2021

Evaluating PMO Sync Implementation for Persistent Memory Object

Faishal Wahiduddin
University of Central Florida

EVALUATING PMO SYNC IMPLEMENTATION FOR PERSISTENT MEMORY OBJECT

by

FAISHAL WAHIDUDDIN
B.S. Padjadjaran University, 2016

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2021

Major Professor: Yan Solihin

ABSTRACT

Persistent Memory, in the form of byte-addressable Non-Volatile Memories (NVMs), provides a low-cost and high-capacity main memory, and provides the ability to store and retain data even when the system is powered off, along with improved performance over traditional storage. Persistent Memory Direct Access (DAX) enables applications to perform byte-addressable operations such as load and store. Filesystem-DAX can store persistent data in NVMs with system call overheads. In order to reduce filesystem overheads, this study utilizes Persistent Memory Object (PMO) as an abstraction for persistent data containers on Non-Volatile Memory (NVM). Persisting data in Persistent Memory Object requires that the system supports atomic transaction and crash consistency.

This study proposes PMO Sync implementation to achieve atomic transaction and crash consistency in Persistent Memory Object. We evaluate three different instructions support: CLFLUSH, CLFLUSHOPT, CLWB; two persistency models, and two atomic update mechanisms: undo logging and shadow copy as a basis of PMO Sync and introduce PMO Sync Table to keep track of modified data. We abstract memory operation, persistency model, and atomic update mechanism as PMO Sync call to persist updates to NVM. PMO Sync reduces performance overheads by avoiding flush and fence operations for every write and providing atomic transaction and gives flexibility to programmer to persist changes. Experimental results show that the PMO Sync implementation reduces execution time and the number of writes compared to undo logging and shadow copy mechanisms.

I dedicate this thesis work for the sake of Allah. All the praises and thanks be to Allah who is the Lord of the universe. A special thanks to my parents who have support me all the time. My beloved brothers. My friends who encourage and support me. All the people in my life.

ACKNOWLEDGMENTS

In the Name of Allah, the Most Merciful, the Most Compassionate all praise be to Allah, the Lord of the worlds; and prayers and peace be upon Mohamed His servant and messenger.

First and foremost, I acknowledge my thankfulness to Allah, the Ever-Magnificent; the Ever-Thankful, for His help and bless. I would like to express my heartfelt gratitude to my advisor and committee chairman, Dr. Yan Solihin, for his excellent support and countless hours invested in ensuring the success of this research.

A special thanks to Dr. Damian Dechev, Dr. Paul Gazzillo for being generous with their time in agreeing to serve on my thesis committee, I would also like to acknowledge UCF College of Engineering and Computer Science for providing an environment that promotes growth and excellence.

I would like to thank all ARPERS members, especially Ardhi Yudha, my best friend, for sharing their knowledge while working on the computer lab. Furthermore, I would like to thank Lisa Putri Kusuma for her support in completing my studies.

I am grateful to Dr. Setiawan Hadi, Dr. Juli Rejito, and Dr. Atje Setiawan Abdullah, who were my professors during my bachelor's degree and encouraged me to pursue a master's degree.

Last but not least, AMINEF (American Indonesian Exchange Foundation) and the Fulbright Program, who have supported me throughout my time in Indonesia and until I completed this study. I sincerely thank them.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Persistent Memory has been shown to complement or replace DRAM as main memory fabric. Persistent Memory supports non-volatility, byte addressability and access through load and store instructions. In addition to advantages of persistent Memory, compared to DRAM, it has limited write endurance, writes are slow, and expensive [3]. The benefit of persistent Memory is hosting data persistently alongside with performance of DRAM and also supports recoverability after failure[4].

Persistent Memory that is available today, for example Intel Optane DC Persistent Memory, can be placed on DIMM slot along with DRAM. Persistent Memory can be accessed using file system storage interface and standard file API similar to managing data on solid-state disk. Another way to utilize persistent Memory is by using memory instructions. Intel has been already providing memory instructions support for storing data into persistent memory [5].

Currently, utilizing persistent Memory to provide persistent data needs involvement of file system. File system requires program to serialize the data from a file to work with data and has higher overhead from system call. Recent works already propose some approaches to optimize file system with persistent Memory such as PMFS [6] that provides access to persistent Memory with memory-mapped I/O, NOVA [7] as log-structured file system, and HMVFS [8] with versioning-mmap. Instead of using file system, we can utilize persistent Memory by avoiding file system and directly using memory-mapped IO. Mnemosyne [9] proposes lightweight transaction mechanism in which we can perform load and store without involvement of file system.

To be able to use persistent Memory directly without file system, we need to use persistent data structure. Persistent Memory Object (PMO) is a persistent data structure that can be accessed across process lifetime. PMO may combine some features of file system such as naming and dura-

bility and need to have metadata. Although some studies have been done on Persistent Memory Object [10] [11], there is no persistency mechanism provided to implement on Persistent Memory Object.

To achieve atomic durability, Persistent Memory Object needs to have a mechanism that guarantees durability and crash consistency. In this research, we explore some design options for Persistent Memory Object persistency mechanism and propose PMO Sync call as synchronization point to persist all changes into NVM.

CHAPTER 2: BACKGROUND & RELATED WORKS

In this section, we present the development of persistent memory and several mechanisms that are mentioned during the explanation of proposed PMO Sync design later in this thesis. This should give the idea of the other solutions to the basic problems in persistent memory. We start by looking at the detail of persistent memory programming model therefore we see the basic idea of application in the persistent memory. Moreover, we show other mechanisms for comparison with our proposed design.

Non-volatile Memory

Over the past few decades, computer systems on memory technology have been advanced and provided many advantages mainly in increased efficiency, capacity, density, and reduced cost. As critical component of computers, main memory can store sets of accessed data and supply it to the processors. Computer storage has been faced two types of storage media between volatile and non-volatile storage. Persistent data stored on non-volatile media such as flash drive or disk and can be buffered in fast byte-addressable DRAM, however, there is a possibility of losing data during a crash or a power failure when data is still on DRAM. On the other hand, Non-Volatile Memory (NVM) has better durability in which it can retain data without power, but it has lower access latencies than volatile memory [12]. Eventually, the existing memory systems are either sacrificing durability, performance, or consistency. Modern applications are raising challenges and complexities in designing high-performance, and energy-efficient memory. Nowadays devices are accessing and processing infinite data in which the data sets can be gigabytes, terabytes, or even larger in size. Memory technologies such as SRAM and DRAM, are facing challenges in scalability.

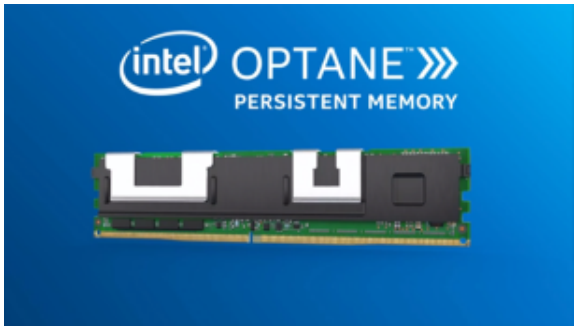

Figure 2.1: Optane DC Persistent Memory [1]

NVMs can replace traditional memory technologies, such as DRAM, with promising characteristics [13]. NVMs outperform DRAM in terms of power consumption. NVMs do not require refresh operations and have near-zero standby power due to their nonvolatility. Furthermore, NVMs can support byte-addressable, fast data accesses as well as nonvolatile data storage. As a result, they combine memory (fast) and storage (data retention without power supply) capabilities in a single device. With the ability to accommodate fast accesses to permanent data storage in a unified nonvolatile memory, this feature has the potential to disrupt the current two-level memory/storage stack. Table 2.1 shows the comparison of Persistent Memory with HDD, SSD, and DRAM.

Table 2.1: Comparison memory and storage

Aspects	**HDD**	**NAND SSD**	**Persistent Memory**	**DRAM**
Latency	High	High	Low	Low
Cost	Low	Low	Medium	High
Byte-Addressable	No	No	Yes	Yes
Volatile	Non-Volatile	Non-Volatile	Non-Volatile	Volatile

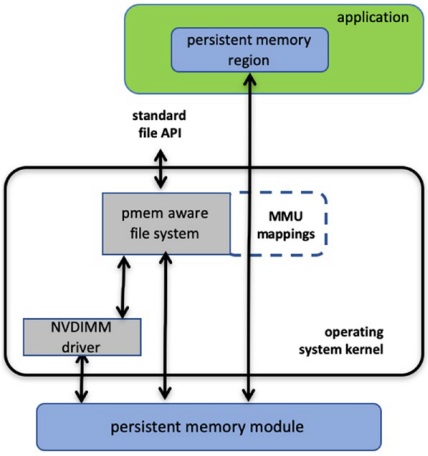

Figure 2.2: Persistent Memory Access [2]

According to the persistent memory programming model [14] and the related DAX feature, persistent memory files can be mapped into memory using standard calls such as mmap(). As a result, the application has direct load/store access to the persistence, as indicated by the far-right arrow in Figure 2.2. Once these mappings are established, and after any initial minor page faults that may be required to create the mappings in the MMU, this provides the shortest possible code path to persistence, allowing applications to perform direct loads and stores on persistent media with no kernel involvement. For media access, no interrupts, no context switching, and no kernel code are required.

On persistent memory, there are two direct access modes [15]. The first is Filesystem-DAX mode, which is the default mode of a namespace when create-namespace is specified with no options.

It creates a block device (/dev/pmemX[.Y]) that supports Linux file systems' DAX capabilities (xfs and ext4 to date). The page cache is removed from the I/O path by DAX, allowing mmap(2) to create direct mappings to persistent memory media. Workloads and working sets that would otherwise exceed the capacity of the page cache can now be scaled up to the capacity of persistent memory thanks to the DAX capability. DAX may not benefit from workloads that fit in page cache or perform bulk data transfers. The second mode is Device-DAX, which provides mmap(2) DAX mapping capabilities similar to Filesystem-DAX. This mode, however, emits a single character device file (/dev/daxX.Y) rather than a block-device that can support a DAX-enabled file system.

Persistency Model

Memory persistency is a the concept of memory consistency to additionally order persists and provides explanation to persist order related to failures. A persistency model allows software to identify persist-order constraints necessarily for recovery-correctness while enabling concurrency among other persists [16]. Failure states are defined as a recovery observer which is able to read all of persistent memory at the moment of failures. In recovery observer view, ordering is constraint for correct recovery, which also becomes constraints on memory and persist operation. As we focus on exploring the semantics, persistency models are divided into two classes, strict and relaxed persistency.

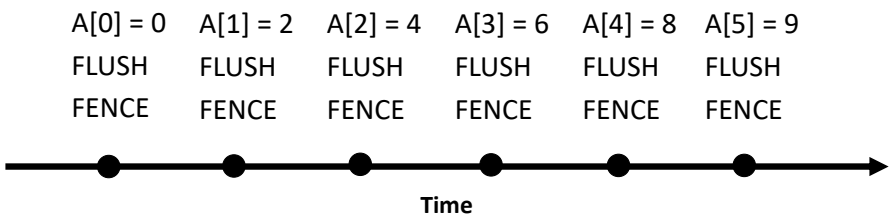

Figure 2.3: Strict Persistency

Strict persistency facilitates mechanisms to explain persist behavior using preexistent memory consistency models. By using the current consistency model, strict persistency unifies memory persistency to the memory consistency model to determine persist ordering. The recovery observer is involved in the memory consistency model specifically as if it were an additional processor, in which persistent memory order is indistinguishable from volatile memory order. Thus, observing volatile memory order can conclude any store ordering which implies persist ordering constraints. The persist order must correlate the order to which the stores are performed in a particular execution. The order of persists using strict persistency shows in Figure 2.3

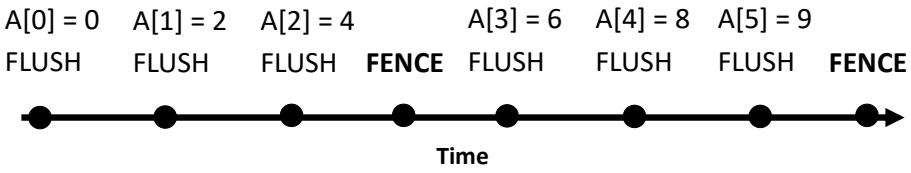

Figure 2.4: Relaxed Persistency

Figure 2.4 illustrates persist ordering using relaxed persistency. Relaxed persistency separates memory consistency and persistency models. It can broaden the persist ordering constraints dealing with the memory consistency model. In short, relaxed persistency and consistency enable the visible loads and stores to reorder among processors, also enables order persists to deviate from the order of stores. One of the implementations of relaxed persistency is Lazy Persistency [17], an application persistency mechanism that enables caches to gradually send dirty blocks to the NVMM via natural evictions. By reducing NVM write constraints, relaxed persistency models increase system throughput by 30 times [16].

Nonetheless, pre-existing memory models used in the strict persistency are recurrently inappropriate for persistence performance. The high latency of NVM persists proposes that relaxed persistency models are preferable.

Persistent Memory Transaction

A transaction [2] is typically defined as the combination of multiple operations into a single atomic operation. A persistent region is a contiguous portion of an application's address space that is populated by a memory-mapped file hosted in persistent memory. The region is accessible via the load/store interface and contains a heap as well as a user-initialized root pointer. Persistent Memory Transactions consisted of load and store instructions on persistent regions.

When a transaction is complete, it is only considered durable when data persisted on persistent media. If the system goes out of power or the system crashes after the transaction and all changes already durable, the transaction is still complete. Typically, this means that the changes must be flushed from the CPU caches to make them durable. This can be accomplished through the use of standard APIs such as the Linux msync() function or platform-specific instructions such as Intel's CLWB. Implementing an atomic update mechanism can result in a more durable transaction.

Atomic update mechanism

Undo Log

Undo log is a mechanism to ensure durable transaction by performing a backup using a log for the data changes. Undo log utilize log to store data changes before performing modification as a backup. Figure 2.5 shows how to create a log and utilize log bit. Log bit is used as an indicator of whether the log of data is already set up. When the log already persists and the log bit changed into 1, then we can modify and persist the data change into persistent memory. After the modification persisted, we unset the log bit by changing the value to 0. Finally, we completed durable transaction using Undo Log. The data and log shows in Figure 2.6.

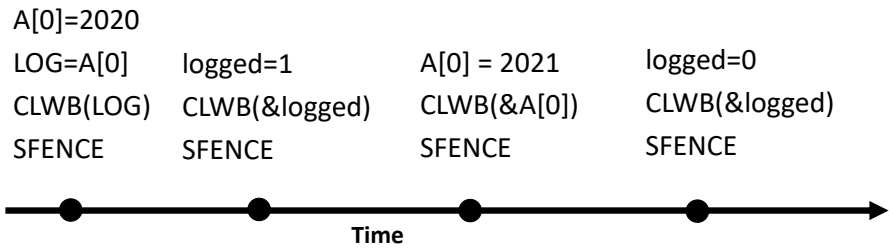

Figure 2.5: Durable Undo Log

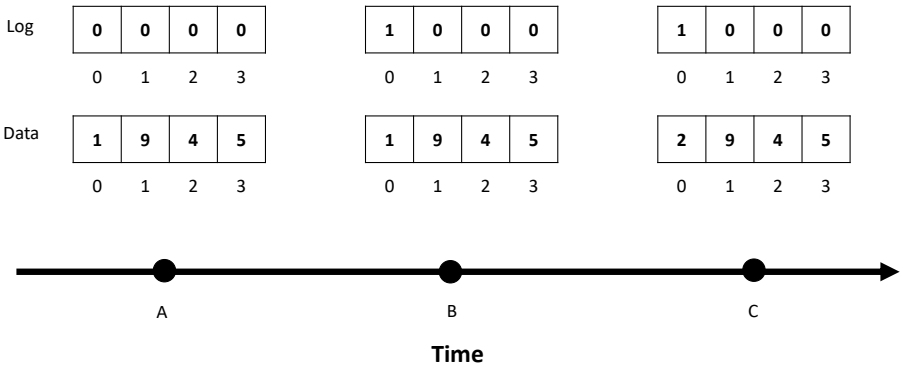

Figure 2.6: Undo Log Flow

An Undo log durable transaction consists of four persist states [18]. The first one when we persist the log of the modified data. The second is when we set the log bit. The third is during we persist the modification. Last we unset the log bit and persist it. Each persists state need to be performed in order to make sure the transaction durable which means we cannot set the log bit before the log already persist. For setting the log bit we need to wait until the log persisted in persistent memory and it is also applied to other persist states that need to wait for the previous operation to be completed. This condition will incur waiting time between operations in a transaction.

```
1  for(i=0; i<N; i++){
2      // Create Log
3      Log = createLog(&C[i]);
4      CLFLUSH(&Log);
5      // Set Log Bit into 1
6      logged = 1;
7      CLFLUSH(&logged);
8      // Persist Change
9      C[i] = rand();
10     CLFLUSH(&C[i]);
11     // Unset Log Bit
12     logged = 0;
13     CLFLUSH(&logged);
14 }
```

Figure 2.7: Undo Log Programming Model

The figure shows a simple example of undo log implementation. Line 3-4 creates the log and persists it by flushing the cache line and a durable barrier that ensures the log is persisted before continuing to the next operation. Some work that utilizes Log are ATOM [19] is a hardware log manager that uses undo logging to move the logging operation out of the critical path. Proteus [20] adds two new instructions: log-load and log-flush. Log-load creates a log entry by reading the original data, and log-flush stores the log entry to the log.

Shadow Copy

By utilizing virtual memory indirection, shadow copying can eliminate data copying. Shadow copying differs from undo log in that there is no separate log and data area. When a transaction completes, it keeps the old data in place, writes the new data to any other free physical page, and atomically updates the persistent virtual-to-physical mappings. However, one disadvantage of using shadow copying is the increased overhead caused by the gap between the large page size and the number of bytes modified, which could be as few as a few bytes. Simply downsizing the page size would result in an unacceptably large increase in virtual-to-physical mapping and page table walk overhead. Figure 3.10 shows how shadow copy performed.

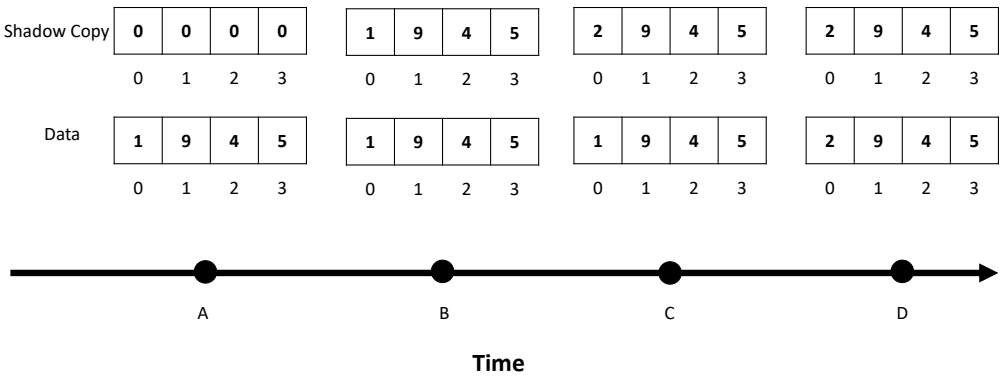

Figure 2.8: Shadow Copy

Some works already used shadow copy for nonvolatile memory such as OSP [21] that employs shadow paging with cache line level mappings, enabling the implementation of durable transac-

tions so no need to log real data for each update and without causing extra persisting overhead. MOD [22] data structures use shadow paging to reduce internal ordering in update operations, with only one sfence per failure-atomic operation. Shadow Sub-Paging (SSP) [23] is a version of shadow paging that is optimized for persistent memory. Each active virtual page is linked with a second physical page when SSP is used to perform atomic updates.

Persistent Memory Object

Persistent Memory Object is a novel data structure for persistent memory that utilizes file system features and file benefits such as pointer address, load and store [3]. PMO performs byte-addressable load and store which do not require a file system and can maximize the performance of persistent memory. The comparison of PMO shows in Figure 2.2.

Table 2.2: PMO Comparison

Aspect	**Data Structures**	**Files**	**Persistent Memory Object**
Access	Load and Store	Sys Calls	Load and Store
Durability	No	Yes	Yes
Granularity	byte	block	byte
Namespace	No	Yes	Yes
Metadata	Vm	FS	POT

Current PMO primitives are attach and detach shows in Figure 2.9. PMO data is only accessible when the program request it by attaching it to the process address space, and it is inaccessible at other times by detaching it from the address space. Detaching data from address space provides

extremely strong protection because the adversary cannot exploit even virtual memory (VM) implementation flaws [10]. Attaching as well as detaching data requires data managed and maintained by the Operating System (OS), which includes namespace and permission mechanisms. Memory-mapped files, which are mapped to and unmapped from the process address space, as an object that fits the attach and detach model, allowing the process to directly access file data. The programmer can probe, read, or write data in the PMO while it is attached.

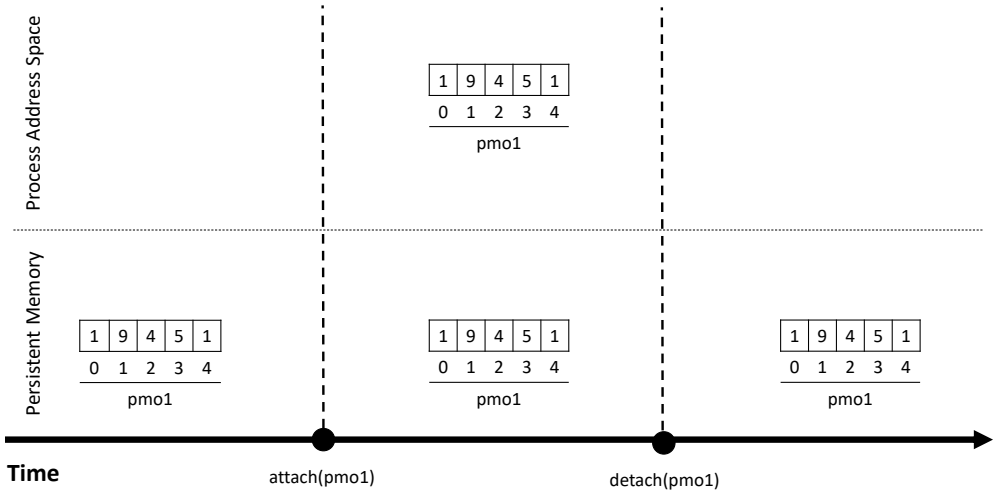

Figure 2.9: PMO attach and detach

Performing writes in Persistent Memory Object to a page and then persist it by cache flush does not guarantee the store operation persisted on persistent memory. While performing persist, there is a probability that a system failure occurs during that time. Therefore, these results could be different from the data that want to be persisted and considered corrupted. The issue is shown in Figure 2.10 when flushing changes to persistent memory and suddenly crash occurs.

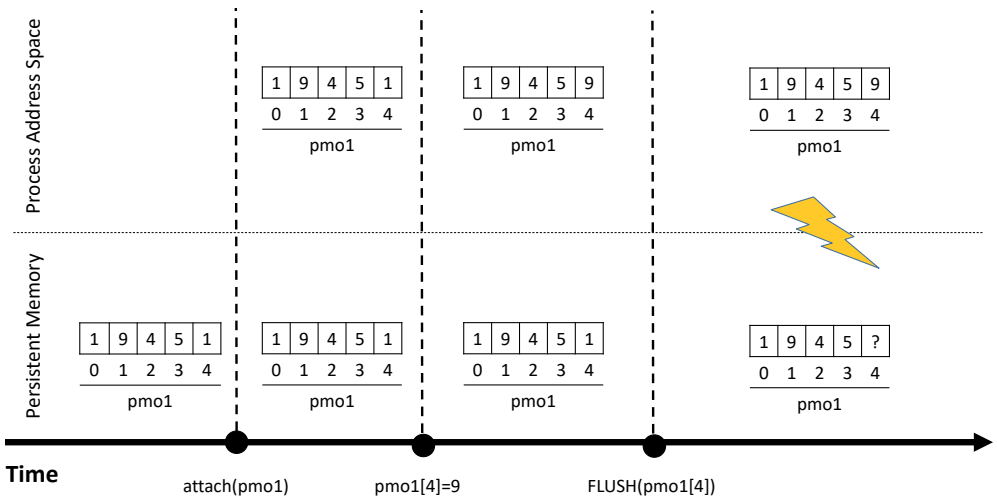

Figure 2.10: PMO Write Crash

This issue leads to the need to provide a persistency mechanism for Persistent Memory Objects. The store to PMO has to be made durable together or nothing at all.

Additional Tools

FIO [24] is a benchmark that launches several threads or processes to perform a specific type of I/O action specified by the user. FIO accepts some global parameters, each of which is inherited by the thread unless other parameters overriding that setting is provided. FIO is typically used to generate a job file that corresponds to the I/O load being simulated. FIO employs pthread mutexes for signaling and locking, but some platforms do not support process shared pthread mutexes. As a result, only threads are supported on such platforms.

The daxio utility is used to perform I/O on Device DAX devices or to zero them. Data is transferred using a memory-mapped device because the standard I/O APIs (read/write) cannot be used with Device DAX. The daxio can dump Device DAX data to a file, restore data from a backup copy, move/copy data to another device, or wipe data from a device [25].

Valgrind [26] is a memory-related bug detection tool that has been adapted for persistent memory. Valgrind is an instrumentation framework for creating dynamic analysis tools. Some Valgrind tools can detect and profile many memory management and threading bugs automatically. We can also use Valgrind to create new tools. To run pmemcheck, we need a modified version of Valgrind that supports the new CLFLUSHOPT and CLWB flushing instructions. Valgrind's persistent memory version includes the pmemcheck tool and is available from https://github.com/pmem/valgrind.

CHAPTER 3: PMO SYNC DESIGN AND IMPLEMENTATION

In this section, we present the design and implementation of PMO Sync. We evaluate memory instruction, persistency model, and atomic update mechanism as the basis of the PMO Sync. We introduce a novel PMO Sync Table. Following that, we present the flow and implementation of PMO Sync.

Direct Access Mode

In order to create an efficient implementation of a Persistent Memory Object on Non-Volatile Memory, we must choose the most effective Direct Access Mode of Persistent Memory. We will use the FIO benchmark to measure the access speed of Filesystem-DAX and Device-DAX. Table 3.1 shows the FIO benchmark configuration.

Table 3.1: FIO Configuration

Option	Configuration
Workload	Sequential Read
Number of Jobs	1 Job
File size	1 GB
Block size	1 MB
Engine	dev-dax and libpmem

The benchmark employs a file size of 1 gigabyte and a block size of 1 megabyte. This configuration

employs two distinct engines based on direct access mode. Filesystem-DAX will be evaluated by the libpmem engine, while Device-DAX will be evaluated by the dev-dax engine. Each engine is limited to a single direct access mode and cannot be switched. The benchmark will be run ten times for each engine, and the average result will be calculated. The one with the best performance will be chosen as the direct access mode for Persistent Memory Object.

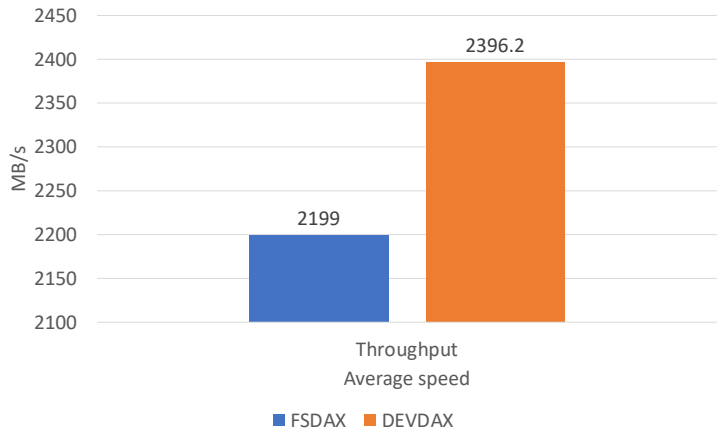

Figure 3.1: FIO Benchmark Result

The evaluation has been run and we can see the result in Figure 3.1 shown above that Device-DAX shows better throughput than Filesystem-DAX. Device-DAX throughput reach throughput around 2396 MB/s compared to Filesystem-DAX that has lower throughput at 2199MB/s. This result indicates that Device-DAX has around 11% better throughput. Based on the evaluation we can utilize Device-DAX as direct access mode for Persistent Memory Object.

Memory Instructions

The modification of data from Persistent Memory Object resides in the cache. We need to perform a flush for every dirty cache line to persist on persistent memory. Intel provides memory instruction that forces modified data to write back to memory. Three memory instructions will be evaluated to be used for PMO Sync: CLFLUSH, CLFLUSHOPT, and CLWB. Each memory instruction uses a similar flow and the same number of store and flush instructions for comparison.

```
1 // Looping for MAX number of writes
2 for (int i = 0; i < MAX; i++)
3 {
4     // Write data to array
5     ptr[i] = i * 2;
6     // Flushing array element
7     CLFLUSH(&ptr[i]);
8 }
```

Figure 3.2: CLFLUSH

CLFLUSH is a memory instruction that perform flushing cache line of modified data to persistent memory in order. In this evaluation, the program does not need to call SFENCE to serialize the order of store. We can perform modification to the array element and perform CLFLUSH to the address of dirty cache line. Due to this instruction serialize the store, the modified data that written back to memory will be in order.

```
// Looping for MAX number of writes
for (int i = 0; i < MAX; i++)
{
    // Write data to array
    ptr[i] = i * 2;
    // Flushing array element
    CLFLUSHOPT(&ptr[i]);
}
    // Store Fence
    SFENCE();
```

Figure 3.3: CLFLUSHOPT

The optimized version of CLFLUSH is CLFLUSHOPT that performs cache line flush without any serialization. The code shown on Figure 3.3 that performs store to array elements will be flushed to memory in any order. There is one SFENCE placed to make sure that all previous updates will persist to persistent memory.

```
// Looping for MAX number of writes
for (int i = 0; i < MAX; i++)
{
    // Write data to array
    ptr[i] = i * 2;
    // Cache Line Write Back array element
    CLWB(&ptr[i]);
}
    // Store Fence
    SFENCE();
```

Figure 3.4: CLWB

The snippet code is shown above as Figure 3.4 represents store program with memory instruction CLWB. Suppose that we want to modify the array element of i and perform write to that array element. The modification will be not written back to memory without any memory instruction. Therefore, the next line of code performs a flush using CLWB that will write back modified data to persistent memory.

All three memory instructions will be evaluated based on execution time. The evaluation will be run ten times and the average will be analyzed and used for PMO Sync abstraction. The direct access mode for this evaluation based on the best performance of direct access mode evaluation using FIO benchmark that is Device-DAX direct access mode.

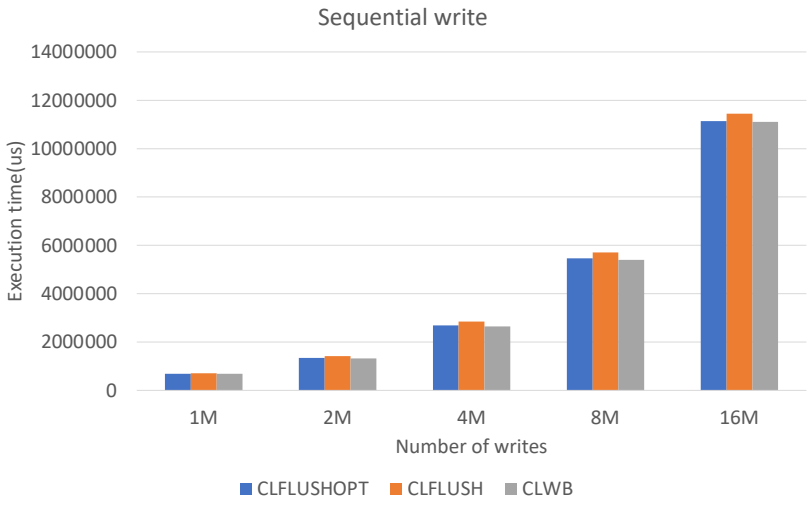

Figure 3.5: Memory Instructions Result

The comparison of memory instructions in terms of execution time is shown in Figure 3.5 above. Overall, it can be seen that the lowest execution time of memory instructions evaluation is CLWB,

whereas CLFLUSH is the highest among all different stores. Based on the results, we utilize CLWB as the basis of memory instruction for PMO Sync abstraction. This CLWB also provides benefits such as cache lines that are already flushed may remain valid.

Persistency Model

The memory persistency model of PMO Sync will guarantee the ordering of persisting modified data to NVM. In this case, we want to evaluate strict persistency and relaxed persistency. Strict persistency will guarantee the persist performed in order while relaxed persistency enables persist order different with store operation of the program.

```
1  // Looping for MAX number of writes
2  for (int i = 0; i < MAX; i++)
3  {
4      // Write data to array
5      ptr[i] = i * 2;
6      // Cache Line Write Back array element
7      CLWB(&ptr[i]);
8      // Store Fence
9      SFENCE();
10 }
```

Figure 3.6: Strict Persistency

The Figure 3.6 shows how strict persistency performing flush and fence. The code performs write operation to an array element and followed by cache line write back to flush modified data to persistent memory. The next code is SFENCE that has a role to guarantee previous changes persisted before continuing to the next code. This considers as strict persistency because it only has one

update and change and there is no reordering of persist operation.

```
// Looping for MAX number of writes
for (int i = 0; i < MAX_N; i++)
{
    // Write data to array
    ptr[i] = i * 2;
    // Cache Line Write Back array element
    CLWB(&ptr[i]);
    // Memory barrier every 1024 array element writes
    if (i % 1023 == 0)
    {
        // Store Fence
        SFENCE();
    }
}
```

Figure 3.7: Relaxed Persistency

The relaxed persistency model shown on snippet code of Figure 3.7 has similar write to an array element. The location of SFENCE in this persistency differs with strict persistency. Instead of performing SFENCE directly after updating the data, relaxed persistency will do several write operations and flushing cache and later perform SFENCE to make sure previous updates persist before continuing to the next code. This code of relaxed persistency allows the program to write until 1024 writes, and the persist operation is not necessarily in order from array element number one to the last which means the persist operation allows reordering of data persisted to NVM.

We evaluate two persistency models above based on the sequential write of N number of array elements. We analyze which persistency model that aptly fits to be applied as persistency model for PMO Sync implementation.

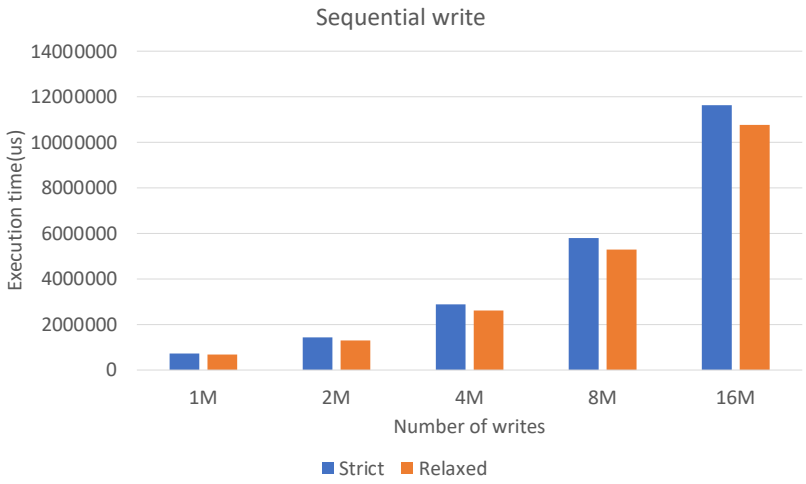

Figure 3.8: Persistency Result

Figure 3.8 shows the comparison of the execution time of two different persistency models. It is obvious that with a different number of stores, relaxed persistency achieves faster execution time than strict persistency. This result indicates that if we do relax persistency and allow reordering of some operation it will give speed up compared to directly persist every store operation. Based on this result we chose the relaxed persistency model for PMO Sync abstraction.

Atomic Update Mechanisms

PMO Sync combines multiple operations as one transaction. The challenge to perform transaction in persistent memory is the changes of multiple operations can be torn by system crashes or power failure between any instruction which leads to partial data persisted in NVM. We need to implement a mechanism that guarantees atomic update that is the all of the changes persisted in

NVM or nothing at all. There are two atomic update mechanisms that can be considered for PMO Sync. The first one is Undo Log that writes logging ahead before persisted changes and the second mechanism is shadow copy that performs out-of-place changes on shadow copy.

```
// Create Undo Log for array element
LOG_data[N] = ptr[i];
// Flushing Log
CLWB((void *)&LOG_data[i]);
// Memory barrier
SFENCE();

//Set logged bit
logged = 1;
// Flushing log bit to 1
CLWB((void *)&logged);
// Memory barrier
SFENCE();

// Write a new data to array element
ptr[i] = rand();
// Flushing modified data
CLWB((void *)&ptr[i]);
// Memory barrier
SFENCE();

//Unset logged bit
logged = 0;
// Flushing log bit to 0
CLWB((void *)&logged);
// Memory barrier
SFENCE();
```

Figure 3.9: Undo Log

Atomic update with Undo Log mechanism relies on Log as the backup of original data. To implement Undo Log for persistent memory, we need to make sure that Undo Log operation is performed in an order which leads to multiple flush and fence operation only for one store or writes changes ash shown in Figure 3.9. At the beginning of the update, the program needs to create backup data as Undo Log. The next operation is to set a logged bit as 1 that indicates the program starts to perform the modification. After the modification is completed, the operation followed by unset the log bit into zero that indicates all of the changes already completed and data persisted in NVM. There are four places that crashes could be happened during executing this program. If a crash happened after creating a log and before setting up the log bit, the data has not been modified yet and we can rerun the code. When the logged bit is already set to 1 or already reached the operation to modify data and the system is interrupted, the program needs to roll back the changes using log. If the crash happened after the logged bit was already unset to 0, the update was already completed, and no need to perform recovery.

```
// Create shadow copy of a 4KB page
memcpy((void *)&shadow[i], (void *)&ptr[i], 4096);
for (int k = 0; k < 64; k++)
{
    // Flushing shadow copy
    CLWB((void *)&shadow[k*16]);
}
// Memory barrier
SFENCE();

//Write a number to array element
shadow[i] = rand() ;
// Flushing new data to shadow copy
CLWB((void *)&shadow[i]);
// Memory barrier
SFENCE();


// Copy back shadow content to original page
memcpy((void *)&ptr[i], (void *)&shadow[i], 4096);
// Loop each cache line
for (int k = 0; k < 64; k++)
{
    // Flushing data to original page
    CLWB((void *)&ptr[k*16]);
}
// Memory barrier
SFENCE();
```

Figure 3.10: Shadow Copy

The Shadow copy atomic update mechanism utilizes a private copy of the data that the program

wants to modify is shown as snippet code in Figure 3.10. Instead of directly perform store operations to the data, this mechanism performs out-of-place write that creates a shadow copy of the data and updates the date on the shadow copy. When the program completes the modification, the modified data on shadow copy transfers back to original data. If a crash occurs during performing change on shadow copy, the original data remains untouched and would not be affected. When the system crash during copying back from the shadow copy, we already have the last changes on the shadow page and original page partially done, the program can perform the copying again to complete the changes.

Undo Log and Shadow Copy mechanisms to achieve atomic updates evaluated based on the number of stores performed. The execution time of the evaluation will be recorded and analyzed as the basis of comparison for implementation in PMO Sync.

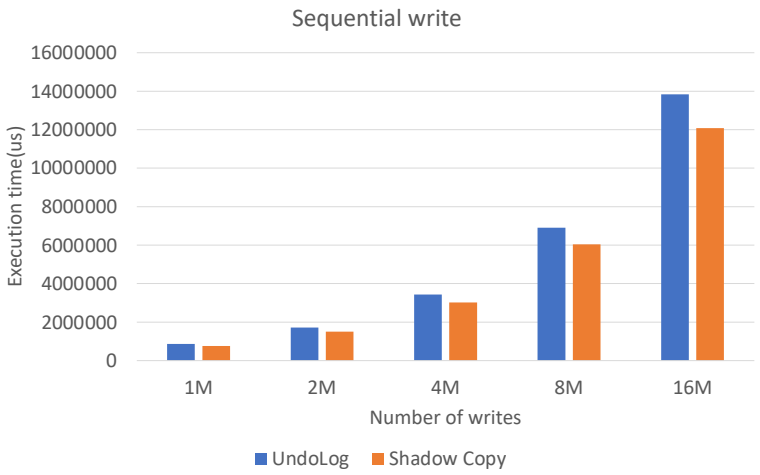

Figure 3.11: Atomic Update Result

The bar chart shown as Figure 3.11 presents the comparison of two atomic update mechanisms. As can be seen, the shadow copy mechanism outperforms undo log in the different number of stores. Both mechanisms increase overhead by incurring addition store to persistent memory however in this comparison undo log has higher execution time. Therefore we apply shadow copy as an atomic update for PMO Sync.

PMO Implementation

Persistent Memory Object (PMO)[10] is a data structure that can be accessible across process lifetime, system failure, and reboot similar to a file. PMO can be managed, allocated in Persistent Memory by Persistent Memory Object System. This Persistent Memory Object System supposes to be implemented at the level of system implementation and can be managed by Operating System. The operating system needs to organize the area of persistent memory to utilize the Persistent Memory Object. This research simulates the Persistent Memory Object System from the user-space program by performing the same operation without interrupting the operating system.

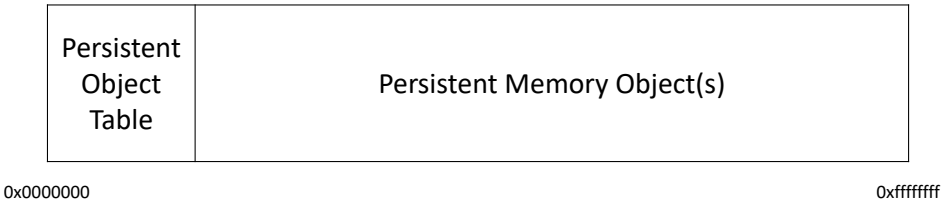

Figure 3.12: PMO Memory Layout

As the Figure 3.12 shown above, Persistent Memory area will be divided into two major areas to be able to store Persistent Memory Object. The first area is managed to store the Persistent Object Table, a data structure to manage Persistent Memory Object. Persistent Object Table is located in starting area of Persistent Memory. The starting address of Persistent Memory Object Area is after the end of the address of Persistent Object Table.

Table 3.2: Persistent Object Table

name	**address**	**size**
pmo1	0x7ffcc8abe770	4096
pmo2	0x7ffcc8ac2770	2097152
pmo3	0x7ffcc92c2770	1048576
...	...	...
...	...	...

Persistent Object Table (POT) [10] is managing and organizing Persistent Memory Object. Persistent Object Table Entry records the name of the PMO, its starting address, and size. When a program requests a new PMO with a specific name as "pmo1" and the size of 4KB, the operating system traverses the POT and checks whether the name is already taken or still available. When the name is usable then the system allocates "pmo1" to a specific address that has free space as the size of the requested PMO. In case that the name is already taken or there is no available space for the requested PMO, the operating system should return the message indicating the issue.

PMO attach() [10] is the primitive of Persistent Memory Object to mapping PMO into process address space. When attach called, Operating System checks Persistent Object Table and finds the entry based on PMO name then obtains the address of the PMO and map into process address space to be able to access by the program. Moreover, the programmer can specify the permission for PMO during attach call.

When the programmer completes modification to PMO and wants to remove PMO from process

address space, the programmer can call detach() to directly unmap PMO to be inaccessible. Therefore after detach() called for a specific PMO, the PMO is no longer accessible until the programmer calls attach to the PMO again. This detach() [10] secure Persistent Memory Object from unauthorized access when PMO is not attached in process address space.

PMO Transaction

PMO Transaction is a collection of operation load and store to Persistent Memory Object that needs to be persisted in NVM. A transaction started when a PMO attached. This PMO Transaction requires data to be atomically updated to NVM and crash consistent. PMO Transaction adapts ACID properties that are Atomicity, Consistency, Isolation, and Durability. The atomicity property of PMO Transaction is combining multiple operations into one operation that completes entirely or does not happen at all. By using the shadow copy mechanism that was already evaluated before, PMO Transaction achieves atomicity that guarantees all operations completed or nothing at all. With the shadow copy mechanism, a transaction creates shadow copy of PMO, and all modifications are performed out-of-place, and leave the original data untouched. The second property is consistency achieved by performing PMO Sync call that makes all changes durable in persistent memory. The durability of PMO Transaction indicates that when transaction complete data already persisted in NVM. This durability achieves by the abstraction of PMO Sync that is consisted of cache flush and fence for every modified data.

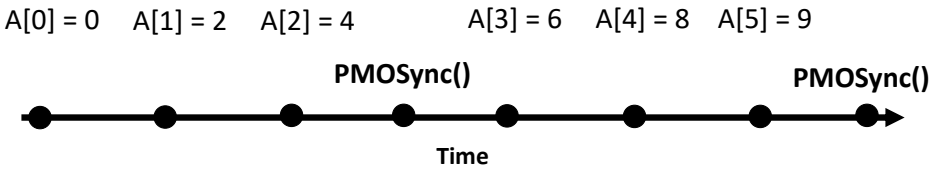

Figure 3.13: PMO Transaction and Sync

Based on the Figure 3.13 above, we can assume that a programmer wants to perform update into three array elements and followed by PMOSync() call. Those operations before PMOSync() are considered as a transaction that needs to be persisted in NVM. This transaction would not allow only partially persisted data. Therefore PMOSync() as a Sync point needs to guarantee that transaction is durable atomic and crash consistent. In addition, to guarantee durable transaction and crash-consistent, PMOSync() plays a role as a barrier of operations that makes sure all of the previous operations are already completed before continuing to the next transaction. In this case, if we have two transactions followed by two PMOSync() calls, those transactions cannot be reordered and need to be performed based on sequential order. The first transaction needs to be completed before the second transaction started.

PMO Sync

PMO Sync is a persistency semantic of Persistent Memory Object that provides durable update and crash-consistent. PMO Sync called when a programmer already completed all operations and wants to persist the changes into NVM. All modifications in a PMO transaction are performed on shadow copy. Therefore the operating system has to track the address of PMO and its shadow. We propose a table to record all of PMO addresses and their shadow copy addresses and give a status whether PMO already swapped or not during PMO Sync call.

Table 3.3: PMO Sync Table

original_address	**shadow_address**	**swap_status**
0x7ffcc8abe770	0x7ffcc96c2770	0
0x7ffcc8ac2770	0x7ffcc96c6770	0
0x7ffcc92c2770	0x7ffcc9ec6770	0
...	...	...
...	...	...

PMO Sync Table as shown as Figure 3.3 contains all addresses of PMO and its shadow copy during the transaction. PMO Sync Table entry created when an operation occurs to PMO and shadow PMO created. PMO Sync Table as persistent table located between Persistent Object Table and Persistent Memory Object Area as shown Figure 3.14 below.

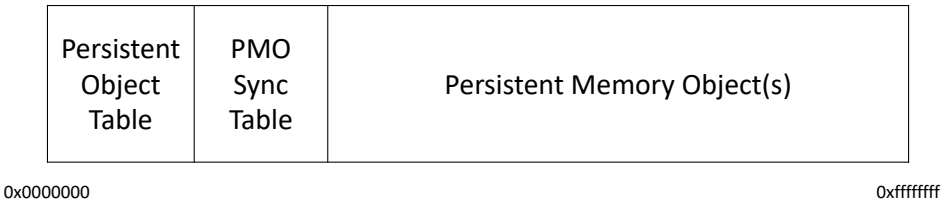

Figure 3.14: PMO Sync Table Layout

Assume that we incorporate PMO Sync Table inside of persistent memory, we need to adjust the location of Persistent Memory Object Area. The starting address of the PMO Sync Table is after the end of Persistent Object Table address, and Persistent Memory Object Area started after the end of PMO Sync Table address. All PMO Sync Table entries are created and persisted directly when a shadow copy is created. All the entries record the PMO addresses and their shadow addresses.

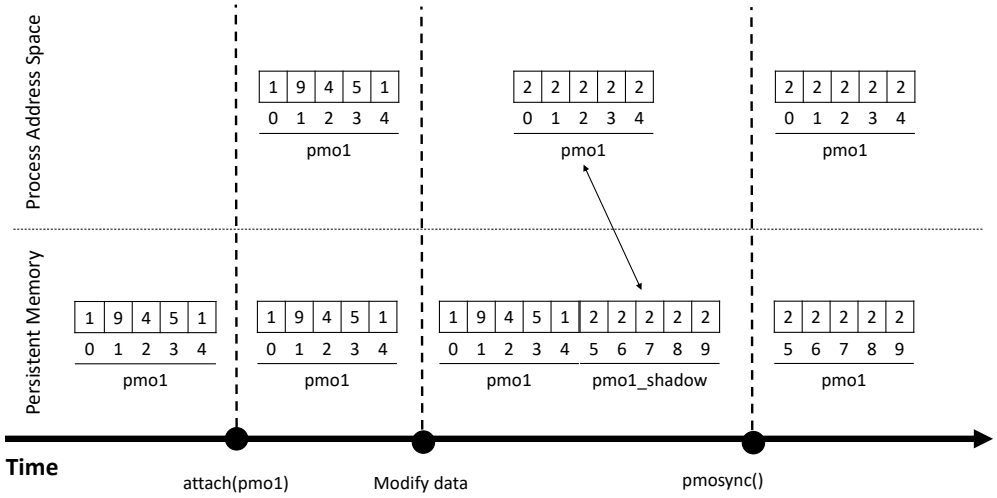

Figure 3.15: PMO Sync Flow

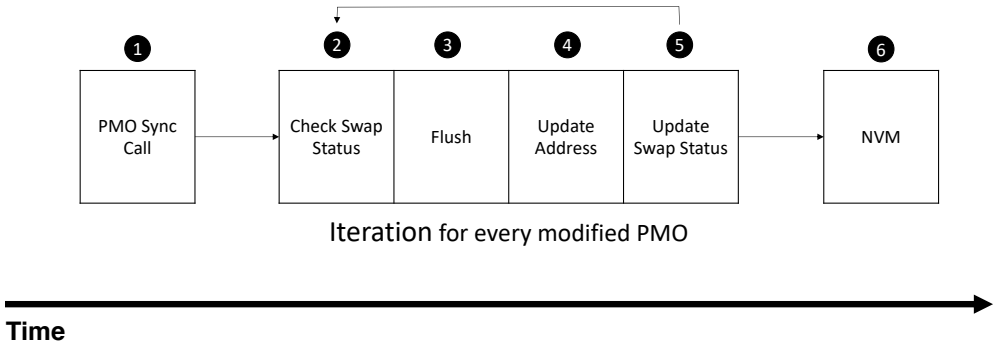

Figure 3.16: PMO Sync: Iteration for every modified PMO

PMO Sync Flow explained by Figure 3.15 depicts that the beginning of PMO transaction is when a programmer performs attach to pmo1. When a programmer performs write to pmo1, it will trigger the operating system to create a shadow copy of the PMO and pointing the modification to shadow PMO and also create an entry that records PMO address and shadow address to PMO Sync Table. After the programmer completes the modification and calls PMOSync(), the operating system accesses PMO Sync Table and flushes every cache line from shadow copy then changes the address of PMO in Persistent Object Table into the address of shadow copy. The transaction is completed when all the changes already persisted in shadow copy and the original PMO address is already swapped.

```
1  // POT = Persistent Object Table
2  // PST = PMO Sync Table
3  // pmo1 already pointing to shadow copy
4  // For each cache line of pmo1 modified data
5  for (int k = 0; k < 64; k++)
6  {
7      // Flushing modified data
8      CLWB((void *)&pmo1[k*16]);
9  }
10 // Memory barrier
11 SFENCE();
12 // Change Persistent Object Table entry
13 // of pmo1 address to shadow address
14 POT.address = PST.shadow_address;
15 // Flushing shadow address
16 CLWB((void *)&POT.address);
17 // Memory barrier
18 SFENCE();
19 // Change PMO Sync Table entry swap_status to 1
20 PST.swap_status = 1;
21 // Flushing swap status
22 CLWB((void *)&PST);
23 // Memory barrier
24 SFENCE();
```

Figure 3.17: Flush and swap address

Snippet code on Figure 3.17 shows some part of the abstraction of PMO Sync call. Line 5-11 represent relaxed persistency model that performs flushing cache line and store fence for shadow copy after completing the modification of a PMO. The next operation is to change the original PMO address into a shadow PMO address in Persistent Object Table. After updating the address

of PMO, the swap_status changed into 1 that indicates the PMO address already swapped with the shadow copy address.

Recover from System Failure

There are three potential locations where a system failure or crash could occur. The first is when a programmer called attach() and made some changes, which resulted in a crash. In this case, PMO's data remains unchanged, and the programmer can redo the attachment and modification. The second scenario is that the system fails after the programmer completes the modification and calls PMOSync(). In this case, the operating system checks the PMO Sync Table and rolls back any PMOs that have already been swapped. The final condition is that the PMOSync() call has already completed and the updates have been saved in NVM. As a result, the data has been persisted, and there is no need to roll back or recover it.

```
// POT = Persistent Object Table
// PST = PMO Sync Table
// Foreach PST entry
for (int i = 0; i < N; i++)
{
    // Find POT that already swapped address
    POT = find_pot(PST[i].shadow_address);
    // Rollback to original address
    POT.address = PST[i].original_address;
    // Flushing POT entry
    CLWB((void *)&POT);
    // Memory barrier
    SFENCE();
}
```

Figure 3.18: Recovery

Figure 3.18 gives idea of how to perform rollback during the crash. The operating system needs to check every PMO Sync Table Entry and check whether the address of PMO in Persistent Object Table is already swapped. If PMO address already changed into shadow address, then roll back POT entry to use original PMO address based on PMO Sync Table.

Microbenchmark

For evaluating PMO Sync implementation, we create microbenchmark that performs store as integer array. The store performs as sequential write starting from array element 0 until the N number of the store as shown on Figure 3.19.

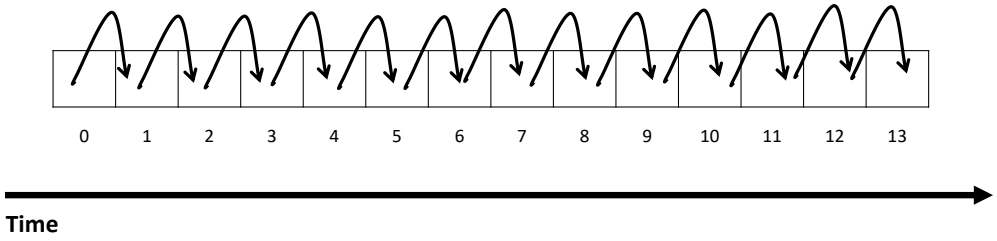

Figure 3.19: Sequential Write

In this evaluation we assume that PMO size is 4KB, an integer array element is 4B, therefore a PMO can store 1024 integer array elements. The microbenchmark creates N number of PMOs based on the number of the store that we want to evaluate. Suppose that we want to perform 1,000,000 stores, then microbenchmark will create 1.000.000 divided by 1024 that is 977 PMOs created, and allocated in contiguous address. Microbenchmark will attach PMO when store reaches the first element of PMO, and perform PMO Sync when store reaches the last element that indicates transaction for a PMO already completed. This microbenchmark performs stores until the last PMO.

Persistency Validation

PMO Sync call suppose to persist all store to become durable to NVM. We want to validate whether the stores performed by microbenchmark reside in NVM, and accessible across process lifetime. Since we utilize Device-DAX mode for Persistent Memory Object, we need to use specific tools that can read data from Device-DAX. Intel provides daxio utility that performs IO on Device-DAX.

```
// Save Persistent Memory data to file
daxio --input=/dev/dax0.0 --output=datafile --len=100M --seek=0
```

Figure 3.20: Read PMO data with daxio

Assume that we use /dev/dax0.0 as Persistent Memory Object location. We read data from PMO and save it to a file named datafile. We will see whether the data are persisted or not. The second tool that we use to validate PMO Sync is Valgrind.

```
// Start debugging
if (DEBUGING == 1)
        VALGRIND_PMC_REGISTER_PMEM_MAPPING(pmo, SIZE);

        /*
        Store operations
        */

// End debugging
if (DEBUGING == 1)
        VALGRIND_PMC_REMOVE_PMEM_MAPPING(pmo, SIZE);
```

Figure 3.21: Valgrind

We embedded Valgrind into the code and compare a program that performs stores without any persistency and with PMO Sync.

CHAPTER 4: RESULTS & DISCUSSION

We evaluated the performance of PMO Sync based on execution time compare to Undo Log and Shadow Copy.

Machine Specification

The evaluation of PMO Sync was conducted on Persistent Memory Machine 1 of ARPERS Research Lab at the University of Central Florida. This Persistent Memory Machine 1 has the following configuration:

Table 4.1: Experimental Setting

Processor	Intel(R) Xeon(R) Gold 6230 CPU x2
Cache	1280KiB L1 cache 20MiB L2 cache 27MiB L3 cache
Memory	32GiB DIMM DDR4 2666 x4
Persistent Memory	128GiB Optane DC x4
OS	Fedora release 33 (Thirty Three)

This research utilizes /dev/dax0.0 on Persistent Memory Machine 1 with the size of 128 GB. The evaluation runs on a single processor and single thread.

PMO Sync Result

Microbenchmark was used to evaluate PMO Sync. In our experiment, the benchmark was set at various numbers ranging from 1 million to 16 million stores. The execution time is measured in microseconds. The benchmark is run ten times, and the result is the average of the ten results. Figure 4.1 shows the end result.

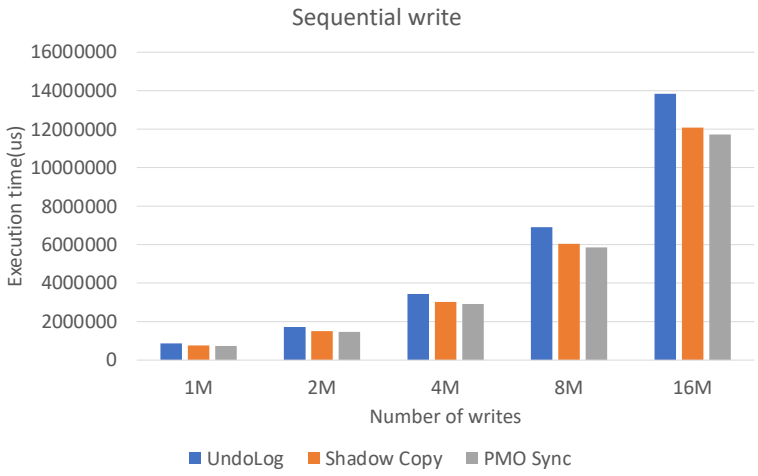

Figure 4.1: PMO Sync Benchmark Result

Figure 4.1 depicts the microbenchmark results of PMO Sync, Undo Log, and Shadow Copy mechanisms. It is obvious that the outcome is consistent across a variety of stores. PMO Sync outperformed shadow copy and undo log in terms of performance and execution time.

To provide an equal comparison, we set PMO Sync call on a specific number of stores with the same number of stores that shadow copy performed. When it comes to actual programming, the

programmer can decide when PMO Sync should be called. It will give programmers the option of having either fine or coarse granularity.

PMO Sync achieved atomic updates and crash recovery by utilizing optimized shadow copy, which performs shadow PMO modification. By avoiding copying data back to the original PMO, the optimized shadow copy reduced the number of flushes and fences. PMO Sync Table plays an atomic update role, providing a shadow PMO address for swapping with the original PMO address and ensuring recovery in the event of a system failure. Although optimized shadow copy can reduce the number of flushes and fences, there is additional overhead when accessing the PMO Sync Table from a user-space program. Better performance could be obtained if the PMO Sync Table was managed by the operating system without user intervention.

The PMO Sync programming model allows the programmer to manage the granularity of transactions on Persistent Memory Objects. PMO Sync simplifies programming by combining memory instruction, a persistency model, and an atomic update mechanism into a single PMO Sync call, allowing the programmer to focus on modification data. The programmer does not need to specify the address that needs to be flushed in a PMO Sync call by reason of it is already abstracted by the PMO Sync call, which prevents some errors from the programmer side, such as forgetting to flush some data. In summary, the PMO Sync comparison shown below:

Table 4.2: PMO Sync Comparison

Aspect	**Persist+Barrier**	**Undo Log**	**Shadow Copy**	**PMO Sync**
Atomic Update	No	Yes	Yes	Yes
Crash Recovery	No	Log	Shadow Copy	Optimized Shadow Copy + PMO Sync Table
Write Amplification	Low	High	High	Medium
Execution time overhead	Low	High	High	Medium
Programming Model	Flush+Fence	Flush+Fence	Flush+Fence	Sync

Persistency Validation

Daxio was the only utility we used to perform IO for Device-DAX. We read data from persistent memory and save it to a file.

The Figure 4.2 show the content of persistent memory after PMO Sync call:

```
0000040          0          2          4          6
0000060          8         10         12         14
0000100         16         18         20         22
0000120         24         26         28         30
0000140         32         34         36         38
0000160         40         42         44         46
0000200         48         50         52         54
0000220         56         58         60         62
0000240         64         66         68         70
0000260         72         74         76         78
0000300         80         82         84         86
0000320         88         90         92         94
0000340         96         98        100        102
```

Figure 4.2: Daxio copy data from Device-DAX

We can see that the stores performed by PMO Sync that stored even numbers to array elements were persistent on NVM and durable across process life cycles.

Moreover, we validate persistency using Valgrind. We used Valgrind to count the number of stores that were not made persistent as a result of the PMO Sync call. The number of stores in this validation is set to 1,000,000 integer array elements equal to 4,000,000 bytes of data. First, we see what happens when we perform stores without any cache line flushing or fencing, as shown below:

```
==362811==      Address: 0x7faea2600000 size: 4000000   state: DIRTY
==362811== Total memory not made persistent: 4000000
==362811== ERROR SUMMARY: 1 errors
```

Figure 4.3: Store without persistency

We compare the outcomes of the PMO Sync call:

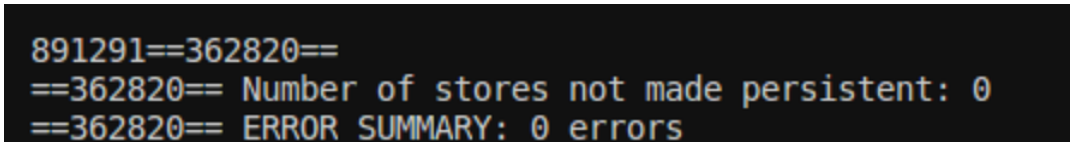

Figure 4.4: Store with PMO Sync

It is obvious based on Figure 4.4 that the PMO Sync call persists all of the stores and makes them persistent in persistent memory. There is no more dirty cache line similar to Figure 4.3, and all stores on Persistent Memory have been made durable.

CHAPTER 5: CONCLUSION

Conclusion

We discussed the implementation of Persistent Memory Object on Device-DAX in this study. We assess the performance of three memory instructions: CLFLUSH, CLFLUSHOPT, and CLWB. We discuss the persistency model for storing data in NVM. The performance of two atomic update mechanisms, Undo Log and Shadow Copy, is then compared.

Based on these findings, we proposed PMO Sync, a Sync call that specific to Persistent Memory Objects in order to achieve durable transaction and crash consistency. PMO Sync abstract memory operation CLWB flushes all modified cache lines. Relaxed persistency was used to reduce overhead while increasing the granularity of PMO Sync. PMO Sync uses optimized shadow copy to achieve durable transactions by replacing the original address with the shadow address. According to our findings, the execution time overheads of PMO Sync are lower than those of the Undo Log and Shadow Copy mechanisms.

Future Work

This study simulates the Persistent Memory Object System at the software level using a user-space program. To achieve real performance, the Persistent Memory Object System can be implemented as a system-level managed by the Operating System in future work.

LIST OF REFERENCES

[1] I. Corporation, "Intel® Optane™ Persistent Memory." https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. Accessed: 2021-03-11.

[2] S. Scargall, *Programming Persistent Memory*. Apress, 2020.

[3] Y. Solihin, "Persistent Memory: Abstractions, Abstractions, and Abstractions," *IEEE Micro*, vol. 39, no. 1, pp. 65–66, 2019.

[4] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," *Proceedings - International Symposium on Computer Architecture*, pp. 265–276, 2014.

[5] I. Corporation, "Intel ® 64 and IA-32 Architectures Software Developer ' s Manual Documentation Changes," *System*, vol. 3, no. 253665, 2011.

[6] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014*, 2014.

[7] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile," *14th USENIX Conference on File and Storage Technologies (FAST)*, pp. 323–338, 2016.

[8] Z. Mao, S. Zheng, L. Huang, and Y. Shen, "A DAX-enabled mmap mechanism for log-structured in-memory file systems," *2017 IEEE 36th International Performance Computing and Communications Conference, IPCCC 2017*, vol. 2018-Janua, pp. 1–8, 2018.

[9] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.

[10] Y. Xu, Y. Solihin, and X. Shen, "MERR: Improving security of persistent memory objects via efficient memory exposure reduction and randomization," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 987–1000, 2020.

[11] Y. Xu, C. Ye, Y. Solihin, and X. Shen, "Hardware-Based Domain Virtualization for Intra-Process Isolation of Persistent Memory Objects," *Proceedings of the 47th International Symposium on Computer Architecture (ISCA)*, pp. 680–692, 2020.

[12] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *SOSP'09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, (New York, New York, USA), pp. 133–146, ACM Press, 2009.

[13] J. Zhao, O. Mutlu, and Y. Xie, "Firm: Fair and high-performance memory control for persistent memory systems," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 153–165, IEEE, 2014.

[14] A. Rudoff, "Persistent memory programming," *Login: The Usenix Magazine*, vol. 42, no. 2, pp. 34–40, 2017.

[15] Pmem.io, "Managing Namespaces - NDCTL User Guide." `https://docs.pmem.io/ndctl-user-guide/managing-namespaces`. Accessed: 2021-03-5.

[16] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," *Proceedings - International Symposium on Computer Architecture*, pp. 265–276, 2014.

[17] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-Performing and write-Efficient software persistency technique," *Proceedings - International Symposium on Computer Architecture*, pp. 439–451, 2018.

[18] S. Shin, J. Tuck, and Y. Solihin, "Hiding the long latency of persist barriers using speculative execution," *Proceedings - International Symposium on Computer Architecture*, vol. Part F1286, pp. 175–186, 2017.

[19] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "ATOM: Atomic Durability in Non-volatile Memory through Hardware Logging," *Proceedings - International Symposium on High-Performance Computer Architecture*, pp. 361–372, 2017.

[20] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, vol. F1312, (New York, NY, USA), pp. 178–190, ACM, oct 2017.

[21] Y. Ni, J. Zhao, D. Bittman, and E. L. Miller, "Reducing NVM writes with optimized shadow paging," *10th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2018, co-located with USENIX ATC 2018*, 2018.

[22] S. Haria, M. D. Hill, and M. M. Swift, "MOD: Minimally ordered durable datastructures for persistent memory," *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*, pp. 775–788, 2020.

[23] S. Sub-paging, Y. Ni, U. C. S. Diego, D. Bittman, and E. L. Miller, "SSP : Eliminating Redundant Writes in Failure-Atomic NVRAMs," *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 836–848, 2019.

[24] Jens Axboe, "Flexible I/O Tester." `https://github.com/axboe/fio`. Accessed: 2021-03-7.

[25] I. Corporation, "daxio - Perform I/O on Device DAX devices or zero a Device DAX device." `https://pmem.io/pmdk/manpages/linux/v1.10/daxio/daxio.1.html`. Accessed: 2021-03-11.

[26] I. Corporation, “pmem/valgrind: Enhanced Valgrind for Persistent Memory.” https://github.com/pmem/valgrind. Accessed: 2021-03-19.