IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 16, 2021, accepted May 26, 2021, date of publication June 8, 2021, date of current version June 18, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3087502

MOSIQS: Persistent Memory Object Storage With Metadata Indexing and Querying for Scientific Computing

AWAIS KHAN[1], HYOGI SIM[2], SUDHARSHAN S. VAZHKUDAI[3], AND YOUNGJAE KIM[1]

[1]Department of Computer Science and Engineering, Sogang University, Seoul 04107, South Korea
[2]Computer Science, Virginia Tech, Blacksburg, VA 24061, USA
[3]Micron Technology Inc., Boise, ID 83716, USA

Corresponding author: Youngjae Kim (youkim@sogang.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant funded by the Korean Government [Ministry of Science and ICT (MSIT)] (Researches on Next Generation Memory-Centric Computing System Architecture) under Grant 2018-0-00503, and in part by the National Research Foundation of Korea (NRF) Grant funded by the Korean Government (MSIT) under Grant NRF-2021R1A2C2014386.

ABSTRACT Scientific applications often require high-bandwidth shared storage to perform joint simulations and collaborative data analytics. Shared memory pools provide a chance to satisfy such needs. Recently, a high-speed network such as Gen-Z utilizing persistent memory (PM) offers an opportunity to create a shared memory pool connected to compute nodes. However, there are several challenges to use scientific applications on the shared memory pool directly such as scalability, failure-atomicity, and lack of scientific metadata-based search and query. In this paper, we propose MOSIQS, a persistent memory object storage framework with metadata indexing and querying for scientific computing. We design MOSIQS based on the key idea that memory objects on PM pool can live beyond the application lifetime and can become the sharing currency for applications and scientists. MOSIQS provides an aggregate memory pool atop an array of persistent memory devices to store and access memory objects to accelerate scientific computing. MOSIQS uses a lightweight persistent memory key-value store to manage the metadata of memory objects, which enables memory object sharing. To facilitate metadata search and query over millions of memory objects resident on memory pool, we introduce Group Split and Merge (GSM), a novel persistent index data structure designed primarily for scientific datasets. GSM splits and merges dynamically to minimize the query search space and maintains low query processing time while overcoming the index storage overhead. MOSIQS is implemented on top of PMDK. We evaluate the proposed approach on many-core server with an array of real PM devices. Experimental results show that MOSIQS gains a 100% write performance improvement and executes multi-attribute queries efficiently with 2.7× less index storage overhead offering significant potential to speed up scientific computing applications.

INDEX TERMS Memory-centric computing and HPC, persistent memory storage, scientific metadata indexing and search, PM index data structures.

I. INTRODUCTION

Large-scale scientific applications, including simulations, experiments, and observations, generate tens of petabytes of data objects and are forecasted to grow even further [1]–[3]. The critical attributes required by such applications include parallel I/O for high-performance and minimal I/O latency in accessing the data objects from storage systems [4]. In addition, the scientific applications, whether running on a single server, small clusters, or HPC systems, all deal with creating, modifying, and processing data objects in memory [5]. The bottleneck between storage and memory has arisen because data must be loaded into memory from slow storage.

Memory centric computing (MCC) has recently emerged to overcome such memory and storage bottlenecks [6]. The HPC has attempted to adopt MCC by enabling a shared memory storage abstraction across the hundreds of compute

The associate editor coordinating the review of this manuscript and approving it for publication was Weipeng Jing.

nodes [6]–[9]. Thus, the upcoming construction of larger MCC infrastructures is expected to be equipped with an array of persistent memory devices co-located with DRAM on each node or shared among all the nodes via high-speed interconnects such as Gen-Z [6] and Infiniband to improvise MCC [6]–[8]. In MCC, the nodes are equipped with non-volatile memories (NVMs), such as Intel Optane DC Persistent Memory (PM) which offers high capacity at low cost, byte-addressability, low idle power, persistence, and performance closer to DRAM than SSD or disks [10]–[12]. A single machine can be equipped with up to 6 TB of PM providing an opportunity to build rack-scale shared memory pools for scientific computing applications [9], [11], [13]. Recently, several studies have shown PM as a full or partial substitute for DRAM [14], [15]. For instance, pVM [14] employs NVRAM to seamlessly expand virtual memory for memory-intensive applications. Similarly, [16] proposed a data-centric OS based on PM. Due to such properties, they are considered a major contender for future main memory fabric and MCC [9], [10].

Therefore, PM given its properties, offers an opportunity to store and manage the millions and billions of objects beyond the lifetime of application in shared memory pool [17]–[19]. Such management of application memory objects on shared PM pool enables multiple benefits, i.e., i) low access latency, ii) low serialization and deserialization overhead, and iii) efficient computation via direct byte-addressability. We refer application objects on PM as Persistent Memory Objects (PMO[1]). Such PM application model also brings us the opportunity to enable PM level object sharing across different users/scientists and applications to facilitate effective scientific collaborations.

Unfortunately, the PM application model stated above creates new data and metadata management challenges. First, there is a need to ensure data and metadata consistency, i.e., data is modified atomically when moving from one consistent state to another. Applications should be able to access PMOs after a crash or ungraceful shutdowns [20]–[23]. Second, scientific application data objects are self-described and packed in versatile scientific data formats, i.e., metadata is embedded inside the data object [24]–[26]. Without additional descriptive metadata, PMO may become unidentifiable, siloed, and in general, not useful to either scientists who own the data or the broader scientific community. Third, where and how to manage, store, and associate object metadata along with user-defined custom metadata is challenging. It is a common standard in the scientific community to tag or annotate data objects with additional descriptive metadata for a better understanding of data for collaborators [2], [25], [27]. Fourth, with ever-changing data analysis scenarios, to select a subset of PMOs from million and billions of PMOs in a shared PM pool based on metadata or user-defined tags without additional indexing becomes highly challenging [27]–[30]. These challenges drive the need for essentially effective scientific metadata search services and querying on top of persistent memory object storage abstraction.

An approach to build a high-performance scientific metadata search service on top of the PM pool is storing, indexing, and querying all of its metadata and at least significant data in the main memory [25], [26], [31], [32]. Although, there exist in-memory scientific data management solutions [25], [30], [31]. However, in-memory solution suffers from failure-tolerance and recurring recovery cost in case of failures. MIQS [25] is the state-of-the-art research, offering an effective in-memory metadata indexing and querying for self-described scientific data formats such as HDF5 [33] and netCDF [34] for scientific applications. It extracts metadata in the form of `<key,value>` pair from scientific data formats and uses multiple tree hierarchies such as a Self-balancing Search Tree (SBST) and Adaptive Radix Tree (ART) to maintain file, location, path, and attributes inside scientific data file. However, a single update to a scientific object makes the whole MIQS index go stale/inconsistent and requires reconstruction of the index, which incurs high recovery overhead. Further, most importantly, the query search space is highly inefficient, i.e., when a search query spans over multiple metadata attributes, then there is ample query search space to find the desired memory objects.

In this paper, we propose to build MOSIQS, an application framework that enables applications, scientists, and researchers to create, modify, search, and delete memory objects on a large shared PM pool. A PMO is a self-described object, i.e., an object can contain a single value, multi-dimensional array or composite value similar to scientific data formats such as HDF5 and netCDF data objects. We design MOSIQS based on the key idea that *memory objects on PM pool can live beyond the application lifetime and can become the sharing currency for applications and scientists*. Moreover, providing controls and annotations to memory objects will bring more friendly storage model in scientific computing environments.

Our key contributions in this paper are:

- We propose an application framework for PM to store and access memory objects via persistent pointers beyond the application lifetime and to share objects across applications, scientists, and collaborators with flexible data sharing controls (Section III). We designed a self-described metadata object and use PMEMKV [35], a light-weight persistent memory key-value store to manage PMO's metadata.
- To enable efficient multi-attribute metadata search and querying for scientific data objects resident in PM pool, we introduce Group Split-and-Merge (GSM) index data structure (Section IV). GSM dynamically split and merge based on sharing frequency, application/user's defined threshold and query patterns to minimize the query search space. Further, GSM also reduces the additional per object index metadata storage overhead.

[1]PMO refers to application memory objects resident on persistent memory.

- For effective storage and easier data sharing, we provide namespace abstraction. Such an abstraction enables a process to share its PMOs with other processes accessing the namespace. We also provide post-storage attribute tagging and annotation to PMO and enable indexing on such application or user-defined metadata attributes annotations(Section III).
- We develop a prototype implementation of the proposed PM application framework using Intel's PMDK [20]. We conduct preliminary evaluations on a Intel many-core server equipped with 1.5 TB real Intel Optane DC 3D-XPoint PM. (Section V). Experimental results show that MOSIQS gains a 100% performance improvement compared to the PM-aware file system approach and executes multi-attribute queries efficiently with 2.7× less index storage overhead.

II. BACKGROUND AND MOTIVATION

In this section, we briefly describe the self-describing scientific data formats. Then, we present the background on PM and elaborate a need for object storage abstraction and search services.

A. SCIENTIFIC DATA FORMATS

Many applications from various scientific domains tend to store experimental, simulation, and analytical data in domain-specific scientific data formats such as HDF5 [33], netCDF [34], FITS [36], Plot3D [37], and GRIB [38] in the underlying parallel file system. These scientific data formats are often referred to as self-described and self-contained, i.e., the metadata is stored alongside the data objects [26]. This metadata enables use and reuse of scientific data [30]. HDF5 is one of the most widely used data formats in the scientific community. It stores data in binary file organized hierarchically for high-performance access [25]. Here, we show the HDF5 internal layout in Table 1. Several optimizations and improvements have been made on top of HDF5 in recent years such as HDF5 virtual object layer (HDF5-VOL) to enable HDF5 data model on different storage backends [39]. Baryon Oscillation Spectroscopic Survey (BOSS) from Sloan Digital Sky Survey (SDSS) [40], typically produces a single data file per object and store it in FITS format [36]. Originally, FITS API lacks parallel IO and data query support and externally relies on HDF5 library [28]. H5Boss [28] tool converts multiple FITS files to HDF5 format to achieve parallel IO and data query.

B. MEMORY CENTRIC COMPUTING

The memory centric computing (MCC) has emerged recently to satisfy the requirements of memory-intensive scientific computing applications [9]. MCC architecture benefits scientific applications in many ways. First, MCC provides a high storage capacity and can store large scientific datasets that could not traditionally fit in the memory. Second, MCC mitigates the performance gap between storage and memory, i.e., fast computation is provided on in-memory large datasets. Third, MCC enables in-memory data sharing across the applications and processes. In particular, MCC operates on the principle of *memory-first*, i.e., the data resides in memory to provide in-memory speeds to deliver tremendous performance. In MCC, each node is equipped with a storage-class non-volatile memory such as Intel Optane DC PM. The PM technology can potentially reduce latency and increase bandwidth of I/O operations by many orders of magnitude, but fully harnessing the device capability requires overcoming the legacy IO stack of disk-based storage systems [11]. A few studies have enabled the use of PM in scientific applications, e.g., NV-Process [41] proposed a fault tolerance process model based on PM and provides an elegant way for the applications to tolerate system crashes. Similarly, [17] evaluates different fault-tolerance approaches for porting scientific applications to use PM. DAOS-M [18] employs PM to store metadata and small writes, whereas larger writes are redirected to NVMe SSDs. Similarly, [16] proposed a data-centric OS based on PM.

TABLE 1. An overview of data abstraction layers in HDF5 [33].

Abstraction	Description
Group	Builds hierarchical representation between one or more objects such as datasets, dataspace and attribute in HDF5 file
Datatype	Defines type of the data element
Dataspace	Logical layout of data elements inside an HDF5 file
Dataset	Single-valued variables, multi-dimensional arrays, complex and user-defined data elements
Attributes	Key-value pair abstraction that provide metadata to annotate or tag dataset or group objects in HDF5 file

C. SERIALIZATION/DESERIALIZATION ON PM

In the conventional scientific computing model, application relies on the CPU to handle the task of deserializing file contents into memory objects. Such an approach requires the application to first load raw data into the system main memory from the storage. Then, the CPU parses and transforms the file data to objects in other main memory locations for the rest of the computation in the application [5], [9]. Such deserialization takes up almost 64% of the application's total execution time [5], [42]. Our previous work [9] provides a conceptual overview of the scientific computing model based on PM, where application objects persist in PM address space, and direct computation is performed, avoiding additional serial- and deserialization operations. Such usage of PM-based storage and computing model also minimizes the decades-old file system IO stack overhead (paging, context switching, kernel code executions).

D. OBJECT MANAGEMENT ON PM

Employing PM directly for legacy scientific applications is challenging. As, the existing applications are built on notion of block-based file system interface and are a clear mismatch

with PM hardware, i.e., byte-addressable. A simple solution is to deploy a PM-aware file system and enable applications to use PM but as reported in [13], ext4-DAX [43] specially designed for PM incurs up to 13× overhead compared to raw PM device write bandwidth. Thus, deploying file system is not an optimal choice for PM. Whereas, an object storage model offers much simpler interface but requires additional metadata book keeping and object sharing controls.

E. MOTIVATION

Arguably, storing application objects directly on persistent memory without a file system interface provides multiple benefits such as faster storage without file system overhead and direct computations. But, it poses several challenges at the same time. First, data sharing across applications and other collaborators is an essential requirement of the scientific community [27], [44]. It is challenging to access, select and share a PMO without additional descriptive metadata. As, object access and sharing require object semantics such as object name, size, and owner provided by the application, user or scientists, whereas, PMOs are simple memory allocated objects and can only be accessed and shared via persistent pointers. For instance, with Intel's PMDK `libpmemobj` API, each stored object on PM is represented by an object handle of type `PMEMoid` as shown in Figure 1.

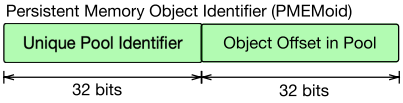


FIGURE 1. Layout of PM object identifier (`PMEMoid`) [20].

The `PMEMoid` for given object does not change during the life of an object/application unless a `realloc()` operation is invoked [10]. Therefore, accessing and sharing a PMO requires an additional metadata mapping or index of objects with user or application provided semantics. Furthermore, self-describing metadata for scientific files, i.e., metadata embedded inside the scientific data file and tags/annotations to data objects by the scientist, also needs to be persisted along with memory objects. Second, a persistent memory object should be crash consistent, i.e., system should ensure access and consistency of memory object in case of application crash or ungraceful power failures. Third, creating and managing index metadata for each PMO incurs high index storage space overhead and large query search space.

To this end, we intend to build an application framework with object storage abstraction on top of the shared PM pool. The proposed application model employs PMDK provided transactions to ensure atomicity and consistency. The metadata is managed in a lightweight persistent key-value store. For multi-attribute indexing, search, and query, we introduce Group Split and Merge (GSM) index data structure. GSM minimizes the query search space and storage overhead by grouping various memory objects based on a specific sharing set of attributes.

III. MOSIQS: DESIGN AND IMPLEMENTATION

In this section, we present our key design goals, target architecture and system overview.

A. DESIGN GOALS

Our key design goals include:

- **Simple and Generic Storage Model:** MOSIQS should have a simple, generic, and schema-less storage model to ensure the compliance to diverse scientific formats and applications, i.e., persistent memory objects should be orthogonal to a domain-specific datatype or format.
- **High-Performance and Scalability:** One critical goal of MOSIQS is to meet the performance and scalability requirements of scientific applications by fully exploiting the underlying hardware architecture, i.e., Shared PM Pool. Furthermore, MOSIQS should be capable of handling concurrent workloads in a scalable manner while ensuring the correctness of individual transactions.
- **Multi-attribute Indexing and Query Support:** Self-described scientific data formats such as HDF5 and NetCDF contain additional descriptive metadata. Oftentimes data is retrieved based on additionally stored metadata. Thus, MOSIQS should provide a capability to search based on object metadata.
- **Flexible Data Sharing and Controls:** Another important goal of MOSIQS is to facilitate scientists and researchers with easier data sharing controls, i.e., ability to export or publish a particular PMO or a collection of PMOs based on certain criteria with other scientists and collaborators. Such PMO sharing also minimizes data movement overhead.
- **Application and User-provided Hints:** The scientific-friendly storage model provided by MOSIQS intends to empower scientists and applications to pass hints to control and set application specific configurations such as creating a shareable and read-only PMO, post-storage tagging of PMO, and excluding metadata extraction or index construction for specific PMOs.

B. TARGET ARCHITECTURE

MOSIQS is a PM object storage framework providing a scalable data management and metadata search service for scientific applications. MOSIQS's target architecture is an array of PM devices distributed across hundreds of compute nodes. PM on each compute node is shared with other compute nodes via a shared PM pool abstraction via high-speed fabric attached memory (FAM) interconnect such as Gen-Z [6], [45]. Figure 2 depicts a high-level architectural overview of the MOSIQS. Multiple compute nodes can create a shared namespace abstraction atop the shared PM pool via the MOSIQS library and directly store and manage memory objects on these namespaces. Multiple processes running at these compute nodes can access and share PMOs via namespace abstraction. Figure 2 shows that a process running at compute node 2 accesses two PMOs from namespace 1 and 2.

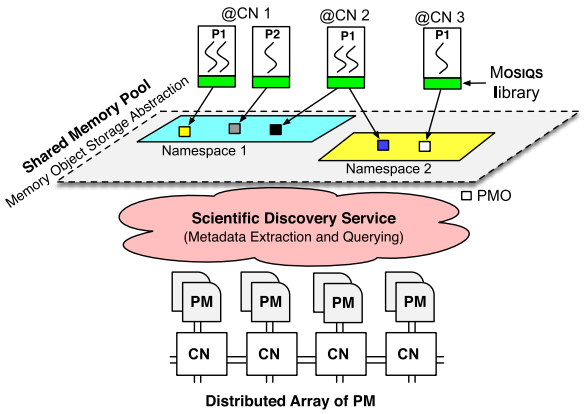


FIGURE 2. MOSIQS target architecture and memory object storage abstraction integrated with scientific discovery service.

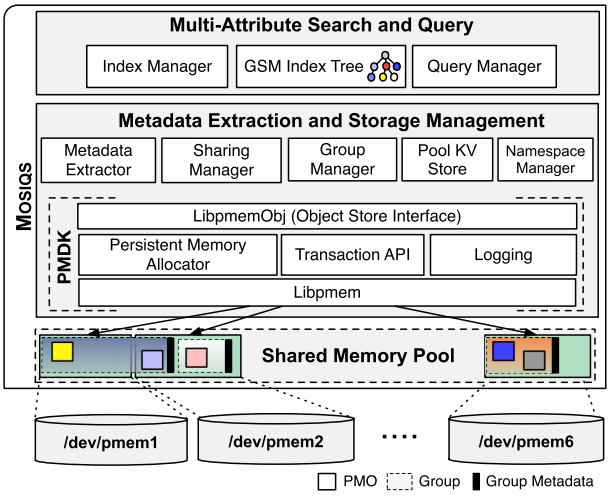


FIGURE 3. An inner layout of MOSIQS architecture.

To enable memory-level object storage abstraction to applications, we employ Intel's PMDK provided `lipmemobj` library, an open source PM object storage interface [20]. On top of the shared memory object storage abstraction, MOSIQS provides applications and scientists with scientific metadata search service to further accelerate the performance and overcome the challenge to find a particular PMO or subset of PMOs.

C. SYSTEM OVERVIEW

MOSIQS primarily consists of the following key abstractions: Shared memory pool, Namespace manager, Metadata extractor, Sharing manager, Group manager, Index manager, and Query manager. Figure 3 presents the multi-layered architecture of MOSIQS. The bottom layer is the shared PM pool, which aggregates all PM devices and exposes them as a single PM pool. Next, the PMDK [20] layer provides low-level primitives, e.g., transactional and reliable object manipulation, via `libpmemobj` and `libpmem`. All the applications attach and detach memory objects from PM pool via MOSIQS, which internally relies on `libpmemobj` interface.

The metadata extraction and storage management layer is stacked on top of the bottom layer. The metadata extractor is responsible to extract and populate the object name and `PMEMoid` mappings. Furthermore, it extracts the annotations, user provided tags and other metadata from the object as well. All the extracted metadata resides in form of key-value paired metadata objects. The sharing manager is responsible to enable the data sharing among applications and collaborators. Group Manager provides logical organization of PMOs defined by application and/or scientists. The group manager in our design plays a crucial role in providing easier and space efficient index metadata. Further, it also benefits in reducing the query search space spanning over multiple attributes. The pool KV store is metadata storage backend for all the metadata of MOSIQS objects. The namespace manager enables flexible controls via partitioning large shared PM pool into application or user-defined namespaces. The third layer provides multi-attribute metadata search and query service and relies on the PMO metadata. The index manager controls the Group Split-Merge (GSM) index data structure. GSM index is a tree data structure designed to provide efficient querying atop pool KV store where all the metadata is stored. The main responsibility of query manager is to serve the query requests from the users/scientists and applications.

D. DATA MODEL

MOSIQS data model consists of three major building blocks.

- **Persistent Memory Object (PMO):** A PMO is a self-described entity and represents a single-value, an array or a compound datatype. It can be created by application or user. A PMO is placed in a group, and additional annotations and hints can be specified. In MOSIQS, a PMO is the minimum sharing currency between applications and users. A PMO requires several properties to be supported: crash consistency to ensure consistent state, system naming, and permission controls to enable PMO to be discovered and shared with other processes and collaborators.
- **Group:** A group represents a collection of PMOs that share common properties and attributes. MOSIQS supports inclusive relationships between groups, i.e., a group can have nested groups similar to nested directories in file systems. Specifically, the group provides a logical organization and a way to store and share collection of PMOs. A key property of a group is its Sharing Frequency (SF), which we discuss later in Section IV.
- **Attribute:** An attribute is a `<key,value>` pair which enables annotations, user-defined tags, and properties of groups and objects. Our attribute concept is the same with attributes in scientific data formats, i.e., HDF5 and netCDF. Each attribute also maintains SF which we explain later in Section IV.

The prior version of this work [9] shows an example application for group and PMO creation with attribute annotations in Listing 1.

E. SHARED PERSISTENT MEMORY POOL

The shared persistent memory pool empowers MOSIQS to provide applications with collective view and an aggregate capacity of an array of PM devices. This satisfies the intense capacity desire of scientific applications [15]. Internally, MOSIQS creates the shared PM pool via `libpmempool` API [20], where the device files, i.e., `/dev/pmem[1-6]` as shown in Figure 3, are concatenated to form a single PM pool. Any object inside the PM pool is reachable via `Root` object pointer. When an application opens a pool, it is given a privilege to access the global memory `Root` pointer, which allows applications to locate the PMOs by accessing metadata stored in the pool KV store. The memory allocations and de-allocations are conducted via `libpmem` at the lower level inside `libpmemobj`.

1) NAMESPACE MANAGEMENT

MOSIQS provides a namespace abstraction atop its data model to enable easier storage for applications using a shared PM pool. A namespace in our design is the same as memory address space for a process except that our namespace is persistent and stays beyond the application lifetime. Each namespace has its own metadata KV storage engine to store and locate PMOs inside the namespace. Applications or scientists using a shared PM pool can access PMOs in another namespace, provided awareness of namespace metadata such as name, owner and access permissions. Such namespace management offers an easier and simpler storage model per application or scientist.

F. METADATA EXTRACTION AND STORAGE

1) METADATA EXTRACTION

We analyzed that a general design technique that proved crucial for MOSIQS is simplifying and minimizing the number of operations in critical I/O path. The key idea to extract and store PMO metadata and user/application annotated tags is to enable sharing and to build indexes for quick access, efficient retrieval of PMO and to enable future analysis. The metadata extractor is implemented as a service by which application or user annotated tags can be extracted from group or PMO. It creates a single metadata KV object for each PMO or group and inserts it in pool KV store. MOSIQS defines its own layout of metadata object for PMO and group.

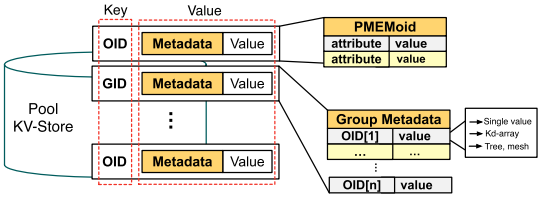


FIGURE 4. The self-described metadata KV objects in pool KV store.

Figure 4 shows an overview of extracted and stored metadata KV object of both types, PMO metadata and group metadata object in pool KV store. The OID denotes the PMO object, whereas GID refers to the group metadata object. The value in `<OID|GID,Value>` pair as shown in Figure 4 itself represents an additional self-described entity, i.e., motivated by scientific data formats [33], [34]. We further partition the value part into header part and data part, as shown in Figure 4. For each OID, the header contains the metadata information such as `PMEMoid`, whereas the data part contains associated attributes and annotated values provided by the user or application to a particular PMO. Note that, each OID points to a single PMO stored in MOSIQS. For each GID, the header contains the metadata of group such as annotated attributes and sharing scope of the group as shown in Figure 4. Whereas, the data part contains the list of PMOs sharing the same set of attributes, along with their unique values which can be a single-valued string or an integer or a complex composite data structure such as a tree or a mesh. The motivation behind storing OID and GID as `<k,v>` metadata objects in pool KV store provides multiple benefits, i) easier access to PMO, ii) flexible and extensible tagging, iii) efficient metadata search queries.

To ensure the consistency of metadata extraction, we encapsulate each operation as a transaction backed by a logging approach. To minimize the performance degradation, we perform metadata extraction in the background and `<OID|GID,Value>` pair populates synchronously in pool KV store. Both metadata extraction operation and `<OID|GID,Value>` pair population is executed in parallel. For data object consistency, we rely on `libpmemobj` provided consistency semantics. All the PMOs annotated with bypass index hint are excluded by metadata extractor from extraction operations. For such objects, only object mapping, i.e., object name to `PMEMoid` is stored.

2) OBJECT SHARING CONTROLS

We design MOSIQS aiming to make it as simple as possible for scientists and applications to enable fast memory-level object sharing. PMDK [20] provides persistent pointers, i.e., `PMEMoid` and handles an internal virtual address mapping indirection to the memory base address to tolerate application crashes and ungraceful shutdowns. Therefore, sharing a PMO beyond the application bounds to other applications or scientists requires storing the persistent pointer of PMO. Whereas, other applications or scientists are unaware of such memory pointer addresses and instead use object naming semantics to share objects. For this reason, we keep object mapping information in the pool KV store as explained earlier (Subsection III-F). We provide sharing controls at two levels, i.e., object and group level. For object-level sharing, an application or scientist requests an object. The sharing manager receives the request and checks the requested object mapping in the pool KV store. If the object entry is found, the sharing manager checks the object scope and properties. If the object is shareable, then the sharing manager returns the `PMEMoid` to requesting application or scientist.

To further ease the sharing controls and bring similarity closer to POSIX like permissions controls, a group can be marked as a shared group that minimizes the data sharing

overhead, i.e., sharing a directory in file system compared to sharing an individual file. An application or a collaborator initiates a sharing request for a group. In such a case, the sharing manager validates the group scope and properties from the pool KV store. If the group is annotated with a global and shared scope then, returns the list of OIDs enclosed in the group data part to the requesting application or collaborator. Note that, the group-level abstraction provides file system like semantics, e.g., `ls -l` on a shared group works similar to `ls -l` on a shared file system directory.

G. METADATA SEARCH AND QUERY

MOSIQS enables scientists and applications to search and query PMOs using the metadata stored in the pool key-value store. However, despite the simpler API for metadata storage, an inherited limitation of key-value store is; lack of multi-attribute or multi-dimensional search queries, which is a common trend in scientific applications and communities [2], [27], [30], [46]. Scientific data is largely unstructured and contains a lot of descriptive metadata in the form of key-value pair attributes [26], and retrieving the desired dataset usually depends on such multiple metadata attributes [30], [47]. Besides, such retrieval often includes additional tags and annotations in search query provided by scientists and applications [2], [26]. Hence, persistent memory objects also require such multi-attribute metadata search services to accelerate the performance of scientific applications. However, simply utilizing existing index data structures for billions of memory objects on an aggregate memory pool entails ample query search space and index metadata storage overhead.

To overcome large query search space and index storage overhead challenges, we intend to introduce and embed a persistent yet simple index data structure on top of PMEMKV [35] to improve scientific queries spanning over multiple metadata attributes. We present the proposed persistent index data structure in the next Section IV.

H. PMO PORTABILITY AND MOBILITY

MOSIQS offers high portability and mobility of PMOs, i.e., PMOs can be packed into any scientific data format and flushed to underlying storage for later use in longer future. Therefore, we consider the portability and mobility of PMOs in MOSIQS for scientific applications to be favorable when compared to domain-specific solutions. However, it is not in scope of this study that how objects will be drained to underlying storage system.

IV. GSM: MULTI-ATTRIBUTE INDEXING AND QUERYING

Group Split and Merge (GSM) is a dynamic index data structure designed for PMO metadata indexing and querying. It leverages the property that memory objects have many common attributes shared among each other and query can benefit from such shared associations. Realistically, a scientific object can be associated with more than a hundred attributes. For example, MIQS [25] stated that an HDF5 file

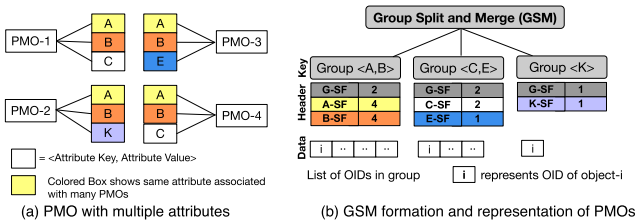


FIGURE 5. Representation of PMOs and overview of group split and merge index tree. SF denotes the sharing frequency.

object contains more than 200 unique attributes, which overlap with other objects in the same file. We elaborate it by using an example of scientific objects associated with multiple attributes. For the sake of simplicity, we use only 3 attributes per each object. Figure 5(a) presents multiple objects with multiple attributes. The white box attributes are unique to each object, whereas, colored box are duplicate attributes which are common among objects as shown in Figure 5(a). Note that, as mentioned earlier (Section III-D), the attribute is a key-value pair and the same attributes across many objects not necessarily have the same value, i.e., the attribute value can vary from object to object in the pool key-value store.

To transform such associations to the GSM index, we extract metadata and associated properties/tags from each object when it is created and pass it to the index manager for further processing. The index manager determines the location of the object in the GSM tree. The location is determined based on the associated attributes of the object. We strive to keep objects with sharing or overlapping attributes into the same group. Recalling, a group object is defined as a set-associative structure to accommodate multiple objects in a single key-value entry, as shown in the Figure 4. A single object may overlap multiple groups. As there is a high probability that future queries will be performed on the attributes with a higher number of associations. Figure 5(b) shows the GSM index tree build from the objects with unique and overlapping attributes. We introduce and bind an important element to each group and attribute, i.e., Sharing Frequency (SF). The group SF is defined as: the number of attributes contained in a group, whereas attribute SF denotes the number of objects associated with a particular attribute contained in a group. For example, as shown in Figure 5(b) the Group <A, B>'s SF is 2 because it maintains two attributes. On the contrary, attribute A and B sharing frequency is 4 because there are 4 PMOs associated with these two attributes, as shown in Figure 5(a).

A. GSM: INSERT OPERATION

When the application or user creates a group, the group manager simply adds it to the first level of the index tree unless an existing group is annotated. The insert operation starts upon receiving an object along with its tags and attributes. There are two possible ways to associate an object to a group in the GSM index tree. First, an application or user-defined group

for the object, where group annotation is provided at object creation, as shown in Listing 1 in our prior work [9]. We refer to such an object as *Tagged Object*. Second, if no group is annotated to object by application or user, then we search the appropriate group for object placement based on associated attributes. We refer to such an object as *Free Object*.

Therefore, the insert algorithm treats each object differently, i.e., ❶ Tagged Object, and ❷ Free object. For ❶, the insert operation is simple; we select the group and update the attribute's SF, and check if the group or attribute SF meets split and merge threshold value. If split and merge threshold is not violated, then we update the group value accordingly in the pool key-value store.

For ❷ Free object, we first scan the groups based on matching attributes; if a group is found, the rest of the procedure is carried out in the same fashion with the tagged object. Note that the insertion of a free object is carried on per attribute. There is a high possibility that free objects with multiple attributes/tags will overlap in multiple groups. Figure 6 presents the insertion of a free object in the GSM index tree. If no group is found then, we create and initialize a new group, update the header of the newly created group, associate object to the group and write the group metadata object in pool key-value store.

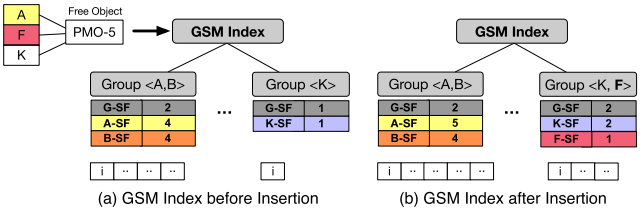


FIGURE 6. A free object insertion in GSM index tree. The changes are shown in bold and red box. Note that, for simplicity we only show the groups with changes.

B. GSM: SPLIT OPERATION

When a thread has exhausted all the options to insert an entry into GSM tree without violating the split threshold or SF value, it triggers a group split operation, which is possibly the expansion of the GSM index tree. A split threshold is a tunable parameter configured by application or user to minimize the query search space. Figure 7 shows the split operation in GSM index tree. Such group split not only offers high parallelism for querying but also enables a broad-range of queries, such as splitting the group based on provided query hint. For example, `Group<A1>` contains objects with temperature less than a certain value, and `Group<A2>` contains the objects with temperature value greater than a certain value, which is a common querying pattern in the scientific community. Such a split trims the query cost and latency by excluding the undesired results from the query.

C. GSM: MERGE OPERATION

On the contrary to split operation, when the sharing frequency drops below a threshold, groups can be merged to minimize

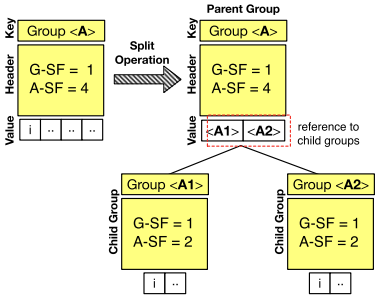


FIGURE 7. A split operation in GSM index tree.

index storage overhead and to accelerate the query. The merge operation is required in cases when objects are overlapping in multiple groups, i.e., sharing attributes from multiple groups. The merge operation squeezes the GSM tree. A merge operation is useful in minimizing the number of additional I/Os where a query criterion contains multiple attributes, and irrespective of query search space, multiple groups, are iterated to find the desired datasets. Another use-case of the merge operation is; when an application provides a query hint in advance, then groups can be merged to execute the query efficiently.

Both split and merge operations can be carried out depending on application, sharing frequency or user-defined query patterns.

D. CRASH CONSISTENCY

There is a need to ensure that GSM index tree is modified atomically when an insert, split or merge operation is performed, in order to provide consistency after a crash due to application error, power loss, or hardware failure. For this reason, we employ a unified logging (ulog) approach, i.e., an undo log for metadata operation and redo log for data operation. The undo log prevents the inconsistency of group metadata such as attribute and group sharing frequency. Whereas, a redo log preserves the group, attribute, and object association. The size of our log records is 8 byte and we mostly write two log records for each operation in GSM tree. Note that, the redo log hits group data part and can sustain duplicate results, whereas we avoid duplication in group header as query relies on header portion and such duplication in header results in population of false positive results.

E. MULTI-ATTRIBUTE SEARCH AND QUERYING

As stated in previous studies [26], [30], the queries bound on multiple attributes are necessarily the most interesting or complex type of queries encountered in real scientific applications. In fact, it is usually the case that specific properties may largely be the unknown parts of queries; indeed, one may query for relationships between objects spanned over multiple files (consider, for example, to retrieve all the objects having temperature and pressure attribute generated at timestamp *t1* from simulation *s1*). Therefore, we intend to

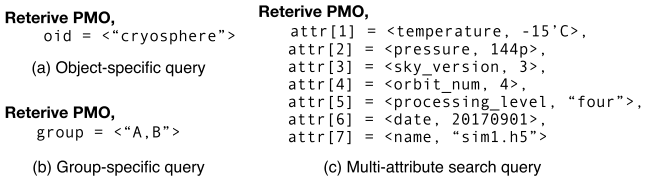


FIGURE 8. An example of versatile querying provided by MOSIQS. In particular, (a) and (b) shows a point query.

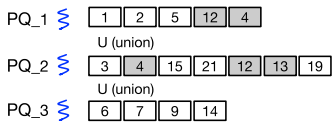


FIGURE 9. An example of union operation in parallel querying. The gray-color boxes represent duplicate OIDs in result. PQ_i denotes parallel queries issued to multiple groups.

provide such multi-attribute querying on persistent memory objects in MOSIQS.

Here, we describe how MOSIQS efficiently responds to queries spanning over multi-attribute ample search space. A scientist or an application provides a query in a string form, comprising of a specific field or spanning over multiple attributes, as shown in Figure 8. For instance, *retrieve all the PMOs associated with set of attributes as defined in Figure 8(c).* The query manager scans the group for query attributes in the GSM index tree. If all the attributes are in a single group header, then we read the group metadata object from the pool key-value store and return the list of `OIDs`. However, if different groups are responsible for storing the attributes and parallel queries are initiated to each group, the result can contain the duplicates OIDs, as shown in Figure 9.

The duplication may arise due to overlapping `OID` across many groups, i.e., PMO associated with many attributes. For such queries, it is critical to minimize the duplication from the query results. Therefore, the query manager performs the union and sort operation on the resulting list of `OIDs` and returns the list with unique OIDs.

Note that, we consider extending and improving the GSM index tree for attribute-value based querying on persistent memory objects as our future work.

V. EVALUATION

This section presents MOSIQS performance evaluation.

A. EXPERIMENTAL SETUP

1) TESTBED

We perform our experiments on a machine equipped with second-generation Intel Xeon scalable dual-socket 56-core processor (hyper-threading enabled) with 1.5 TB Intel Optane DC 3D-XPoint PM, and 768 GB DRAM. Each socket has two memory controllers, six memory channels, and twelve memory DIMM slots installed with 12 × 128 GB PM. Optane DC PM is configured in 100% App direct mode without DRAM intervention, so that application has direct byte-addressable access to the PM. We used PMDK 1.7 and Linux Kernel version 5.4.30 (Ubuntu 18.04.2) for compiling and running the experiments. Note that our target architecture is a distributed shared PM pool. However, for evaluation, we consider a single PM device as a shared PM pool where it is shared multiple processes on the Intel Xeon scalable server. The peak write throughput for 28 cores is 6.6 GB/s and peak read throughput is 23 GB/s measured with Memory Latency Checker [48].

TABLE 2. HDF5 dataset used for evaluation [49].

Item	Description
Total Dataset Size	101 GB
Total File Count	4137 HDF5 Files
Average File Size	25 MB
Average Objects per File	2167 HDF5 Objects
Average Attributes per Object	37

2) IMPLEMENTATION

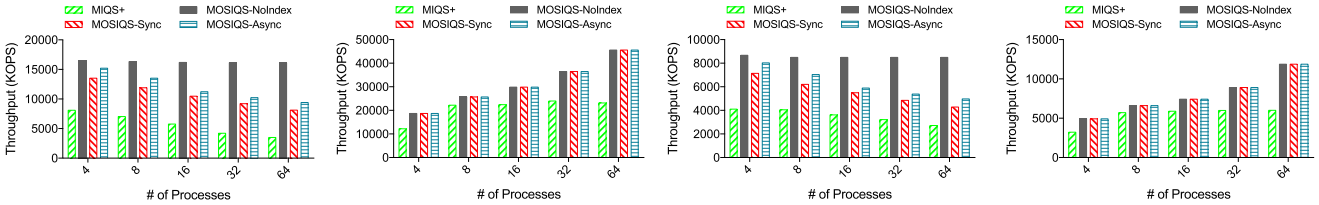
We modified Intel's PMDK library and provide a wrapper API on top of persistent memory object API. For metadata backend storage, we used PMEMKV [35] with a fully persistent B+-Tree storage engine. Though it is not production-ready yet, we still used it as the goal of our study is to show the feasibility of the proposed PM application framework. We use R-trees to implement our GSM index data structure and internally, rely on PMDK transactions for GSM split and merge operations to ensure atomicity. This frees us from handling low-level memory management while guaranteeing safe and atomic allocations.

3) BENCHMARK AND WORKLOADS

We use PIOK [24] benchmark to evaluate the performance of MOSIQS. In particular, we use two PIOK provided kernels, i.e., VPIC-IO and BDCATS-IO to show the read and write performance. VPIC-IO is an extracted kernel that simulates the particle data write behavior by the real VPIC scientific application [50]. Similarly, BDCATS-IO demonstrates the data read patterns of a parallel program that analyze the particle data generated by VPIC [50]. We modified the two kernels using MOSIQS object storage abstraction API, where each particle denotes a PMO and is associated with a fixed metadata. We changed 120 and 270 lines in VPIC-IO and BDCATS-IO to adopt MOSIQS. We show the efficiency of MOSIQS's GSM indexing and multi-attribute querying by using the most commonly used HDF5 scientific data format. We download real-world publicly available NASA's GLAS/ICESat L2 Sea Ice Altimetry HDF5 dataset [49]. The dataset details are provided in Table 2. We run each experiment multiple times and report the mean. In all cases, the standard deviation was less than five percent of the mean.

We compare our approach with the following systems:

- **MIQS+:** We implement and emulate MIQS [25] on top of ext4-DAX file system mounted PM and refer to it as



(a) VPIC (Write IO, 256B PMO) (b) BDCATS-IO (Read IO, 256B PMO) (c) VPIC (Write IO, 512KB PMO) (d) BDCATS-IO (Read IO, 512KB PMO)

FIGURE 10. MOSIQS I/O performance analysis by varying number of processes using 256B and 512KB PMO size.

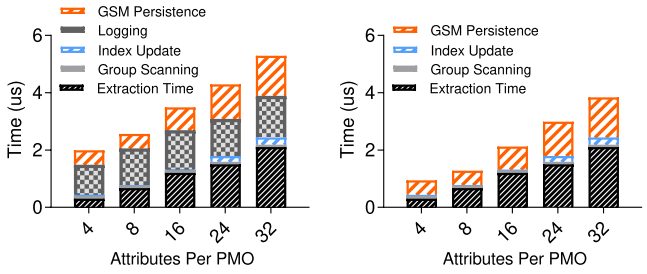
MIQS+. MIQS [25] implements various DRAM-based index data structures such as ART and SBST trees to maintain HDF5 file indexes for querying on self-describing scientific datasets, stored in parallel and distributed file systems. The metadata indexing is conducted after the data is written successfully.

- **MOSIQS-NoIndex:** MOSIQS with no metadata indexing and search service, but including the software implementation overhead of MOSIQS on top of PMDK [20].
- **MOSIQS-Sync:** MOSIQS with metadata extraction enabled in inline synchronous mode, i.e., metadata populates in pool key-value store and write operation finishes.
- **MOSIQS-Async:** MOSIQS with metadata extraction enabled in inline asynchronous mode, i.e., metadata populates in pool key-value store after the write I/O. We use separate dedicated threads executing concurrently, one for processing application I/O and another for metadata extraction.

B. I/O PERFORMANCE ANALYSIS

1) THROUGHPUT ANALYSIS

Figure 10 shows the peak throughput of read and write operations with varied number of processes using a fixed PMO (i.e., 256 Bytes). As, observed from the Figure 10 (a) and (c), MIQS+ performs poorly compared to the proposed MOSIQS variants. MIQS+ access data in the block size granularity exposed to the OS, which is typically 4KB. Further, MIQS+ always needs to go through the I/O stack to fetch data, adding extra system call overheads. We observed that in MIQS+, the IO stack overhead has a much higher impact than the write amplification due to block size mismatch, i.e., MIQS+ wastes I/O bandwidth if the required I/O size is smaller than the block size. If the block size is bigger, MIQS+achieve better bandwidth, but I/O stack overhead remains the same. On the other hand, this overhead can be easily amortized in MOSIQS variants as there is no file system or kernel involved. However, throughput difference in MOSIQS variants is mainly attributed to the additional metadata extraction and management in the critical I/O path. MOSIQS-NoIndex shows a consistent performance trend with varied processes. It reaches the peak write bandwidth including our software implementation overhead. It incurs a single metadata insertion operation per I/O to populate a mapping entry in pool key-value store compared to MOSIQS-Sync and Async approach. Therefore, with varying processes we can see performance drop in MOSIQS-Sync and Async. For read throughput, we observe



(a) With logging (b) Without logging

FIGURE 11. MOSIQS's index construction with and without consistency overhead.

a scalable performance trend for all the approaches as shown in Figure 10(b) and (d).

2) BANDWIDTH ANALYSIS

We have already discussed the read/write bandwidth of MOSIQS with varying PMO size in our previous work [9]. The peak write bandwidth of MOSIQS-NoIndex is 63% of 6.6 GB/s due to PMDK's internal transaction management, atomic memory allocations, pointer assignments and MOSIQS's object to persistent pointer mappings. Its peak read bandwidth is 40% of 23GB/s with 512KB PMO size. It is mainly due to iMC's cache misses, accessing object's persistent pointer and PMDK's internal persistent pointer to memory address translation. As expected, we observe a scalable read performance trend in all MOSIQS variants. The difference in read and write throughput of MOSIQS and its variants is mainly derived from PM properties, i.e., the read and write performance of PM is highly asymmetric.

C. MULTI-ATTRIBUTE INDEXING AND QUERYING

Here, we present the evaluation of metadata search and query service provided by MOSIQS.

1) INDEX CONSTRUCTION AND CONSISTENCY OVERHEAD

Figure 11(a) & (b) depict the average index construction runtime of a single PMO with and without ensuring the consistency. We varied the number of attributes per PMO to measure the time spent on each operation for constructing index. We observed that all operations such as metadata extraction, group scanning, and GSM metadata object persistence linearly increase with varying number of attributes except the index update sub-operation. The index update incurs unified logging overhead, i.e., a redo log and an undo log for data and

TABLE 3. Versatile search queries to measure the query performance. Attribute column shows metadata object required to answer the query.

	Description	Attributes	Query Search Space
Q1	Locate PMO with name containing '9610/Inf'.	OID	PMO-specific
Q2	List all PMOs contained in group 'GLAH_634_2121'.	GID	group-specific
Q3	Count the number of all attributes in a group 'GLAH_634_2121'.	GID	group metadata-specific
Q4	Find set of PMOs associated with 'orbit_num' attribute and name containing '9610/Inf'.	GID, OID[ ]	single group, multiple PMOs
Q5	List all PMOs which have attribute 'orbit_num, sky_version, processing_level' generated on '20140928' and 'd_lat, d_lon'.	GID[ ], OID[ ]	multiple groups, multiple PMOs

metadata updates. However, excluding logging overhead can benefit from improved runtime but at the cost of inconsistency tolerance.

2) INDEX STORAGE OVERHEAD

Here, we show the index storage overhead of MIQS+ and MOSIQS variants with variable group sharing frequency values. Recall, group sharing frequency defines the number of attributes contained in a group, which directly impacts the index storage space overhead. Figure 12 presents the index storage space consumed by MIQS+ and MOSIQS variants. #A/G in Figure 12 denotes group sharing frequency, i.e., number of attributes per each group. The distribution of attributes in group and attribute sharing frequency affects the index storage overhead. For instance, if attribute sharing frequency is less per group, then there is a high probability that attribute will overlap across the groups, which in turn increases the index storage overhead. Similarly, if a group has many attributes and attribute sharing is very high that all associated PMOs can fit in a single group, then index storage overhead will be very minimum. We use real HDF5 dataset files and observe that MOSIQS-4A/G (4 attributes per group) consume almost half index storage space compared to MIQS+.

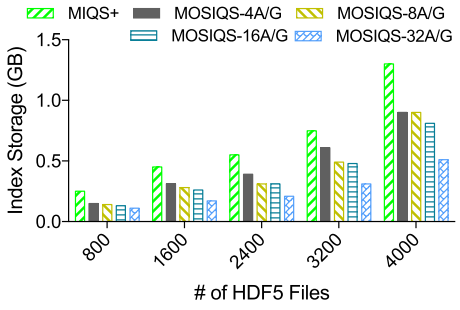
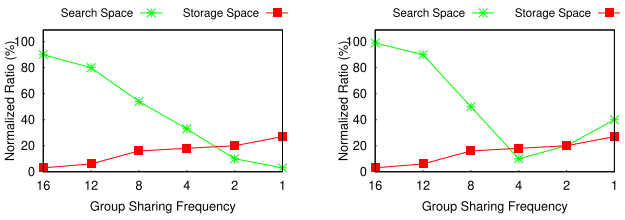


FIGURE 12. MOSIQS's index storage space overhead. #A/G denotes number of attributes per group.

Surprisingly, with increasing count of HDF5 files, i.e., 4K, MIQS+ takes up almost 1.35GB. On the contrary MOSIQS–4A/G and 32A/G consume only 920MB and 500MB of index storage space for a total of (`37 Attributes × 2167 PMOs × 4K Files`) PMOs. Note that, high sharing of attributes among the group can increase the overlapping PMOs across the groups. Therefore, GSM index tree intelligently split and merge based on the attribute sharing frequencies to minimize the storage space overhead. However, to clearly show the storage overhead, we do not split or merge the GSM index tree. Further, MIQS+ incurs index storage overhead in DRAM, whereas MOSIQS's storage overheard is PM overhead. DRAM is more expensive and lower density than PM.

3) GSM LOAD FACTOR

Here, we explain the correlation between attribute and group sharing frequency while taking into account multi-attribute query, index storage overhead, and most importantly, query search space. To analyze the trade-off between query search space and index storage overhead, we insert 1 Million zero-size PMOs, each tagged with 16 attributes. We consider two multi-attribute queries, i) Q1 includes a single metadata attribute and Q2 uses only 4 attributes with sharing frequency ≤ 20. Figure 13(a) & (b) show the query search space and storage overhead for each query type. For Q1, the query search space for PMOs is very high when a single group is managing all the attributes. However, the index storage overhead is very small because GSM manages attribute indexes at group abstraction instead of individual PMO as shown in Figure 13(a). On the contrary in Figure 13(b) we observed that for Q2, attribute sharing frequency based group creation is highly important as attribute sharing frequency enables us to trim the query search space. We observed that an optimal group sharing frequency value for our workload and query is around 3-4, i.e., 4 attributes per group considering query search space and storage space overhead.



(a) Q1: Single attribute per query (b) Q2: 4 attributes per query

FIGURE 13. Load factor between query search space and storage space. The storage and query search space % is normalized with respect to MIQS+ (100%).

Note that, the MIQS+ has an imbalance load factor among the index trees, i.e., when a particular attribute shows a higher association with PMOs, the resulting tree can be of higher depth thus overloading a particular index tree.

4) MULTI-ATTRIBUTE QUERY PERFORMANCE

To analyze the query performance using realistic scientific HDF5 datasets, we use previously populated PMO mappings and GSM index tree from the experiment. The group sharing

TABLE 4. Query throughput for queries defined in Table 3. EE shows an end-to-end query and data retrieval time.

Query	MIQS+		MOSIQS-Fixed		MOSIQS-Dynamic	
	kQPS	EE Time(s)	kQPS	EE Time(s)	kQPS	EE Time(s)
Q1	28184	1.25	25478	0.10	25478	0.10
Q2	37766	8.25	30590	0.19	30590	0.19
Q3	37756	12.25	29827	1.1	34357	0.19
Q4	19892	17.25	14123	2.3	18499	2.3
Q5	17895	43.25	10915	3.4	16642	3.4

frequency is set to 4 as explained earlier. We define five realistic PMO metadata, and associated attribute queries, as shown in Table 4. In particular, **Q4** and **Q5** represent a query spanning across multiple groups. Therefore, the query search space will profoundly impact Q4 and Q5 throughput as compared to Q1-Q3. For this experiment, we compare MIQS+ with two variants of MOSIQS. MOSIQS-Fixed, manages all attributes in a single group and do not exhibit any split or merge operation. Whereas, MOSIQS-Dynamic adjusts group and attribute sharing frequency dynamically. For example, if two groups have high overlapping objects or many groups containing low attribute sharing frequency, we merge the two groups dynamically to minimize index storage overhead and number of additional I/Os. Similarly, if a group has high attribute sharing frequency than pre-configured (4 in this case), MOSIQS-Dynamic splits the group in two sub-groups, i.e., child groups to benefit with query parallelism. Table 4 shows the average query throughput and end-to-end time (query time + time to read the data object). In real scientific usecases, such queries are used to find and retrieve the data items. Therefore, we measure end-to-end time because MIQS+ caches the indexes in DRAM, whereas data item retrieval requires accessing disk storage system. We read varied number of PMOs against each query for MIQS+ and MOSIQS variants. On average, MOSIQS-Dynamic only shows 7% query throughput degradation and MOSIQS-Fixed shows 27% compared to MIQS+. However, for end-to-end time, MOSIQS outperforms the MIQS+ due to dataset storage location, i.e., PM pool against the file system.

VI. RELATED WORK

In this section, we review the related work on in-memory indexes, PM in scientific computing along with significance of metadata indexing and querying.

A. IN-MEMORY VOLATILE INDEXES

An approach to build a high performance scientific data management system is to store and index all of its metadata and at least significant data in the main memory [31]. Several scientific and database communities have adopted such in-memory ordered and unordered index data structures including B/B+-trees, Hashes, Tries, and Radix trees to mitigate the data access problems [30], [31]. MIQS [25] provides a classical example of building such in-memory indexes to provide efficient querying support on scientific datasets. MIQS employs multiple trees such as ART and SBST altogether to ensure quick retrieval of dataset. However, in-memory volatile data structures or DRAM-resident indexes have an inherent limitation, i.e., they cannot survive power failures and unexpected crashes [32]. A simple power-failure makes the index unreachable and requires rebuilding or recovering the whole index.

B. PM IN SCIENTIFIC COMPUTING

A few recent studies employ PM to benefit scientific applications [4], [14], [17], [41]. NV-Process [41] proposed a fault tolerance process model based on PM and provides an elegant way for the applications to tolerate system crashes. Similarly, [17] evaluates different fault-tolerance approaches for scientific applications to use PM. pVM [14] provides the dynamic expansion of the virtual memory technique with an object storage abstraction on PM. DAOS-M [18] proposed an object storage model on PM and NVMe SSDs, using PM for small writes and metadata, whereas large objects are written directly on SSDs. However, pVM and DAOS-M lack scientific metadata indexing and querying for memory-level objects. Further, these studies also lack data sharing controls, an important requirement of scientific computing [27].

C. SCIENTIFIC DATA INDEXING AND QUERYING

Several studies have proposed various solutions and optimizations to improve the scientific indexing and querying tightly integrating into the file system or using external databases [2], [9], [25]–[27], [30]–[32], [46], [51]–[56]. SoMeta [51] provides scientific metadata querying via scalable self-described object and map objects to files internally at the file system level. GUFI [55] embeds SQLite in every directory hierarchy to improvise indexing and metadata query in deeply nested file system directories. Brindexer [46] employs a relational database to track and manage metadata changes on the lustre. TagIt [2] offers a file system integrated indexing and querying for a large collection of scientific files stored on a shared-nothing distributed file system. Spyglass [56] proposed a hierarchical partitioning approach to exploit locality for effective querying. EMPRESS [54] stores and manages metadata in PostgreSQL. Dataspaces [57] proposed a general programming model for data exchange using virtual shared space abstraction with online data indexing support and querying for coupled scientific workflows. Similarly, ExaHDF5 [24] provides metadata extraction and querying for scientific data formats such as HDF5 and NetCDF. Furthermore, scientific data queries vary from simple point query to prefix, suffix and range query based on complex selection criteria involving multi-attributes as described in [26]. A few common challenges solved by previous studies involve: effective metadata extraction and indexing, efficient metadata management, and querying over domain-specific datasets.

A common goal among previous studies and our work is to enable metadata indexing, and querying over a large collection of diverse and complex layout scientific datasets. However, it is very challenging to build a general framework due to a variety of applications, diverse scientific data

formats, and APIs. Moreover, most of the aforementioned solutions are built on the notion of underlying parallel file system, which is block-addressable. Thus, requires a lot of engineering and complex implementation efforts to apply on persistent memory. Many of the challenges we face are not radically new, but take on new characteristics when coupled with object storage abstraction on shared PM pool with high-speed interconnect such as Gen-Z [6], e.g., ensuring consistency and failure-atomicity in case of application crashes and ungraceful power failures.

Therefore, to this end, MOSIQS provides a memory-level persistent object storage abstraction via a shared memory pool on top of distributed array of PM devices. The scientists and collaborators can create and remove persistent objects from shared memory pool via MOSIQS. Moreover, MOSIQS is equipped with metadata indexing and efficient multi-attribute querying mechanism to accelerate scientific computing applications.

VII. CONCLUSION

In this paper, we present MOSIQS, a persistent memory object management system to accelerate scientific computing. MOSIQS provides application to efficiently attach and detach memory objects into their address space and enables effective sharing of persistent memory resident objects across different applications and collaborators. The proposed PM-based application model not only allows effective metadata extraction and tagging of memory objects but is also equipped with indexing and querying service aimed at scientific datasets to further accelerate scientific experiments, simulations and analysis. The preliminary evaluation confirms a 100% improvement for write and 30% in read performance against a PM-aware file system approach. Further, GSM index data structure incurs 2.7× less storage index space compared to existing state-of-the-art in-memory indexing approach. MOSIQS application model has proven to be extremely capable, and usable in a wide variety of applications. We consider minimizing the translational overhead to deference persistent pointers for direct load and store accesses as our future work.

REFERENCES

[1] H. Tang, S. Byna, S. Bailey, Z. Lukic, J. Liu, Q. Koziol, and B. Dong, "Tuning object-centric data management systems for large scale scientific applications," in *Proc. IEEE 26th Int. Conf. High Perform. Comput., Data, Anal. (HiPC)*, Dec. 2019, pp. 103–112.

[2] H. Sim, A. Khan, S. S. Vazhkudai, S.-H. Lim, A. R. Butt, and Y. Kim, "An integrated indexing and search service for distributed file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 10, pp. 2375–2391, Oct. 2020.

[3] A. Khan, H. Sim, S. S. Vazhkudai, A. R. Butt, and Y. Kim, "An analysis of system balance and architectural trends based on Top500 supercomputers," in *Proc. Int. Conf. High Perform. Comput. Asia–Pacific Region (HPC)*, New York, NY, USA, Jan. 2021, pp. 11–22, doi: 10.1145/3432261.3432263.

[4] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang, "Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using Intel optane DC persistent memory modules," in *Proc. Int. Symp. Memory Syst. (MEMSYS)*, New York, NY, USA, 2019, pp. 288–303, doi: 10.1145/3357526.3357541.

[5] H.-W. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: Creating application objects efficiently for heterogeneous computing," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 53–65.

[6] K. Keeton and S. Spence. (2017). *Persistent Memory: A New Tier or Storage Replacement*. Accessed: Mar. 19, 2020. [Online]. Available: https://www.snia.org/sites/default/files/SDC/2017/presentations/General_Session/Keeton_Kimberly_Spence_Susan_Persistent_Memory_New_Tier_or_Storage_Replacement.pdf

[7] K. Asanović, *Firebox: A Hardware Building Block for 2020 Warehouse-Scale Computers*. Santa Clara, CA, USA: USENIX Association, Feb. 2014.

[8] Intel. (2020). *Intel Rack Scale Design Architecture*. Accessed: Apr. 10, 2020. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf

[9] A. Khan, H. Sim, S. S. Vazhkudai, J. Ma, M.-H. Oh, and Y. Kim, "Persistent memory object storage and indexing for scientific computing," in *Proc. IEEE/ACM Workshop Memory Centric High Perform. Comput. (MCHPC)*, Nov. 2020, pp. 1–9.

[10] Y. Xu, Y. Solihin, and X. Shen, "MERR: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst. ASPLOS*, New York, NY, USA, Mar. 2020, pp. 987–1000, doi: 10.1145/3373376.3378492.

[11] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, Santa Clara, CA, USA, Feb. 2020, pp. 169–182. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/yang

[12] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *Proc. ASPLOS*, 2019, pp. 427–439.

[13] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "SplitFS: Reducing software overhead in file systems for persistent memory," in *Proc. 27th ACM Symp. Operating Syst. Princ. (SOSP)*, New York, NY, USA, 2019, pp. 494–508, doi: 10.1145/3341301.3359631.

[14] S. Kannan, A. Gavrilovska, and K. Schwan, "PVM: Persistent virtual memory for efficient capacity scaling and object storage," in *Proc. 11th Eur. Conf. Comput. Syst. (EuroSys)*, New York, NY, USA, 2016, doi: 10.1145/2901318.2901325

[15] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann, "NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 957–968.

[16] D. Bittman, P. Alvaro, P. Mehra, D. D. E. Long, and E. L. Miller, "Twizzler: A data-centric OS for non-volatile memory," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Jul. 2020, pp. 65–80. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/bittman

[17] B. Nesterenko, X. Liu, Q. Yi, J. Zhao, and J. Zhang, "Transitioning scientific applications to using non-volatile memory for resilience," in *Proc. Int. Symp. Memory Syst. (MEMSYS)*, New York, NY, USA, 2019, pp. 114–125, doi: 10.1145/3357526.3357563.

[18] M. Scot Breitenfeld, N. Fortner, J. Henderson, J. Soumagne, M. Chaarawi, J. Lombardi, and Q. Koziol, "DAOS for extreme-scale systems in scientific applications," 2017, *arXiv:1712.00423*. [Online]. Available: http://arxiv.org/abs/1712.00423

[19] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "DAOS and friends: A proposal for an exascale storage system," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, Nov. 2016, pp. 585–596.

[20] (2020). *Persistent Memory Development Kit*. Accessed: Mar. 19, 2020. [Online]. Available: https://pmem.io/pmdk/

[21] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 91–104, Mar. 2011, doi: 10.1145/1961295.1950379.

[22] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "NVthreads: Practical persistence for multi-threaded applications," in *Proc. 12th Eur. Conf. Comput. Syst. (EuroSys)*, New York, NY, USA, 2017, pp. 468–482, doi: 10.1145/3064176.3064204.

[23] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "NV-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *Proc. 16th Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS XVI)*, New York, NY, USA, 2011, pp. 105–118, doi: 10.1145/1950365.1950380.

[24] S. Byna and M. Howison. (2015). *Parallel I/O Kernel (PIOK) Suite*. Accessed: Mar. 16, 2020. [Online]. Available: https://sdm.lbl.gov/exahdf5/software.html
[25] W. Zhang, S. Byna, H. Tang, B. Williams, and Y. Chen, "MIQS: Metadata indexing and querying service for self-describing file formats," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC)*, New York, NY, USA, 2019, doi: 10.1145/3295500.3356146.
[26] W. Zhang, S. Byna, C. Niu, and Y. Chen, "Exploring metadata search essentials for scientific data management," in *Proc. IEEE 26th Int. Conf. High Perform. Comput., Data, Analytics (HiPC)*, Dec. 2019, pp. 83–92.
[27] A. Khan, T. Kim, H. Byun, and Y. Kim, "SciSpace: A scientific collaboration workspace for geo-distributed HPC data centers," *Future Gener. Comput. Syst.*, vol. 101, pp. 398–409, Dec. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18326025
[28] J. Liu, D. Bard, Q. Koziol, S. Bailey, and Prabhat, "Searching for millions of objects in the BOSS spectroscopic survey data with H5Boss," in *Proc. New York Sci. Data Summit (NYSDS)*, Aug. 2017, pp. 1–9.
[29] S. Byna, J. Chou, O. Rubel, Prabhat, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, A. Shoshani, A. Uselton, and K. Wu, "Parallel I/O, analysis, and visualization of a trillion particle simulation," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2012, pp. 1–12.
[30] T. J. Skluzacek, R. Chard, R. Wong, Z. Li, Y. N. Babuji, L. Ward, B. Blaiszik, K. Chard, and I. Foster, "Serverless workflows for indexing large scientific data," in *Proc. 5th Int. Workshop Serverless Comput. (WOSC)*, New York, NY, USA, 2019, pp. 43–48, doi: 10.1145/3366623.3368140.
[31] X. Wu, F. Ni, and S. Jiang, "Wormhole: A fast ordered index for in-memory data management," in *Proc. 14th EuroSys Conf. (EuroSys)*, New York, NY, USA, 2019, pp. 1–16, doi: 10.1145/3302424.3303955.
[32] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proc. VLDB Endowment*, vol. 13, no. 4, pp. 574–587, Dec. 2019, doi: 10.14778/3372716.3372728.
[33] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proc. EDBT/ICDT Workshop Array Databases (AD)*, New York, NY, USA, 2011, pp. 36–47, doi: 10.1145/1966895.1966900.
[34] (2020). *Network Common Data Form*. Accessed: Mar. 20, 2020. [Online]. Available: https://www.unidata.ucar.edu/software/netcdf/
[35] (2019). *PMEMKV: Key/Value Datastore for Persistent Memory*. Accessed: Mar. 19, 2020. [Online]. Available: https://github.com/pmem/pmemkv
[36] (2020). *FITS Astronomical Image and Table Format*. Accessed: Mar. 19, 2020. [Online]. Available: https://fits.gsfc.nasa.gov/
[37] (2020). *Plot3d File Format for Grid and Solution Files*. Accessed: Mar. 20, 2020. [Online]. Available: https://www.grc.nasa.gov/WWW/wind/valid/plot3d.html
[38] (2020). *GRIB Data Format*. [Online]. Available: http://www.cosmomodel.org/content/model/documentation/grib/default.html
[39] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen, "Data elevator: Low-contention data movement in hierarchical storage system," in *Proc. IEEE 23rd Int. Conf. High Perform. Comput. (HiPC)*, Dec. 2016, pp. 152–161.
[40] (2020). *SSDS Survey*. Accessed: Mar. 19, 2020. [Online]. Available: http://www.sdss.org/
[41] X. Li, K. Lu, X. Wang, and X. Zhou, "NV-process: A fault-tolerance process model based on non-volatile memory," in *Proc. Asia–Pacific Workshop Syst. (APSYS)*, New York, NY, USA, 2012, pp. 1–6, doi: 10.1145/2349896.2349897.
[42] J. Jang, S. Junjung, S. Jeong, J. Heo, S. Hoon, H. Tae-Jun, and J. W. Lee, "A specialized architecture for object serialization with applications to big data analytics," in *Proc. 47th Int. Symp. Comput. Archit.*, ser. ISCA '20, 2020.
[43] M. Wilcox. (2014). *Add support for NV-DIMMs to ext4*. Accessed: Mar. 26, 2020. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9028138&tag=1
[44] A. Bhardwaj, A. Deshpande, A. J. Elmore, D. Karger, S. Madden, A. Parameswaran, H. Subramanyam, E. Wu, and R. Zhang, "Collaborative data analytics with DataHub," *Proc. VLDB Endowment*, vol. 8, no. 12, pp. 1916–1919, Aug. 2015, doi: 10.14778/2824032.2824100.
[45] C. C. Chou, Y. Chen, D. Milojicic, N. Reddy, and P. Gratz, "Optimizing post-copy live migration with system-level checkpoint using fabric-attached memory," in *Proc. MCHPC*, 2019, pp. 16–24.
[46] A. K. Paul, B. Wang, N. Rutman, C. Spitz, and A. R. Butt, "Efficient metadata indexing for HPC storage systems," in *Proc. 20th IEEE/ACM Int. Symp. Cluster, Cloud Internet Comput. (CCGRID)*, May 2020, pp. 162–171.
[47] T. J. Skluzacek, R. Kumar, R. Chard, G. Harrison, P. Beckman, K. Chard, and I. T. Foster, "Skluma: An extensible metadata extraction pipeline for disorganized data," in *Proc. IEEE 14th Int. Conf. e-Sci. (e-Sci.)*, Oct. 2018, pp. 256–266.
[48] Intel. (2020). *Intel Memory Latency Checker v3.8*. [Online]. Available: https://software.intel.com/en-us/articles/intelr-memory-latency-checker
[49] H. J. Zwally, R. Schutz, D. Hancock, and J. Dimarzio. (2014). *GLAS/ICESat l2 Sea Ice Altimetry Data (HDF5), Version 34*. Accessed: Jul. 20, 2020. [Online]. Available: https://nsidc.org/data/glah13
[50] K. J. Bowers, B. J. Albright, L. Yin, B. Bergen, and T. J. T. Kwan, "Ultrahigh performance three-dimensional electromagnetic relativistic kinetic plasma simulation," *Phys. Plasmas*, vol. 15, no. 5, May 2008, Art. no. 055703.
[51] H. Tang, S. Byna, B. Dong, J. Liu, and Q. Koziol, "SoMeta: Scalable object-centric metadata management for high performance computing," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Sep. 2017, pp. 359–369.
[52] W. Zhang, H. Tang, S. Byna, and Y. Chen, "DART: Distributed adaptive radix tree for efficient affix-based keyword search on HPC systems," in *Proc. 27th Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2018, pp. 1–2.
[53] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Surf: Practical range query filtering with fast succinct tries," in *Proc. SIGMOD*, 2018, pp. 323–336.
[54] M. Lawson and J. Lofstead, "Using a robust metadata management system to accelerate scientific discovery at extreme scales," in *Proc. IEEE/ACM 3rd Int. Workshop Parallel Data Storage Data Intensive Scalable Comput. Syst. (PDSW-DISCS)*, Nov. 2018, pp. 13–23.
[55] (2019). *GUFI: Grand Unified File Index*. Accessed: Mar. 13, 2020. [Online]. Available: https://github.com/mar-file-system/GUFI
[56] A. W. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. L. Miller, "Spyglass: Fast, scalable metadata search for large-scale storage systems," in *Proc. 7th Conf. File Storage Technol. (FAST)*, New York, NY, USA, 2009, pp. 153–166.
[57] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An interaction and coordination framework for coupled simulation workflows," in *Proc. HPDC*, 2010, doi: 10.1007/s10586-011-0162-y.

AWAIS KHAN received the B.S. degree in bioinformatics from Mohammad Ali Jinnah University, Islamabad, Pakistan, and the Ph.D. degree in computer science and engineering from Sogang University, Seoul, South Korea, in 2021. From 2012 to 2015, he worked with Digital Research Laboratories, as a Software Engineer. He is currently working as a Postdoctoral Research Associate with the DISCOS Laboratory, Department of Computer Science and Engineering, Sogang University. His research interests include memory-centric computing and HPC, data management services in HPC, object storage systems, cluster-scale deduplication, and parallel and distributed file systems.

HYOGI SIM received the B.S. degree in civil engineering and the M.S. degree in computer engineering from Hanyang University, South Korea, and the M.S. degree in computer science from Virginia Tech, in 2014, where he is currently pursuing the Ph.D. degree. He has been working as an HPC Systems Engineer with the Oak Ridge National Laboratory, where he conducted research and development on active storage systems, burst buffer storage systems, and scientific data management for HPC systems. His research interests include storage systems and distributed systems.

SUDHARSHAN S. VAZHKUDAI is currently the Director of Micron, where he works on understanding data center workloads to improve products in the deep-memory/storage hierarchy and build novel solutions therein. Prior to this, he worked for two decades with the U.S. Department of Energy's (DOE) National Laboratory System, working on the supercomputing and extreme-scale data center programs. He was the Director of the Scalable Data Infrastructure for Science Program, Computing and Computational Sciences Directorate, Oak Ridge National Laboratory (a U.S. DOE Laboratory). In this role, he led an initiative to build and deploy scalable, distributed storage infrastructure, and rich data/metadata management services, to capture and support mountains of scientific data emanating from computer simulations, experiments, and observations, conducted at the various ORNL facilities. In addition, he also contributed to the co-design and deployment of supercomputers and associated solutions for the Nation's Premier Supercomputing Center, Oak Ridge Leadership Computing Facility (OLCF). OLCF is a home to the world's No. 2 supercomputer, Summit, the future Frontier exascale system and the fastest storage system; and Spider, providing billions of core hours to a scientific user base from academia, government, and industry, to perform breakthrough research in science. For around eight years, he led the Technology Integration (TechInt) Group of around 16 researchers, building solutions for supercomputers in several areas, such as file and storage systems, non-volatile memory, data management, system architecture, networking, and distributed systems. He was also a Distinguished Scientist with ORNL and studied the fundamental underpinnings of supercomputers and data centers in many of the aforementioned areas.

YOUNGJAE KIM received the B.S. degree in computer science from Sogang University, in 2001, the M.S. degree from the Korea Advanced Institute of Science and Technology (KAIST), in 2003, and the Ph.D. degree in computer science and engineering from Pennsylvania State University, University Park, PA, USA, in 2009. He was a Staff Scientist with the Oak Ridge National Laboratory, U.S. Department of Energy, from 2009 to 2015, and an Assistant Professor with Ajou University, Suwon, Republic of Korea, from 2015 to 2016. He is currently an Associate Professor with the Department of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea. His research interests include distributed file and storage, parallel I/O, operating systems, emerging storage technologies, and performance evaluation.