sy8wg, Shunyu Yao

Persistent Memory in HPC Applications

Project Report

1. Problem Statement

The introduction of the usage of persistent memory enriches the storage stack, bringing in one more flexible layer to be utilized. As a memory device, it has a higher capacity than traditional DRAM; as a persistent storage, it is faster than block devices. In datacenters where High Performance Computing (HPC) applications are taking up hundreds of hours of computing time with huge amount of disk I/O, persistent memory can take roles to streamline the dataflow between host and permanent storage.

More importantly, because of its non-volatile nature, and also because it is closer to CPU than disk is, persistent memory makes an ideal vault to drain the jobs from host when crashes happen, or to checkpoint them just to prevent so because it has enough space. This is especially true for HPC applications since the majority of them have prolonged execution time, crashes and system failure are fatal and could lead to massive waste of computation time and resources if the execution progresses are left unpreserved or unable to be recovered. A good example would be the emerging green energy datacenters located on the extensive flat lands of the Great Plains in the mid-west United States, where clusters are plugged into wind farm energy in order to reduce the computing carbon footprint brought by soaring amount of data consumed by applications nowadays. The paradigm shift from computation-centric to data-centric stimulates the growth of could providers that would like to come up with cheaper, more environment-friendly solutions. One of the challenges they are facing is that the wind does not blow all year long, thus they need to constantly shutdown the clusters proportionally accordingly, The ability to checkpoint the on-going massive amount of jobs in given time and the ability to bring them back online later become vital to their operations. Persistent memory would satisfy such requirements well if used as a draining storage for checkpointed job images.

Containerizing HPC applications are widely gaining popularity among datacenters. Contrast to the comparatively full stack virtualization by Virtual Machines, containers' kernel-level services are provided by the host OS, only the environmental libraries / binaries are packaged with the absent of a virtual OS. Such properties enable the lightweight virtualization, leading to more migration flexibility, and HPC applications are undoubtedly among the most benefitted ones. There has been userspace checkpointing projects that are targeting containers, with CRIU the most popular project starting 2011.

As the fault tolerance solution to HPC application grows more lucid, persistent memory's integration into the container sounds really promising. But there are several issues to be tackled with:

- How should we expose the persistent memory to containers? What are the trade-offs?
- What's the extra work needed on integrating persistent memory with containers so that it can be considered "properly contained"?
- How much can HPC applications benefit from persistent memory?

Each one of the above questions are big enough to be answered with an individual research project. But this project aims to conclude a brief answer that we can learn the research & development directions from.

2. Key Idea

a. PM Exposure

There poses two possible ways for a container to access persistent memory region available on the host machines: One way is to directly expose it like passing in a normal device; the other is to mount it onto the host OS's file system as a volume.

Presumably, the former way includes less translation from the device to containers, but requires both in-container environment to be setup and device driver to be installed on the host machine, which loses the portability of containers. What's more, direct exposure rises a challenge that the device won't be well "contained", it needs root privilege to fire up a container who wants access to the device on host. The second way, which is exposing persistent memory as a volume storage, might introduce

performance penalty because its byte-addressability is not made use of, and imposing extra layers of consistency guarantee are expensive.
To confirm the presumptions, we need to run benchmark against inside vs. outside container scenarios to see whether the performance penalty from virtual filesystem mapping of container itself matters; we also need to run benchmark against the way persistent memory region is mapped in the first place: Direct I/O as a device (DEV-DAX) vs. file system mapped region (FSDAX), to see the performance difference of the two.

b. PM Containerization

Containerization needs to separate users from affecting each other, usually it means to use chroot() (now pivot root) to change the root of the file system to a different location for that contained process, and to use Cgroups to throttle physical resources such CPU cycles, memory, device I/O, and to use namespace to control abstract resources such as PID, Network, Mount, etc. The combination of all three creates a jail for the contained process. But in terms of what exactly happens to the jailed process, containerization technologies have broadly diversified, for instance, Docker used to have less constraints to prevent root escalation attack but focuses on service applications; Singularity, on the other hand, focuses on datacenter scenarios and puts relinquishing jailed process's privilege as priority.

To conclude, containers are just more isolated processes. NDCTL has already provided certain level of isolation for PMEM access from bare machine processes, such as page faults on illegal memory access, read/write protection, etc., but to "contain" it, we need NDCTL to be container-aware, which could mean a Linux kernel module that allocates persistent memory regions through Linux namespace and Cgroups on behalf of the contained process. In this project, only Cgroup throttling is explored.

c. PM for HPC Applications

Since persistent memory is both closer to CPU and non-volatile, it would function more like a disk cache, saving the time to flush modifications back to disk on every write. To test and demonstrate the potential speedup persistent memory can bring to HPC application checkpointing, based the original LULESH serial code, two versions of checkpointing are implemented: the baseline version of checkpointing into files on disk vs. the PMDK checkpointing to persistent-memory-mapped files. This project uses serial LULESH code, and checkpoints at the end of each iteration with all the updated variables of cube nodes and elements written back.

3. Goal

The goal of this project aims to answer the three questions proposed in the Problem Statement section in an exploring manner. The following tasks are formulated separately:

- Benchmarking the persistent memory region with two variances: inside of container vs. outside of container; I/O into persistent memory using DAX, file-system-mapped DAX with Libpmem, default mmap, and Pmemblk with benchmarking tool FIO.
- Throttling persistent memory devices I/O using Cgroups with two variances: bytes per second (BPS); I/O device weights. Two PMEM devices are virtualized using Intel NDCTL utilities.
- Adding checkpoints to LULESH, a hydrodynamic simulation model, with 2 implementations: Benchmarking into devices, benchmarking into persistent memory region with Libpmem.

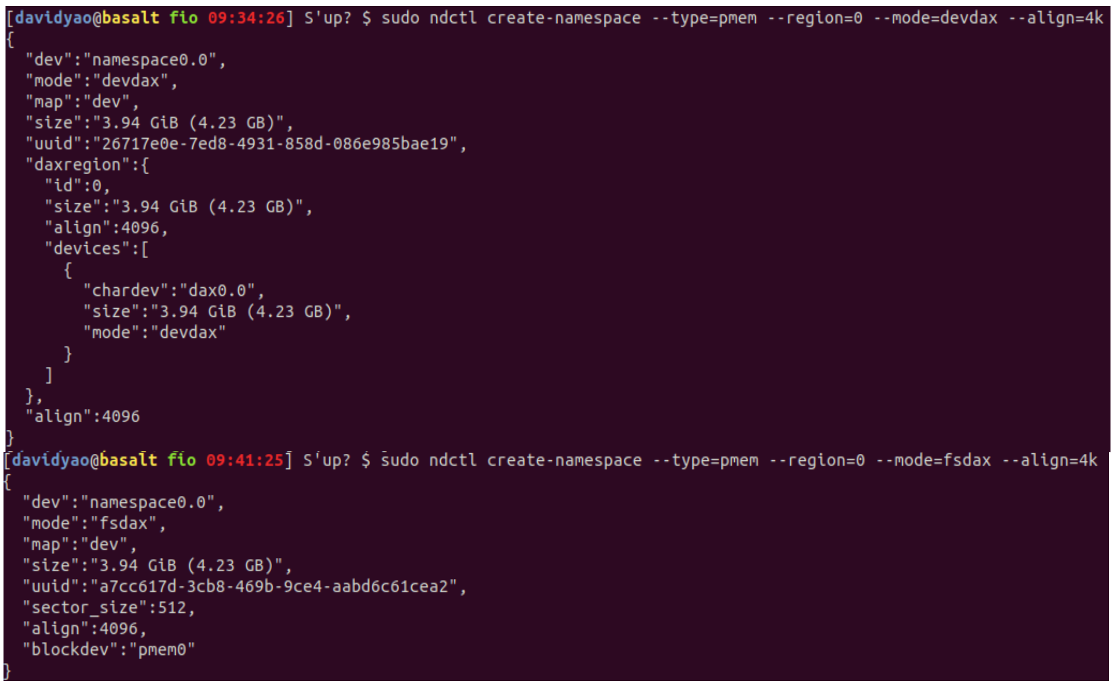
4. Mechanism / Tools

a. DAX

Direct Access, a Linux Kernel feature that removes the page cache from the I/O path and allows mmap() to establish direct mappings to persistent memory media.

b. PMDK & NDCTL

The Persistent Memory Development Kit is a collection of libraries and tools built based on the DAX feature of Linux to allow applications to access persistent memory regions as memory-mapped files, with Libpmem the major library of it. The development team built another tool to help manage LIBNVDIMM (non-volatile memory device) sub-system in the Linux kernel, which is called NDCTL, a CLI utility. It can partition persistent memory into unified regions (called "namespace"), and present them to userspace applications as individual PMEM devices. Two DAX modes can be applied when creating namespaces: DEV-DAX, creating a single character device file; FSDAX, creating a block device file that supports DAX capabilities of Linux FS.



```
[davidyao@basalt fio 09:34:26] S'up? $ sudo ndctl create-namespace --type=pmem --region=0 --mode=devdax --align=4k
{
  "dev":"namespace0.0",
  "mode":"devdax",
  "map":"dev",
  "size":"3.94 GiB (4.23 GB)",
  "uuid":"26717e0e-7ed8-4931-858d-086e985bae19",
  "daxregion":{
    "id":0,
    "size":"3.94 GiB (4.23 GB)",
    "align":4096,
    "devices":[
      {
        "chardev":"dax0.0",
        "size":"3.94 GiB (4.23 GB)",
        "mode":"devdax"
      }
    ]
  },
  "align":4096
}
[davidyao@basalt fio 09:41:25] S'up? $ sudo ndctl create-namespace --type=pmem --region=0 --mode=fsdax --align=4k
{
  "dev":"namespace0.0",
  "mode":"fsdax",
  "map":"dev",
  "size":"3.94 GiB (4.23 GB)",
  "uuid":"a7cc617d-3cb8-469b-9ce4-aabd6c61cea2",
  "sector_size":512,
  "align":4096,
  "blockdev":"pmem0"
}
```

In this project, the implementation of checkpointing an HPC application involves persistent memory programming; NDCTL helps to create PMEM namespace in different modes to suit the benchmarking/implementing purpose. Since this project doesn't have the budget for a real persistent memory, a 4GB DRAM is reserved by bootloader and appears to be a persistent memory region. To create a DEV-DAX NDCTL namespace, only "ndctl create-namespace" command is needed; But to create a FSDAX region, we need to mount a file system on it. All namespaces created in this project are 4K aligned.

c. FIO

A handy to benchmark I/O of persistent memory is FIO, it takes in a test case specification file that explicitly specify parameters such as which device to use, how many I/O processes to fire up, how big of a file each I/O process is in charge, I/O depth, etc.

Another parameter needs to be specified in the specification is the I/O engine. It defines the general style of how I/O is going to be issued during the test. There are 4 engines used in this project:

- mmap: File is memory mapped with mmap(2) and data copied to/from using memcpy(3).
- libpmem: Read and write using mmap I/O to a file on a filesystem mounted with DAX on a persistent memory device through the PMDK libpmem library.
- dev-dax: Read and write using device DAX to a persistent memory device (e.g., /dev/dax0.0) through the PMDK libpmem library.
- pmemblk: Read and write using filesystem DAX to a file on a filesystem mounted with DAX on a persistent memory device through the PMDK libpmemblk library.

Mmap is used as a baseline test case, mainly for the comparison against libpmem. A FSDAX namespace is created but the device is not mounted into the host file system. Libpmem and dev-dax are tested to see the difference between libpmem mmap() access vs. libpmem DAX access. Libpmem requires FSDAX namespace to be created and the PMEM device file should be mounted, and dev-dax requires a DEVDAX namespace created. Pmemblk is using PMDK pmemblk library, but the use case is the same as libpmem.

d. Cgroups

Cgroups uses a tree structure to manage control groups. Different categories of resources are in their own Cgroup subsystems, and subsystems can be grouped into (or by their own to form) hierarchies, which are trees of control groups and appear as trees of paths in the file system, thus, each node directory of such trees is a Cgroup. Depending on the hierarchy tree the Cgroup is in, different parameter files are in its directory specifying how much of what resource to constraint. The tree structure decides that the rules in parent nodes will be applied with a higher priority than its children (only true for Cgroup v1).

e. LULESH

LULESH is a hydrodynamic model developed by Lawrance Livermore National Laboratory that simulates a cube of nodes with kinetic and electric parameters. An initial condition will be given, and during each iteration, the condition on each node will be calculated to simulate the propagation of these parameters, and a time constraint function dynamically determines the end of the program according the result discrepancy between current iteration and last iteration. LULESH is a classic stencil model, and has a typical HPC application's I/O pattern that will make a great target to test the impact of persistent memory programming can bring. Thus, checkpointing is implemented at the end of each iteration with both PMDK and disk read/write.

5. Results

a. FIO Benchmark

With all the parameters set to the same except the I/O machines, after dozens of tests for each test suite to get steady results, the results are abstract out as below:

	Max Bandwidth	**Avg Bandwidth**	**IOPS**
Mmap	5840MB/sec	5569MB/sec	1426K
Libpmem	6603MB/sec	6297MB/sec	1612K
Dax	6613MB/sec	6307MB/sec	1615K
Pmemblk	3087MB/sec	2944MB/sec	754K

Recall the section 4.c for the takeaways we can abstract out from comparisons between different I/O engine tests. comparing Libpmem to Mmap, the normal mmap I/O is actually slower than PMDK mmap I/O. This result is expected since the normal memory mapping I/O is not tailor-made for persistent memory region, it does not take advantages of persistent memory region like Libpmem does; Comparing Libpmem to DEV-DAX I/O, the DEV-DAX statistically turns out to be slightly faster than Libpmem, since under Libpmem I/O engine, the persistent memory region is mounted with a file system, adding extra layer of penalty and does not make use of the byte-addressability of persistent memory; Comparing Pmemblk to the Libpmem, the results of Pmemblk library I/O is significantly slower, which is also expected since the it does block I/O which can be significantly slower than I/O by bytes that has finer granularity.

The same workload is also tested within the container, and It turns out that there is little difference between containerized persistent memory usage vs. out of container results, and test results fluctuation caused by memory bandwidth occupancy from the test machine environment is large enough to hide it. Thus, we can drop the assumption that the extra layer of the virtual file system mapping in the container causes trivial performance penalty, and we are safe to pursue persistent memory containerization.

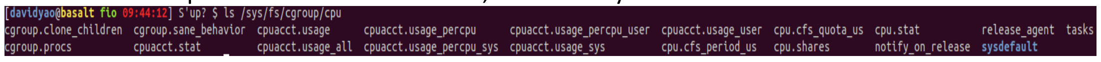
b. Cgroup Throttling

For the scenario of this project, we need to throttle the BPS and I/O weights. Before that, we need to mount the Cgroups onto the paths we want by specifying it in the /etc/cgconfig.conf file (see appendix). It takes effects after the Cgroup client daemon restarts.

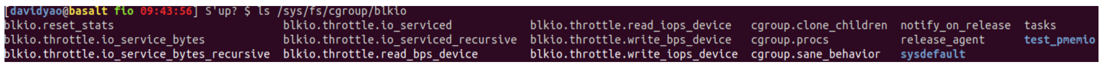
Then we create our own control group by "cgcreate". What we want to control needs to be specified by giving a subsystem in which the created control group will reside.

```
cgcreate -t uid:gid -a uid:gid -g subsystems:path
```

We can see the cgroup and parameter files after the creation. Different cgroups in different subsystems will see different parameter files. In our case, BLKIO subsystem is controlled.

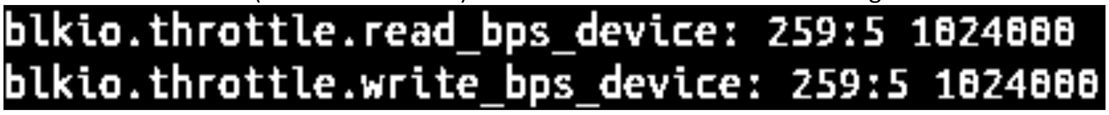


```
[davidyao@basalt fio 09:44:12] S'up? $ ls /sys/fs/cgroup/cpu
cgroup.clone_children  cgroup.sane_behavior  cpuacct.usage      cpuacct.usage_percpu      cpuacct.usage_percpu_user  cpuacct.usage_user  cpu.cfs_quota_us  cpu.stat           release_agent  tasks
cgroup.procs           cpuacct.stat          cpuacct.usage_all  cpuacct.usage_percpu_sys  cpuacct.usage_sys          cpu.cfs_period_us   cpu.shares        notify_on_release  sysdefault
```



```
[davidyao@basalt fio 09:43:56] S'up? $ ls /sys/fs/cgroup/blkio
blkio.reset_stats                      blkio.throttle.io_serviced            blkio.throttle.read_iops_device   cgroup.clone_children  notify_on_release  tasks
blkio.throttle.io_service_bytes        blkio.throttle.io_serviced_recursive  blkio.throttle.write_bps_device   cgroup.procs           release_agent      test_pmemio
blkio.throttle.io_service_bytes_recursive  blkio.throttle.read_bps_device    blkio.throttle.write_iops_device  cgroup.sane_behavior   sysdefault
```

To throttle the BPS, we need to specify which device to throttle, and how many bytes per second we want it to I/O. We write such rules into the corresponding parameter files. The device is identified by Linux Allocated Device (the "259:5" below) number which can be checked through "ls -al".

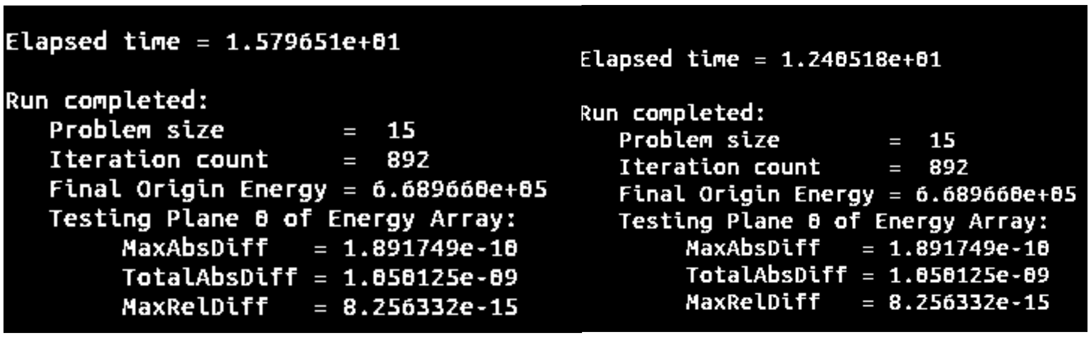


```
blkio.throttle.read_bps_device: 259:5 1024000
blkio.throttle.write_bps_device: 259:5 1024000
```

The result doesn't turn out to be satisfactory. Cgroups do not take effects unless there is high contention on the resources being isolated, whenever there are still free resources left, Cgroups (or the kernel itself) won't try to throttle the block I/O. Many efforts have been tried to push the memory bandwidth of tested machine to its limit, including firing-up to 8 FIO threads simultaneously, but first, the memory bandwidth is deteriorating linearly, it's hard to tell whether it's Cgroup taking effects or not; second, there are only limited cores on a test node, a higher number of simultaneous I/O threads might keep deteriorating block I/O of each thread, but the contention on the CPU clock cycles is further introduced, making it harder to decide whether Cgroups is controlling the I/O.

c. LULESH Checkpointing

The first implementation of both Libpmem checkpointing and disk checkpointing didn't turn out to be satisfactory: the Libpmem turns out to be slower than disk read / write. Even just to regard the physical level, I/O to a persistent memory region emulated on a DRAM region should be significantly faster than I/O onto block devices.

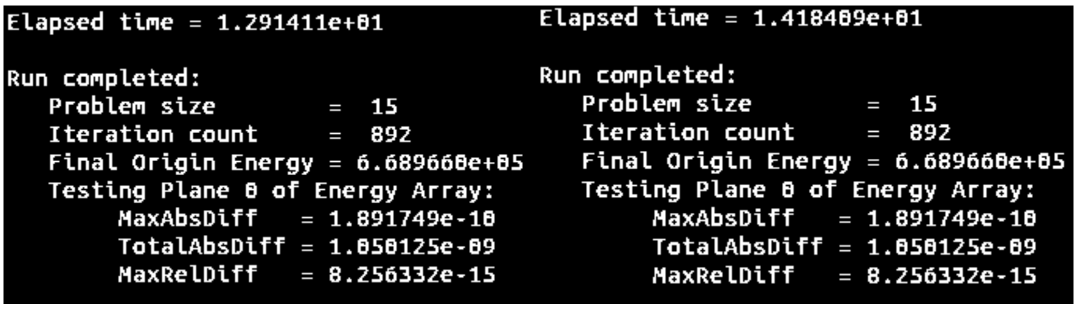


```
Elapsed time = 1.579651e+01

Run completed:
   Problem size         =  15
   Iteration count      =  892
   Final Origin Energy  = 6.689660e+05
   Testing Plane 0 of Energy Array:
        MaxAbsDiff   = 1.891749e-10
        TotalAbsDiff = 1.050125e-09
        MaxRelDiff   = 8.256332e-15
```

```
Elapsed time = 1.240518e+01

Run completed:
   Problem size         =  15
   Iteration count      =  892
   Final Origin Energy  = 6.689660e+05
   Testing Plane 0 of Energy Array:
        MaxAbsDiff   = 1.891749e-10
        TotalAbsDiff = 1.050125e-09
        MaxRelDiff   = 8.256332e-15
```

The first Libpmem version (left) is significantly slower than writing back to disk (right)

To reduce the amount of work to debug this issue, the checkpointing workload was reduced for only one variable. The first implementation code snippets for both versions are shown in the appendix. The disk checkpointing is fairly simple: for each checkpoint of each variable, open the checkpointed image, overwrite the data, flush the modification after writing a data point, and close the file. The Libpmem checkpointing, however, after each data point modification, A pmem_persist() is invoked to sync the change. According to the documentation, a pmem_persist() is actually equivalent to two functions: pmem_flush() that flushes changes out of cache, and pmem_drain() that drains modification into the persistent storage, which will take significantly more time. Therefore, comparing to the disk checkpointing, the extra latency might come from the draining process. This is confirmed by decomposing the pmem_persist() into pmem_flush() and pmem_drain(), keeping the flush in the loop and drain out of the loop. The modified implementation code snippets are shown in the appendix, and the final results are shown below. More runs with more problem sizes are tested, showing the same results.



```
Elapsed time = 1.291411e+01

Run completed:
   Problem size         =  15
   Iteration count      =  892
   Final Origin Energy = 6.689660e+05
   Testing Plane 0 of Energy Array:
        MaxAbsDiff    = 1.891749e-10
        TotalAbsDiff = 1.050125e-09
        MaxRelDiff    = 8.256332e-15
```

```
Elapsed time = 1.418489e+01

Run completed:
   Problem size         =  15
   Iteration count      =  892
   Final Origin Energy = 6.689660e+05
   Testing Plane 0 of Energy Array:
        MaxAbsDiff    = 1.891749e-10
        TotalAbsDiff = 1.050125e-09
        MaxRelDiff    = 8.256332e-15
```

The second Libpmem version (left) is faster than writing back to disk (right)

6. Conclusion / Future Work

Persistent memory has demonstrated its strength in streamlining the data flowing between CPU and persistent storage devices, this project has partially proven the benefit to adapt persistent memory and its programming scheme into future HPC system/application designs. But in terms of isolation and software security and when such non-volatile memory stands alone as a new intermediate storage layer, there is still more integration work to be done.

In the future, it would be great to integrate persistent memory device into Cgroup subsystems so that Linux kernel would handles the persistent memory as a more flexible resource, rather than a special device that needs to be partitioned and managed by a separate device driver and utilities such as NDCTL and its low-level kernel modules.

HPC applications in the future can definitely accommodate persistent memory programming to greatly reduce execution time and save computation resources. Specifically since most HPC applications are highly partitionable/parallelizable, the HPC applications can perform more fine-grained checkpointing at different partitioned spaces on different sets of variables, and tuning the granularity can balance the tradeoff between the amount of rollback to perform when recovering the progress and checkpoint time cost, and with persistent memory supports faster checkpointing, we can pursue a higher level of fault-tolerance.

7. Reference

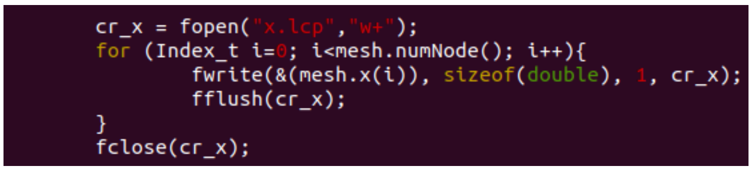
[1] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System software for persistent memory. *In Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 15:1–15:15.

[2] Giles, E. R. (2016, December). *Container-based virtualization for byte-addressable NVM data storage.* In 2016 IEEE International Conference on Big Data (Big Data) (pp. 2754-2763). IEEE.

[3] https://pmem.io/pmdk/

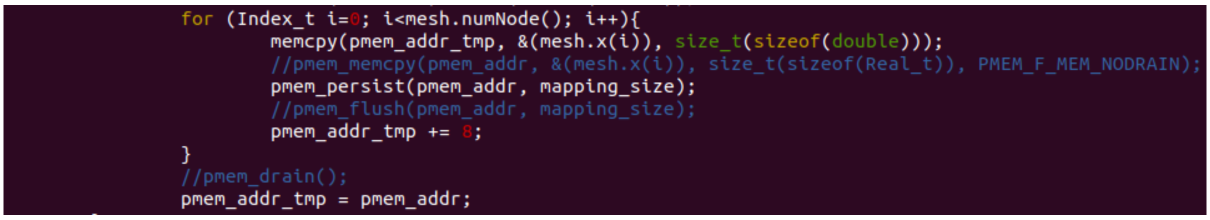
[4] https://docs.pmem.io/persistent-memory/

8. Appendix



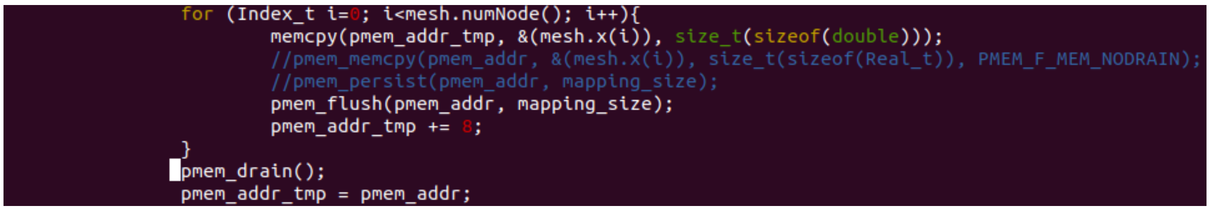
```
cr_x = fopen("x.lcp","w+");
for (Index_t i=0; i<mesh.numNode(); i++){
        fwrite(&(mesh.x(i)), sizeof(double), 1, cr_x);
        fflush(cr_x);
}
fclose(cr_x);
```

Disk checkpointing



```
for (Index_t i=0; i<mesh.numNode(); i++){
        memcpy(pmem_addr_tmp, &(mesh.x(i)), size_t(sizeof(double)));
        //pmem_memcpy(pmem_addr, &(mesh.x(i)), size_t(sizeof(Real_t)), PMEM_F_MEM_NODRAIN);
        pmem_persist(pmem_addr, mapping_size);
        //pmem_flush(pmem_addr, mapping_size);
        pmem_addr_tmp += 8;
}
//pmem_drain();
pmem_addr_tmp = pmem_addr;
```

Libpmem checkpointing (first attempt)



```
for (Index_t i=0; i<mesh.numNode(); i++){
        memcpy(pmem_addr_tmp, &(mesh.x(i)), size_t(sizeof(double)));
        //pmem_memcpy(pmem_addr, &(mesh.x(i)), size_t(sizeof(Real_t)), PMEM_F_MEM_NODRAIN);
        //pmem_persist(pmem_addr, mapping_size);
        pmem_flush(pmem_addr, mapping_size);
        pmem_addr_tmp += 8;
}
pmem_drain();
pmem_addr_tmp = pmem_addr;
```

Libpmem checkpointing (second attempt)