/tier. PoliMOR allows to define any number of diverse policies using an external configuration file or directly encoded into the policy agent.

3.2.3 Message Queues. PoliMOR adopts a *publish-subscribe (pub-sub)* messaging model. A pub-sub model reduces design complexity and substantially simplifies the communication between agents by eliminating point-to-point connections between them. The loose coupling between publishers and subscribers is well suited for the micro-service based design.

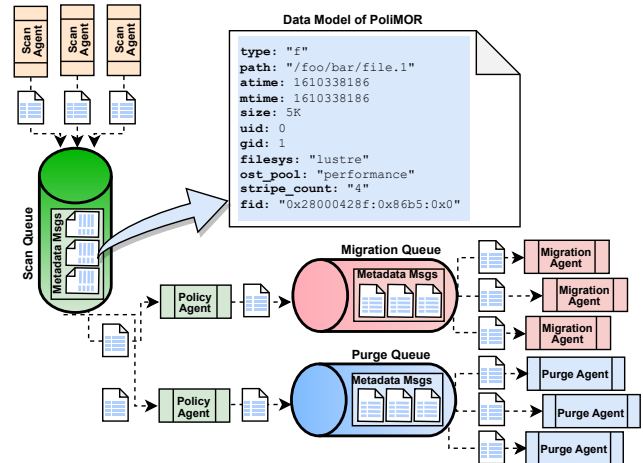


Figure 2: Different types of queues and agents in PoliMOR and their interactions. The data model represents the format of messages in the message queues.

As shown in Figure 2, PoliMOR is composed of two types of queues – **Scan Queue** and **Action Queue**. The scan agents publish messages to the scan queue, which are consumed by policy agents that, in turn, publish to the action queues. Action agents subscribe to these respective action queues (migration, purge, and archive (not shown here)) to receive commands from the policy agents.

Agents use *pull*-based subscriptions to avoid a backlog of messages. Further, to enable one-time consumption of messages and prevent double initiation of actions on the same file, the queues are designed to be *workqueue* based. Table 1 shows various properties of the queues in PoliMOR, their configurations, and rationale behind those configurations. All the message queues store a single type of data, that is, metadata pertaining to files. Therefore we define the

data model of PoliMOR to be key value pairs representing the file’s properties and their values (see Figure 2).

3.3 Fault Tolerance and Reliability

The failure scenarios under which PoliMOR can self-recover are listed in Table 2. We state the failure scenarios, then their mitigation strategy, and possible repercussions due to these scenarios. PoliMOR can tolerate most of these failures without administrator intervention, guaranteeing fully automated operation under normal circumstances.

Table 1: Message queue properties in PoliMOR

Property	Configuration	Rationale
Size	Small ($\approx 10K$)	To enable feedback loop rate limiting between agents and prevent the accumulation of outdated messages
Storage	In-memory, Age limited	To prevent accumulation of outdated messages
Replicas	On all messaging servers	For load balancing among servers and fault tolerance of queues
Retention	Workqueue based	To enable load balancing among subscribers and prevent redundant processing of messages
Duplicate tracking	Enabled	To prevent redundant processing of messages

PoliMOR pushes much of the fault tolerance requirements to the messaging system expecting replicated queue state (consensus protocol) and message retries and acknowledgements at the pub-sub level. The agents themselves are mostly stateless, relying upon the messaging system in order to simplify the design. As a result possible message loss due to these failure scenarios is minimal since each agent handles a single message at any point in time. Reduction in the messaging pipeline throughput can be overcome by dynamically starting additional agents and messaging servers. However, PoliMOR cannot self-recover in case of catastrophic failures that result in all agents or all messaging queues being unavailable. In case of such catastrophic failures, all messages are lost preventing issues arising from old messages remaining in the queues. Administrators will need to restart all agents after the failure scenario has been resolved.

3.4 Implementation Details

To implement PoliMOR for OLCF’s Orion-Lustre file system, we collected these initial two data management requirements for Orion: 1) *Migrate files not accessed in 2 days from the performance tier to the capacity tier.* 2) *Purge files not accessed for the past 90 days from the capacity tier.* We implemented 4 agent types - scan, policy, purge, and migration. Our agents rely on Lustre’s underlying utilities, such as `lfs find` (enhanced version [10, 18]) and `lfs migrate`, and `rm` to collect metadata, migrate, and purge file system entries, respectively. The highly performant and reliable NATS [1] framework serves as the backend for our messaging system with JSON-based messages. NATS is open source and lightweight, offers low latency (in the order of sub-milliseconds), incurs low CPU and memory overhead, and has been utilized in previous Cloud/Edge/HPC system studies [12, 15, 24]. Using the native NATS tools we created scan, purge, and migration queues required for PoliMOR operation adhering to Table 1. All agents are implemented as lightweight, single threaded, C++20 coroutines with additional Boost libraries [25], third party libraries [23, 26], and Python.

4 EVALUATION

We present an evaluation of PoliMOR, characterizing its scalability and performance while measuring resource utilization and load on Lustre MDS and OSS servers.

4.1 Testbed Setup for Evaluation

To evaluate PoliMOR, we used 11 nodes from a small scale HPC testbed at OLCF. The 11 testbed nodes correspond to number of utility nodes available for PoliMOR deployment (in production) on Orion. The operating system is RHEL 8.5 with Lustre version 2.15. All the nodes have identical hardware configurations with a single AMD EPYC 7351 16-core processor, 126 GB of RAM, and communicate with the Lustre file system using EDR Infiniband.

The Lustre file system consists of 6 OSS servers and 32 MDS servers, with only 4 MDSs utilized for our testing. The OSS servers are connected via EDR Infiniband to a DDN 14K storage system providing 800 TB of disk-based storage configured as 12 OSTs (2 per OSS). Additionally, the first four OSS servers are connected to a DDN NV200 storage system providing 13.6 TB of SSD-based storage configured as 4 OSTs (1 per OSS). The 12 disk-based OSTs form the capacity tier OST pool, while the 4 SSD-based OSTs form the performance tier OST pool. Each MDS node uses an internal 1.5 TB NVMe SSD drive for MDT storage.

For the evaluation tests, we allocated 3 nodes to form a NATS Jetstream cluster with 20 GB of in-memory storage per node. The remaining 8 nodes executed the processes and agents required for PoliMOR tests.

4.2 Scalability Tests

For evaluating agent scalability, we independently scaled each type of agent from 1 to 64 (in powers of 2). The agents were round-robin deployed across the 8 testbed nodes with up to 8 agents per node in the maximum configuration. Each configuration was tested 4 times, ensuring all system caches, OSS, MDS, and Lustre client caches were cleared between runs. Our workload consists of 64 directories with each having 1 million files (with all the directories round-robin across 4 MDTs). We adjusted the size of the message queues to accommodate the total message count since the agents were tested in isolation triggering message accumulation. The properties of the file system workload for each agent test and the rationale for their settings are described in the respective test sections.

4.2.1 Scan Agent Scaling. Our preliminary tests on `lfs find` utility revealed that the time it takes to collect metadata from the file system has little dependence on directory tree depth (a max increase of only 5.5% for a tree depth of 10 over 1). However, the collection time for files with stripe count 6 showed an increase of 2.5x over that of files with stripe count 1. Therefore, for scan agent scaling tests we chose a directory tree with depth of 1 and files having a stripe count of 6. Since collection time is independent of file sizes, we randomly distributed the sizes up to 1 MB.

We benchmarked our scan agent performance directly against the native `lfs find` command. Scan agents and the `lfs find` commands both operated on independent directories to avoid delay caused by Lustre lock contention on the same directories. The time that an agent takes to scan one file entry is the sum of metadata

Table 2: PoliMOR Failure Scenarios and Mitigation Strategies. SR: Self recoverable, MI: Manual intervention

Failure Scenario	Mitigation Strategy	SR?	MI?	Repercussions
Loss of some (not all of same type) agents	Agents are scalable and can be run independently	Yes	No	Message loss and slowdown in pipeline
Complete node failure	Agent can be distributed across multiple nodes	Yes	No	Message loss and slowdown in pipeline
Loss of (some) messaging servers / message queues	Queues can be replicated across a cluster of servers	Yes	No	No message loss, reduction in average messaging throughput
Loss of network connectivity	Agents can continue to wait until network access is restored	Yes	No	Message backlog in queues, reduction in average pipeline throughput
File system down	Agents can back off and timeout their respective operations	Yes	No	Message backlog in queues, reduction in average pipeline throughput
Loss of all agents / all messaging servers / all nodes or combination of these	Currently none	No	Yes	Message loss if queues are not persisted to disk

collection, processing and publishing time. Figure 4a provides an average (of 4 runs) of the total number of files scanned for 300 seconds for each configuration. The time taken by the agent is dominated by metadata collection as there is almost no difference in performance between the agent and `lfs find`. 64 agents showed a 46.7x increase in the total number of files scanned compared to one agent, demonstrating our implementation scales effectively as `lfs find`.

4.2.2 Policy Agent Scaling. The performance of the policy agent was tested in isolation by populating the scan queue with 1 million file entry messages of size 512 bytes.

A policy agent subscribes to the scan queue, performs policy matching, and publishes actions to either the purge or migration queue. We determined that the policy agent processing time is dominated by publish and subscribe times. Since message sizes are highly contingent on file path size (max length of a file path is 4 KB in Linux), we tested the impact of message size on publish/subscribe times for different message sizes as shown in Figure 3. There is a gradual decline of messaging throughput for message sizes beyond 512 bytes. We are unlikely to encounter such long path names in a file system, and therefore we chose a typical path length of 200 Bytes (representative of a variable directory tree depth, and directory and file names). Besides path, other fields in the `lfs find` command have an almost fixed length totalling 231 bytes. Combined with the path length, this would result in a message size of 431 Bytes. We chose to round up to 512 bytes size for a message in the scan queue.

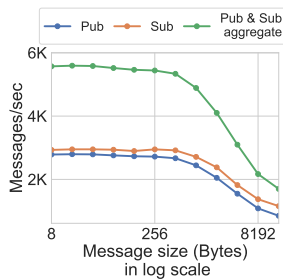


Figure 3: Publish and subscribe messaging throughput for PoliMOR queues.

After populating the scan queue with 1 million messages, we measured the time taken by the policy agent to process the messages. We modified file timestamps to ensure that every file would be published to either the purge or migration queue. Figure 4b shows near perfect linear scaling as the number of policy agents increases from 1 to 64. Rapidly enough, 64 policy agents took only 52.25 seconds to process 1 million messages from the queue.

4.2.3 Purge and Migration Agent Scaling. Like the previous test, purge and migration agent scaling were conducted by populating the respective queues. The queues were populated using 192K actions. Since purge and migration are both dependent on the file size, we populated the file system with fixed size (2MB) files.

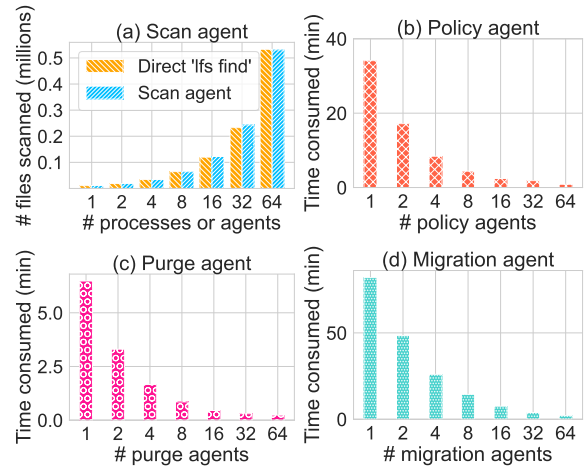


Figure 4: Scaling results for PoliMOR agents.

Each data point is measured as the time taken to purge/migrate 192K files. Figure 4c and 4d show that both purge and migration agents scale well as the number of agents increases up to 64. 64 purge and migration agents operated 27.7x and 38.7x faster than their respective single agents.

It is unfair to make a direct comparison between the migration/purge agents and the `lfs migrate/rm` commands in the same manner as the scan agent and the `lfs find` command. In the latter case, both the scan agents and the `lfs find` commands could be easily configured to operate on independent directories to avoid contention. While `lfs migrate` and `rm` commands could also be configured for independent directories, the same cannot be done for the migration and purge agents. There is nothing to prevent different agents from receiving messages pertaining to files in the same directory, thus leading to directory lock contention. Note that, choosing the optimal number of agents and scan frequency depends on key factors, including file system attributes, choice of scanning tool, resource availability, and performance objectives.

4.3 Performance Tests

To evaluate the performance of PoliMOR agents we created a directory layout with 100 directories uniformly distributed across 4 MDTs (MDT0 to MDT3). The top-level directories in this layout resemble project directories on Orion file system.

Each top-level directory consists of 100K files totaling to 10 million files in the file system. The contents of each project directory are split to have 20K files on the performance tier with a stripe count of 1 and 80K files on the capacity tier with a default stripe count of 6. Since Orion was not in production at the time of these tests, to

get an accurate estimate of the file sizes on Orion we profiled the Alpine [20] file system (Orion’s predecessor at OLCF) and collected the distribution of file sizes over different size buckets. Figure 5 shows the percentage of files distributed over sizes ranging from 0 Bytes to 512 TB. We projected this distribution to 10 million files to obtain the number of files to be created under each size bucket.

For files on the performance tier, the access times are set to make all of them eligible for migration. In production, not all files on the performance tier would be eligible, however, since they have a short lifetime (2 days) on the performance tier, many files are expected to get migrated. With the 90-day purge policy on Orion, purge eligible files would be closer to 1.1% ($1/90^{th}$) in production. But to have slightly higher purge activity during the tests, the access times for files on the capacity tier are set to make 3-5% of them eligible for purging.

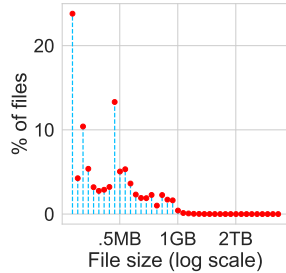


Figure 5: File size distribution in Alpine file system.

for file creation, read, stat, and removal. More testing is needed to fully quantify the performance difference at larger scales.

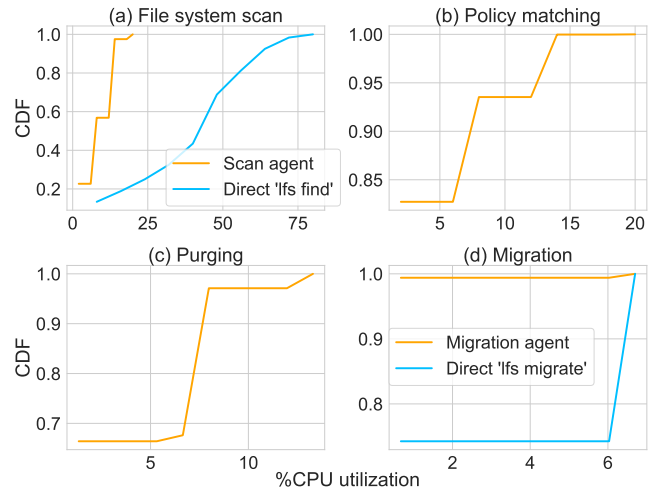


Figure 7: CDF of %CPU utilization for operational phases in PoliMOR.

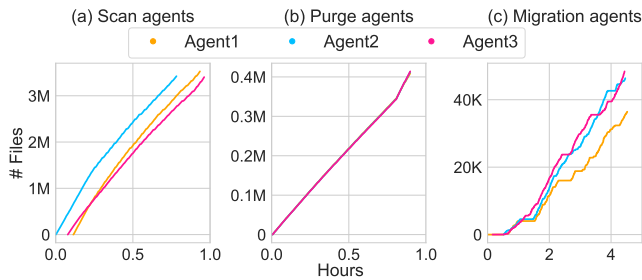


Figure 6: Performance of PoliMOR agents. Number of files scanned, purged and migrated by each of the 3 (a) scan, (b) purge, and (c) migration agents.

A total of 3 scan, 3 purge, 3 migration, and 2 policy agents are allowed to operate on the file system until they finish their respective actions. During the operation, number of files scanned, purged and migrated are measured by sampling the number of underlying task specific system calls made per second. Figure 6 shows the cumulative number of files scanned, purged, and migrated by each of the agents. Each scan agent scanned around $1/3^{rd}$ of the files. The slightly different performance across scan agents attribute to the `lfs find` process collecting metadata from any of the 6 OSS servers depending on the striping of files across OSTs.

All purge agents have identical performance in contrast to migration agents, since purging is less dependent on file sizes. However, median number of bytes migrated by agents showed only a maximum variation of 5.7%. The scan and purge agents completed their runs within 1.5 hours whereas migration agents’ runs consumed 4.25 hours. Since policy agent does not interact with the file system it operates faster than the other agents. In other words, it’s performance is throttled by the rate of operation of other agents and therefore we do not show the results here. Additionally, the MDTest benchmark, run with and without PoliMOR scan agents on four Lustre clients, showed less than a 10% difference in average rates

4.4 Resource Utilization Tests

Using the same workload described in §4.3, we evaluated the resource consumption of the agents by measuring their %CPU and %memory utilization, and load on Lustre MDTs and OSSs. Figure 7 shows the CDF of %CPU utilization of the individual PoliMOR agents. In the file system scan phase, CPU utilization peaked at 80% due to the internal `lfs find` process, with the scan agent below 20%. Policy matching and purging phases recorded CPU utilizations under 20% and 13%, respectively. In the migration phase, both the migration agent and the internal `lfs migrate` process reached a minimal 6.7% max utilization. Additionally, all agents reported nearly 0% memory utilization on all the nodes.

Figure 8 shows the 1-minute load average of the Lustre MDSs and OSSs during the operation of PoliMOR. The OSS load shows a spike during the early part of the operation when scans, purges, and migrations are running simultaneously. After 1.5 hours, the load drops dramatically and remains consistently around 2-3 when only the migrations are running. The MDS load shows a similar spike in load during early operation, but it is much less pronounced since the MDS load during the entire operation is minimal and never exceeding 1.

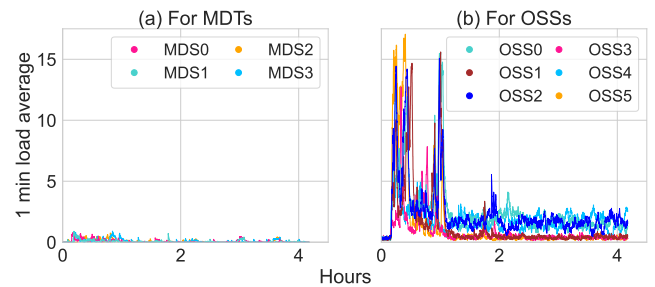


Figure 8: Load average of Lustre MDSs and OSSs during PoliMOR operation.

5 CONCLUSION AND FUTURE WORK

We presented the architecture and design of PoliMOR, a pragmatic and reliable policy-driven data management framework for HPC facilities. PoliMOR has fully automated operation, independently scalable components and allows facilities to customize it as per their storage and policy requirements. We evaluated PoliMOR’s scalability, performance, and resource utilization using an HPC testbed at the OLCF.

As a near-term objective, we plan to evaluate PoliMOR’s performance and resource usage when deployed on Orion file system containing billions of files. In our ongoing work, we have identified several key areas for future enhancement. We plan to implement concrete policy specifications and optimize the interaction between the scan and policy agents, making policies into simpler requirements for efficient evaluation. We would like to also balance the workload among scan agents more effectively using a work sharing algorithm. Additionally, we aim to introduce agents that use auxiliary databases to reduce full tree walks and explore alternatives to Lustre’s lfs find for improved performance and richer data. The migration agent will benefit from a third-party tool for bandwidth optimization and finer-grained control. A future direction is to dynamically prioritize resources based on file system load and available capacity. This includes scaling down PoliMOR during high user load with ample space and scaling up during resource scarcity. We are considering supervisor agents for scalability control and exploring container orchestration like Kubernetes for fault tolerance. Our broader goals involve handling complex policies, expanding compatibility with various storage systems, and accommodating different types of data storage.

ACKNOWLEDGMENTS

This research was sponsored by and used resources of the Oak Ridge Leadership Computing Facility (OLCF), which is a DOE Office of Science User Facility at the Oak Ridge National Laboratory supported by the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

- [1] NATS Authors. 2023. *NATS, Connective Technology for Adaptive Edge and Distributed Systems*. Retrieved March 5, 2023 from <https://nats.io/>
- [2] CEA-HPC. 2017. *Robinhood v3 admin documentation*. Retrieved September 18, 2023 from https://github.com/cea-hpc/robinhood/wiki/robinhood_v3_admin_doc#user-content-Feeding_the_beast
- [3] Cray ClusterStor data services User Guide 2.1.2 (S-1239). 2023. *What ClusterStor data services Provides*. Retrieved March 9, 2023 from https://support.hpe.com/hpsc/public/docDisplay?docId=sd00002127en_us&docLocale=en_US&page=GUID-D1887849-55E8-4B93-9880-5EED70536873.html
- [4] Tina M Declerck et al. 2014. Using Robinhood to purge data from Lustre file systems. *Proceedings of the 2014 Cray User Group, Lugano* (2014).
- [5] IBM Documentation. 2015. *Introducing General Parallel File System*. Retrieved September 13, 2023 from <https://www.ibm.com/docs/en/gpfs/4.1.0.4?topic=guide-introducing-general-parallel-file-system>
- [6] IBM Spectrum Scale Documentation. 2021. *Information lifecycle management for IBM Spectrum Scale*. Retrieved March 9, 2023 from <https://www.ibm.com/docs/en/spectrum-scale/5.0.5?topic=administering-information-lifecycle-management-spectrum-scale>
- [7] OpenSFS; EDFS. 2023. *Lustre® software release 2.x: Operations manual*. Retrieved April 8, 2023 from https://doc.lustre.org/lustre_manual.xhtml
- [8] Oak Ridge Leadership Computing Facility. 2021. *OLCF announces storage specifications for Frontier exascale system*. Retrieved March 29, 2023 from <https://www.olcf.ornl.gov/2021/05/20/olcf-announces-storage-specifications-for-frontier-exascale-system/>
- [9] IT Center for Science (CSC). 2020. *One of the world’s mightiest supercomputers, LUMI, will lift European research and competitiveness to a new level and promotes green transition*. Retrieved March 28, 2023 from <https://www.csc.fi/en/-/lumi-one-of-the-worlds-mightiest-supercomputers>
- [10] Anjus George, Richard Mohr, and James Simmons. 2022. *LU-10378 utils: add formatted printf to lfs find*. Retrieved March 30, 2023 from <https://review.whamcloud.com/c/fs/lustre-release/+45136>
- [11] Anjus George, Rick Mohr, James Simmons, and Sarp Oral. 2021. *Understanding Lustre Internals*. Second Edition. (9 2021). <https://doi.org/10.2172/1824954>
- [12] Anjus George, Arun Ravindran, Matias Mendieta, and Hamed Tabkhi. 2021. Mez: An Adaptive Messaging System for Latency-Sensitive Multi-Camera Machine Vision at the IoT Edge. *IEEE Access* 9 (2021), 21457–21473. <https://doi.org/10.1109/ACCESS.2021.3055775>
- [13] ThinkParQ GmbH. 2023. *BeeGFS, The leading parallel file system*. Retrieved September 13, 2023 from <https://www.beevfs.io/c/>
- [14] HPCWire. 2020. *Exascale Watch: El Capitan Will Use AMD CPUs & GPUs to Reach 2 Exaflops*. Retrieved February 18, 2023 from <https://www.hpcwire.com/2021/02/18/livemores-el-capitan-supercomputer-hpe-rabbit-storage-nodes/>
- [15] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2018. Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 219–230.
- [16] Thomas Leibovici. 2015. Taking back control of HPC file systems with Robinhood Policy Engine. *arXiv preprint arXiv:1505.01448* (2015).
- [17] Fang Liu, Dan Zhou, Ken Suda, Michael D Weiner, Mehmet Belgin, Ruben Lara, and Pam Buffington. 2022. A Fully Automated Scratch Storage Cleanup Tool for Heterogeneous Parallel Filesystems. In *Practice and Experience in Advanced Research Computing*, 1–7.
- [18] Richard Mohr, Anjus George, and James Simmons. 2020. The Improved “lfs find” Command. (2020). https://www.opensfs.org/wp-content/uploads/Mohr-Improved_lfs_find_Command.pdf Lustre User Group Conference (LUG2022).
- [19] DataDirect Networks. 2023. *ddn exa6, Introducing Strategem*. Retrieved April 7, 2023 from <https://www.ddn.com/products/lustre-file-system-exascaler/>
- [20] OLCF. 2018. *ALPINE, OLCF’s center-wide, POSIX-based IBM Spectrum Scale file system*. Retrieved April 10, 2023 from <https://www.olcf.ornl.gov/olcf-resources/data-visualization-resources/alpine/>
- [21] Oak Ridge National Laboratory (ORNL). 2022. *Frontier, Direction of Discovery*. Retrieved March 24, 2023 from <https://www.olcf.ornl.gov/frontier/>
- [22] Arnab K Paul, Brian Wang, Nathan Rutman, Cory Spitz, and Ali R Butt. 2020. Efficient metadata indexing for hpc storage systems. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. IEEE, 162–171.
- [23] Tristan Penman. 2023. *Valijson*. Retrieved March 20, 2023 from <https://github.com/tristanpenman/valijson>
- [24] Waldemar Quevedo. 2018. *Introduction to NATS*. Apress, Berkeley, CA, 1–18. https://doi.org/10.1007/978-1-4842-3570-6_1
- [25] Boris Schäling. 2023. *The Boost C++ Libraries*. Retrieved March 30, 2023 from <https://theboostcpplibraries.com/>
- [26] Kirill Simonov. 2023. *LibYAML - A C library for parsing and emitting YAML*. Retrieved March 17, 2023 from <https://github.com/yaml/libyaml>
- [27] Feiyi Wang, Hyogi Sim, Cameron Harr, and Sarp Oral. 2017. Diving into Petascale Production File Systems through Large Scale Profiling and Analysis. In *Proceedings of the 2nd Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (Denver, Colorado) (PDSW-DJSCS '17)*. Association for Computing Machinery, New York, NY, USA, 37–42. <https://doi.org/10.1145/3149393.3149399>
- [28] Gong Zhang, Lawrence Chiu, Clem Dickey, Ling Liu, Paul Muench, and Sangeetha Seshadri. 2010. Automated lookahead data migration in SSD-enabled multi-tiered storage systems. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–6.