A Simulator for None-Volatile Memory

CS652 Course Project Final Report

Shanhe Yi
College of William and Mary
syi@email.wm.edu

Abstract

As the rapid development of persistent storage and memory, None-Volatile Memory(NVM) becomes a critical factor to be taken into consideration for future system design, compiler design etc in terms of performance, how to leverage its properties. This paper proposed and implement a prototype for NVM simulation. The key idea of this simulator is using a big chunk of memory of Dynamic Random Access Memory(DRAM) to simulate the NVM. Corresponding data structures of NVM such as NVRegion, NVRoot, NVSet are implemented to provide compatible experience for upper layer application. All the NVRegion related APIs are implemented. In order to support dynamic memory allocation inside the data region in NVRegion, a NVMalloc and NVFree is implemented using explicit free list. The evaluation is writing two programs to demonstrate the usage and test the APIs' functions. A trace-driven memory allocation performance benchmark is also used to evaluate the memory performance inside the NVM simulator. The example result verifies the NVM simulation design and the system implementation.

Keywords simulation, none-volatile memory

1. Introduction

None-Volatile Memory(NVM) emerges from the rapid development of persistent storage and memory. In Fig.1 a NVM market report from Yole Development[1] shows that in the year of 2016, the NVM market will be more than the twice of current market.

NVM is so attractive due to its good properties such as persistent, byte addressable, short access time and low power consumption. This hardware evolution brings a lot opportunities in markets as well as in engineering. As a fact, NVM has gained more and more attentions not only in the industry but also academia.

However, since the high cost of NVM and limitation of size, it is not convenient and cost efficient to deploy large amount of NVM in related experiments. Here comes a need for a solution which can simulate the NVM and provide similar performance.

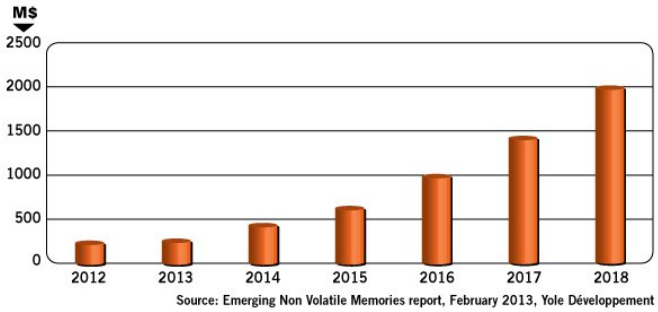

Figure 1. Emerging Non Volatile Memories report

Generally, in this project, we are trying to solve the problem: how to simulate NVM. The solution is to simulate NVM using a chunk of memory. Thus we can get comparable memory performance with little cost. According to the properties of NVM, we can use shared memory to provide global access; we will provide APIs to make it transparent to applications. For the privacy concern, we can provide permission control mechanism. For the consistency, we can use read-write lock. Since we are using DRAM to simulate NVM, an assumption is made that processes won't crash, and there is no power outage. which indicates that in this project, we won't solve this kind of persistence problem.

The rest of this paper is organized as following: In Sec.2 we will introduce some related work. Design of NVM simulator will be discussed in Sec.3. Next, we discuss the system implementation details in 4 and evaluations in 5. Finally, in Sec.6, we conclude this paper and points out some pontential future works.

2. Related Work

There are some related work about the NVM programming interfaces. Mnemosyne[6] is a simple lightweight persistent memory interface. It address two problems: 1) how to create and manage such memory, 2)now to ensure consistency in the presence of failures. Mnemosyne supports both static and dynamic persistent data. It also provides primitives for manipulating those persistent data and keeps consistent updates

[1] `http://www.mram-info.com/`

using a lightweight transaction mechanism. Following evaluation shows that Mnemosyne outperforms persistent design for disks such as BerkeleyDB and Boost serialization. For memory allocation relies on two popular volatile memory allocators: Hoard and dlmalloc[4]. Hoard[2] is a fixed size block allocator and is use for smaller block(less than 8KB) while dlmalloc[3] is unmodified and used for larger block. Mnemosyne utilizes logging to make memory allocations and transaction atmoic. The final implementation and evaluations are also on DRAM

In [2], a lightweight, high-performance persistent object system called NV-heap is implemented. Their motivation is that currently using non-volatile memory just like conventional memory is not a good solution. And neither existing implementation of persistent objects nor the familiar tools are good fit for those memories. They claim that we need new design of persistent data structures creating system, since they were previously restricted by slow speed persistent storage(i.e. disk). NVM can remove those constrains and boost the performance but we need to build a new persistent object system which can harness the performance of those new kind of memories but also ensures safety and robust of application by avoiding dangling pointers, multiple frees and locking errors. They evaluate the new design NV-heap and show that its performance scales with thread. And the data structures in NV-heap can beat BerkelyDB and Stasis significantly. They also investigate the cost for safety guarantees and primitive operations in NV-heap.

There are some good designs in NV-heap can be learned and applied. NV-heap relies on ACID transactions rather than programmer-managed locks to provide robustness. For memory management, NV-heap is using reference counting Whenever a space becomes dead, it will be reclaimed. They also bring in weak pointer to avoiding memory leaks in cycles since weak pointer not affecting reference counts. For pointers, in NV-heap, four new types of pointers are provided to solve the problem of unsafe pointers.

In [3], authors propose a memory abstraction called Software Persistent Memory. In their design, there are two components: Location Independent Memory Allocator(LIMA) and Storage-optimized IO Driver(SID). In LIMA, the Discovery and allocator module moves any data newly made reachable from the root at page level. Write handler tracks updates to those pages at multiple-page chunks. When coming the request, flusher will create persistent pointer and send dirty chunks to SID asynchronously. The SID is an abstract of all kinds of persistent storage. SoftPM can be taken as a kind of "simulation" of NVM using memory and those persistent storage.

NVM simulator is used for evaluation of the proposed file system metadata accelerator(FSMAC) in [1]. It leverages the advantages of Non-volatile Memory to decouple the data and metadata IO, putting data on the disk while metadata in NVM. Since NVDIMM has almost same read/write speed as DRAM, authors use DRAM to simulate NVM in their experiment. Evaluation results demonstrate that this kind of implementation can accelerate the file system up to 49.2 for synchronized IO and 7.22 times for asynchronized IO.

[2] http://www.hoard.org/

[3] http://g.oswego.edu/dl/html/malloc.html

3. Design of NVM Simulator

In this section, we will discuss the design of NVM simulator. First, we will give an overview design of the system, including two type of designs, our decision and rationale. Next, we will follow the structure of NVM to give a detailed design of each part.

3.1 System Design

3.1.1 NVM overview

Before introducing the NVM simulator design, we will briefly describe the NVM logic view. The whole logic view of NVM is showed in Fig.2.

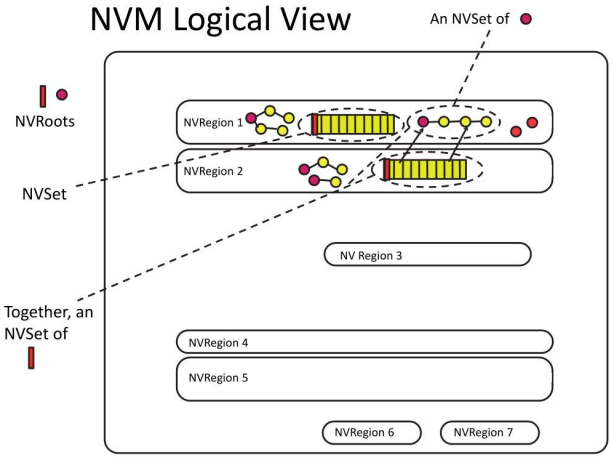

Figure 2. NVM logic view [5]

NVRegion is the basic block in NVM, contains NVRoot and NVSet. Each NVRegion is a consecutive(in virtual address space, not necessarily in physical space) chunk of memory[5]. NVRegion contains one or more NVRoots. NVRoot is defined as a named entity located in the NVRegion, from which, a set of data can be reached through either regular strides(e.g. array elements) or pointer chasing(e.g. linked list, tree, forest). An NVSet may span one or more NVRegions. In Fig.2, there are in total seven NVRegions. In the first region, there are 5 NVRoots, each leads a different NVSet in data structure such as graph, array, linked list and individual variables.

There are two basic designs of NVM. One is centralized, we call it as Type 1, another one is distributed as Type 2.

3.1.2 Type 1

Type 1 design is centralized. Its view is the same as the logic view in Fig.2, where the NVM is an entirely big chunk of shared memory. Each NVRegion is a strip of NVM. Each NVRegion contains multiple type of NVSet and NVRoots.

At the provider side, all the NVM managements should be implemented inside this NVM chunk. For example, the management of current NVRegions needs to be implemented. Since each NVRegion has its own metadata, this will require the provider to maintain lots of stats inside the NVM, which will be extremely complicated when dealing with multiple processes situations.

At the user side, When a process is accessing the NVM, it will map this whole memory chunk to its address space. This kind of mapping is once and for all, there is no further cost for the mapping unless the final unmapping.

3.1.3 Type 2

The type 2 design is distributed at the level of NVRegions. In Type 1, The whole NVM is a piece of shared memory, while in type2, each NVRegion is a piece of shared memory.

At the provider side The management of shared memory will be offloaded to either the kernel(if using System V shared memory) or the file system (if using mmap and munmap). Also, we can directly get some metadata from the system maintained data structures such as file stats and shm stats data structures.

At the user side, the way a user using NVM is also changed, which is called as map-as-you-need. Unless the process needs this NVRegion, there is no need to map this region into the process' address space. This flexibility in the usage pattern can save address space for the process for own use.

3.1.4 Comparison

To compare those two kinds of design, type 2 is simple, light-weight, flexible in management while type 1 is more flexible in function implementations but involving complex implementation and "reinventing the wheel". However, there do have difference in functionalities if choosing different type design.

In current type 2 design, an NVSet may not span more than one NVRegions. If using type 1, this feature can be easily supported since we do not need to maintain relationship between NVRegions because they are all together. Their offsets can be inferred from the size in metadata even if we use offset instead of pointer. However, in type 2, each NVRegion are located distributively. To support NVSet spanning more than one NVRegion, we need to use offset in each NVRegion.

For the pointer in NVM, generally, pointers are not supposed to be used in NVM. The reason is that we can not make sure that each time, the NVM or NVRegion is mounted at exactly the same starting address. Fig.3 illustrates that

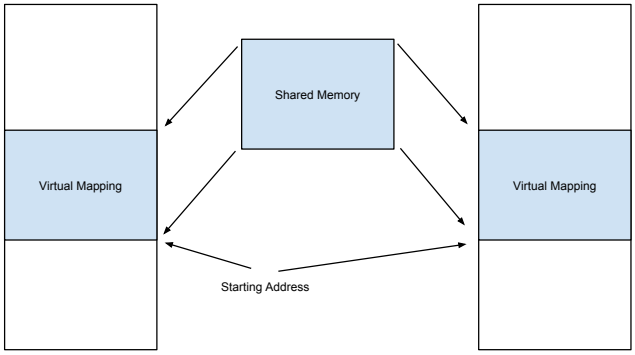

Figure 3. Shared memory mapping

how a shared memory is mapped into different processes' address space. For example, in `shmat`, you should provide an address for the shared memory to attach, otherwise, the kernel will chose a suitable address. This unfixed attach address for shared memory make it impossible to use pointer in those shared memories. Usually, an offset from the base address is used to replace a pointer. However, in the current implementation, we still make assumptions that by carefully choosing the starting address in the process address space, we can meet this requirement.

For the cost of share memory mapping or attaching, in type 1, it is once-and-for-all. However, in type, it is multiple times depending on the needs of the processes. For the metadata of the NVRegions, we need less information in type 2 than type 1 since we can retrieve those information from the kernel in type 2.

Based on those comparisons and tradeoff between complexity and functionalities, we decide to use type 2 design to implement the NVM simulator. Here we summary our assumptions, settings and limitations towards this kind of design: 1)There is no process crash and power outage, 2) an NVSet may not span more than one NVRegions, 3) Each NVRegion should be mapped at the same starting address of process [4]. 4) Shared memory size is currently fixed. 5) Relying on programmer managed locks.

Actually, those limitations are temporal and can be relaxed in our future work. For example, if we replace all the pointer in the NVRegion as offset value, we can relax the limitation such as 3. If we use mmap to backup those memory with files and write through(msync), we may overcome assumption 1 or at least keep consistent and correct.

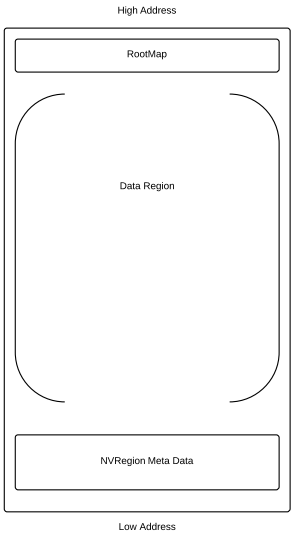

Figure 4. NVRegion Segment

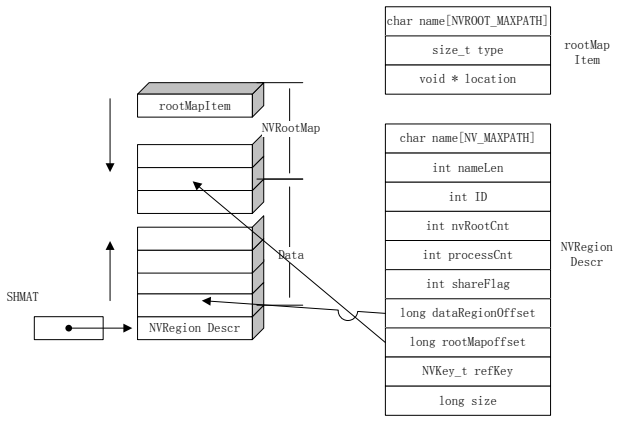

Figure 5. NVRegion Structure

3.2 Detailed Design

3.2.1 NVRegion

As we mentioned before, NVRegion is the base unit of NVM. It has three parts: MetaData(NVRegion Descriptor), the Data Region, the Rootmap Region, as shown in Fig.4. The meta data is stored in the NVRegion Descriptor at the beginning of the NVRegion. Rootmap is a map recording the information of roots in the data region. The NVRegion Descriptor data structure is located at the lower address of NVRegion and grows from bottom to up while the NVRootmap is located at the highest address of NVRegion and grows from top to bottom.

Fig.5 gives the detail data structure of NVRegion Descriptor and rootMapItem. In the NVRegion descriptor, we store two offsets: rootMapOffset and dataRegionOffset. This can be taken as the low heap address and the high heap address. The rootMap grows from the highest address to lower address while the Data Region grows from the lower address to higher address. The NVRegion descriptor data structure definitions are

```
typedef struct NVRDescr_s {
    long size;
    NVKey_t refKey;
    long rootMapOffset;
    long dataRegionOffset; // brk
    int shareFlag;
    int processCnt;
    int nvRootCnt;
    int ID;
    int nameLen;
    char name[NV_MAXPATH];
    // extra meta data for implementation
} NVRDescr,NVRDescr_t;
```

Generally, the metadata includes the size of NVRegions, the shareFlag indicating the sharing status: sharing or exclusive, the reference key for find the shared memory, DataRegion offset to record the brk position, the process counter to count the number of process accessing this NVRegion, the number of NVRoots in this region, the ID of NVRegion, the name length and name for this NVRegion. The APIs related to NVRegion and NVRegion descriptor are NVOpenRegion, NVDeleteRegion and NVCloseRegion. The NVOpenRegion will create or open an NVRegion. Given a NVRegion name, this API will firstly check whether this NVRegion existed. If it exists, NVOpenRegion will just open this region and return its address, otherwise, it will create a region as required.

At the high address of NVRegion is the NVRootMap. NVRootMapItem is the element in NVRootmap. Each NVRootmapItem data structure is like below:

```
typedef struct NVRootmapItem_s {
    void * location;
    // design tag bit
    size_t type;
    char name[NVROOT_MAXPATH];
} NVRootmapItem_t;
```

Currently, there are only three related APIs with NVRoot. We have NVFetchRoot, which get the address of an NVRoot of a given name from a given NVRegion, and NVSetRoot,

[4] For simplicity, we assume that each shared memory will be attach to the same position(`0x80000000(32bit) 0x1000000000L(64bit)`) in the process address space, otherwise, this NVM will not be attached. Thus we can safely use pointer in the NVM, to support more complex data structures such as linked-list, tree etc.

which set a new root in an NVRegion, i.e. recorded in the NVRootmap.

3.2.2 DataRegion and Dynamic Memory Allocation

NVDataRegion is the segment in NVRegion which is used for the data storage. Basically, all the data-related structures will be put into this segment. The Data Region of NVM is similar to the heap segment in a program's address space, since it grows from lower address to higher address and needs its own memory allocator for memory management.

Memory Management To support dynamic memory allocation, we need to 1) track block metadata, 2) search and manage free space and 3) perform allocation.

There are several ways to implement a dynamic memory allocator: naive way, implicit free list and explicit free list.

The naive allocator just allocate memories sequentially. The problem is the utilization of memory which is limited because user can not reuse allocated block. This is not suitable to NVM allocator since NVM needs efficient allocator.

The implicit free list navigates blocks using the size of the block. It can be easily implemented but the allocation is linear time in the number of total blocks. An explicit free list stores size and pointer in free blocks and what we get is actually a double-linked free block list. It can be faster than implicit free list since its allocation is linear time in number of free blocks.

To search and pick for free blocks, there are some heuristics such as first-fit, next-fit and best-fit. First-fit always uses the first free block with its size which is bigger than the required size. Next-fit will looks for free block not from the start of free block list but from where the last allocation happens. So next-fit requires saving the last position. For utilization, intuitively, best-fit will has the best utilization but at the cost of throughput and higher possibility to result in scattering blocks since it is not possible to find the exactly required size free block.

To improve the utilization and avoid different kinds of fragmentation, several improvements are proposed. When the allocated block is much larger than what we required, we will just split it into two blocks: one as used, another one is put back into the free list. By introducing the boundary tag, we can do bidirectional coalescing. The time to do coalesce is critical to the performance of the allocator. For example, we can do coalescing to merge adjacent free block every time after free operation and when there is no enough space.

Previous memory management can be grouped into manual memory management(explicit memory management). It is also possible to use Garbage Collection to implement an automatic memory management(implicit memory management). Generally, To compare those two kinds of memory management, explicit management gives control to the application(programmer) but make the usage prone to errors such as accessing a freed block, multiple-frees and memory leak. Implicit management is easy to use but at the cost of performance and also depends on the GC algorithms.

As required, NVMalloc, NVFree are APIs defined in [5]. NVMalloc allocates required size in a specified NVRegion. NVFree frees the memory held by a data object in the NVRegion.

4. System Implementation

4.1 System V Shared Memory APIs

In order to simulate the NVM and provide it to processes, we rely on the shared memory. Generally, we have serval ways to create a shared memory, among which are System V shared memory APIs such as `shmget, shmat, shmdt, shmctl`, or we can use `mmap, munmap`. In our current implementation, we are using System V shared memory APIs as our preferred method to manipulate shared memory and using the shared memory chunk to simulate the NVM while we provide mmap and munmap as an alternative in case of System V shared memory API not supported in the operating system. By choosing this scheme, we can leverage the convenient shared memory management provided by the kernel. Using `NVOpenRegion` as an example, each nvregion will be backup with a file, by `ftok` we can get a unique key from this file, this key will be directly used as the reference key in NVRDescr. The input name should be the path of the file in file system. If the file is not existed, there will be an errno from `ftok`. For simplicity, the `startingAddr` is fixed. With the key, We can use `shmget` to get `shmid` to test whether this NVRegion exists or not. Additionally, if it does not exist, we can using `shmget` with the IPC_CREAT flag to create a new shared memory in kernel. This shared memory have its own state data structure shm_ds, we can get it by `shmctl` with the IPC_STAT flag.

The shm_ds is:

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* Ownership and permissions */
    size_t          shm_segsz;   /* Size of segment (bytes) */
    time_t          shm_atime;   /* Last attach time */
    time_t          shm_dtime;   /* Last detach time */
    time_t          shm_ctime;   /* Last change time */
    pid_t           shm_cpid;    /* PID of creator */
    pid_t           shm_lpid;    /* PID of last shmat(2)/shmdt(2)
       */
    shmatt_t        shm_nattch;  /* No. of current attaches */
    ...
};
```

while the ipc_perm data structure is:

```
struct ipc_perm {
    key_t          __key;    /* Key supplied to shmget(2) */
    uid_t          uid;      /* Effective UID of owner */
    gid_t          gid;      /* Effective GID of owner */
    uid_t          cuid;     /* Effective UID of creator */
    gid_t          cgid;     /* Effective GID of creator */
    unsigned short mode;     /* Permissions + SHM_DEST and
                                SHM_LOCKED flags */
    unsigned short __seq;    /* Sequence number */
};
```

Some of those information can be directly imported into the NVRDescr structure from above two data structures. If we are using `mmap`, we need construct those meta info by ourselves or get those stats from file descriptor since each mmap-created memory will have a corresponding backing file. Some metadata field can be collected from the file stats data structure.

Identification Generation The ID of NVRegion is directly generated from the shmget. This ID is generated by the kernel and can be used as an unique identification of NVRegions. The refKey is an alternative which we can get it from the ftok.

4.2 MMAP and MUNMAP

MMAP and MUNMAP can also be used to operate the NVRegion, i.e. NVOpenRegion, NVCloseRegion, NVDeleteRegion. The existence of a shared memory is identified by the existence of the backing file. By using file open function and file stats data structure, we can get or set information such as file size, file name, permission, uid, gid, inode, last modified time, etc. Similarity, NVCloseRegion is implemented by MUNMAP, given the address of shared memory and its length, which can be easily got from the NVRegion descriptor.

The implementation of NVDeleteRegion is simpler than the implementation using System V Shared Memory APIs, just deleting the backing file.

Identification Generation If using MMAP and MUNMAP to create, open or close NVRegion, there is no necessary to using ID or RefKey to identify a piece of shared memory. In this case, since the NVRegion Descriptor needs those information, the inode of the memory backing file is used as refKey and the ID is generated by a hashing function from the name of the NVRegion.

4.3 NVRegion-inside Memory Allocator

To implement the NVRegion-inside memory management, we are using explicit free list with splitting, coalescing and first-fit. Our implementation takes the CSAPP course malloclab[5] as a reference. The implementation ensure that the address allocated is aligned to an 8-bytes boundary.

NVMalloc NVMalloc accepts inputs of the specified NVRegion address and the required size. A address in the corresponding NVRegion is returned if there is enough space otherwise it return null. Currently in the implementation, since the NVRegion is fixed size, so the DataRegion is also has a upper bound of size. The reallocation of a NVRegion for a larger size can be our future work. Generally, the NVMalloc will search from the first free block to locate the first suitable free block, set it as used. If the block size is much larger than what we need, splitting will be done to avoid internal fragmentation.

NVFree NVFree frees the corresponding data object as input. The block is set as unused and put back into the free list. We are using a simple coalescing policy which is doing coalescing every time during the free operation to avoid external fragmentation. Usually, after marking the block as freed, it will coalesce adjacent free blocks. Then we insert the free block back to the free list with a FIFO manner.

NVRealloc NVRealloc tries to find a new block which satisfies the new size requirement. Since in our current implementation, the NVRegion size is fixed and there is no method to realloc an NVRegion(This usually involves create a new NVRegion and copy all the data in the old region to this new region which can be our future work.). So this NVRelloc will failed if there is no suitable free block after coalescing, and extent the region to the maximal heap size.

5. Evaluation

In addition to those original required APIs in [5], we also implemented three additional APIs. The whole project has **LoC** (lines of codes) Besides the required APIs, two more dump functions are implemented for debugging and visualization. We also add one more API for better memory management which is named **NVRealloc**.

Performance of Memory Allocator To evaluate the performance of this nvm allocator, we borrow the grade script from the CSAPP course[6] which inputs same traces to both self-implemented allocator and the system memory allocator to get the result. The result is:

```
Using default trace files in traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.000263 21650
 1       yes   92%    5848  0.000206 28416
 2       yes   94%    6648  0.000373 17804
 3       yes   96%    5380  0.000314 17145
 4       yes  100%   14400  0.000289 49861
 5       yes   87%    4800  0.000522  9194
 6       yes   84%    4800  0.000555  8653
 7       yes   55%   12000  0.002910  4123
 8       yes   51%   24000  0.004584  5235
 9       yes   39%   14401  0.000300 47955
10       yes   36%   14401  0.000226 63806
Total          75%  112372  0.010542 10659

Perf index = 45 (util) + 40 (thru) = 85/100
```

Except last two traces which are for realloc, generally, this dynamic memory allocator has a good performance.

Testing Case A testing case is written in this project to test through all the APIs. This test case creates an NVRegion by a given name. The name is actually the path. By defining marco at the global header file, the test case can test the implementation using System V Shared Memory API or the MMAP and MUNMAP. Then an array is NVMalloced in the

[5] `http://csapp.cs.cmu.edu/public/labs.html`

[6] `http://csapp.cs.cmu.edu/`

NVDataRegion. The in a loop, the array is assigned initial value. The NVNewRoot is called to set two roots in the NVRootmap. After that we free this array. The we close this region and delete this region at the end.

Use Cases Next, we write two use cases. This use cases together present the work flow of NVM simulator. In the example 1, it does the following:

1. Create a NVRegion
2. NVMalloc an int arrays(size=100)(array_shm)
3. NVNewRoot sets the array_shm address as the NVRoot.
4. Assignment value to this array
5. Print out the array value
6. Close this region
7. Exit.

Then in the example 2, we do some following up work:

1. Open the NVRegion created in example 1.
2. Sort the int arrays in the NVRegion.
3. Print out the array value
4. Close this region
5. Delete this region
6. Exit.

The example result verifies the NVM simulation design and the system implementation.

6. Conclusion and Future Work

In this project, we design and implement a NVM simulator using DRAM. Corresponding data structures of NVM such as NVRegion, NVRoot, NVSet are implemented to provide compatible experience for upper layer application. All the NVRegion related APIs are implemented. In order to support dynamic memory allocation inside the data region in NVRegion, a NVMalloc and NVFree is implemented using explicit free list. The evaluation is writing two programs to demonstrate the usage and test the APIs' functions. A trace-driven memory allocation performance benchmark is also used to evaluate the memory performance inside the NVM simulator. The example result verifies the NVM simulation design and the system

Besides those future works mentioned previously, there are also some future work in various aspects. The NVRootmap needs better API support such as automatic delete the corresponding root if what the root points is freed. This can be done with a reference counter. To integrate some mature memory allocator such as dlmalloc[4] into our NVM-simulator is also our future work. As stated in [2],"An error in any of these areas will result in permanent corruption that neither restarting the application nor rebooting the system will resolve." How to keep consistency of those NVM even in a erroneous environment is also my future work.

References

[1] J. Chen, Q. Wei, C. Chen, and L. Wu. Fsmac: A file system metadata accelerator with non-volatile memory. 2013.

[2] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 105–118. ACM, 2011.

[3] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software persistent memory. In *Proc. of the USENIX Annual Technical Conf., Boston, MA*, 2012.

[4] D. Lea. Doug leas malloc (dlmalloc).

[5] X. Shen. Nvm simulator api manual. In *CS652, Course Project No.8, Fall*, 2013.

[6] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 91–104. ACM, 2011.