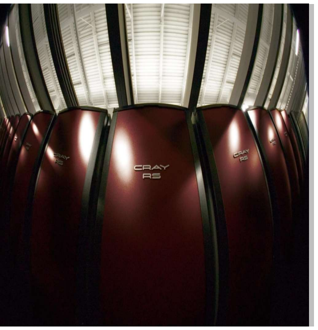


pMEMCPY: Effectively Leveraging Persistent Memory as a Storage Device

Jay Lofstead, Luke Logan, Anthony Kougkas

Sandia National Laboratories

Exceptional service in the national interest

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

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.

Why PMEM?

- Persistent storage with a CPU-level load/store interface?!
  - Intel + Micron, now Intel and Micron
  - PMDK makes programming “portable”
- But the memory bus!
- But it is still on the node. What about failures?

Getting the Performance is Hard?

- IO Stack has many layers
  - Storage system software is important, but not the only problem
- IO libraries are written for spinning media, but are feature rich
  - HDF5, ADIOS, NetCDF/PnetCDF

HDF5

Feature Set

- Hierarchical namespace
- Primitive and compound types
- Data Layout Policies
  - Contiguous
  - Chunked
- Multi-tiering support
- Transparent operations
- Highly extensible through VOL plugins

Limitations

- Complex API & configuration space
- Data is stored in a single binary file
  - Metadata is not human-readable
  - Version control less efficient
- Compound types cannot be nested
- POSIX and MPI-I/O are used to interface with storage, causing overhead

ADIOS

Feature Set

- Unifies I/O transport mechanisms under a simple key-value store API
  - POSIX, MPI-IO, NetCDF, etc.
- Processes store data independently without data rearrangement
  - Avoids data copying and network communication costs
- Support for transparent operations

Limitations

- Data is stored in a single binary file
  - Metadata is not human-readable
  - Version control less efficient
- POSIX and MPI-I/O are used to interface with storage, causing overhead

NetCDF/PnetCDF

Feature Set

- Simpler API than HDF5
- Support for transparent operations
- Stores data using a contiguous layout policy

Limitations

- Data rearrangement can cause significant data copying & network communication costs
- Data is stored in a single binary file
  - Metadata is not human-readable
  - Version control less efficient
- POSIX and MPI-I/O are used to interface with storage, causing overhead

Optimized approach

API

- Simple key-value store API
- Can store primitive types and arrays of those types
- Templating allows for storing complex data types
- Support for transparent compression by changing serializer type

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

pmem.store<T>(std::string id, T &data,
 pmemcpy::SerializerType s = Default);
pmem.alloc<T>(std::string id,
 int ndims, size_t *dims,
 pmemcpy::SerializerType s = Default);
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);
```


7

Usage Example

```
#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_COMM_WORLD);
   pmem.alloc<double>("A", 1, &dimsf);
   pmem.store<double>("A", data, 1, &off, &count);
   MPI_Finalize();
}
```

pMEMCPY for writing a 1-D array of 100 doubles per-process

API Comparison

% increase in # tokens
relative to pMEMCPY

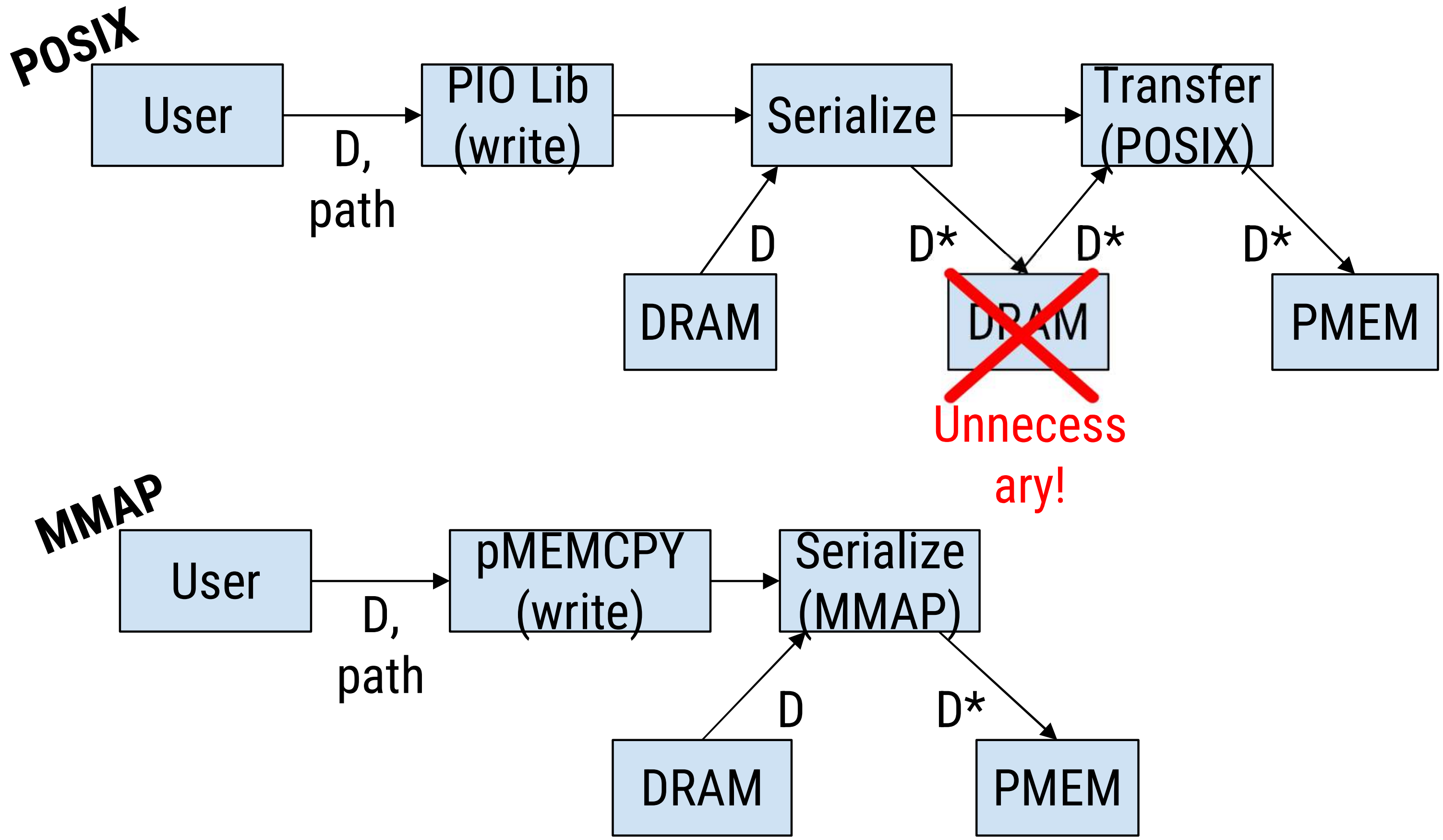
	LOC	# Tokens	% Larger
pMEMCPY	16	132	0%
ADIOS	24	164	24%
NetCDF	26	180	36%
HDF5	42	253	92%

pMEMCPY requires the least user effort to store basic data structures.

Avoid Unnecessary Data Copies

D is a data structure

D* is the serialized form of D



Memory Mapping Caveats

- Data stored in CPU caches may not be written to PMEM immediately
- Filesystems may change file metadata while it is also memory mapped
  - E.g., one process may decrease the size of a file while another process has the same file memory mapped, causing inconsistency
- There are two general approaches to guarantee consistency:
  - The MAP_SYNC flag
  - The msync() system call
- MAP_SYNC flushes updates to PMEM during I/O
  - Can cause many small I/Os and page faults to occur
  - Performance degradation due to increased latency
- MAP_SYNC is best when most I/O is smaller than a page
  - E.g., updating persistent pointers in a B-tree
- pMEMCPY mainly performs large, sequential reads and writes
  - MAP_SYNC is overkill in most cases for this workload
- msync() can be called after every I/O operation instead

Evaluation

S3D Combustion Emulation

Writes a 40GB 3-D rectangle to PMEM

The 40GB is divided equally among each process

S3D Combustion Analysis Emulation

Reads the 40GB 3-D rectangle from PMEM

Each process reads the same region that was originally written

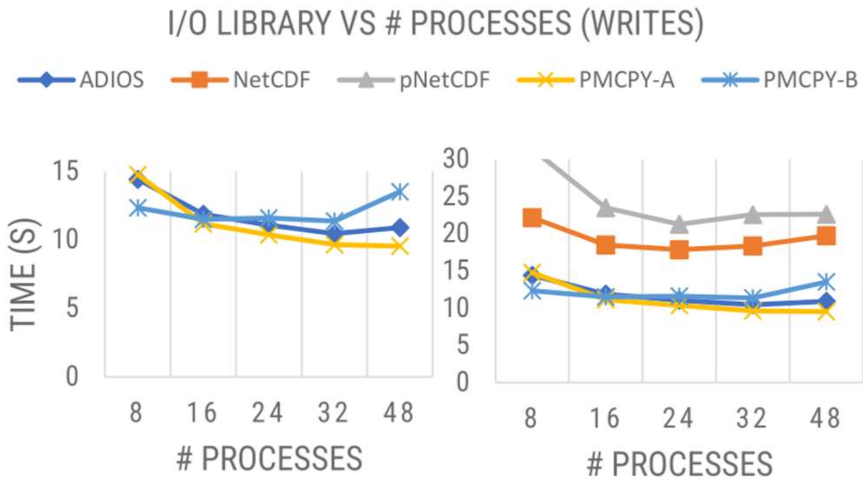
Test Environment

PMEM Emulation	
Capacity	80GB
Read BW	30GB/s
Write BW	8GB/s
Read Latency	300ns
Write Latency	125ns

Chameleon Skylake	
DRAM	192GB
CPU Cores	24
CPU Threads	48
OS	Ubuntu 20.04
Kernel	5.4.0-70-generic

- Experiments were conducted on a single Skylake node using emulated PMEM
- Only one node is used because pMEMCPY performs independent I/O, and will scale with an increase in number of nodes

Writing Results



* pMCPY-A has MAP_SYNC disabled
* pMCPY-B has MAP_SYNC enabled

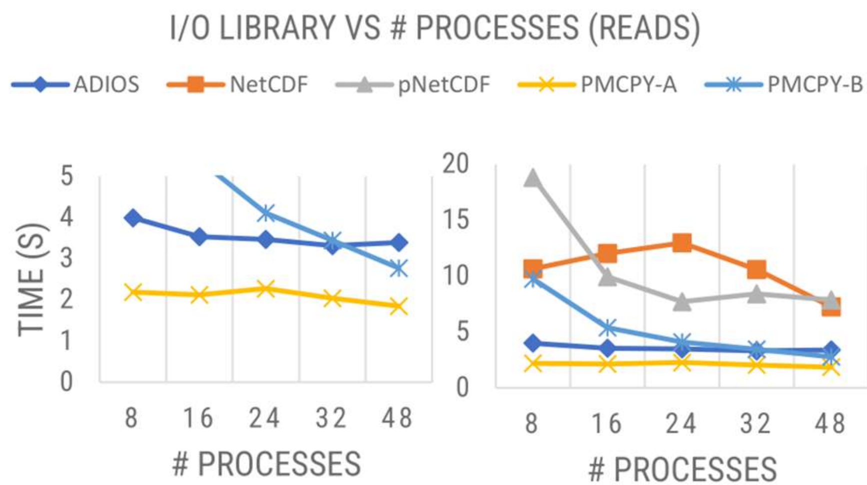
pMCPY-A is **15%** faster than ADIOS

pMCPY-A is **2.5x** faster than NetCDF and pNetCDF

pMCPY-B is no faster than ADIOS

Reading Results

Sandia National Laboratories



* pMCPY-A has MAP_SYNC disabled
* pMCPY-B has MAP_SYNC enabled

pMCPY-A is **2x** faster than ADIOS

pMCPY-A is **5x** faster than NetCDF and pNetCDF

pMCPY-B is no faster than ADIOS

Conclusions

- PMEM can be used as a fast, temporary storage area
- PIO libraries introduce significant software overhead when interacting with PMEM
- PIO libraries also tend to have complex interfaces that put strain on application developers

- We found that pMEMCPY can reduce code size by as much as **92%** over other PIO libraries by providing a simple key-value store interface
- We found that pMEMCPY can improve writes by **15%** and reads by **2x** over other PIO libraries through memory mapping

Questions?

pMEMCPY on github.com/sandialabs/pMEMCPY shortly

gflofst@sanda.gov or pmemcpy@sandia.gov for questions