CS490 - Autumn 2022

Implementation & Review of Persistent Memory based Applications

(R & D Report)

Submitted By: Raja Gond (190050096)

Under the guidance of
Prof. Purushottam Kulkarni & Prof. Umesh Bellur

Department of Computer Science and Engineering
Indian Institute of Technology Bombay

2022-2023

Acknowledgements

I am grateful to Prof. Purushottam Kulkarni & Prof. Umesh Bellur for giving me my first taste of research. I would also like to thank them for their guidance and encouragement during the course of my project.

I would like to thank Rahul for helping me configure the optane machine.

Place: Mumbai, India [Raja Gond]

Abstract

Non-volatile memory (NVM) or persistent memory (PMEM) is a new type of memory module that is capable of data persistence. Pmem provides a unique combination of affordable larger capacity. With Innovative Technology offering distinctive operating modes it adapts to your needs across workloads. It provides near-DRAM data access latency and can be directly accessed in bytes through the memory bus using CPU load. Intel Optane DC has become the first commercially available PMEM device in 2019.

Intel's Optane Persistent Memory modules support two modes: volatile and byte-addressable persistent memory. DRAM acts as a cache for the most frequently accessed data, while Optane persistent memory provides large memory capacity. Cache management operations are handled by Intel's integrated memory controller.

We explored persistent memory in this project and did a survey of various persistent memory libraries and write the reader-writer program to understand the working of these libraries. To know how persistent memory affects large applications(such as Redis) performance we conduct performance evaluations of both pmem-redis and redis-4.0.0 using the redis benchmark.

Contents

Acknowledgements **i**

Abstract **ii**

List of Figures **iv**

1 Life Before Persistent Memory **1**

2 Introduction to Persistent Memory(PMem) **2**

2.1 Introduction - Data can't be lost, if it is on pmem 2

2.2 Persistent Memory Use Cases[2] 2

2.3 Some new programming concerns introduced by persistent memory[8] 3

3 Persistent Memory Programming *&* PMDK Libraries **4**

3.1 Operating System Support for Persistent Memory 4

3.1.1 Configure Intel's Optane for direct access in Linux 6

3.2 Persistent Memory Development Kit(PMDK) Libraries[6, 10] 6

4 Applications of Persistent Memory: PMem-Redis *&* reader-writer **8**

4.1 Reader-Writer 8

4.2 Enabling Redis for Persistent Memory: Pmem-Redis 11

4.2.1 Redis 11

4.2.2 Pmem-Redis 11

4.2.3 TieredMemDB 11

4.3 Analysis of Pmem-Redis *&* Redis-4.0.0 12

5 Discussions *&* Future Work **16**

List of Figures

1.1 Memory Storage Hierarchy[8] 1

2.1 Memory-Storage Hierarchy with Persistent Memory Tier[8] 2

3.1 Storage and volatile memory in the operating system[10] 4
3.2 Persistent Memory As Block Storage [10] 5
3.3 Persistent Memory-Aware File Systems [10] 5

4.1 Redis-benchmark:- SET: Throughput vs. number of parallel clients 12
4.2 Redis-benchmark:- GET: Throughput vs. number of parallel clients 13
4.3 Redis-benchmark:- SET: Throughput vs. payload size 13
4.4 Redis-benchmark:- GET: Throughput vs. payload size 14
4.5 Redis-benchmark:- SET: Throughput vs. pipeline 14
4.6 Redis-benchmark:- GET: Throughput vs. pipeline 15

1. Life Before Persistent Memory

Over the last few decades, computer systems have implemented the memory-storage hierarchy shown in Figure 1.1. Successive generations of technologies have iterated on the number, size, and speed of caches to ensure the CPUs have access to the most frequently used data. CPU speeds have continued to get faster, adding more cores and threads with each new CPU generation as they try to maintain Moore's Law. The capacity, price, and speed of volatile DRAM and non-volatile storage such as NAND SSDs or Hard Disk Drives have not kept pace and quickly become a bottleneck for system and application performance.[8]

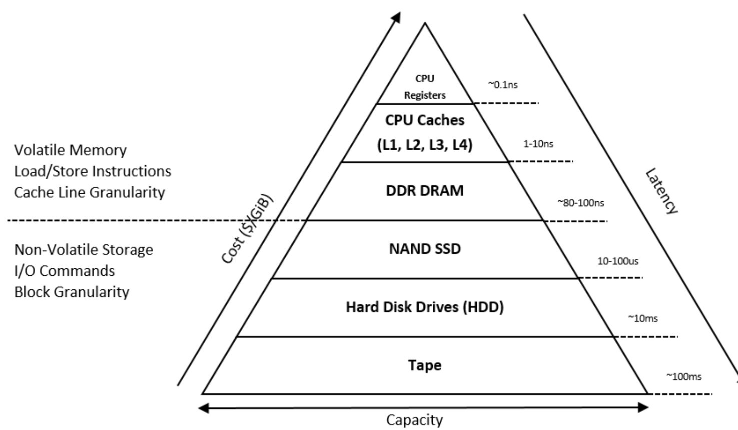

Figure 1.1: Memory Storage Hierarchy[8]

Dynamic Random-Access Memory (`DRAM`) is a common type of random access memory (RAM) that is used in personal computers (PCs), workstations and servers. Dynamic comes from the fact that DRAM must be refreshed after a fixed time-quanta since its cells lose their state over time. DRAM is located close to a computer's processor and enables faster access to data than storage media such as hard disk drives and solid state drives but has a long access time as compared to CPU Cache and Registers.

`DRAM` is very durable and fast but not able to retain data(i.e does not provide power failure tolerance). What does this mean? If we cut off the power source(or DRAM loses its power source for any reason) the data it was processing is lost and needs to be retrieved from disk storage. This inability to retain data is known as volatility.[1]

DRAM is byte addressable, and provides low latency in accessing data(as compared to pmem-more on pmem in the later section, SSDs and Hard Disks) but it is volatile and has low capacity.

In the contrast, solid-state drives and Hard-Disk Drives(HDD) is a "non-volatile" storage drives, which means they can retain the stored data even when no power is supplied but they have low read/write speeds and have high latency when dealing with heavy workloads. Operating systems (OS) tell the HDD to read and write data as needed by programs. The speed that the drive reads and writes this data is solely dependent on the drive itself.[1]

SSD and HDD are not byte addressable and have high latency as compared to DRAM but they are cheap and non-volatile.

2. Introduction to Persistent Memory(PMem)

2.1 Introduction - Data can't be lost, if it is on pmem

Non-volatile memory (NVM) or persistent memory (PMEM) is a new kind of memory device that provides both near-DRAM data access latency and data persistence capability. Different from block-based devices, PMEM can be directly accessed in bytes through the memory bus using CPU load and store instruction without using block-based interfaces. Due to its high density, low cost, and near-zero standby power cost PMEM devices have been considered as a promising part of the next-generation memory hierarchy. Among all the persistent memory solutions, the Intel Optane DC Persistent Memory (or Optane DC for short) has become the first commercially available PMEM device on the market in 2019.[9]

As the NVM device (e.g., Optane DC) becomes available, software developers start to consider porting their applications to persistent memory. However, to make it work efficiently, they need to have an accurate expectation of their applications performance on PMEM, as well as know how to re-design their applications to achieve the best performance.[9]

Persistent Memory (PMEM), also referred to as Non-Volatile Memory (NVM), or Storage Class Memory (SCM), provides a new entry in the memory-storage hierarchy shown in Figure 2.1 that fills the performance/capacity gap.[8]

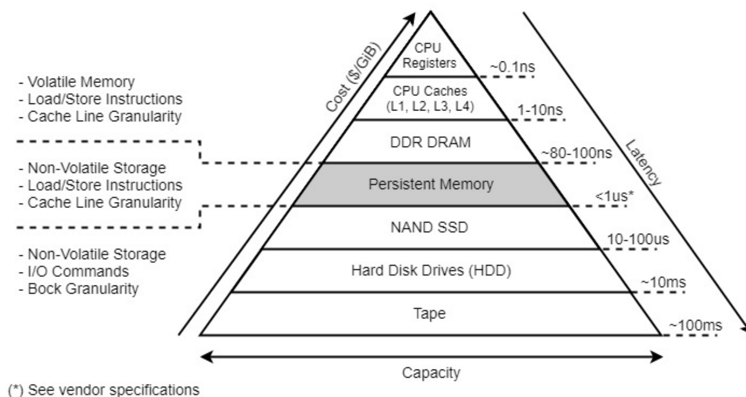

Figure 2.1: Memory-Storage Hierarchy with Persistent Memory Tier[8]

With persistent memory, applications have a new tier available for data placement. In addition to the memory and storage tiers, the persistent memory tier offers greater capacity than DRAM and significantly faster performance than storage. Applications can access persistent memory like they do with traditional memory, eliminating the need to page blocks of data back and forth between memory and storage.[8]

2.2 Persistent Memory Use Cases[2]

- **Cloud Service Providers Cost Reduction**

  The key metric for a cloud service operator is how many VMs they can deliver to their customers, and at what cost. The size of the memory on the servers becomes the bottleneck of how many VMs they can allocate per server, limiting how low their price per VM can go.

PMem(such as Intel's Optane) is cheap compared to DRAM and delivers a larger amount of memory per server, allocating a larger number of VMs, therefore lowering the cost per VM and increasing the competitiveness of cloud service providers.

- **Reliability with Large Memory Data Bases**

  Financial customers such as stock exchanges, banks, and mutual funds use a lot of memory databases and in-memory applications.

  - `Fraud detection`

    Financial Institutions have very large amounts of data of transactions, customer records etc. Accessing data from disks and then performing various analytics (such as illegal transactions detection) on them is slow and time-consuming. Persistent Memory can help in improving speed.

2.3 Some new programming concerns introduced by persistent memory[8]

It did not apply to traditional volatile memory

- `Data Persistence`

  Stores are not guaranteed to be persistent until flushed. Although this is also true for the decades-old memory-mapped file APIs (like mmap() and msync() on Linux), many programmers have not dealt with the need to flush to persistence for memory. CPUs have out-of-order CPU execution and cache access/flushing. This means if two values are stored by the application, the order in which they become persistent may not be the order that the application wrote them.

- `Data Consistency`

  8-byte stores are powerfail atomic on the x86 architecture – if a powerfail happens during an aligned, 8-byte store to PMEM, either the old 8-bytes or the new 8-bytes (not a combination of the two) will be found in that location after reboot. Anything larger than 8-bytes on x86 is not powerfail atomic.

- `Memory Leaks`

  Memory leaks to persistent storage are persistent. Rebooting the server doesn't change the on-device contents.

- `Byte Level Access`

  Application developers can read and write at the byte level according to the application requirements. The read/writes no longer need to be aligned or equal to storage block boundaries, eg: 512byte, 4KiB, or 8KiB. The storage doesn't need to read an entire block to modify a few bytes, to then write that entire block back to persistent storage. Applications are free to read/write as much or as little as required. This improves performance and reduces memory footprint overheads.

- `Error Handling`

  Applications may need to detect and handle hardware errors directly. Since applications have direct access to the persistent memory media, any errors will be returned back to the application as memory errors.

3. Persistent Memory Programming & PMDK Libraries

3.1 Operating System Support for Persistent Memory

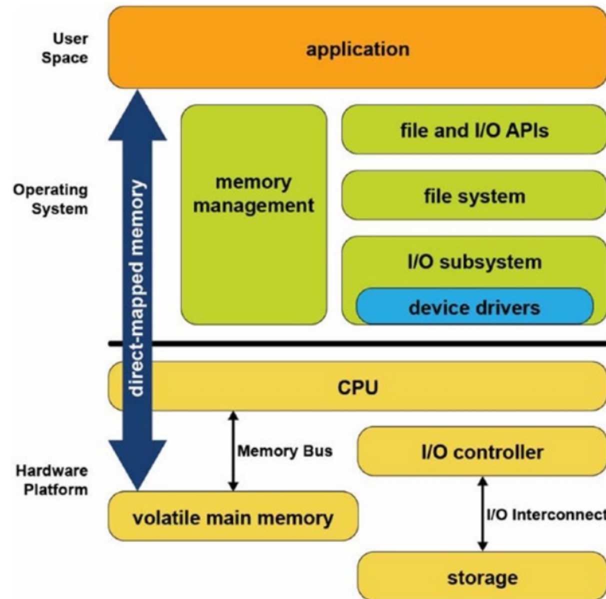

Figure 3.1: Storage and volatile memory in the operating system[10]

As shown in figure 3.1, the volatile main memory is attached directly to the CPU through a memory bus. The operating system manages the mapping of memory regions directly into the application's visible memory address space. Storage, which usually operates at speeds much slower than the CPU, is attached through an I/O controller. The operating system handles access to the storage through device driver modules loaded into the operating system's I/O subsystem.

Since persistent memory can be accessed directly by applications and can persist data in place, it allows operating systems to support a new programming model that combines the performance of memory while persisting data like a non-volatile storage device.[10]

- `Persistent Memory As Block Storage` The first operating system extension for persistent memory is the ability to detect the existence of persistent memory modules and load a device driver into the operating system's I/O subsystem as shown in Figure 3.2. This NVDIMM driver serves two important functions. First, it provides an interface for management and system administrator utilities to configure and monitor the state of the persistent memory hardware. Second, it functions similarly to the storage device drivers.[10]

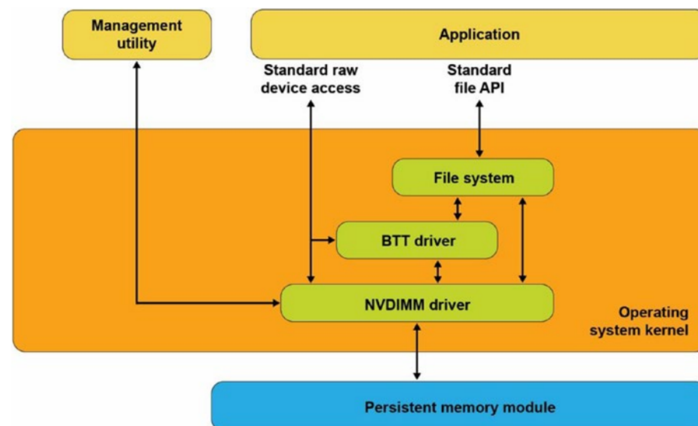

Figure 3.2: Persistent Memory As Block Storage [10]

- `Persistent Memory-Aware File Systems`

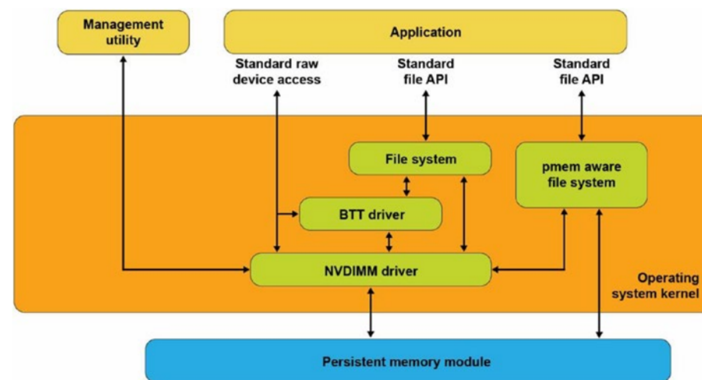

Figure 3.3: Persistent Memory-Aware File Systems [10]

Make the operating system is to file system aware of and be optimized for persistent memory. File systems that have been extended for persistent memory include Linux ext4 and XFS, and Microsoft Windows NTFS. As shown in Figure 3.3, these file systems can either use the block driver in the I/O subsystem or bypass the I/O subsystem to directly use persistent memory as byte-addressable load/store memory as the fastest and shortest path to data stored in persistent memory.

- `Memory-Mapped Files`

  When memory mapping a file, the operating system adds a range to the application's virtual address space which corresponds to a range of the file, paging file data into physical memory as required. This allows an application to access and modify file data as byte-addressable in-memory data structures. This has the potential to improve performance and simplify application development, especially for applications that make frequent, small updates to file data.

- Persistent Memory Direct Access (DAX)

The persistent memory direct access feature in operating systems, referred to as DAX in Linux and Windows, uses the memory-mapped file interfaces but takes advantage of persistent memory's native ability to both store data and to be used as memory.[10]

3.1.1 Configure Intel's Optane for direct access in Linux

Intel's Optane Persistent Memory modules support two modes: *Memory Mode*, which is volatile, and *App Direct mode*, which is byte-addressable persistent memory.

In Memory Mode, the DRAM acts as a cache for the most frequently accessed data, while Intel's Optane persistent memory provides large memory capacity. Cache management operations are handled by Intel's Xeon Scalable processor's integrated memory controller.

In App Direct Mode, applications and the Operating System are explicitly aware there are two types of direct load/store memory in the platform and can direct which type of data read or write is suitable for DRAM or Intel's Optane persistent memory.[5]

Displaying persistent memory physical devices and regions on Linux

```
~$ sudo ipmctl show -dimm
```

Displaying persistent memory physical devices, regions, and namespaces on Linux

```
~$ ndctl list -DRN
```

Locating persistent memory on Linux

```
~$ df -h /dev/pmem*
```

The Show Topology command displays both the PMem and DDR4 DRAM DIMMs.

```
~$ sudo ipmctl show -topology
```

The Show Memory Resources command displays how the DDR/PMem capacity is allocated at the system level.

```
~$ sudo ipmctl show -memoryresources
```

Create memory allocation goal

```
~$ sudo ipmctl create -goal PersistentMemoryType=AppDirect -y
```

Create namespaces DAX support

```
~$ ndctl create-namespace --mode fsdax --region region0
```

To get the DAX functionality, mount the file system with the dax mount option.

```
~$ sudo mount -o dax /dev/pmem0 /mnt/pmem/
```

3.2 Persistent Memory Development Kit(PMDK) Libraries[6, 10]

The Persistent Memory Development Kit (PMDK) is a growing collection of libraries and tools. Tuned and validated on both Linux and Windows, the libraries build on the DAX (Direct Access) feature of those operating systems which allows applications to access persistent memory as memory-mapped files, as described in the SNIA NVM Programming Model.

The PMDK offers two library categories:

1. Volatile libraries are for use cases that only wish to exploit the capacity of persistent memory. Volatile libraries are typically simpler to use because they can fall back to dynamic random-access memory (DRAM) when persistent memory is not available. This provides a

more straightforward implementation. Depending on the workload, they may also have lower overall overhead compared to similar persistent libraries because they do not need to ensure consistency of data in the presence of failures. for example,

   - `libmemkind`

     The memkind library, called libmemkind, is a user-extensible heap manager built on top of jemalloc. The memkind library provides familiar malloc() and free() semantics.

   - `libvmemcache`

     libvmemcache is an embeddable and lightweight in-memory caching solution that takes full advantage of large-capacity memory, such as persistent memory with direct memory access (DAX), through memory mapping in an efficient and scalable way.

2. Persistent libraries are for use in software that wishes to implement fail-safe persistent memory algorithms. Persistent libraries help applications maintain data structure consistency in the presence of failures.

   - `libpmem`

     libpmem is a low-level C library that provides a basic abstraction over the primitives exposed by the operating system. It automatically detects features available in the platform and chooses the right durability semantics and memory transfer (memcpy()) methods optimized for persistent memory. Most applications will need at least parts of this library.

   - `libpmemobj`

     libpmemobj is a C library that provides a transactional object store, with a manual dynamic memory allocator, transactions, and general facilities for persistent memory programming. libpmemobj turns a persistent memory file into a flexible object store supporting transactions, memory management, locking, lists, and a number of other features.

   - `libpmemobj-cpp`

     also known as libpmemobj++, is a C++ header-only library that uses the metaprogramming features of C++ to provide a simpler, less error-prone interface to libpmemobj. It is a C++ idiomatic bindings for libpmemobj. Implementing containers from scratch will be a long effort, order of their implementation is quite important. libpmemobj++ has its first container - an array. It is included in the pmem::obj::experimental namespace. *libpmemobj++ persistent pointer* wraps around a type and provides the implementation of operator*, operator-> and operator[]. (persistent_ptr<>)

   - `libpmemkv`

     libpmemkv is a generic embedded local key-value store optimized for persistent memory.

   - `libpmemlog`

     C library that implements a persistent memory append-only log file with power fail-safe operations.

   - `libpmemblk`

     C library for managing fixed-size arrays of blocks. It provides fail-safe interfaces to update the blocks through buffer-based functions.

4. Applications of Persistent Memory: PMem-Redis & reader-writer

4.1 Reader-Writer

Reader-Writer using array-based approach

```c
/*
* Resources:  Programming Persistent Memory
                - A Comprehensive Guide for Developers by Steve Scargall
* Full code can be found [here]
(https://github.com/rajagond/pmem_cxl/blob/main/reader_writer/read_write.c)
*/
#define MAX_BUF_LEN 50
struct my_root {
    int rp; //for read
    int wp; //for write
    int buf[MAX_BUF_LEN]; //buffer
};
int main(int argc, char *argv[]) {
    char path_obj[50];
    scanf("%s",path_obj);//reading string
    PMEMobjpool *pop= pmemobj_open(path_obj, POBJ_LAYOUT_NAME(rweg));
    if (pop == NULL) {
        pop = pmemobj_create(path_obj, POBJ_LAYOUT_NAME(rweg), (100000000), 0666);
       /*
        * Initialisation of Fibonacci Series
        * I have used transactions to read or write an element.
      */
    }
    PMEMoid root = pmemobj_root(pop, sizeof(struct my_root));
    struct my_root *rootp = pmemobj_direct(root);
    int i = rootp->wp;
    for (; ; i++)
    {
        sleep(1);
        TX_BEGIN(pop) {/* Add the previous 2 numbers in the series and store it */
            TOID(struct my_root) root = POBJ_ROOT(pop, struct my_root);
            TX_ADD(root); // adding full root to the transaction
            printf("%d ", D_RO(root)->buf[D_RO(root)->rp]);
            fflush(stdout);
            D_RW(root)->rp = D_RO(root)->rp + 1;
            int curr = D_RO(root)->buf[D_RO(root)->wp - 1];
            int prev = D_RO(root)->buf[D_RO(root)->wp - 2];
            D_RW(root)->buf[D_RO(root)->wp] = curr + prev;
            D_RW(root)->wp = D_RO(root)->wp + 1;
        } TX_END
    }
    pmemobj_close(pop);
    return 0;
}
```

Reader-Writer using pointer-based approach

```
/*
* Resources:  Programming Persistent Memory
                - A Comprehensive Guide for Developers by Steve Scargall
* Full code can be found [here]
(https://github.com/rajagond/pmem_cxl/blob/main/reader_writer/read_write_linked_list.cc)
* This program will print the Lucas series
*/
#define CREATE_MODE_RW (S_IWUSR | S_IRUSR)
inline int file_exists (const std::string& name) {
    return access( name.c_str(), F_OK );
}
using namespace std;                    namespace pobj = pmem::obj;
#define LAYOUT "demo"
struct Node {
    pobj::p<int> prev_data;             pobj::p<int> data;
    pobj::persistent_ptr<Node> next;
};
class root {
    public:
        pobj::persistent_ptr<Node> wp = nullptr;
        pobj::persistent_ptr<Node> rp = nullptr;
};
int main() {
    string path = "/mnt/pmem/po/poolfile";
    pobj::pool<root> pop;    pobj::persistent_ptr<root> proot;
    try {
                if (file_exists(path) != 0) {//File not exist
                pop = pobj::pool<root>::create(path, LAYOUT, 100000000,
                      CREATE_MODE_RW);
                proot = pop.root();
                /*
                * Initialisation using transactions
                */
                } else {//File exists
                        pop = pobj::pool<root>::open(path, LAYOUT);
                    proot = pop.root();
                }
        } catch (const pmem::pool_error &e) {
                std::cerr << "Exception: " << e.what() << std::endl; return 1;
```

Reader-Writer using pointer-based approach(Cont.)

```
        } catch (const pmem::transaction_error &e) {
                std::cerr << "Exception: " << e.what() << std::endl; return 1;
    }
    while(1){//run continuously
        sleep(1); //sleeping for one second
        pobj::transaction::run(pop, [&] { //write
            auto n = pobj::make_persistent<Node>();
            if( proot->wp != nullptr){
                    n->prev_data = proot->wp->data;
                n->data = proot->wp->data + proot->wp->prev_data;
            }
            else{
                n->data = 1;
            }
            n->next = nullptr;
            if (proot->rp == nullptr && proot->wp == nullptr) {
                proot->rp = proot->wp = n;
            } else {
                proot->wp->next = n;
                proot->wp = n;
            }
            std::cout << "\t\t\t Write: " << proot->wp->data << std::endl;
        });
        pobj::transaction::run(pop, [&] {// read
            if (proot->rp == nullptr)
                pobj::transaction::abort(EINVAL);
            uint64_t ret = proot->rp->data;
            auto n = proot->rp->next;
            pobj::delete_persistent<Node>(proot->rp);
            std::cout << "\t\t\t\t\t Read: " << ret << std::endl;
            proot->rp = n;
            if (proot->rp == nullptr)
                proot->wp = nullptr;
        });
```

The above reader-writer program is implemented with the help of pmdk libraries(more specifically, the array-based uses libpmemobj and the pointer-based program uses libpmemobj++). TX or transaction used in both programs make sure that update happens in one go. In pointer-based program, `make_persistent` creates a persistent Node and `delete_persistence` deletes the persistent Node. `pmemobj_create` and `pobj::pool<root>::create` creates the object file with given permission and size passed as arguments. `file_exists` is a custom `C` function that return `true` if file exists.

Many APIs such as `TX_ADD`, `D_RO`, `D_RW`, `TOID` are provided by the pmdk libraries for correct implementation of the program on persistent memory. Details explanation about these APIs can be found in this [10] book.

4.2 Enabling Redis for Persistent Memory: Pmem-Redis

4.2.1 Redis

- The open-source, in-memory data store implements a distributed, in-memory key value used by millions of developers as a database, cache, streaming engine, and message broker. External programs talk to Redis using a TCP socket and a Redis-specific protocol.[4]
- Redis is a data structure server. At its core, Redis provides a collection of native data types that help you solve a wide variety of problems, from caching to queuing to event processing(An event is anything that happens at a clearly defined time and that can be specifically recorded. Event processing is the process that takes events or streams of events, analyzes them and takes automatic action.)
- `Redis-Persistence` How Redis writes data to disk (append-only files, snapshots, etc.)?
  - RDB (Redis Database): The RDB persistence performs point-in-time snapshots of your dataset at specified intervals.
  - AOF (Append Only File): The AOF persistence logs every write operation received by the server, which will be played again at server startup, reconstructing the original dataset.
  - No persistence: we can disable persistence completely,
  - RDB + AOF: It is possible to combine both AOF and RDB in the same instance.

4.2.2 Pmem-Redis

What is Pmem-Redis[3]? Pmem-Redis is one redis version that support Intel DCPMM(Data Center Persistent Memory) based on open source redis-4.0.0. It benefits the redis's performance by taking advantage of DCPMM competitive performance and persistence. Basically Pmem-Redis covers many aspects that related to DCPMM usage:

- Five typical data structures optimization including: String, List, Hash, Set, Zset.
- DCPMM copy-on-write
- Redis LRU for DCPMM
- Redis defragmentation support for DCPMM
- Pointer-based redis AOF
- Persistent ring buffer

4.2.3 TieredMemDB

TieredMemDB is a Redis branch that fully uses the advantages of DRAM and Intel Optane Persistent Memory (PMEM). It is fully compatible with Redis and supports all its structures and features. The main idea is to use a large PMEM capacity to store user data and DRAM speed for latency-sensitive structures. They also offer the possibility of defining the DRAM to PMEM ratio, which will be

automatically monitored and maintained by the application. This allows us to fully adapt the utilization of the memory to your hardware configuration.[7]

The source code of TieredMemDB can be found in this GitHub repository.

`Features:`

- Different types of memory - use both DRAM and PMEM in one application.
- Memory allocation policy - define how each memory should be used. You can specify DRAM / PMEM ratio that will be monitored and maintained by the application.
- Redis compatible - supports all features and structures of Redis.
- Configurable - use simple parameters to configure new features.

4.3 Analysis of Pmem-Redis & Redis-4.0.0

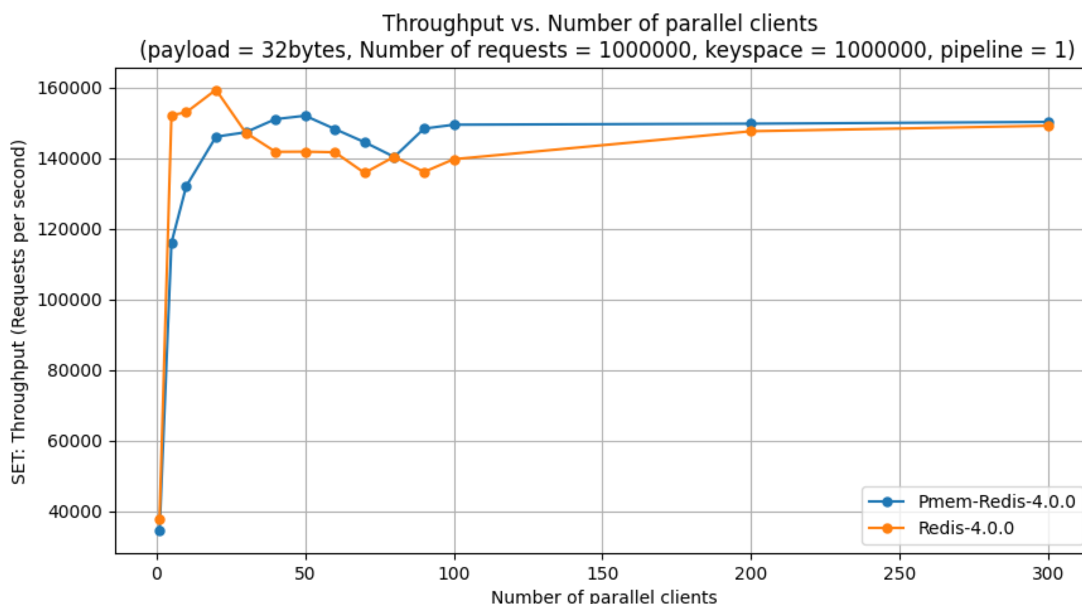

Figure 4.1: Redis-benchmark:- SET: Throughput vs. number of parallel clients

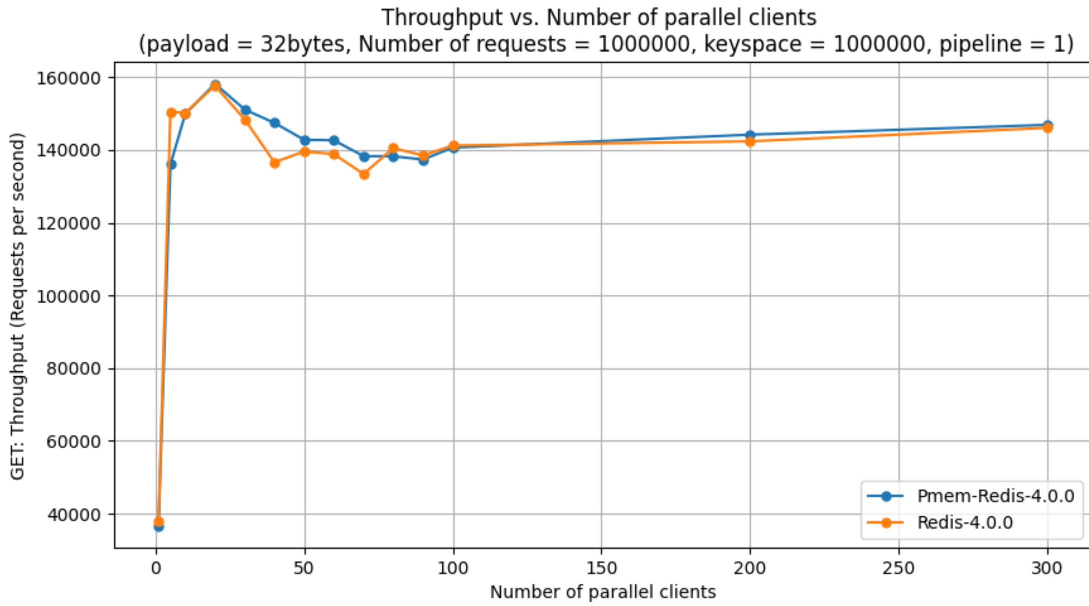

Figure 4.2: Redis-benchmark:- GET: Throughput vs. number of parallel clients

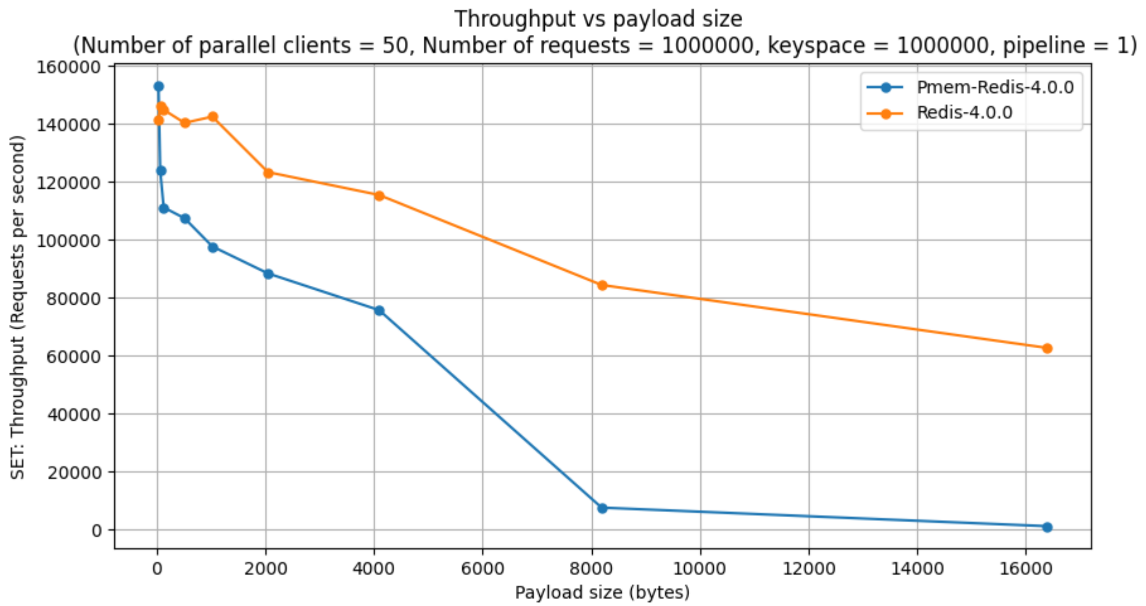

Figure 4.3: Redis-benchmark:- SET: Throughput vs. payload size

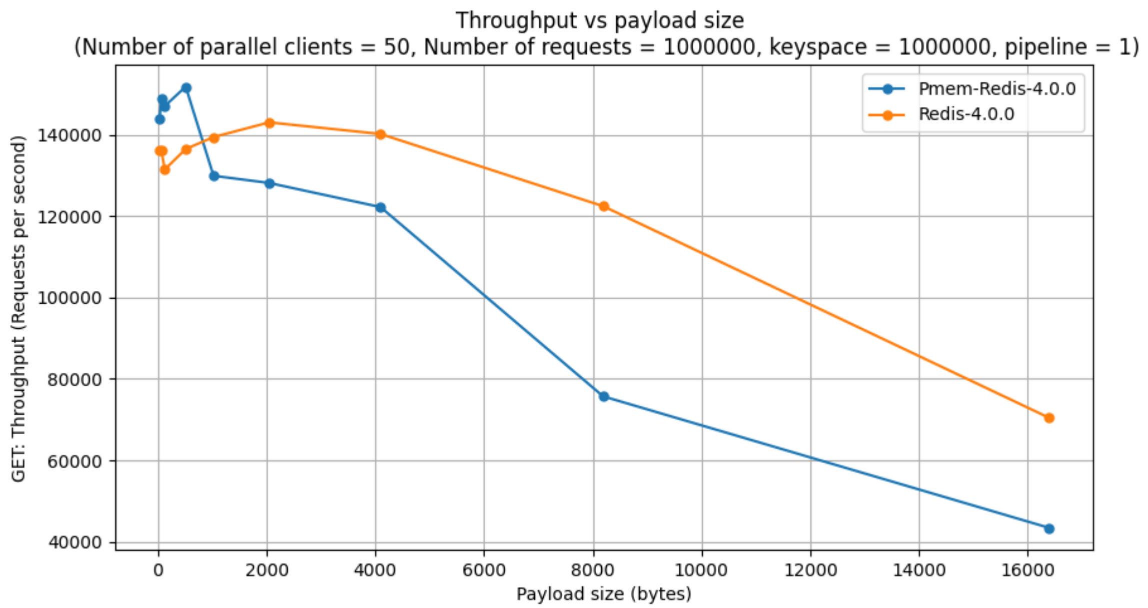

Figure 4.4: Redis-benchmark:- GET: Throughput vs. payload size

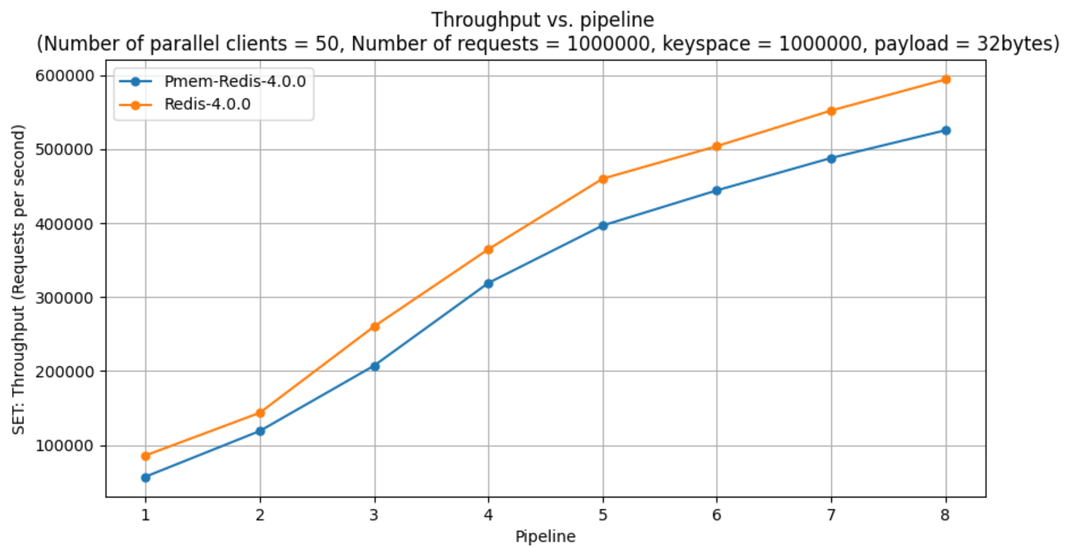

Figure 4.5: Redis-benchmark:- SET: Throughput vs. pipeline

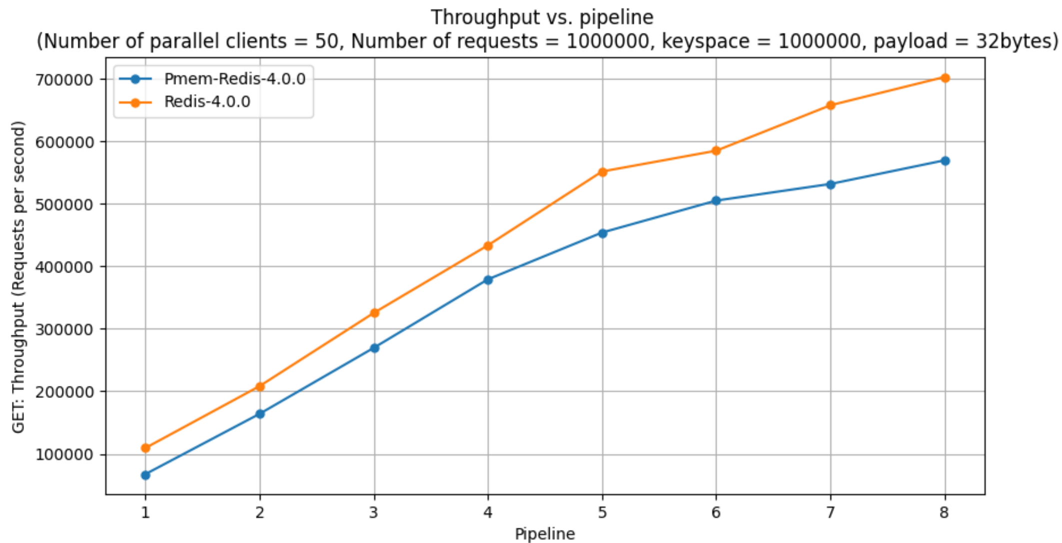

Figure 4.6: Redis-benchmark:- GET: Throughput vs. pipeline

I have used redis-benchmark and have run it on Pmem-Redis-4.0.0 and Redis-4.0.0 to record the data. On pmem-redis I have used to below command to start the redis server

```
sudo ./src/redis-server --nvm-maxcapacity 100 --nvm-dir /mnt/pmem --nvm-threshold 0
```

The above command will start a Redis server with 100GB capacity and will use the /mnt/pmem directory as the NVM pool. The –nvm-threshold 0 option will disable the NVM threshold, which means that the server will not evict any data from NVM to DRAM.

and `redis-server` to start server on Redis-4.0.0.

Both have used **jemalloc** to allocate memory(more specifically pmem-redis use jemalloc-4.1.0).

- Figure 4.1 *&* Figure 4.2, both Pmem-redis-4.0.0 and redis-4.0.0 are showing almost same behaviour. Fixing payload size(data size) and the number of requests with no pipelining both are giving almost the same throughput if we vary the number of parallel clients.
- Figure 4.3 *&* Figure 4.4, these are the interesting plots. As we increase the payload size(data size) pmem-redis-4.0.0 goes down as compared to redis-4.0.0. For the SET case, pmem-redis-4.0.0 is poor for large payloads.
- Figure 4.5 *&* Figure 4.6, both Pmem-redis-4.0.0 and redis-4.0.0 are showing almost same behaviour(redis-4.0.0 have somewhat better throughput). Fixing payload size(data size), the number of requests and the number of clients, we can observe that redis-4.0.0 perform better if we increase pipelining.

5. Discussions & Future Work

Since I was not familiar with persistent memory and redis. So, I spent most of the time building familiarity with the background material, configuring and setting up different applications (like pmem-redis, TieredmemDB) on the optane machine and trying out the basic persistent memory programming.

I plan to continue working on the project and below is the possible trajectory on which I will be working in the coming months.

- Rearchitect any popular middleware that uses persistence via disks today to take advantage of persistent memory(PMem) where it exists.
- Exploring the specification of Compute Express Link(CXL). Compute Express Link (CXL) is an open standard for high-speed central processing unit (CPU)-to-device and CPU-to-memory connections, designed for high-performance data centre computers.

Bibliography

[1] Hard disk drive (hdd) vs. solid state drive (ssd): What's the difference? https://www.ibm.com/cloud/blog/hard-disk-drive-vs-solid-state-drive. [Online; accessed 22-November-2022].

[2] Persistent memory use cases. https://www.intel.com/content/www/us/en/government/podcasts/embracing-digital-transformation-episode51.html. [Online; accessed 22-November-2022].

[3] Pmem-redis. https://github.com/pmem/pmem-redis. [Online; accessed 22-November-2022].

[4] Redis. https://redis.io/docs/about/. [Online; accessed 22-November-2022].

[5] Intel. Optane memory start guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/optane-memory/intel-optane-memory-quick-start.pdf. [Online; accessed 22-November-2022].

[6] Intel. Persistent memory development kit. https://pmem.io/pmdk/. [Online; accessed 22-November-2022].

[7] Intel. Tieredmemdb. https://github.com/TieredMemDB/TieredMemDB. [Online; accessed 22-November-2022].

[8] Intel. Persistent memory documentation. https://docs.pmem.io/persistent-memory/getting-started-guide/introduction, 2020. [Online; accessed 22-November-2022].

[9] Abdullah Al Raqibul Islam, Anirudh Narayanan, Christopher York, and Dong Dai. A performance study of optane persistent memory: From indexing data structures' perspective. https://webpages.charlotte.edu/ddai/papers/MSST20_Pmem_CameraReady.pdf.

[10] Steve Scargall. *Introduction to Persistent Memory Programming.* Apress, Berkeley, CA, 2020.