**Persistent memory emulation to implement simplified NV-Heaps.**

**Project Report**

Submitted in partial fulfilment of the requirements for

**CS 695**

**Submitted by**

Abhik Bose, 184050002

Akash Trehan, 150050031

**Instructor: Prof. Purushottam Kulkarni**



**Computer Science and Engineering**

**Indian Institute of Technology Bombay**

**May 2019**

**Abstract**

Persistent memory (PM) is a newly emerging memory technology residing between fast byte addressable volatile DRAM and slow persistent block devices like hard disk or storages. Data written to this memory is persistent across power cycles. Due to its comparable access time and byte addressability like DRAM, it sits on the processor’s memory bus. So, a chunk of persistent memory can be mapped to the process’s virtual address space and can be directly accessed by the process without intervention of the operating system. This opens a new window for persistent user-defined data structures while avoiding slow block I/O based initialization operations, hence speeding up programs.

However, persistent memory poses several programming challenges due to this persistent nature itself. Firstly, the mapping of persistent memory to a process’ virtual address is likely to change at every execution cycle. Thus, there exists a need for a mechanism to safely store and retrieve data from persistent memory even when the mapping changes. This ever-changing persistent memory mapping also requires a special technique to handle pointers. Secondly, a volatile pointer if stored in persistent memory becomes invalid and potentially dangerous in future executions. Lastly, an e Handling all this manually is a burden for regular programmers.

Existing works like Persistent Memory Development Kit (PMDK), NV-Heap have mature and complex functionality to hide these low-level details from users and make their life easier. In this project, we are implementing a simplified version of NV-Heap. We provide a C/C++ based general purpose programming library for safe storage and retrieval of user-specific objects to and from persistent memory. Our library allows users to specify and work with custom data structures. Our implementation includes a bitmap-based stub persistent memory allocator, safety against invalid pointers and ACID transactions.

We have evaluated our implementation with several known data structures like linked list and binary search tree. The library also allows a single program to store multiple data structures of the same or different type (e.g. a linked list and a BST) in the same NV-Heap.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Table of Contents** | |  |
| **Section 1** | **Introductions** | | **1** |
| **Section 2** | **Motivation and problem context** | | **3** |
|  | 2.1 | Store and retrieval | 3 |
|  | 2.2 | Safe handling of pointers | 3 |
|  | 2.3 | Atomicity | 3 |
|  | 2.4 | Thread safety | 4 |
|  | 2.5 | Existing libraries and their functionality | 4 |
| **Section 3** | **Simplified NV-Heap components and features** | | **5** |
|  | 3.1 | Support for user-specified data structure | 5 |
|  | 3.2 | Storage and retrieval of root pointer | 5 |
|  | 3.3 | Program initialization across execution cycle | 5 |
|  | 3.4 | Memory allocation | 5 |
|  | 3.5 | Pointers and safety | 6 |
|  | 3.6 | Atomicity | 6 |
| **Section 4** | **NVH Library Implementation details** | | **7** |
|  | 4.1 | Non-volatile memory initialization | 7 |
|  | 4.2 | Getting and setting the root | 8 |
|  | 4.3 | Persistent memory allocation | 8 |
|  | 4.4 | NVPtr and pointer safety | 8 |
|  | 4.5 | Atomicity and transaction management | 9 |
| **Section 5** | **Experimental evaluations** | | **11** |
|  | 5.1 | Store and retrieve functionality | 11 |
|  | 5.2 | Pointer safety evaluation | 11 |
|  | 5.3 | Transaction management evaluation | 12 |
| **Section 6** | **Summary and interesting aspects** | | **13** |
| **Section 7** | **Future plan** | | **14** |
|  | 7.1 | Mature memory manager | 14 |
|  | 7.2 | Dynamically re allocable non-volatile memory chunk | 14 |
|  | 7.3 | Multiple non-volatile memory chunk and shared memory | 14 |
|  | 7.4 | Thread safety and locking | 14 |

**1. Introduction**

Conventional memory hierarchy primarily consists of fast volatile memories e.g. byte addressable DRAM and slow non-volatile block devices e.g. Hard disks, Flash storages etc. At the top of the memory hierarchy CPU Registers are as fast as CPU, however, block devices are thousands to million times slower than CPU and the entire hierarchy is designed to use locality of memory references to mask the slowness of non-volatile block devices. Recent years seen several fold speed improvement of block devices, however, these devices are still 100 to 1000 times slower than DRAM based main memory. Further, block addressability of this non-volatile storage devices limits the minimum granularity of memory access (usually 512 Bytes) and negatively impact performance of small read and writes.

|  |
| --- |
| https://lh6.googleusercontent.com/i2cukAzOnxDazvfdOb7zlMRJwb0f0aN9HWkLBAwU28UJLqw4YzOv-xY6TyWOPM737aYZwFDDUou66lQw_edDipm-TRMu0IGzlGG559THoFpLuJuU8c1UYAamSw8XfOT9LKaPyG3c |
| **Figure 1.1:** Memory hierarchy highlighting the large difference between access time of volatile DRAM and non-volatile block storage devices. |

Persistent memory technologies with speed comparable to volatile DRAM are slowly emerging. Significant developments are happening in the field of Spin-transfer torque memory (STTM) and Phase change memory (PCM). Unlike non-volatile block storage, persistent memories are byte addressable like volatile DRAM, and the data on them is persistent across power cycles. Being a fundamental change in the memory hierarchy, persistent memory is expected to open new avenues in several fields including operating systems, compiler design, and others.

With these advancements in persistent memory technology, we can now store a program’s execution state on persistent memory. Multiple approaches to persistent memory programming are possible. In the persistent program approach, the entire program (its stack, heap, etc.) are in persistent memory. In this approach program’s execution state is virtually preserved during a power cycle. However, this approach introduces a notion of persistent process and non-persistent process and needs significant deviation from the conventional approach of operating systems towards processes.

So, most persistent programming approaches let a conventional process store its variables and user-defined objects on persistent memory. In the absence of persistent memory programming, a process needs to store its data on block devices as files, going through the file system and operating system. The file system transparently caches all reads and writes on a file into DRAM. These overheads are justified considering the 512 Byte read/write granularity and performance improvement over slow block devices. However, with modern persistent memories, caching persistent memory accesses into DRAM leads to unjustified overhead and performance degradation. Further, programs often copy most of the data from file to memory before modification. This also leads to unnecessary overheads while using persistent memory. This clearly points out the inefficiency of conventional block-device oriented file systems when used with persistent memory and demands a new paradigm for persistent memory programming.

Most modern operating systems allow mapping file-backed memory in the user’s virtual address space e.g. “mmap” system call for Linux. This comes in handy for persistent memory programming. A user program trying to deal with persistent memory faces challenges specific to the persistent environment. The program needs to define a metadata layer to store and retrieve data on a power cycle. It also needs to ensure the correctness and integrity of data after a restart. Any inconsistency leads to permanent damage.

The goal of persistent memory libraries is to provide an easy to use interface to use byte addressable persistent memory mapped to the user program’s virtual address space. Significant works in persistent memory programming include persistent memory development kit (PMDK), Non-volatile heap (NV-Heap) and Mnemosyne. These libraries provide mature functionalities like safe storage and retrieval, pointer safety (discussed in more details later), concurrency, and atomicity. In this work, we implement a simplified version of NV-Heap C/C++ based library for safe storage and retrieval of user-specific objects to and from persistent memory.

**2. Motivation and problem context**

Persistent memory programming means is programmatically storing and retrieving data as variables and user-defined objects from persistent memory safely and efficiently. The challenges it poses to programmers are described below.

**2.1. Storage and retrieval:** Persistent memory programs retrieve data and state of last execution from persistent memory. For example, a program implementing a conventional linked list executing in volatile memory will always initialize its head pointer to null during a fresh execution, irrespective of the value during its previous execution. However, a similar program executing on persistent memory must initialize its head pointer based on its last execution state. It is important to note that a block of persistent memory may not always be mapped to the same virtual memory address range. Hence, metadata-based architectures are often used for safe storage and retrieval of data from persistent storage. This comes with extra programming overhead.

**2.2. Safe handling of pointers:** Pointers stored in persistent memory often become invalid and such invalid pointers are potentially dangerous across program execution cycles. A pointer saved in persistent memory holding a volatile memory address (NV to V pointer) will become invalid after program restart. Further, a pointer from non-volatile to non-volatile (NV to NV) also becomes invalid when a chunk of persistent memory is mapped to a new virtual address range on its next execution. Such wild pointers once stored are permanent and it is tough to debug and remove them. Thus, pointers need special care when stored in persistent memory.

|  |
| --- |
| 1. struct node \* temp = head -> next  2. free (head);  3. head = temp; |
| **Figure 2.1:** Code snippet showing conventional “delete\_front” operation on a linked list. |

**2.3. Atomicity:** Atomicity is especially important in the context of persistent memory programming. Consider a conventional “delete\_front” operation on a persistent linked list shown in Figure 2.1. If a program accidentally terminates after line 2 and before line 3, the entire linked list will be permanently lost. Further, this will lead to memory leaks which are persistent across program execution cycles. This implies that either the user program or the persistent memory library must ensure the atomicity of critical code blocks. A critical block of code must either finish execution completely or have no effect at all. Unfortunately, persistent memory hardware guarantee atomicity for 8-byte writes only. Atomicity of sequential writes must be programmatically ensured.

**2.4. Thread safety:** A chunk of persistent memory can be mapped to multiple processes’ address space and can also be shared between threads. This leads to a need for strong thread safety to prevent read-write conflicts.

**2.5 Existing libraries and their functionality:** PMDK, Mnemosyne and NV-Heap are a few user-level persistent memory development tools and libraries. PMDK provides a set of libraries such as libpmem and libpmemobj. Libpmem deals with low-level persistent memory details whereas libpmemobj comes with advanced functionalities for pointers and transactions among other things. NV-Heap is a C/C++ persistent memory library more focused on user-defined objects. It provides a set of rules and prototypes to define custom data structures. This includes a prototype class for object definition and a prototype class for pointers. Libpmemobj and NV-Heap libraries also implement pointer safety and journal-based transaction management to provide atomicity. Currently they are partially thread-safe, with provisions for using mutex like persistent memory specific lock.

In this project, we implement a simplified version of NV-Heap library on top of the libpmem library provided by PMDK. This provides safe object storage and retrieval, pointer safety and log-based transaction management to implement atomicity. We designed a metadata layout to initialize a user program based on its last execution state. This provided a great learning vehicle for persistent memory programming.

**3. Simplified NV-Heap components and features**

**3.1. Support for user specified data structure:**  Our implementation of NV-Heap does not put any restrictions on the nature of the data structure. The library and the programming layout provided is the same for all different persistent memory objects in a program. User has the full ability to define custom structures and classes without providing any specific information to the library.

3.2. Storage and retrieval of root pointer: The library provides an interface to store and retrieve a single root pointer into persistent memory. The library guarantees consistency of the data pointed to by the root pointer over program execution cycles. The user decides which variable or object the root pointer will point to. The root is intended to point to the head node of the persistent memory data structure. In case the program implements multiple data structures, the user can store the head pointer of all data structure in another structure which the root then points to.

3.3. Program initialization across execution cycle: All persistent data of a program is stored into a file (in linux). The file should ideally be on a persistent memory device with a direct accessible (DAX) byte addressable file system like NOVA. The library is also backward compatible with block devices like HDD and FLASH storage. However, that will lead to operating system overheads causing performance degradation.

The user program gives a file name and a password specific to the file. The library sets the password when a file corresponding to an NV-Heap is accessed the first time. All subsequent accesses to the same file must use the same password. Based on the state of the file the library performs several sanity checks, determines last execution status and maps the file to executing process’ virtual address space. Now, the program is ready to retrieve the root pointer and use the library with full force.

3.4. Memory allocation: NVH library takes all responsibility for persistent memory bookkeeping. It provides “nvh\_malloc()” and “nvh\_free()” functions just like conventional “malloc()” and “free()” to allocate and deallocate memory chunks on persistent memory. Unlike the “stdlibc” provided free function, “nvh\_free()” must be supplied with both the base address and size of the memory to be freed. Now the program can choose to allocate persistent and volatile memory, as required, during its execution.

3.5. Pointers and safety: The library provides an “NVPtr” class for managing non-volatile pointers. The user must use NVPtr to implement pointer in any custom persistent data object. The library abstracts the implementation details of NVPtr. A dereferenceable pointer pointing to non-volatile memory can be directly assigned to an NVPtr object. The user must use the “dptr()” function to get a dereferenceable pointer from an NVPtr. The address of the dereferenceable pointer corresponding to an NVPtr may differ across program execution cycles. However, the library ensures that the object pointed to by an NVPtr is unchanged irrespective of its dereferenceable address. The library allows the user to assign only valid non-volatile pointers into NVPtr. Any wrong memory address, if assigned to NVPtr, is caught and execution is terminated.

**3.6. Atomicity:** The library provides atomicity for any user-defined block of code. It ensures the completeness or restoration of the full code block, as the case may be. The library also undoes any persistent memory allocations and deallocations that happen during a failed transaction to prevent persistent memory leaks. The program can start an atomic block using the “tx\_begin()” and end using the “tx\_commit()” function. The user must supply the address of the objects which they want to protect and their sizes using “tx\_add()” or “tx\_add\_direct()” function to the library.

**4. NVH Library Implementation details**

Our implementation of simple NVH library use a metadata-based approach for safe store and retrieve of object on persistent memory and for transection management. Following sections describes the metadata layout and roles of its’ important fields in library operation.

**4.1. Non-volatile memory initialization:**

|  |
| --- |
|  |
| **Figure 4.1:** Metadata layout of a 64 K Byte NV Heap library in our implementation |

Simplified NV Heap library is capable to handle dynamic size persistent memory chunk (size of the NV Heap), however the size must be mentioned statically before compilation. Dynamic resizing of NV Heap during execution is not yet supported. Figure 4.1 shows metadata layout of a 64 Kbyte NV Heap. Each NV Heap is initialized with a user defined password. The library stores an 8-byte hash of the password in initial 8 byte. Another 8-byte hash is generated from the previous hash and stored into next 8 byte. This is for sanity checking and provides some redundancy against corruption. When user call “nvh\_init” with a file name and password to initialize a NV heap, the library compute hash of the user provided password and check the one stored. If this matches and hash of this also matches, then it proceeds to open the NV heap using “nvh\_open” function and store the base address of the mapped virtual address in a volatile global variable (nvh\_base\_address). Else a new NV heap is created using “nvh\_create” function. This creates a fresh metadata layout on the non-volatile memory chunk and store the base address mapping as previously for further use. An 8-byte length field is intended to store exact size of NV heap in future versions with support for dynamic resizing.

**4.2. Getting and setting the root:** Offset of the root object is stored into 1024 – 1031th byte. A Null root object is indicated with -1. “nvh\_getroot” returns dereferenceable pointer by adding the “nvh\_base\_address” with the offset and returns Null if offset is -1. Program may need to change its root object during execution. This is handled by “nvh\_setroot” in exact opposite way.

**4.3. Persistent memory allocation:** We implemented a simple stub allocator to handle all allocation and deallocation requests for the persistent memory. The allocator can handle memory allocation request of any size (limited by available free space and size of the NV Heap), however it allocates to a granularity of 8-bytes i.e. for a 9-byte allocation request 16 byte will be allocated. We adopted this approach to avoid memory allocation overhead and for simplicity. The allocator mentions a bitmap array to keep track of the allocated and free bits. 1-bit per 8-byte is used. The bitmap array resides in a fixed position starting from 24th Byte. The allocator keeps track of overall allocation status, however, size of individual memory requests is not mentioned. It’s the user’s responsibility to keep track of each allocation size and provide that during deallocation.

**4.4. NVPtr and pointer safety:** In our implementation non-volatile pointer is basically the offset of a memory location form “nvh\_base\_address”. The offset of any memory location doesn’t change across program execution cycle. Same as root pointer -1 indicates Null pointer. We implemented NVPtr class and stored the offset in 8-byte signed integer to achieve hardware atomicity during update. “=” operator was overloaded to implement pointer safety. Programs may either assign a offset value or a dereferenceable pointer type casted to void \*. The assignment operator checks whether its’ inside the range of non-volatile memory and accepts only valid pointers and offsets. Only the offset corresponding to dereferenceable pointer is stored in NVPtr. “nvh\_dptr” function do exact opposite.

**4.5. Atomicity and transaction management:** Providingautomaticity for a block of code is a crucial and challenging part of any persistent memory library. An unexpected program termination and inconsistency is mainly caused due to 3 reasons

1. Power and/or hardware (other than non-volatile storage) failure.
2. Operating system failure.
3. Non volatile memory failure i.e. disk or persistent memory failure.

Providing consistency against disk failure is often handles with multiple disks and complex techniques which is not a part of simplified NVH library. The library in current version provides atomicity and consistency only against the first 2 reason of failure.

We’ve taken transection log-based approach to implement atomicity on a non-volatile memory. Transection log contains all changes done by a block of code seeking atomicity on persistent memory. During program initialization transection manager checks for any unfinished transections and either finish them. A failed transection may be finished by looking the log or can be undone to provide atomicity. In redo approach entire block of code seeking atomicity must be logged before the code block starts execution. Then, in case of a program crash during execution of the atomic block, transection manager checks the log during next program initialization and complete them. It’s possible because the entire transaction is logged before execution of the code block starts. This guarantees that if the block has started execution then the logging is completed and enough for the transection manager to complete it in case of failure. On completion of an atomic block of code, either during first execution or by transection manager, the transection log corresponding to the transaction is deleted.

Undo based transection manager don’t need the program to write entire block of transection ahead execution. Each changes of the memory location must be logged before making that changes on memory. The program may choose to log them before starting execution of a block of code, however, the transection manager only demands any change must be logged before it is happening. Once the program commits the transection log is deleted. In case of a failed transection the log is checked by transection manager during program initialization and is undone.

Both undo and redo log-based transection manager is individually functionally complete to provide atomicity for a block of code. A persistent memory program or library may choose to implement any of them, or a mixed version where based on the transection progress its fate is decided during recovery. A complex transection manager with both redo and undo capacity is expected to perform faster than a simple one with only one functionality. However, considering implementation complexity we’ve chosen to implement a simple undo log-based transection manager. Further, logging a transection completely before execution possess some programming challenges during implementation which is not present in undo-based transection manager.

We’ve reserved half of the non-volatile memory for logging transections. An 8-byte structure was made to keep track of running status and number of transactions. Any update needs to be done on the structure is done in volatile memory first and then written back on persistent memory to use the 8-byte write atomicity provided by intel x86-64 ISA. A program may start a transection using “tx\_start” function. Before any memory write program must pass the memory location and size to the API using “tx\_add” function. The library stores the entire content of the memory in a transection object (struct “tx\_obj”, a fixed size structure) and then update the transection status. If any of this fails in between the transection will not be executed at all.

The transection manager also deals with V to NV pointer. If a memory allocation on non-volatile memory is done during an atomic block of execution it must be undone in case of a failure to ensure no memory leak. This is done by modifying “nvh\_malloc” function. During each allocation the function checks whether a transection is running. If so, it logs the entire region of the allocation table correspond to the allocated region. In case of failure this is also undone making the memory region free and thus preventing any memory leak. Once a program call “tx\_comit” the transection is marked as inactive by flipping the running status. In case of failure the transection manager undoes it and then change the running status.

**5. Experimental evaluations**

We’ve checked each component of the NV Heap library implementation with several persistent memory programs. This includes implementation of some known data structures and evaluation of pointer safety and atomicity. A true non-volatile memory was not yet available with us. So, we assumed the underlying “libpmem” library will extend our implemented on a true non-volatile memory without requiring any change from our side. We used a file residing on traditional ext4 file system on this block device to emulate persistent memory and mapped it to program’s virtual address space using “libpmem” library provided by pmdk.io. “libpmem” can detect the nature of underlying storage and abstract a non-volatile like storage on top of it. This will definitely perform poor with respect to true persistent memory; however, correctness will be conserved.

**5.1. Store and retrieve functionality:** We first evaluated our implementation with a user program implementing a simple linked list residing on non-volatile using the NV Heap library. We implemented insertion in linked list, deletion traversal etc common operations. It was found that the library successfully managed a long link list of more than 1000 nodes on non-volatile storage. We terminated the user program and found the non-volatile list remained intact between 100s of execution cycles.

We also implemented a bit more complex data structures like binary search tree etc. We further demonstrated that a program maintaining multiple non-volatile data structure may also be implemented using the library. We saved the root node of a linked list and a binary search tree in another structure and stored it in the root of the NV Heap. Both data structure with more than 100 nodes remained intact over multiple execution cycle.

**5.2 Pointer safety evaluation:** The library prevents users from storing any memory address not corresponding to the NV heap in a non-volatile pointer (a NVPtr object stored on persistent memory). To check this, we tried to purposefully assign some volatile pointer returned by “malloc” function into a NV pointer. The library was found successfully caught those errors. However, the library doesn’t prevent storing non-volatile memory address in volatile pointer, a crucial point to avoid memory leak. This is because we need to store NV memory address in volatile pointer variable during normal execution of the program as per current abstraction provided by the library. User program must allocate any non-volatile memory in an atomic transection to prevent any memory leak.

**5.3. Transection management evaluation:** A transection manager can best be evaluated by simulating failure between each transection and checking correctness. However, we evaluated transection management in a rather simple way due to lack of time. We introduced “exit” function in multiple if else conditions. A particular if else conditions was triggered by value given from command line argument during each execution. This block of codes including an exit was placed inside an atomic block of code. The program was caused to exit based on command line input. It was observed that each atomic transection was restored back as it was not started. We also terminated the program several times using SIGINT while execution of an atomic block. The persistent memory was never found to be corrupted by all such interruptions.

Checking the pointer safety was a bit tricky. We adopted a simple visual inspection of the allocation bit in sublime text. We copied the allocation bits in a text file before an allocation. We then interrupted a non-volatile memory allocation using the method described previously. We’ve executed the program and stopped before the allocation and found the allocation bits were same before the execution was started. This roughly says that the transection manager is restoring any allocation during a running transection and thus preventing memory leak.

**6. Summary and interesting aspects**

In this work we’ve implemented a simple non-volatile memory management library called NV-Heap on the top of “libpmem” library, a low-level persistent memory management library, provided by pmdk.io. The NV Heap library allows users to store and retrieve custom object in non-volatile memory using a root pointer provided by the library. We also implemented pointer safety, an essential feature to deal with non-volatile pointer. The library prevents assigning a memory address not corresponding to the non-volatile memory chunk into the persistent memory. The library also allows a program to execute any block of code atomically. This was implemented using undo transection log on top of 8-byte atomic write provided by intel x86\_64 ISA.

This is the first time we worked with persistent memory. It felt great to design a file system like metadata structure to store and retrieve an object and then it working with user programs. The idea of storing custom objects on persistent memory directly without using a file system API was interesting. This comes with persistent memory specific challenges, which we have already discussed, and handling them was challenging. It was exciting to work on something which is in its early stages and is seeing active development these days. We read several online documents, watched videos. Those were interesting. Then we came across pmdk.io and libpmem, libpmemobj library. We studied their features and studied some code also. It’s a big project and understanding the code was challenging. Although most of our work is already done, this project served as an important learning vehicle to understand persistent memory programming..

**7. Future plan**

The following extensions are possible:

**7.1. Mature memory manager:** Currently we implemented a stub allocator which can allocate only in chunks of 8 bytes leading to inefficiency. This could be replaced with a better version which allocated in chunks of 1 byte.

**7.2. Dynamically re-allocable non-volatile memory chunk:** The current version of NV-Heap uses a memory size of 64KB per program. The library allocates the full chuck of the non-volatile memory for the program and maps it to the program’s virtual address space during program initialization. This leads to inefficiency since not all the memory is in use. Ideally, we would want the size to dynamically double when a certain threshold is reached. We’ve reserved some bit in our NV-heap metadata layout to extend this to a dynamically allocable non-volatile memory allocator.

**7.3. Multiple non-volatile memory chunks and shared memory:** A program may need to keep data structures separated on a separate chunk of non-volatile memory or multiple programs may want to share a memory. Our current version support only one contiguous memory chunk per program. Currently, multiple programs can share memory but since locks have not been implemented read-write errors may happen.

**7.4. Thread safety and locking:** Our NV heap implementation doesn’t deal with multi-thread programs. A full feature NV Heap library should be robust against multiple threads and provides non-volatile locks to users. These are our future goals.