SAND2021-8135C

pMEMCPY: a simple, lightweight, and portable I/O library for storing data in persistent memory

Luke Logan, Jay Lofstead, Scott Levy, Patrick Widener, Xian-He Sun, Anthony Kougkas
Illinois Institute of Technology and Sandia National Labs
United States
llogan@hawk.iit.edu,{glofst,sllevy,pwidene}@sandia.gov,{sun,akougkas}@iit.edu

ABSTRACT

Persistent memory (PMEM) devices can achieve comparable performance to DRAM while providing significantly more capacity. This has made the technology compelling as an expansion to main memory. Rethinking PMEM as storage devices can offer a high performance buffering layer for HPC applications to temporarily, but safely store data. However, modern parallel I/O libraries, such as HDF5 and pNetCDF, are complicated and introduce significant software and metadata overheads when persisting data to these storage devices, wasting much of their potential. In this work, we explore the potential of PMEM as storage through pMEMCPY: a simple, lightweight, and portable I/O library for storing data in persistent memory. We demonstrate that our approach is up to 2x faster than other popular parallel I/O libraries under real workloads.

1 INTRODUCTION

Scientific applications generate massive amounts of data; however, storage performance lags behind CPU performance resulting in applications being bottlenecked by I/O both with other nodes as well as with storage. One approach to alleviate this problem is to expand the memory capacity of the nodes, enabling more local processing before requiring communication. PMEM (e.g., phase change memory and Intel DC Persistent Memory) offers an excellent solution that is cheaper than DRAM, but offers reasonably similar performance characteristics. This technology has driven considerable work into using DRAM as a working cache for an expanded PMEM main memory [30]. As compelling as this case is, it only addresses the inter-node portion of the bottleneck. For applications where communication with storage is a more serious concern, using that same PMEM technology as fast storage (instead of slower memory) offers a flexible resource that can address multiple kinds of workloads. For example, various works investigate the use of storage hierarchies in order to combat the I/O bottleneck [5, 21, 34]. In these works, storage such as PMEM, NVMe, SSD, and HDD are arranged in a hierarchy based on performance and capacity characteristics. Data is initially buffered in faster storage tiers and then asynchronously flushed to slower mass storage, which helps avoid costly data stalls. While there has been considerable work examining the use of node hosted storage technology with more favorable performance characteristics than hard drives, the interfaces for PMEM offer another potential performance gain, but only if the software uses the devices with these more efficient interfaces.

Due to the DRAM-like performance of PMEM, software overheads are no longer negligible on the I/O path. For this reason, researchers have started rethinking the design of node-local storage stacks [1, 19, 23, 40, 41], which had previously been designed for slow storage technologies, such as hard drives. EXT4/XFS DAX [1] allows applications to directly store data in PMEM without first copying to DRAM using memory mapped I/O. SplitFS [19] aims to improve the performance of DAX by splitting metadata and storage operations between kernel space and user space respectively, allowing the majority of I/O operations to avoid the kernel entirely. NOVA [40] is a log-structured filesystem that aims to exploit the parallelism and random access properties of PMEM by storing logs per-inode as opposed to a global log. These works avoid many of the overheads introduced by the Linux kernel, such as context switching, splitting/merging, lock contention, and request reordering. However, improving node-local storage stacks is not enough. HPC applications typically use parallel I/O (PIO) libraries on top of node-local storage stacks to persist data. Fundamental changes in the design of PIO libraries must be made to gain the full benefits of PMEM for I/O.

Various PIO libraries exist, such as ADIOS [13, 29], HDF5 [22], and pNetCDF [24]. However, these libraries introduce significant programming burden, software overhead, and complex configuration spaces. In order to maximize the performance of these libraries and reduce the user's burden, researchers have investigated the use of auto-tuning to identify optimal parameters specific to the characteristics of applications and parallel filesystems [3, 4, 6, 7], with approaches such as genetic algorithms and Bayesian optimization. However, at a fundamental level, existing PIO libraries do not interact with PMEM efficiently, regardless of how well they are tuned. For example, all existing work depends on the use of MPI-IO and POSIX, which causes unnecessary networking communication and data copies that degrade the performance of I/O to PMEM [20]. Furthermore, PIO libraries tend to have complicated APIs, requiring many lines of code to store simple data structures, such as arrays. A simple *memcpy* interface is more desirable. PIO libraries should be designed with awareness of the underlying device characteristics in mind in addition to being more user-friendly.

In this work, we present pMEMCPY: a simple, lightweight, and portable I/O library for storing data in persistent memory. Using the Persistent Memory Development Kit (PMDK) [32], applications have direct access to PMEM while maintaining consistency guarantees. Users can store data structures with a simple key-value store interface that adds the minimal metadata necessary to deserialize the data structures in addition to avoiding costly network communications and data copies that other PIO libraries introduce.

Our contribution offers an optimized approach for parallel I/O library design that can store application data structures in node-local PMEM directly with minimal overhead using a simple key-value store interface similar to *memcpy*. Through this style of I/O library, users can achieve the best possible PMEM performance for their storage operations and enjoy an API much closer to memcpy.

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

The rest of this paper is organized as follows. First in Section 2 is a deeper discussion of background and related work. Next in Section 3 we detail the reasoning and design decisions for our demonstration. Section 4 presents a collection of evaluations comparing this approach against alternatives. Finally, in Section 5 we summarize the work.

2 BACKGROUND & RELATED WORK

There are various existing parallel I/O libraries, including HDF5, ADIOS, and pNetCDF. Furthermore, there are various libraries and APIs that exist to efficiently interact with PMEM. However, there has been no published approach, to our knowledge, that demonstrates how to optimize the I/O library for PMEM interfaces and simplify the API to a most basic memcpy-like approach.

2.1 Parallel I/O (PIO) Libraries

HDF5 [22] is a popular PIO library, and is used as the foundations for other popular PIO libraries, such as NetCDF4 [31]. HDF5 exposes a hierarchical namespace to users, where H5Groups are analogous to directories. HDF5 can store primitive types (ints, floats, doubles, etc.), compound data types (structures), and arrays (H5Datasets) of those types. Subsets of datasets can be taken using the Hyperslab APIs. HDF5 can store datasets using various data layout policies: contiguous, chunked, and compact. The contiguous layout stores arrays as a 1-D sequence of data, and is the default layout for HDF5. The chunked mode divides the array into fixed-size sub-arrays (i.e., chunks) where the dimensions of the sub-arrays are user-defined. In chunked mode, HDF5 also allows for the definition of filters, which are operations to perform on individual chunks, such as compression [10, 11]. Lastly, if the dataset is less than 64KB, the compact mode stores the dataset in its corresponding metadata entry. In order to persist data to storage, HDF5 allows multiple approaches: MPI Independent I/O, MPI Collective I/O, and POSIX I/O. In each of these cases, the final output of HDF5 is a single binary file. Furthermore, HDF5 introduced a multi-tiered buffer management system, Hermes [21], that allows users to manage the complexity of heterogeneous, multi-tiered storage environments without changing application code. While HDF5 is a feature-rich library that has specific functionality for buffering and prefetching, it has many limitations. The Neuroscience community, for example, has noted multiple flaws in the user-friendliness of this library [12]. HDF5 stores data in a single binary file, where metadata is not human readable. This also makes version control systems less efficient. Furthermore, HDF5 compound types do not support the nesting of compound types or dynamically sized arrays. Furthermore, MPI-IO relies on the underlying filesystem (for Linux, read/write) APIs in order to store data. However, read/write perform data copies which introduces unnecessary overhead [20].

An alternative to HDF5 and NetCDF4 is the pNetCDF [24] library. This was developed around the same time as NetCDF4 as an effective demonstration on how to maintain the NetCDF3 compatibility as much as possible while extending for 64-bit support. While the two libraries "compete", the reality is that they co-exist peacefully and are widely supported as a pair rather than individually. For exmaple, the NCAR PIO library [9] offers a single API that can switch to use either NetCDF4 or pNetCDF underneath. As with HDF5, pNetCDF is designed with MPI-IO as the primary IO interface for parallel IO and optimized for slow storage devices through additional work to prepare data to more efficiently be moved into storage. However, the performance gains of PMEM shifts the bottleneck of the storage device that afforded such optimizations without noticeably hurting performance to the I/O library itself. NVMe devices have had a similar effect [2], but PMEM offers additional performance exasperating the performance overhead the software layer imposes.

ADIOS is an alternative PIO library to HDF5, NetCDF and pNetCDF. ADIOS aims to encompass various I/O transport mechanisms (e.g., MPI-IO, POSIX, HDF5, and netCDF) under a simplified interface that is easily configurable and requires little change to application code to change which implementation is used. The original design of ADIOS was based on trying to reduce the code complexity of HDF5 and acknowledge that some of the performance optimizations employed by HDF5 and other libraries that use similar PIO techniques ultimately do not scale for writing or reading as well as hoped [28]. One approach ADIOS uses to address the performance gap is to use its own BP format whenever possible. BP offers delayed consistency, lightweight data characterization, and data resilience. Unlike HDF5, ADIOS stores data in the same format as it was produced on a process-by-process basis rather than constructing a global linearization of complex datastructures. For example, a 3D domain decomposition is stored as a single item in HDF5 with all three dimensions across all processes being linearized through a data rearrangement phase prior to hitting storage. This has the advantage of eliminating any potential artifacts from unusual process decompositions. ADIOS has each process write the data it owns with no coordination with other processes. This eliminates the data rearrangement phase, which can improving performance greatly. In particular, large 3D domain decompositions see radical performance improvements for both writing and reading [28]. ADIOS also supports transparent and custom operators, similar to HDF5. Recently, ADIOS2 [13] was released, which provides a C++ interface that is more simplistic and extensible than that of the original ADIOS. Their recent revision includes a key-value store API for storing data. However, ADIOS2 suffers from the same drawbacks as the original when it comes to PMEM as it is storage device agnostic.

A more recent effort, Proactive Data Containers [33] offers a similar key-value store approach for data management. However, it is also designed to assume storage devices, such as SSDs with an assumption that non-volatile memory devices will have compatible interfaces to get the full performance benefits.

One system recognizing the need for a different interface to non-volatile memory is DStore [14]. However, DStore is intended as a way to store a log for an in-DRAM key-value store. Unlike other attempts to optimize key-value stores with PMEM, such as MongoDB-PMEM [17] and PMEM-RocksDB [39], DStore uses PMEM to store the logs rather than as the main store offering greater performance while still offering predictable consistency.

In all cases, effectively using PMEM using efficient interfaces is a relatively new endeavour that popular HPC I/O libraries have yet to embrace. While some progress has been made in the scale-out space, the recent DStore paper demonstrates that a simple "switch to the PMDK interface" may not be the most efficient nor optimal approach for achieving both performance and price/performance.

2.2 Accessing PMEM

PMEM can be exposed like any other storage device. Application developers can use traditional filesystem APIs such as POSIX, stdlib, and iostream in order to store data in PMEM. However, these interfaces introduce significant software overheads. For example, these interfaces will cause unnecessary data copies and memory allocations to occur. To avoid this, applications can access PMEM directly using memory mapped I/O (MMIO) and DAX. However, managing memory-mapped regions requires application developers to provide their own memory allocation functions and concurrency control mechanisms, which can cause data consistency and reliability concerns.

The Persistent Memory Development Kit (PMDK) [32] is a collection of libraries and tools for managing PMEM devices. It provides low-level primitives for interacting with PMEM and a transactional object store that utilizes memory mapping in order to interface with PMEM devices. What this really means is that PMEM is mapped directly in the memory space for a process enabling direct access. Unlike MPI-IO and POSIX I/O, this approach allows applications direct, zero-copy access to PMEM while providing consistency guarantees. PMDK provides optimized memory allocation functions, persistent locks, basic data structures (e.g., thread-safe lists), and transactions. This allows applications to have efficient and safe access to PMEM while reducing the complexity of managing memory-mapped files.

2.3 New Filesystems

In the Introduction, we covered many of the newer generation storage systems written from the ground up to take advantage of solid state, node local storage. However, these have all been written for NVMe devices, at best, and still assume a more traditional device interface. One major exception to this is DAOS [16]. The original design of DAOS [27] was to offer a new storage architecture, but still assuming non-PMEM storage devices. The current DAOS generations have been reimagined using Intel Optane PMEM devices as a core component. Using these devices, DAOS was able to achieve top marks on the IO500 list at sc19 [18]. More recent conversations with the DAOS team about Optane and DAOS or other storage use recommended at most 1% of the capacity using the PMEM devices as a way to ensure top performance for the most critical operations while keeping costs from spiraling out of control [26]. This makes DAOS a good potential candidate for using PMEM as a storage device, but it does not address the I/O library layer entirely. The plug-ins for HDF5 for speaking directly with DAOS and the DAOS native APIs may offer better support. However, the interfaces are still complex and focused on a container-like structure with POSIX-structures layered on top.

3 DESIGN & IMPLEMENTATION

This work offers pMEMCPY, a simplistic and portable I/O library for managing node-local PMEM. Our design assumes that the compute nodes running the application also contain PMEM. Data structures in memory are stored directly on PMEM without extra metadata, context switching, or data copies beyond what is necessary to reload the data during a different application run or for an analysis job. Our assumed, basic machine architecture is illustrated in Figure 1.

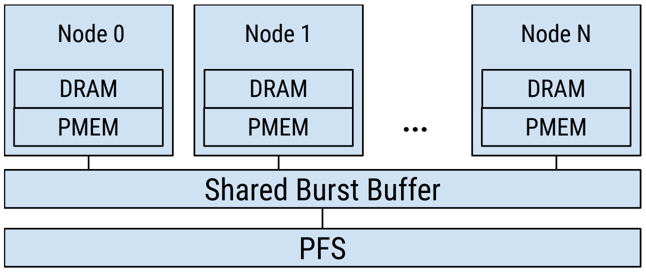

Figure 1: Basic Machine Architecture

```
#include <pmemcpy/pmemcpy.hpp>
pmemcpy::PMEM pmem;
pmem.mmap(std::string filename, int comm);
pmem.munmap();

pmem.store<T>(std::string id, T &data);
pmem.alloc<T>(std::string id,
  int ndims, size_t *dims);
pmem.alloc<T>(std::string id,
  pmemcpy::Dimensions dims);
pmem.store<T>(std::string id, T *data,
  int ndims, size_t *offsets, size_t *dimspp);

pmem.load<T>(std::string id);
pmem.load<T>(std::string id, T &num);
pmem.load<T>(std::string id, T *data,
  int ndims, size_t *offsets, size_t *dimspp);
pmem.load_dims(std::string id,
  int *ndims, size_t *dim);
```

Figure 2: pMEMCPY API

```
#include <pmemcpy/pmemcpy.h>
int main(int argc, char** argv) {
    int rank, nprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    pmemcpy::PMEM pmem;
    size_t count = 100;
    size_t off = 100*rank;
    size_t dimsf = 100*nprocs;
    char *path = argv[1];

    double data[100] = {0};
    pmem.mmap(path, MPI_COM_WORLD);
    pmem.alloc<double>("A", 1, &dimsf);
    pmem.store<double>("A", data, 1, &off, &count);
    MPI_Finalize();
}
```

Figure 3: pMEMCPY API Usage Example

API: pMEMCPY exposes a key-value interface for storing and loading data from PMEM. Users can store primitive types, structured types, and arrays of these types using the templated load/store APIs. The C++ API is shown in Figure 2. In Figure 3, we demonstrate the usage of pMEMCPY for writing a 1-D array of data in parallel. In the example, each process writes 100 doubles to non-overlapping offsets in the array directly to PMEM. *alloc* is used to specify the final dimensions of the array, and *store* is used to

```c
#include <hdf5.h>
int main (int argc, char **argv) {
  int nprocs, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  hid_t file_id, dset_id;
  hid_t filespace, memspace;
  hsize_t       count = 100;
  hsize_t offset = rank*100;
  hsize_t dimsf = nprocs*100;
  hid_t plist_id;
  herr_t        status;
  char *path = argv[1];
  int data[100];

  plist_id = H5Pcreate(H5P_FILE_ACCESS);
  H5Pset_fapl_mpio(plist_id,
    MPI_COMM_WORLD, MPI_INFO_NULL);
  file_id = H5Fcreate(path,
    H5F_ACC_TRUNC, H5P_DEFAULT, plist_id);
  H5Pclose(plist_id);

  filespace = H5Screate_simple(1, &dimsf, NULL);
  dset_id = H5Dcreate(file_id, "dataset",
    H5T_NATIVE_INT, filespace, H5P_DEFAULT,
    H5P_DEFAULT, H5P_DEFAULT);
  H5Sclose(filespace);
  memspace = H5Screate_simple(1, &count, NULL);
  filespace = H5Dget_space(dset_id);
  H5Sselect_hyperslab(filespace,
    H5S_SELECT_SET, &offset,
    NULL, &count, NULL);

  plist_id = H5Pcreate(H5P_DATASET_XFER);
  status = H5Dwrite(dset_id, H5T_NATIVE_INT,
    memspace, filespace, plist_id, data);

  H5Dclose(dset_id);
  H5Sclose(filespace);
  H5Sclose(memspace);
  H5Pclose(plist_id);
  H5Fclose(file_id);
  MPI_Finalize();
  return 0;
}
```

Figure 4: Equivalent HDF5 Example

persist pieces of the array generated by each process. In Figure 4, we show the equivalent HDF5 code. HDF5 requires a user to create and free dataspace and dataset objects in addition to subsetting the dataset, and each of these interfaces contain many parameters. The dataspace defines the dimensions of the array, and the dataset represents the array within HDF5. The HDF5 version is 42 lines of code and 253 tokens, whereas our code is 16 lines and 132 tokens, which is a 92% reduction in the number of tokens. Similar to HDF5, NetCDF and pNetCDF requires users to define and allocate the dimensions of the array using special APIs, which adds unnecessary complexity. While ADIOS simplifies this, it still requires the user to store the dimensions of the array separately and then associate those variables with the array. pMEMCPY automatically stores the dimensions of the array and the per-process subarrays in the *store* API by appending "#dims" to the id; dimensions can be queried using *load_dims*. In Figure 5, we show the equivalent ADIOS code,

```c
#include <adios.h>
int main(int argc, char **argv) {
    int rank, nprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    char *path = argv[1];
    char *config = argv[2];
    double data[100];
    int64_t adios_handle;
    size_t count = 100;
    size_t offset = 100*rank;
    size_t dimsf = 100*nprocs;

    adios_init(config, MPI_COMM_WORLD);
    adios_open (&adios_handle, "dataset",
      path, "w", MPI_COMM_WORLD);
    adios_write (adios_handle, "count", &count);
    adios_write (adios_handle, "dimsf", &dimsf);
    adios_write (adios_handle, "offset", &offset);
    adios_write (adios_handle, "A", data);
    adios_close (adios_handle);
    adios_finalize (rank);
    MPI_Finalize ();
    return 0;
}
```

Figure 5: Equivalent ADIOS Example. Note, there is a separate ADIOS config file that defines "A" in terms of count, off, and dimsf.

which is 24 lines and 164 tokens. Overall, we see that pMEMCPY provides a more simplified and compact API than other libraries.

Data Transfer and Serialization: Unlike ADIOS, NetCDF, and pNetCDF which depend on POSIX and MPI-IO, pMEMCPY uses memory mapping and independent I/O to store data in the node-local PMEM, which avoids unnecessary data copies, network/inter-process communications, and kernel interventions. When storing a data structure in PMEM, pMEMCPY serializes the data using well-known, portable serialization libraries, such as BP4 [13], CapnProto [36], and cereal [38]. By default, the BP4 serialization (same as ADIOS) is used; however, other serialization tools can be added, and serialization can be completely disabled. Unlike similar work which serializes data structures into an in-memory buffer and then copies to PMEM, pMEMCPY can serialize the data directly into PMEM without first placing it in DRAM, avoiding a significant data copying cost. Furthermore, we allow users to configure whether or not the MAP_SYNC flag is enabled when storing serialized data structures in a region of PMEM. The MAP_SYNC flag guarantees that, after a crash, a block that has been mapped into memory with write permissions will still be at the same offset within the file [8]. While this improves crash consistency, this can introduce significant latency penalties that severely degrade performance, as shown in our evaluations. After serialization, a burst buffer, such as DataWarp [15], will then be triggered to asynchronously flush the buffered data to mass storage. The data will be stored in the same format as it was produced, similar to ADIOS, which avoids the network and inter-process communication required to restructure the data.

Data Layout: By default, pMEMCPY stores all application data in a single file similar to ADIOS, NetCDF, and pNetCDF. However, pMEMCPY uses the PMDK [32] to manage PMEM, which provides direct access to PMEM in addition to data consistency guarantees,

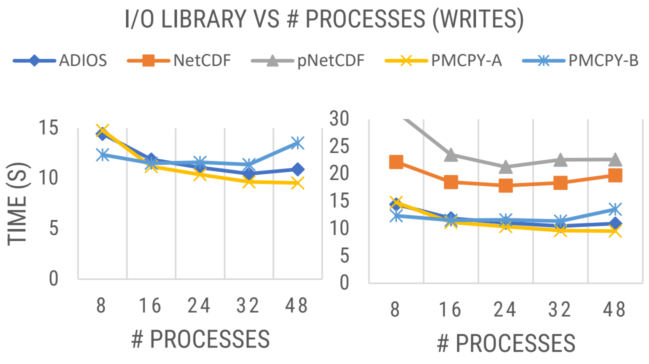

Figure 6: Performance of writing a 40GB 3-D domain to PMEM for a varying number of processes. PMCPY-A has MAP_SYNC disabled, whereas PMCPY-B has it enabled. Each process writes an equal amount of data. pMEMCPY is 2.5x faster than pNetCDF and NetCDF by avoiding network communications and data copying costs. At 24 cores, pMEMCPY is faster than ADIOS by 15% when MAP_SYNC is disabled, and slightly slower when MAP_SYNC is enabled.

concurrency control, and memory allocation policies. Metadata is stored in a flat namespace using a hashtable with chaining. This utilizes the high parallelism and random access characteristics of PMEM. Alternatively, unlike ADIOS, NetCDF, and pNetCDF, pMEMCPY can layout data hierarchically using the PMEM's filesystem. In this approach, instead of writing to a single file, pMEMCPY stores the data structures in a directory and creates a file for each variable. Whenever a "/" is used in the id of the variable, a directory is created if it didn't already exist.

4 EVALUATIONS

Testbed: All tests were conducted in Chameleon Cloud using a Compute Skylake node. Compute Skylake nodes come with 192GB of RAM and 2x Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz, for a total of 24 cores/48 threads. The OS used was Ubuntu 20.04 with kernel 5.4.0-70-generic. We used openmpi 3.1.6.

Emulating PMEM: Since we do not have access to PMEM, we emulate it using the approach presented in the Strata PMEM filesystem paper [23]. We utilize Linux's PMEM emulator to treat 80GB of DRAM as PMEM and format the resulting PMEM device using EXT4 with DAX enabled. We assume that PMEM has a read latency of 300ns, write latency of 125ns, read bandwidth of 30GB/s, and write bandwidth of 8GB/s [35]. We benchmarked DRAM bandwidth and latency using Intel's MLC [37] and use nanosecond-accurate monotonic timers to add the additional latency and bandwidth constraints.

4.1 Real-App Evaluation

In this test, we demonstrate the performance impact of pMEMCPY over other popular PIO libraries using real workloads. In this evaluation, we use two workloads that were obtained through the help of scientists [28]. The first workload is a write-only 3-D domain decomposition problem where each process writes a rectangular region of data to storage. The second workload is a read-only workload that reads the regions from storage. For both tests, we use

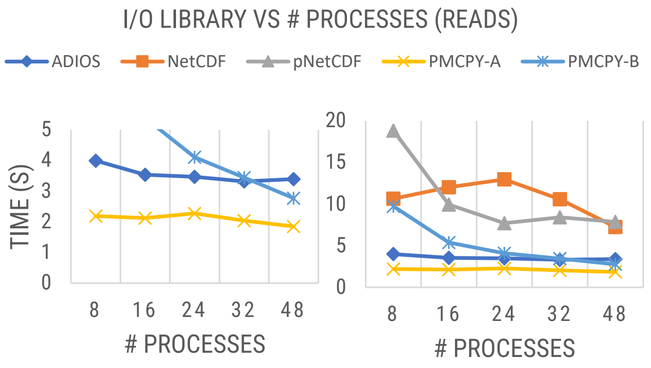

Figure 7: Performance of reading a 40GB 3-D domain from PMEM for a varying number of processes. PMCPY-A has MAP_SYNC disabled, whereas PMCPY-B has it enabled. Each process reads an equal amount of data. pMEMCPY is 5x faster than pNetCDF and NetCDF by avoiding network communications and data copying costs. pMEMCPY is 2x faster than ADIOS when MAP_SYNC is disabled. When enabled, pMEMCPY performs no better than ADIOS.

between 8 and 48 processes. This model represents a large memory regular stencil code common in compute models today. One example is the S3D combustion code [25] that was the inspiration for this configuration. This model has been previously used [28] to demonstrate potential I/O performance. In the write-only case, we generate 10 3-D rectangles. For each test, a total of 40GB of data is generated and the 40GB is divided equally among the processes. Each element in the rectangle is a double precision floating point value (8 bytes). The read workload is completely symmetrical to the write workload, where each process reads the same data that had been written. We measure the wall-clock time from the point at which the file is opened/mmapped to when the it is closed. We perform the I/O using ADIOS, NetCDF-4, pNetCDF, and pMEMCPY and compare the runtime between the different approaches. For NetCDF-4, we make sure to call *nc_def_var_fill()* with NC_NOFILL in order to prevent it from initializing variables with a default value, which causes significant overhead for write workloads. For pMEMCPY, we use BP4 serialization with the PMDK hashtable layout. We run each experiment 3 times and take the average of the runs.

The results of the experiment are shown in Figures 6 and 7. From these figures, we see the effects of concurrency due to the CPU and PMEM wear off after 24 cores in the write case and for most of the reads, with the exception of PMCPY-B and NetCDF4. This makes sense considering the node has 24 physical CPU cores in total. For NetCDF, the performance differences were largely due to differences in the dimensions of the cube being read for the different process counts. For PMCPY-B, this was because the metadata updates were parallelized, which caused fewer stalls. Overall, we see that pMEMCPY outperforms ADIOS, NetCDF, and pNetCDF in both workloads when MAP_SYNC is disabled. This is because pMEMCPY avoids unnecessary communications and data copies that other PIO libraries introduce. In the case of writes, all other PIO libraries first generate the cube in DRAM, serialize the cube into another DRAM buffer, and then copy the serialized cube to the PMEM whereas pMEMCPY generates the cube in DRAM and then serializes the cube directly into the PMEM, avoiding an entire copy of the cube. From these figures, we see ADIOS performs far better than NetCDF and pNetCDF in both read and write performance. This is because, similar to pMEMCPY, ADIOS stores data in the same format as it was produced, which avoids costly network communications and data copies during the write phase. Furthermore, since the workload is symmetrical, ADIOS does not need to realign any data, which mitigates data shuffling costs in the read phase. However, pNetCDF and NetCDF store data contiguously, which requires data to be shuffled during both reads and writes, incurring significant overhead. While ADIOS performs much better than pNetCDF and NetCDF, it still introduces data copying overheads that pMEMCPY avoids, causing its performance to be suboptimal. For example, in the case of reads, ADIOS requires the serialized data to be copied from PMEM into DRAM and then deserialized into another DRAM buffer. pMEMCPY deserializes the data directly from PMEM, avoiding the initial copy from PMEM to DRAM. Within pMEMCPY, we see that the choice of flags has a significant impact on performance. When MAP_SYNC is enabled, the performance benefit of serializing/deserializing directly from PMEM is completely lost, and can even cause performance to be worse than simply using POSIX read()/write(). Overall, we see that pMEMCPY can perform at least 15% better for writes and 2x better for reads depending on the level of security the user requires.

4.2 Discussion

While standard I/O libraries offer a familiar interface, that can come at a cost. ADIOS, with the design break from the previous generation demonstrates better performance, but is still not optimal by a margin of 15% - 100%. Only by using an approach such as the one we demonstrate in pMEMCPY can the full potential of PMEM as a storage device be achieved.

5 CONCLUSIONS

Persistent memory (PMEM) is an extraordinarily fast persistent storage device typically thought of as an extension of DRAM main memory. However, using PMEM for storage requires revisiting the design of parallel I/O (PIO) libraries. With PMEM being integrated into compute nodes, PIO libraries should take full advantage of the characteristics of these devices. However, popular libraries, such as HDF5, ADIOS, and pNetCDF, introduce significant overheads when applications store and load data. Furthermore, they introduce complex interfaces and parameters that add unnecessary burden on programmers. In this paper, we introduced pMEMCPY: a simple, lightweight, and portable I/O library for storing data in persistent memory. We compared our design with ADIOS, NetCDF-4, and pNetCDF, and found that write speeds improved at least 15% and reads improved up to 2x.

ACKNOWLEDGEMENTS

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] 2014. Direct Access for files. https://www.kernel.org/doc/Documentation/filesystems/dax.txt
[2] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. 2019. File systems unfit as distributed storage backends: lessons from 10 years of Ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 353–369.
[3] Babak Behzad, Surendra Byna, and Marc Snir. 2019. Optimizing I/O performance of HPC applications with autotuning. *ACM Transactions on Parallel Computing (TOPC)* 5, 4 (2019), 1–27.
[4] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Ruth Aydt, Quincey Koziol, Marc Snir, et al. 2013. Taming parallel I/O complexity with auto-tuning. In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.
[5] Suren Byna, Quincey Koziol, Venkat Vishwanath, Jerome Soumagne, Houjun Tang, Jingqing Mu, Bin Dong, Richard A Warren, François Tessier, Teng Wang, et al. 2018. Proactive Data Containers (PDC): An Object-centric Data Store for Large-scale Computing Systems. In *AGU Fall Meeting Abstracts*, Vol. 2018. IN34B–09.
[6] Zhen Cao. 2019. *A Practical, Real-Time Auto-Tuning Framework for Storage Systems*. Ph.D. Dissertation. State University of New York at Stony Brook.
[7] Zhen Cao, Vasily Tarasov, Sachin Tiwari, and Erez Zadok. 2018. Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 893–907.
[8] Jonathan Corbet. 2017. Two more approaches to persistent-memory writes.
[9] John M Dennis, Jim Edwards, Ray Loy, Robert Jacob, Arthur A Mirin, Anthony P Craig, and Mariana Vertenstein. 2012. An application-level parallel I/O library for Earth system models. *The International Journal of High Performance Computing Applications* 26, 1 (2012), 43–53.
[10] Hariharan Devarajan, Anthony Kougkas, Luke Logan, and Xian-He Sun. 2020. HCompress: Hierarchical Data Compression for Multi-Tiered Storage Environments. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 557–566.
[11] Hariharan Devarajan, Anthony Kougkas, and Xian-He Sun. 2019. An intelligent, adaptive, and flexible data compression framework. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 82–91.
[12] Svenn-Arne Dragly, Milad Hobbi Mobarhan, Mikkel E Lepperød, Simen Tennøe, Marianne Fyhn, Torkel Hafting, and Anders Malthe-Sørenssen. 2018. Experimental Directory Structure (Exdir): An alternative to HDF5 without introducing a new file format. *Frontiers in neuroinformatics* 12 (2018), 16.
[13] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. 2020. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX* 12 (2020), 100561.
[14] Shashank Gugnani and Xiaoyi Lu. 2020. DStore: A Fast, Tailless, and Quiescent-Free Object Store for PMEM. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing* (Virtual Event, Sweden) *(HPDC '21)*. Association for Computing Machinery, New York, NY, USA, 31–43. https://doi.org/10.1145/3431379.3460649
[15] Dave Henseler, Benjamin Landsteiner, Doug Petesch, Cornell Wright, and Nicholas J Wright. 2016. Architecture and design of cray datawarp. *Cray User Group CUG* (2016).
[16] Intel. [n.d.]. DAOS: Revolutionizing High-Performance Storage with Intel Optane Technology. https://www.intel.com/content/dam/www/public/us/en/documents/solution-briefs/high-performance-storage-brief.pdf
[17] Intel. 2021. MongoDB Persistent Memory Storage Engine. Github. https://github.com/pmem/pmse
[18] io500. [n.d.]. 10 Node Challenge, I0500-SC19. https://www.vi4io.org/io500/list/19-11/10node?fields=information__system,information__institution,information__storage_vendor,information__filesystem_type,information__client_nodes,information__client_total_procs,io500__score,io500__bw,io500__md,information__data,information__list_id&equation=&sort_asc=false&sort_by=io500__score&radarmax=6&query=
[19] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 494–508.
[20] Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. 2020. SubZero: zero-copy IO for persistent main memory file systems. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. 1–8.
[21] Anthony Kougkas, Hariharan Devarajan, and Xian-He Sun. 2018. Hermes: a heterogeneous-aware multi-tiered distributed I/O buffering system. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. 219–230.

[22] Quincey Koziol, Dana Robinson, et al. 2018. *HDF5*. Technical Report. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States).

[23] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 460–477.

[24] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A high-performance scientific I/O interface. In *SC'03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. IEEE, 39–39.

[25] David Lignell, C Yoo, Jacqueline Chen, Ramanan Sankaran, and M Fahey. 2007. S3D: Petascale combustion science, performance, and optimization. In *Proceedings of the Cray Scaling Workshop, Oak Ridge National Laboratory, TN*.

[26] Jay Lofstead. 2021. PMEM for Storage Conversation in IO500 general Slack channel. Slack. https://io500workspace.slack.com/archives/C01BMTNT56K/p1612291357026500

[27] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. 2016. 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*. IEEE, 585–596.

[28] Jay Lofstead, Milo Polte, Garth Gibson, Scott Klasky, Karsten Schwan, Ron Oldfield, Matthew Wolf, and Qing Liu. 2011. Six degrees of scientific data: Reading patterns for extreme scale science io. In *Proceedings of the 20th international symposium on High performance distributed computing*. 49–60.

[29] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. 2008. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. 15–24.

[30] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. 2021. Optimizing large-scale plasma simulations on persistent memory-based heterogeneous memory with effective data placement across memory hierarchy. In *Proceedings of the ACM International Conference on Supercomputing*. 203–214.

[31] Russ Rew and Glenn Davis. 1990. NetCDF: an interface for scientific data access. *IEEE computer graphics and applications* 10, 4 (1990), 76–82.

[32] Steve Scargall. 2020. PMDK Internals: Important Algorithms and Data Structures. In *Programming Persistent Memory*. Springer, 313–331.

[33] Jerome Soumagne, Richard Warren, Jingqing Mu, Venkat Vishwanath, Francois Tessier, Suren Byna, Quincey Koziol, Houjun Tang, Teng Wang, Bin Dong, and Jialin Liu. [n.d.]. Final Technical Report - Proactive Data Containers for Scientific Storage. ([n. d.]). https://doi.org/10.2172/1577855

[34] Kun Tang, Ping Huang, Xubin He, Tao Lu, Sudharshan S Vazhkudai, and Devesh Tiwari. 2017. Toward managing hpc burst buffers effectively: Draining strategy to regulate bursty i/o behavior. In *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 87–98.

[35] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory I/O primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–7.

[36] Kenton Varda. 2013. Capn'n Proto Cerealization Protocol. https://capnproto.org/

[37] Thomas Willhalm Patrick Lu Blazej Filipiak Sri Sakthivelu Vish Viswanathan, Karthik Kumar. 2015. Memory Latency Checker. https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html

[38] Randolph Voorhies. 2014. cereal - A C++11 library for serialization. https://uscilab.github.io/cereal/

[39] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 427–439.

[40] Jian Xu and Steven Swanson. 2016. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 323–338.

[41] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 207–219.