Persistent Memory Object Storage and Indexing for Scientific Computing

Awais Khan†, Hyogi Sim*, Sudharshan S. Vazhkudai¶, Jinsuk Ma‡, Myeong-Hoon Oh‡, Youngjae Kim†
Dept. of Computer Science and Engineering, Sogang University, Seoul, South Korea†
Oak Ridge National Laboratory, TN, USA*,
Micron Technology Inc, USA¶, ETRI, Daejon, South Korea‡
{awais,youkim}@sogang.ac.kr, simh@ornl.gov, svazhkudai@micron.com, {majinsuk, mhoonoh}@etri.re.kr

***Abstract*—This paper presents 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 shared 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. MOSIQS uses a lightweight persistent memory key-value store to manage the metadata of memory objects such as persistent pointer mappings, which enables memory object sharing for effective scientific collaborations. MOSIQS is implemented atop PMDK. We evaluate the proposed approach on many-core server with an array of real PM devices. The preliminary evaluation confirms a 100% improvement for write and 30% in read performance against a PM-aware file system approach.**

***Index Terms*—Memory-centric Computing, Persistent Memory Storage, Scientific Metadata Indexing and Search**

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], [2]. 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 [3]. 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 [4]. 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 [5]. The HPC has attempted to adopt MCC by enabling a shared memory storage abstraction across the hundreds of compute nodes [5], [6], [7]. 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 [5] and Infiniband to improvise MCC [7], [6], [5]. However, simply porting scientific applications to MCC infrastructure is challenging. As, applications are tightly coupled to the file system interface, i.e., block-addressable, limits the performance gain expected from MCC. For instance, it has been reported that scientific applications spend 64% of total execution time deserializing file data into memory objects for further processing and computations [4], [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 [9], [10], [11]. 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 [10], [12]. Recently, several studies have shown PM as a full or partial substitute for DRAM [13], [14]. For instance, pVM [13] employs NVRAM to seamlessly expand virtual memory for memory-intensive applications. Similarly, [15] 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].

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 [16], [17], [18]. 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 [19], [20], [21], [22]. Second, scientific application data objects are self-described and packed in versatile scientific data formats, i.e., metadata is embedded inside the data object [23], [24], [25]. Without additional descriptive metadata, PMO may become unidenti-

Y. Kim is the Corresponding Author.

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

fiable, 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], [26], [24]. Fourth, to select a subset of PMOs from millions of PMOs in a shared PM pool based on metadata or user-defined tags without additional indexing becomes highly challenging [27], [28], [26], [29].

To address the aforementioned challenges, 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. Such attractive properties drive the scientists and research communities to have a new memory object style management system which offers scalability, high-performance, easy and flexible data sharing controls.

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).
- 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 [19]. We conduct preliminary evaluations on a Intel many-core server equipped with 1.5 TB real Intel Optane DC 3D-XPoint PM. (Section IV). Experimental results show that MOSIQS gains a 100% performance improvement compared to the PM-aware file system approach.

II. Background and Motivation

In this section, we present the background on emerging persistent memory (PM) and elaborate a need for object storage abstraction to manage scientific data objects on PM pool.

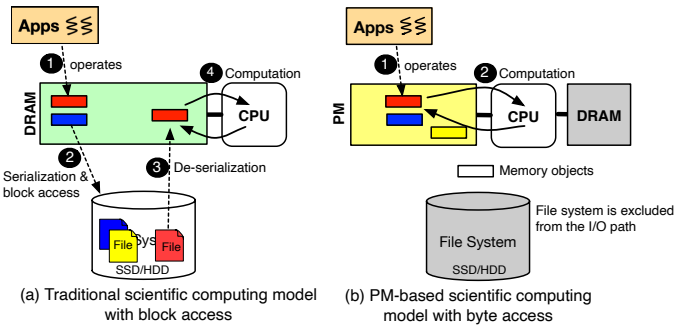


Fig. 1: Traditional scientific computing model vs PM-aware computing model.

A. Memory Centric Computing

The memory centric computing (MCC) has emerged recently to satisfy the requirements of memory-intensive scientific computing applications [30], [31]. 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 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 [10]. A few studies have enabled the use of PM in scientific applications, e.g., NV-Process [32] proposed a fault tolerance process model based on PM and provides an elegant way for the applications to tolerate system crashes. Similarly, [16] evaluates different fault-tolerance approaches for porting scientific applications to use PM. DAOS-M [17] employs PM to store metadata and small writes, whereas larger writes are redirected to NVMe SSDs. Similarly, [15] proposed a data-centric OS based on PM. There are also a few other applications of PM hosting key-value stores and various index data structures to accelerate performance of applications [33], [34].

B. 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 [4]. Such deserialization takes up almost 64% of the application's total execution

time [4], [8]. Figure 1(a) demonstrates the traditional DRAM-based computation model. Figure 1(b) 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), as reported in [12].

C. 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 [12], ext4-DAX [35] specially designed for PM incurs up to 13x 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.

D. Motivation

Arguably, storing application objects directly on PM 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, sharing/protection semantics of PMOs across applications and other scientists are an essential requirement of the scientific community [26], [36]. 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 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 2.

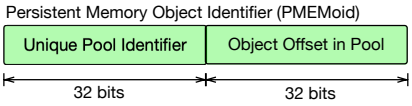


Fig. 2: The layout of PM object identifier (`PMEMoid`) [19], [9].

The `PMEMoid` value for given object does not change during the life of an object/application unless a `realloc()` operation is invoked. 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

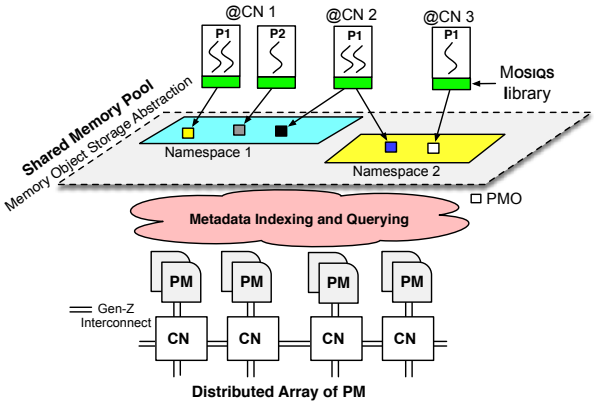


Fig. 3: MOSIQS target architecture and memory object storage abstraction integrated with scientific discovery service.

be crash consistent, i.e., system should ensure access and consistency of memory object in case of application crash or ungraceful power failures.

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 indexed and managed in a lightweight persistent key-value (KV) store with a persistent B+-tree storage backend. Note that, our focus is not to provide an optimal PM programming API. Instead, we focus on building an application model for the PM to accelerate memory centric scientific computing.

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.
- **Metadata 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.

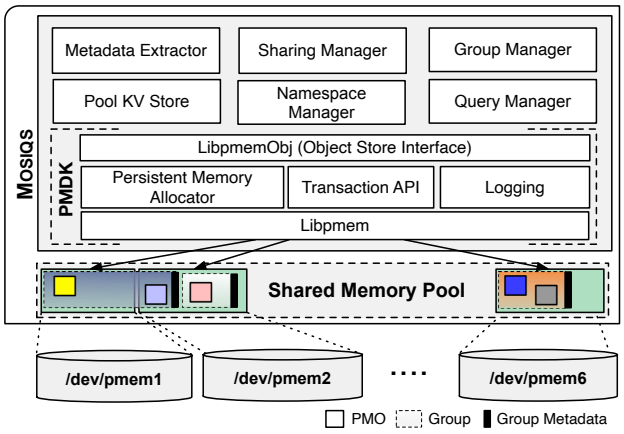


Fig. 4: An inner layout of MOSIQS architecture.

- **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.

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 [5], [30]. Figure 3 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 3 shows that a process running at compute node 2 accesses two PMOs from namespace 1 and 2. To enable memory-level object storage abstraction to applications, we employ Intel's PMDK provided `lipmemobj` library, an open source PM object storage interface [19]. 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.

The main motivation behind our work is to provide an application model for scientific applications to benefit with emerging persistent memory devices which are fast, persistent, byte-addressable, higher in capacity and cheaper than DRAM.

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 4 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 [19] 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 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 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 allow users to organize and share a collection of PMOs.
- *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.

Listing 1 shows an example application that creates a group and a PMO with attribute annotations.

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 [14]. Internally,

MOSIQS creates the shared PM pool via `libpmempool` API [19], where the device files, i.e., `/dev/pmem[1-6]` as shown in Figure 4, 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`.

```
/** create, initialize and annotate properties to group **/
struct group_info_t my_group;
group_id = create_group("group-name", parent_group|NULL);
my_group.set_groupid(group_id);
my_group.set_scope(SHARE|PRIVATE);
struct mosiqs_attribute_t group_attr[1];
group_attr[1].key = "file"; group_attr[1].value="sim-v0.1";
my_group.set_attr(group_attr, group_attr.size());
my_group.set_split_value(100);
group_init(my_group);

/** create and annotate PMOs **/
struct pmo_info_t my_pmo;
pmo_id = create_pmo("pmo-name", group_id|NULL);
my_pmo.set_objectid(pmo_id);
my_pmo.set_scope(SHARE|PRIVATE);
my_pmo.set_type(String|Int|Float|Struct);
my_pmo.set_pmovalue(fits); // struct FITS fits = {...};
my_pmo.annotate("file=foo.hdf5");
struct mosiqs_attribute_t pmo_attr[2];
pmo_attr[1].key = "timestamp"; pmo_attr[1].value="t4";
pmo_attr[2].key = "iteration"; pmo_attr[2].value="1401";
my_pmo.set_attr(pmo_attr, pmo_attr.size()));
pmo_persist(my_pmo);
```

Listing 1: An example of group and object creation with metadata attribute annotations.

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 5 shows an overview of extracted and stored metadata KV object of both types, PMO metadata and group

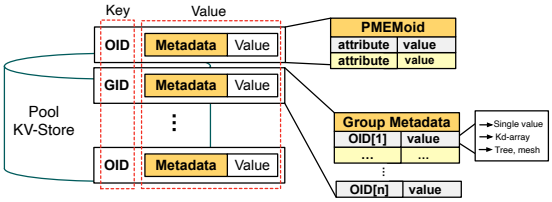


Fig. 5: The self-described metadata KV objects in pool KV store.

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 5 itself represents an additional self-described entity, i.e., motivated by scientific data formats [37], [38]. We further partition the value part into header part and data part, as shown in Figure 5. 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 5. 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 [19] 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

With the availability of large memory capacities, in- memory index structures have become an inevitable need. However, in-memory volatile structures or DRAM-resident indexes have an inherent limitation, i.e., they cannot survive power failures and unexpected crashes [39]. A simple power-failure makes the index unreachable and requires rebuilding or recovering the whole index. For instance, MIQS [24] is a state-of-the-art research, offering an effective in-memory metadata indexing and querying for scientific data formats such as HDF5 [37] and netCDF [38]. 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 for fast object retrieval. 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.

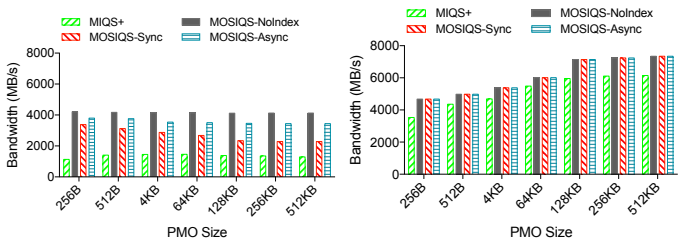
Therefore, we intend to employ a persistent index data structure for metadata search and querying. In current scope of the work, we provide search and query via a fully persistent B+-Tree, storage backend of PMEMKV [40]. However, it is not limited to B+-trees only and other persistent indexes can be integrated atop MOSIQS to further accelerate the query performance, e.g., NV-Tree [41], LSM-Trees [33] FP-Tree [39], and CCEH [42].

IV. Preliminary Evaluation

This section presents MOSIQS performance evaluation.

A. Experimental Setup

Testbed: We perform our experiments on a machine equipped with 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. PM is configured in 100% App direct mode, 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). 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 and read throughput of 28 cores measured via Intel's MLC tool [43] is 6.6 GB/s and 23 GB/s respectively.



(a) VPIC (Write IO) (b) BDCATS-IO (Read IO)

Fig. 6: MOSIQS bandwidth analysis via varying PMO size with 4 processes.

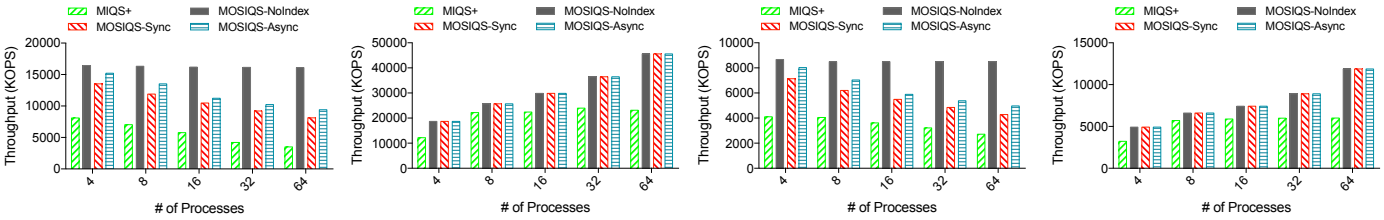
Benchmark and Workloads: We use two PIOK [23] benchmark provided kernels, i.e., VPIC-IO and BDCATS-IO to show the read and write performance. VPIC-IO is an extracted kernel that simulates the particle data write behavior by the real VPIC scientific application [23]. Similarly, BDCATS-IO demonstrates the data read patterns of a parallel program that analyze the particle data generated by VPIC [23]. We modified the two kernels using MOSIQS object storage abstraction API.

We compare our approach with the following systems:

- **MIQS+:** We implement and emulate MIQS [24] on top of ext4-DAX file system mounted PM and refer to it as MIQS+. MIQS [24] implements various DRAM-based indexes such as ART and SBST trees to maintain HDF5 file indexes for querying on scientific datasets, stored in parallel file systems. The metadata indexing is conducted after the data is written successfully.
- **MOSIQS-NoIndex:** MOSIQS with no metadata indexing and search service, but includes the software implementation overhead of MOSIQS on top of PMDK [19].
- **MOSIQS-Sync:** MOSIQS with metadata extraction enabled in inline synchronous mode, i.e., metadata populates in pool KV store and write operation finishes.
- **MOSIQS-Async:** MOSIQS with metadata extraction enabled in inline asynchronous mode, i.e., metadata populates in pool KV store after the write I/O. We use separate dedicated threads executing concurrently, one for processing application I/O and another for metadata extraction.

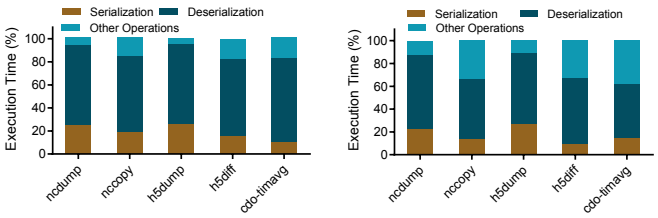
B. Bandwidth Analysis

Figure 6 (a) & (b) show the peak bandwidth of read and write operations with varied PMO sizes on 4 processes. The



(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)

Fig. 7: MOSIQS performance analysis by varying number of processes using 256B and 512KB PMO size.



(a) Ext4 on SSD (b) Ext4-DAX on PM

Fig. 8: Analysis of serialization and deserialization overhead.

reason to use 4 processes is to enable a realistic, moderate contention among processes. 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.

Figure 6 (a) presents the write bandwidth with varying number of processes. All MOSIQS variants outperform MIQS+ in varied PMO sizes. With small PMO size (i.e., $\leq$ 4KB), MOSIQS shows significant performance gain compared to MIQS+. For instance, with 512B PMO, MOSIQS-NoIndex achieves 100% higher bandwidth than MIQS+ respectively. Further, with varied PMO sizes, there is slight performance degradation in MOSIQS-Sync and Async till performance gets saturated at 512KB PMO. It is because all MOSIQS variants internally rely on PMDK provided `libpmemobj` for memory object allocation [19]. For large memory allocations (such as $\geq$1 MB), we observed a high thread contention inside the global heap of `libpmem` (PMDK's memory space management library). We believe that such overhead can be amortized by adopting pre-memory allocation techniques.

Figure 6 (b) presents the read bandwidth with varying number of processes. 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 device characteristics, i.e., the read and write performance of PM is highly asymmetric. Hence, shows a big throughput difference.

C. Throughput Analysis

Figure 7 (a) & (b) show 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 7 (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 scalable performance trend as shown in Figure 7 (b) and (d).

D. Serialization and Deserialization (S/D) Overhead Analysis

We perform a small set of experiments with several scientific utilities provided by middleware I/O libraries such as HDF5 [37] and netCDF [38] to validate the S/D time of MIQS+, as shown in Figure 8. We observe that, on average, the application's 70% of the execution time is spent on file system level cumulative serialization and deserialization operations. Few other studies have made such observations as well [4], [8]. Even with PM-aware file systems such as ext4-DAX and XFS-DAX, the serialization and deserialization overhead cannot be omitted, which drives the need for an object storage abstraction atop PM devices.

E. Metadata Search Query Performance

To analyze the query performance using realistic scientific HDF5 datasets, we download NASA's GLAS/ICESat L2 Sea

#	Query	MIQS+		MOSIQS	
		kQPS	EE	kQPS	EE
Q1	Locate PMO with name containing '9610/Inf'.	28184	1.25	25478	0.10
Q2	Find PMOs under group 'GLAH_634_2121'.	37766	8.25	30590	0.19
Q3	Count attributes annotated to group 'GLAH_634_2121'.	37756	12.25	29827	1.1

TABLE I: MOSIQS multi-attribute query throughput. EE shows an end-to-end query and data retrieval time in seconds.

Ice Altimetry real HDF5 dataset [44] and populated PMO metadata mappings in pool KV store. The dataset contains 4137 HDF5 files (Total size 101GB, Avg. File size 25MB, Avg. objects/File 2167, and Avg. attributes/Object 37). we define three realistic PMO metadata based queries, as shown in Table I. For this experiment, we compare MIQS+ with MOSIQS. Table I 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 object retrieval requires accessing disk storage system. We read varied number of PMOs against each query for MIQS+ and MOSIQS. On average, MOSIQS shows 27% query throughput degradation compared to MIQS+. However, for end-to-end time, MOSIQS outperforms the MIQS+ due to dataset storage location, i.e., PM pool vs parallel file system.

V. 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 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.

ACKNOWLEDGMENT

This work was supported by Institute for Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00503, researches on next generation memory-centric computing system architecture). This work was also supported by, and used the resources of, the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at ORNL, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725).

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 *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 103–112.

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

[3] 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 *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 288–303. [Online]. Available: https://doi.org/10.1145/3357526.3357541

[4] H. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, and S. Swanson, "Morpheus: Creating application objects efficiently for heterogeneous computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 53–65.

[5] Kimberly Keeton and Susan Spence, "Persistent Memory: a new tier or storage replacement," https://www.snia.org/sites/default/files/SDC/2017/presentations/General_Session/Keeton_Kimberly_Spence_Susan_Persistent_Memory_New_Tier_or_Storage_Replacement.pdf, 2017, (Accessed on 03/19/2020).

[6] K. Asanović, "Firebox: A hardware building block for 2020 warehouse-scale computers." Santa Clara, CA: USENIX Association, Feb. 2014.

[7] Intel, "Intel® Rack Scale Design Architecture," https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/rack-scale-design-architecture-white-paper.pdf, 2020, (Accessed on 04/10/2020).

[8] 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 *Proceedings of the 47th International Symposium on Computer Architecture*, ser. ISCA '20, 2020.

[9] Y. Xu, Y. Solihin, and X. Shen, "Merr: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 987–1000. [Online]. Available: https://doi.org/10.1145/3373376.3378492

[10] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 169–182. [Online]. Available: https://www.usenix.org/conference/fast20/presentation/yang

[11] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *ASPLOS '19*, 2019.

[12] 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 *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 494–508. [Online]. Available: https://doi.org/10.1145/3341301.3359631

[13] S. Kannan, A. Gavrilovska, and K. Schwan, "Pvm: Persistent virtual memory for efficient capacity scaling and object storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2901318.2901325

[14] 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 *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012, pp. 957–968.

[15] D. Bittman, P. Alvaro, P. Mehra, D. D. E. Long, and E. L. Miller, "Twizzler: a data-centric OS for non-volatile memory," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 65–80. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/bittman

[16] B. Nesterenko, X. Liu, Q. Yi, J. Zhao, and J. Zhang, "Transitioning scientific applications to using non-volatile memory for resilience," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 114–125. [Online]. Available: https://doi.org/10.1145/3357526.3357563

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

[18] J. Lofstead, I. Jimenez, C. Maltzahn, Q. Koziol, J. Bent, and E. Barton, "Daos and friends: A proposal for an exascale storage system," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2016, pp. 585–596.

[19] Persistent Memory Development Kit, https://pmem.io/pmdk/, 2020, (Accessed on 03/19/2020).

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

[21] T. C.-H. Hsu, H. Brügner, I. Roy, K. Keeton, and P. Eugster, "Nvthreads: Practical persistence for multi-threaded applications," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 468–482. [Online]. Available: https://doi.org/10.1145/3064176.3064204

[22] 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 *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, p. 105–118. [Online]. Available: https://doi.org/10.1145/1950365.1950380

[23] Suren Byna and Mark Howison, ""Parallel I/O Kernel (PIOK) Suite", ," https://sdm.lbl.gov/exahdf5/software.html, 2015, (Accessed on 03/16/2020).

[24] W. Zhang, S. Byna, H. Tang, B. Williams, and Y. Chen, "Miqs: Metadata indexing and querying service for self-describing file formats," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356146

[25] W. Zhang, S. Byna, C. Niu, and Y. Chen, "Exploring metadata search essentials for scientific data management," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, Dec 2019, pp. 83–92.

[26] A. Khan, T. Kim, H. Byun, and Y. Kim, "Scispace: A scientific collaboration workspace for geo-distributed hpc data centers," *Future Generation Computer Systems*, vol. 101, pp. 398 – 409, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X18326025

[27] J. Liu, D. Bard, Q. Koziol, S. Bailey, and Prabhat, "Searching for millions of objects in the boss spectroscopic survey data with h5boss," in *2017 New York Scientific Data Summit (NYSDS)*, Aug 2017, pp. 1–9.

[28] S. Byna, J. Chou, O. Rubel, Prabhat, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K. Hsu, K. Lin, A. Shoshani, A. Uselton, and K. Wu, "Parallel i/o, analysis, and visualization of a trillion particle simulation," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov 2012, pp. 1–12.

[29] 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 *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 43–48. [Online]. Available: https://doi.org/10.1145/3366623.3368140

[30] 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 *MCHPC '19*, 2019.

[31] I. Choi, J. Nelson, L. Peterson, and J. Hartman, "Sdm: A scientific dataset delivery platform," in *2019 15th International Conference on eScience (eScience)*, 2019, pp. 378–387.

[32] X. Li, K. Lu, X. Wang, and X. Zhou, "Nv-process: A fault-tolerance process model based on non-volatile memory," in *Proceedings of the Asia-Pacific Workshop on Systems*, ser. APSYS '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2349896.2349897

[33] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y. ri Choi, "Slm-db: Single-level key-value store with persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 191–205. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[34] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 349–362. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[35] M.Wilcox, "Add support for nvdimms to ext4," https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9028138&tag=1, 2014, (Accessed on 03/26/2020).

[36] 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 Endow.*, vol. 8, no. 12, p. 1916–1919, Aug. 2015. [Online]. Available: https://doi.org/10.14778/2824032.2824100

[37] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson, "An overview of the hdf5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, ser. AD '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 36–47. [Online]. Available: https://doi.org/10.1145/1966895.1966900

[38] N. N. C. D. Form, https://www.unidata.ucar.edu/software/netcdf/, 2020, (Accessed on 03/20/2020).

[39] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proc. VLDB Endow.*, vol. 13, no. 4, p. 574–587, Dec. 2019. [Online]. Available: https://doi.org/10.14778/3372716.3372728

[40] PMEMKV: Key/Value Datastore for Persistent Memory, https://github.com/pmem/pmemkv, 2019, (Accessed on 03/19/2020).

[41] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 167–181. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/yang

[42] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 31–44. [Online]. Available: https://www.usenix.org/conference/fast19/presentation/nam

[43] Intel, "Intel Memory Latency Checker v3.8," https://software.intel.com/en-us/articles/intelr-memory-latency-checker, 2020.

[44] H. J. Zwally, R. Schutz, D. Hancock, and J. Dimarzio, "Glas/icesat l2 sea ice altimetry data (hdf5), version 34. [indicate subset used]," 2014, (Accessed on 02/27/2020).