Utilizing Persistent Memory in Parallel I/O Libraries

Luke Logan
Illinois Institute of Technology
Chicago, IL, USA
llogan@hawk.iit.edu

Jay Lofstead
Sandia National Labs
Albuquerque, NM, USA
glofst@sandia.gov

Scott Levy
Sandia National Labs
Albuquerque, NM, USA
sllevy@sandia.gov

Patrick Widener
Sandia National Labs
Albuquerque, NM, USA
pwidene@sandia.gov

Xian-He Sun
Illinois Institute of Technology
Chicago, IL, USA
sun@iit.edu

Anthony Kougkas
Illinois Institute of Technology
Chicago, IL, USA
akougkas@iit.edu

KEYWORDS

Persistent Memory, Libraries, Memory Management, Memory Mapped I/O

1 EXTENDED ABSTRACT

Scientific applications use parallel I/O (PIO) libraries, such as ADIOS [7], HDF5 [3], and NetCDF [8], to manage the complexity of reading and writing mass amounts of data to storage efficiently. A common and effective approach to improving I/O performance is multi-tiering, which utilizes fast storage to absorb writes and store data expected to be accessed in the near future. However, PIO libraries have not adequately adapted to the emergence of persistent memory (PMEM) as a new tier of storage, which provides comparable performance to DRAM and the ability to be accessed directly using CPU load/stores. Existing research on utilizing PMEM as fast storage [1, 4, 10, 11], mainly focuses on redesigning single-node filesystems to reduce the software overhead incurred by traditional storage stacks designed for slower storage media. While PIO libraries currently depend on single-node filesystems to read and persist data, simply improving the performance of these filesystems is not sufficient to gain the full benefits of PMEM. At a fundamental level, the MPI-IO and POSIX interfaces typically used by PIO libraries to interact with storage cause significant overhead due to unnecessary network communication and data copying when PMEM is available. For example, persisting an in-memory data structure to storage using POSIX requires it to be serialized into another in-memory buffer and then copied to PMEM as opposed to simply serializing the data directly into PMEM, resulting in an unneeded copy of the entire data structure, wasting both precious space and performance. **PIO libraries must adapt in order to realize the full potential of PMEM as fast storage.** In addition, PIO libraries tend to introduce complex APIs and configuration spaces that cause significant burden on programmers to store basic data structures, such as integer arrays. In many cases, more simplistic interfaces, such as key-value stores, are preferable.

In this work, we explore the use of memory mapping as an alternative to POSIX and MPI-IO for interacting with PMEM through pMAP: a lightweight I/O library with a simplistic key-value store interface. We demonstrate that our approach can yield up to 2x improvement in performance over other popular PIO libraries by eliminating unnecessary data copies and network communications.

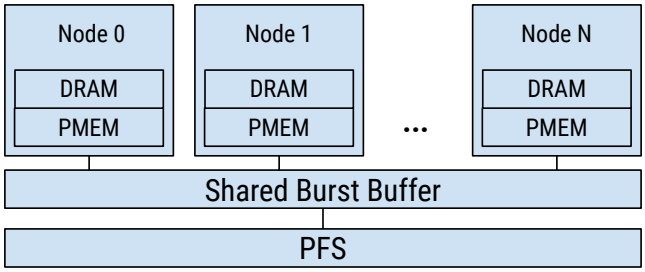

Figure 1: Basic Machine Architecture

```
1. #include <pmap/pmap.hpp>
2. pmap::PMEM pmem;
3. pmem.mmap(std::string filename, int comm);
4. pmem.munmap();
5.
6. pmem.store<T>(std::string id, T &data);
7. pmem.alloc<T>(std::string id,
8.   int ndims, size_t *dims);
9. pmem.store<T>(std::string id, T *data,
10.  int ndims, size_t *offsets, size_t *dimspp);
11.
12. pmem.load<T>(std::string id);
13. pmem.load<T>(std::string id, T &num);
14. pmem.load<T>(std::string id, T *data,
15.  int ndims, size_t *offsets, size_t *dimspp);
16. pmem.load_dims(std::string id,
17.  int *ndims, size_t *dim);
```

Figure 2: pMAP API

2 DESIGN

pMAP is designed for systems where each node performing I/O contains a local PMEM device, which is depicted in Figure 1, and provides a key-value interface, depicted in Figure 2. In Figure 3 (on the poster), we show an example of pMAP where each process writes 500 doubles to a 1-D array. The pMAP implementation is 16 lines of code (LOC) and 132 tokens. We also implement the example in HDF5, NetCDF, pNetCDF, and ADIOS (not shown due to space). The equivalent HDF5 code requires the creation of multiple property lists, dataspaces, and datasets, forcing users to learn many specialized APIs that contain many parameters. The equivalent HDF5 code for this simple task is 42 LOC and 253 tokens, a 2.62x increase in LOC and 92% increase in the number of tokens. (Note,

LOC ignores lines without significant code.) While NetCDF and pNetCDF reduce code volume significantly compared to HDF5, they still require specialized APIs to allocate array dimensions, adding complexity. While specialized APIs aren't needed to allocate dimensions in ADIOS, users must manually create variable names for each dimension and then associate them with the array being stored, whereas pMAP simply stores the array dimensions in the alloc function. The equivalent ADIOS code is 24 LOC and 164 tokens. Overall, pMAP provides a more compact, user-friendly interface than other representative PIO libraries.

pMAP utilizes memory mapping (mmap) to interface with PMEM as opposed to MPI-IO and POSIX. Typically, PIO libraries depending on POSIX or MPI-IO must first serialize data to an in-memory buffer and then copy the serialized data to PMEM. Conversely, to load data from PMEM, they must first copy the serialized data from PMEM to an in-memory buffer, and then deserialize the data into another in-memory buffer. However, pMAP serializes data directly to an mmapped file using serialization libraries, such as BP4 [7]. For reads, data is deserialized directly from PMEM to an in-memory buffer. This avoids an entire copy of the data structure caused by the POSIX and MPI-IO interfaces. In addition, pMAP makes the MAP_SYNC flag optional when memory mapping the PMEM. This flag prevents the filesystem from silently relocating mapped blocks of the file and guarantees that the metadata regarding block allocations are consistent, which helps improve crash consistency. However, this can cause severe performance degradation, as metadata flushes are much more frequent. The user can choose to enable this depending on their security needs. Lastly, to persist data to mass storage, pMAP will trigger a burst buffer, such as DataWarp, to asynchronously migrate the data in PMEM to permanent storage.

3 EVALUATIONS

Testbed: Tests were conducted on a Compute Skylake node in Chameleon Cloud [2], which are equipped 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 5.4.0-36.

Emulating PMEM: To emulate PMEM, we use the approach presented in Strata [4]. We utilize Linux's PMEM emulator to treat 60GB of DRAM as PMEM and format it with DAX-enabled EXT4. 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 [9].

Workload: In these tests, we compare the performance of pMAP and other PIO libraries using an emulation of the S3D combustion code [5, 6]. First, a 40GB 3-D rectangle is generated and persisted to PMEM. Each process persists an equivalently-sized portion of the rectangle. Next, an analysis code reads the 40GB rectangle back into memory. Each process reads the exact same portion of the rectangle that was originally written. Between 8 and 48 processes are used. We compare the performance of pMAP with ADIOS, NetCDF, and pNetCDF on a single Skylake node. Only a single node is used for the evaluations as pMAP stores all data in node-local PMEM, which requires no inter-node communications, making it scale linearly with an increase in nodes. We measure the time difference between when a file is opened and when it is closed. Each experiment is conducted 3 times, and the average time is shown. We show the performance of pMAP with (pMAP-B) and without (pMAP-A) the MAP_SYNC flag enabled.

Results: The results of the experiments are shown in the poster in Figures 4 and 5. We find that pMAP outperforms NetCDF and pNetCDF dramatically, as much as 5x, in both the write and read workloads. This is because NetCDF and pNetCDF forces the 3-D rectangle to be stored logically as an array, requiring the data to be shuffled, incurring significant and avoidable communication and data copying costs. ADIOS and pMAP store data without any shuffling in local storage, which eliminates the communication cost and most data copying costs seen in NetCDF and pNetCDF. While ADIOS makes significant improvements over NetCDF and pNetCDF, pMAP outperforms it by 15% in the write workload and 2x in the read workload when the MAP_SYNC flag is disabled. This is because pMAP (de)serializes data directly to/from the PMEM, avoiding an entire copy of the rectangle. However, when enabled, MAP_SYNC causes a flurry of metadata requests, eliminating the performance benefits of memory mapping entirely, and potentially causing performance to be worse than simply using POSIX. Overall, we see that pMAP can perform at least 15% better for writes and 2x better for reads depending on its configuration.

4 CONCLUSION

Persistent memory (PMEM) has caused a revolution in the design of storage stacks. Much research has been conducted on optimizing filesystems for accessing PMEM, significantly reducing the overheads imposed by traditional I/O stacks. However, parallel I/O (PIO) libraries have yet to adapt to the emergence of this technology, resulting in severe performance loss on systems where PMEM is available. Existing PIO libraries simply rely on MPI-IO and POSIX in order to interact with storage, incurring significant overhead due to unnecessary data copying and network communication. In this work, we design pMAP: a simple I/O library that efficiently utilizes PMEM to read and persist data with memory mapping. We compared pMAP with multiple PIO libraries and found that our approach yields improvements of up to 15% for writes and 2x for reads.

REFERENCES

[1] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–15.

[2] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

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

[4] 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.

[5] 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*.

[6] 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.

[7] 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.

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

[9] 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.

[10] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. 2014. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–14.

[11] 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.