High-Performance PMEM-Aware Collective I/Os

Keegan Sanchez
Washington State University
Vancouver

Alex Gavin
Washington State University
Vancouver

Suren Byna
The Ohio State University

Kesheng Wu
Lawrence Berkeley National Laboratory

Xuechen Zhang
Washington State University
Vancouver

ABSTRACT

Collective I/Os are widely used to transform small non-contiguous accesses into large contiguous accesses for parallel I/O optimization. The existing collective I/O techniques assume that computer memory is volatile. They are limited both by the size of the buffer, which must be small so data is not lost during a crash, and the communication overhead that occurs during collective I/O. PMIO is a proposed framework to utilize persistent memory (PMEM) for collective I/O, as opposed to DRAM. First, we utilize a log-structured buffer to take advantage of the non-volatility of PMEM. Second, we utilize larger buffers to take advantage of the larger space available on less expensive PMEM. Finally, we implement a two-phase merging algorithm to eliminate the communication overhead. The poster provides an overview of collective I/O and its problems, an introduction to PMEM, an outline of PMIO, and a brief discussion of PMIO's performance.

KEYWORDS

Collective I/O, Persistent Memory

ACM Reference Format:
Keegan Sanchez, Alex Gavin, Suren Byna, Kesheng Wu, and Xuechen Zhang. 2023. High-Performance PMEM-Aware Collective I/Os. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Collective I/O is a performance-critical facility in the MPI-IO library used by most petascale HPC applications. It consists of two major phases: communication phase and I/O phase. In the communication phase, metadata exchange and data shuffling are executed among processes on compute servers. In the I/O phase, I/O operations are executed between compute and I/O servers [7]. With collective I/O optimization, fewer and larger requests are generated to I/O servers improving efficiency.

The existing implementations [7, 9, 10] of collective I/O have the following issues. First, they use DRAM as a collective I/O buffer for data shuffling. Because the per-core DRAM capacity of compute servers is limited on supercomputers, the extra memory used for the buffer is also limited, leading to an excessive amount of data shuffling when the buffer size is significantly smaller than the size of file domains to be accessed. Second, they assume that the collective I/O buffer is volatile. All data written in the collective I/O buffer can be lost upon failures. Consequently, a large collective I/O buffer size increases the risk of losing data, especially in supercomputing systems where frequent system failures can be the norm [6]. They cannot recover the data in volatile I/O buffers. Third, the communication overhead dominates the performance of collective I/O. Our study shows that the communication time of data shuffling accounts for up to 91% of the execution time of the collective I/Os in the MPICH library [1].

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 DESIGN OF PMIO

In this paper, we design and implement a new collective I/O framework, PMIO, which uses persistent memory (e.g., Intel Optane DCPMM) to increase the size of collective I/O buffers for exploring higher I/O efficiency and completely remove the communication bottleneck of data shuffling. It has the following features.

Log-structured I/O buffers. We implement a log-structured collective I/O buffer, to eliminate random accesses and explore the high bandwidth of persistent memory. The log-structured buffer is sequentially appended when serving write requests and cleaned up when it is full using log management threads.

Two-level log merging. We design a two-level log merging approach to replace the original two-phase I/Os to eliminate data shuffling on compute servers. The two-level log merging consists of merging on compute servers and merging on I/O servers respectively. On the compute servers, data in the buffers are segregated according to their destination I/O servers. When the log merging is triggered, data written by different processes on the same compute node are merged locally. No data shuffling is needed. The merged data are then written to respective I/O servers. On the I/O servers, the data from different compute servers are merged by operating systems again before writing to disks. All log merging is executed asynchronously in the background.

Enforcing crash consistency. Crash consistency is the recoverability of persistent data from memory in a consistent state after system failures. PMIO enforces crash consistency by storing both metadata and data of write requests in the collective I/O buffers. It can recover the data by replaying log items in the metadata logs. Consequently, we can safely increase the size of the collective I/O buffer to exploit higher I/O efficiency.

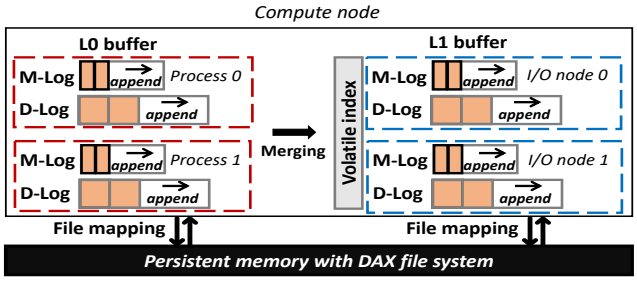


Figure 1: Overview of log-structured PMIO buffer.

Read buffers. PMIO supports reading from the collective I/O buffers in persistent memory by maintaining an index structure in memory. The index can be dynamically partitioned and placed in DRAM or persistent memory according to the memory demand of applications.

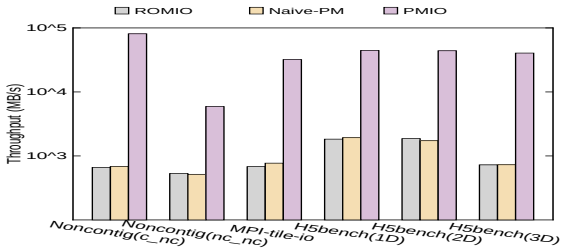
Data movement flow. As shown in Figure 1, each MPI process appends data to one L0 sequentially when PMIO write is served. When the L0 buffers are full, they are then merged to form L1. The merging process is executed by a log management thread asynchronously. Each L1 buffer stores data mapped to one I/O server. When the L1 buffers are full, their data are sent to the corresponding I/O servers in batch. PMIO relies on the operating systems of I/O servers to merge the requests sent from multiple compute servers before writing them to parallel file systems. Consequently, in our design, MPI processes on compute servers only write to node-local L0+L1 and do not need to send data to the processes on other compute servers. This ensures that there is no communication for data shuffling among processes on compute servers.

For PMIO reads, if the requested data are available in the L0+L1 buffers, they are returned immediately. Otherwise, they are served from parallel file systems. We build an index structure in DRAM for searching the requested data in L1.

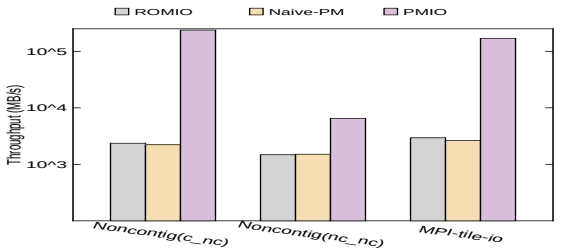
3 EVALUATION

Experimental platforms. We use the Perlmutter supercomputer in the experiments. It is a Cray EX with a peak performance of about 59.9 petaflops [2]. Because Perlmutter does not have local persistent memory on compute nodes, we model node-local persistent memory using DRAM. Specifically, we store M-Logs and D-Logs in in-memory file system tmpfs on compute nodes. We then use mmap() to map these log files to process address space. Finally, PMIO accesses M-Logs and D-Logs in tmpfs in the same way as those in persistent memory. The emulation-based approach has been used in other projects [3–5, 8]. Because we assume the emulated PM has the same performance as DRAM, we did not add additional latency for PM accesses.

Results: We have three observations from the results shown in Figure 2. First, the write throughput of PMIO is 121X and 117X higher than ROMIO and Naive-PM, respectively. This is because PMIO avoids data shuffling among processes on compute servers and removes the communication overhead. The log-structured buffers promote sequential accesses in persistent memory, leading to improved I/O performance. Second, the read throughput of PMIO is 100X and 151X higher than ROMIO and Naive-PM, respectively. Because we can allocate a large space in persistent memory



(a) Write



(b) Read

Figure 2: Performance of the benchmarks on a log scale.

for buffering data, the benchmarks can read the data in persistent memory directly without accessing parallel file systems. As a result, read throughput is also significantly improved for all the benchmarks. Third, PMIO works for representative benchmarks and I/O kernels using HDF5, which is widely adopted in HPC applications.

4 CONCLUSION

This poster, describes PMIO, a PMEM-aware collective I/O framework to reduce communication on compute servers, improve data access locality in PMEM, and enforce crash consistency for failure recovery. PMIO uses log-structured collective I/O buffers in PMEM to achieve high write bandwidth. It supports two-level log merging to transform small, non-contiguous requests into large, contiguous requests without expensive data shuffling. It records the metadata of requests in PMEM to provide recoverability upon failures. Compared to ROMIO collective I/Os using DRAM and PMEM as the collective buffers, PMIO improves the I/O throughput by up to 122X on the Perlmutter supercomputer.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Visiting Faculty Program (VFP) This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and also used resources of the National Energy Research Scientific Computing Center (NERSC). The research was also supported by the US National Science Foundation under CNS 1906541, CNS 2216108, and OAC 2243980.

REFERENCES

[1] [n. d.]. Official MPICH Repository. https://github.com/pmodels/mpich.

[2] [n. d.]. Perlmutter. https://docs.nersc.gov/systems/perlmutter/architecture/.

[3] Bao Nguyen, Hua Tan, Kei Davis, and Xuechen Zhang. 2019. Persistent Octrees for Parallel Mesh Refinement through Non-Volatile Byte-Addressable Memory. IEEE Transactions on Parallel and Distributed Systems 30, 3 (2019), 677–691. https://doi.org/10.1109/TPDS.2018.2867867

[4] Bao Nguyen, Hua Tan, and Xuechen Zhang. 2017. Large-scale Adaptive Mesh Simulations Through Non-volatile Byte-addressable Memory. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17).

[5] Jiaxin Ou, Jiwu Shu, and Youyou Lu. 2016. A High Performance File System for Non-volatile Main Memory. In Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16).

[6] H. Song, C. Leangsuksun, R. Nassar, N.R. Gottumukkala, and S. Scott. 2006. Availability modeling and analysis on high performance cluster computing systems. In First International Conference on Availability, Reliability and Security (ARES'06). 8 pp.–313. https://doi.org/10.1109/ARES.2006.37

[7] R. Thakur, W. Gropp, and E. Lusk. 1999. Data sieving and collective I/O in ROMIO. In Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation. 182–189. https://doi.org/10.1109/FMPC.1999.750599

[8] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. SIGPLAN Not. 47, 4 (March 2011), 91–104.

[9] Xuechen Zhang, Song Jiang, and Kei Davis. 2009. Making resonance a common case: A high-performance implementation of collective I/O on parallel file systems. In 2009 IEEE International Symposium on Parallel & Distributed Processing. 1–12. https://doi.org/10.1109/IPDPS.2009.5161070

[10] Xuechen Zhang, Jianqiang Ou, Kei Davis, and Song Jiang. 2018. Orthrus: A Framework for Implementing High-Performance Collective I/O in the Multicore Clusters. In Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '13). Association for Computing Machinery, New York, NY, USA, 113–114. https://doi.org/10.1145/2462902.2462924