A DAX-enabled Mmap Mechanism for Log-structured In-memory File Systems

Zhixiang Mao*, Shengan Zheng*, Linpeng Huang*, Yanyan Shen*
*Department of Computer Science and Engineering
Shanghai Jiao Tong University
Email: {mzxsusilo, venero1209, lphuang, shenyy}@sjtu.edu.cn

***Abstract*—Emerging byte-addressable Non-Volatile Memory(NVM) technologies offer fine-grained access to persistent data with latency comparable to DRAM. This presents both challenges and opportunities to system software designers. To utilize the fascinating features of NVM, a lot of works have been done to develop NVM-based file systems. Direct access (DAX) is a key feature provided by most NVM-based file systems, it enables applications to directly access NVM without going through the page cache layer. However, DAX-style mmap may cause serious consistency issues in versioning in-memory file systems that adopt CoW to solve block sharing problem among snapshots.**

In this paper, we propose a new mmap mechanism called versioning-mmap which solves the consistency issue of memory-mapped I/O in DAX-enabled versioning in-memory file systems. We implement our proposed mechanism in HMVFS and do a series of experiments based on our implementation. Results show that our mechanism achieves consistency while imposing negligible performance overhead compared to PMFS and NOVA.

I. Introduction

Conventional computer systems usually adopt a two-level storage model, with fast, byte-addressable, volatile DRAM serving as a buffer to cache the working set of programs and files, and slower, non-volatile block storage serving as back-end device to persist data. However, emerging non-volatile-memory(NVM) technologies, such as phase change memory (PCM)[1], spin-torque transfer memory(STT-RAM)[2], and resistive memory(ReRAM)[3], are expected to revolutionize this two-level storage model in the near future. The reason for this insight is that NVM possesses the best features of both DRAM and Disk. Its performance is comparable to DRAM, while the data stored in NVM can persist without power just like disks. This presents a lot of opportunities and challenges for computer architecture designers as well as system software programmers[4][5].

File systems have always been designed and optimized to exploit the characteristics of storage media. For examples, F2FS[6] is designed to fully leverage the usage of the NAND flash media, ramfs[8] and tmpfs[7] are in-memory file systems designed for DRAM to store temporary data. With the development of NVM technology, researchers have been trying to optimize file systems to accommodate the fascinating features that NVM provides. Several works have already been done, such as SCMFS[9], PMFS[14], BPFS[10] and NOVA[15].

Performance and consistency are two key components one should bear in mind when designing a file system. To maximize the performance of NVM-based file system, researchers have proposed a method called XIP(execute-in-place) or DAX(direct access), which bypasses the page cache layer and operates directly on NVM pages. Take memory-mapped I/O as an example, in traditional file systems, the requested page in a memory-mapped file will first be copied to DRAM(page cache) on a page fault and then the buffered page in DRAM will be mapped into application's address space. This is useful in Disk-based file systems since the performance gap between disk and DRAM is huge and disks are not byte-addressable. But when it comes to NVM, page cache seems to be unnecessary or even harmful to the performance because the access latency of NVM is expected to be close to DRAM, and NVM is byte-addressable. Therefore in DAX-mmap, NVM page will be directly mapped to application's address space without copying to DRAM.

As for consistency, snapshotting is a well-known method. Snapshotting provides strong consistency by enabling users to reverse the entire file system to a previous consistent state on a system crash. It is widely adopted by many modern file systems, such as BTRFS[11] and NILFS[12] and it has been proven to be an efficient consistency mechanism which is suitable for NVM-based file systems in HMVFS[17]. The implementation of a snapshotting file system is a little bit tricky because consecutive snapshots usually share a lot of data, and an efficient way of managing those shared data is required. A commonly used approach is Rodeh's hierarchical reference counting mechanism[13], which is adopted by both HMVFS and BTRFS. Rodeh's hierarchical reference counting mechanism is based on the idea of copy-on-write(CoW) friendly B-tree, which means write a file block shared by the previous snapshots will trigger a CoW operation.

To design a high-performance NVM-based file system, one would suggest a DAX enabled versioning file system. However, DAX-style mmap mechanism may cause serious consistency issue in versioning file systems. Consider the following scenario: 1) a file is memory-mapped and a data block in the file is accessed by store operation in the current version, 2) a snapshot is created, and the data block is written in the new version by file system write operation. Step 1 will create a PTE(page table entry) in application's address space that directly maps a virtual address to the PFN(page frame number) of the data block. In step 2, the write operation to the memory-mapped file will trigger a CoW operation since

978-1-5090-6468-7/17/$31.00 ©2017 IEEE

the target data block is shared by both current and previous version. After the CoW operation finishes, the write and read operations will be performed on the newly allocated data block. However, the load and store operations to the data blocks in the memory-mapped file are still performed on the old data block since the page table entry is not updated along with the read or write operations. As a result, memory-mapped I/O and file system read/write operations operate on different files, thus the file system is in an inconsistent state.

To address this problem, we propose versioning-mmap, a new mmap mechanism designed for versioning in-memory file systems which records the mapping between application's address space and file data blocks on page faults and remaps the memory-mapped data blocks during the creation of a snapshot. We propose two remapping methods: passive remapping and initiative remapping. **Passive remapping** invalidates the page table entries(PTE) created in the last version for memory-mapped I/O and leave the page fault handler to handle the consistency issue. **Initiative remapping** creates a new copy of the memory-mapped data block and swaps the version between the old and the copied data blocks so that the memory-mapped data block is always exclusively owned by the current version. Passive remapping introduces extra page faults which result in a performance degradation of memory-mapped I/O, while initiative remapping introduces overhead to snapshot creation. To achieve a balance between memory-mapped I/O and snapshot creation performance, we propose an adaptive remapping method, which dynamically chooses which method to be used based on how frequently the data block is accessed.

In designing and implementing versioning-mmap, we make following contributions:

- We discover a consistency issue related to DAX-style mmap mechanism in some versioning in-memory file systems.
- We propose a new mmap mechanism which solves the problem we've discovered. To the best of our knowledge, we are the first to integrate DAX-style mmap into versioning in-memory file system with consistency guarantee.
- We implement our proposed mechanism in HMVFS and conduct a variety of experiments based on our implementations. Experiments show that versioning-mmap incur minor performance overhead.

The remainder of the paper is organized as follows. Section 2 gives some background knowledge and thorough explanation of our motivation. In section 3, we describe the design and implementation details. In section 4, we evaluate our proposed mechanism. Section 5 lists some related works and finally in section 6, we conclude our work.

II. Background and Motivation

A. HMVFS Overview

HMVFS is a versioning file system designed for NVM, it achieves space-efficient snapshotting with little impact on I/O performance by using so called Stratified File System Tree(SFST). In HMVFS, each version of the file system is a branch of SFST, and data is shared between versions by sharing subtrees of SFST.

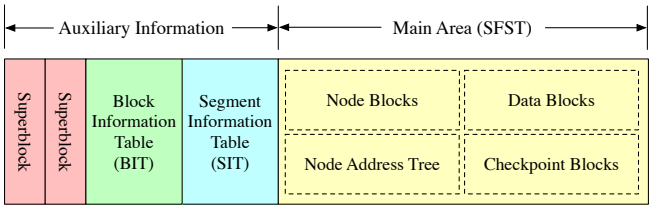


Fig. 1. Physical Layout of HMVFS

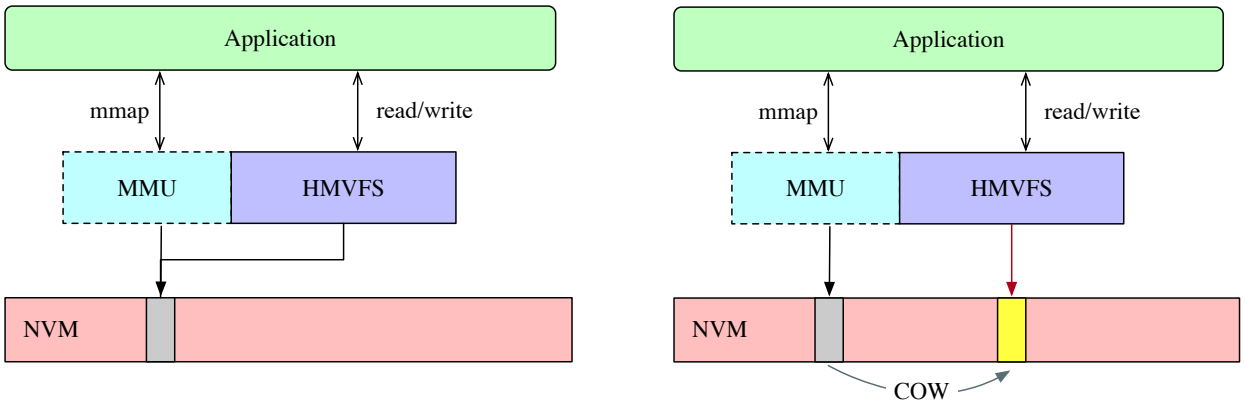
1) Physical Layout: The physical layout of HMVFS is shown in figure 1. As is illustrated in the picture, the entire NVM space managed by HMVFS is divided into two area: auxiliary information and main area. NVM space in the main area is further divided into blocks(NVM pages) with size of 4KB. A block in the main area is either one of the following four types:

- **Data Block** stores the actual file data.
- **Node Block** contains pointers which point to data blocks (if it is a direct node block) or point to a lower level node block (if it is an indirect node block).
- **NAT Block** is a node of NAT(Node Address Tree), which is an extended version of Node Address Table implemented in F2FS. It is used to convert a node id to a physical address in NVM.
- **Checkpoint Block** contains the checkpoint information of a snapshot.

Auxiliary information area stores some supervisory information about HMVFS. **Super Block** contains the overall management information of the entire file system. And **Block Information Table** and **Segment Information Table** respectively store the status of blocks and segments in the main area.

2) Node Address Tree: Unlike traditional file structure in which blocks are indexed by physical address, HMVFS indexes all blocks except data blocks by NID(node id). So in HMVFS, same inode number may appear in different versions, yet they are stored in separate locations. This is possible because of the adoption of NAT. NAT is actually a multi-version node address table which translates NIDs to physical addresses in NVM. Each NAT root in SFST gives a separate virtual NVM address space, thus indicate a version of the file system. Since the translation between NIDs and physical addresses happens frequently, HMVFS caches NAT in DRAM.

3) Snapshotting: When a snapshot is to be taken, HMVFS first flushes all dirty NAT entries cached in DRAM to form a new NAT in NVM and then writes a checkpoint block. Finally, HMVFS modifies the checkpoint pointer stored in superblock to this checkpoint. Once the snapshot is taken, all data belongs to the previous snapshot becomes read-only, and a CoW operation will be triggered if an application tries to write the data.



(a) Before Snapshootting (b) After Snapshotting

Fig. 2. Inconsistency caused by DAX-style mmap int HMVFS

B. Motivation

Suppose a file is memory-mapped with DAX-style mmap in HMVFS. Access to the file will result in page faults, and links between virtual memory addresses and the physical addresses of the file data blocks will be added to application's address space. After that, load/store or read/write operations will all be performed directly on the same data blocks. Figure 2a illustrates this situation.

However, once a snapshot is taken, write to the data blocks that belong to the previous version will trigger CoW operations which result in the following accesses in current version to be performed on newly allocated data blocks. As is shown in figure 2b, if a file is memory-mapped and accessed in previous version and written in the current version, the read/write operation will be performed on the newly allocated data blocks, while load/store operation will still be performed on the old data blocks that belong to the previous version since page table entries remain unchanged after CoW operations. As a result, load/store and read/write to the same data block operate on different physical data blocks, leaving the file system in an inconsistent state.

Besides HMVFS, other in-memory file systems that adopt CoW will face a similar problem if DAX-style memory-mapped I/O is enabled. For example, NOVA, a log-structured file system designed for hybrid volatile/non-volatile memory system, uses CoW to update file. Data blocks in NOVA will be copied to new locations whenever been written while memory-mapped I/O operates on fixed data blocks.

III. Design and Implementation

In this section, we provide design and implementation details of versioning-mmap in HMVFS. Even though our mechanism aims to solve the consistency issue of memory-mapped I/O between snapshots in versioning in-memory file systems, it can be easily adopted by other DAX-enabled log-structured in-memory file systems, such as NOVA, with little modification to solve the similar consistency issue as we discussed before.

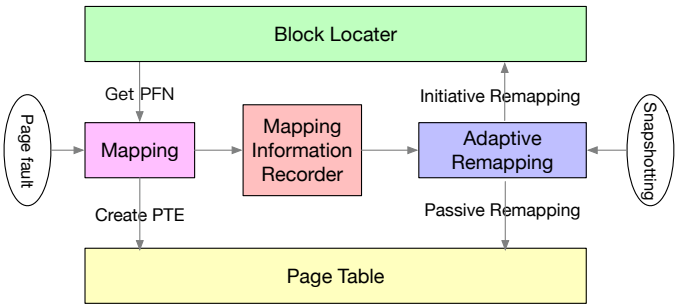


Fig. 3. Architecture

A. Overview

As is shown in figure 3, our overall mechanism can be roughly divided into four modules: *block locator* module, *mapping* module, *mapping information recorder* module and *adaptive remapping* module.

Block locator module is responsible for the acquisition of page frame number(PFN) given the page offset in a file. *Mapping information recorder* module records the mapping information between application's virtual memory address space and NVM address space. *Mapping* module handles page faults, it obtains PFN from Block Locater, adds mapping information to Mapping Information Recorder, and finally, creates page table entry(PTE). *Adaptive remapping* module is called during creation of a snapshot to maintain consistency between snapshots for memory-mapped I/O. It reads mapping information from *mapping information recorder*. For every recorded memory-mapped page, it checks whether the page is frequently accessed among snapshots or not. If the page is not frequently accessed, passive remapping method is adopted, otherwise, initiative remapping is used. Passive remapping is done by finding and invalidating the PTE so that access to the page after a snapshot is taken will re-trigger a page fault,

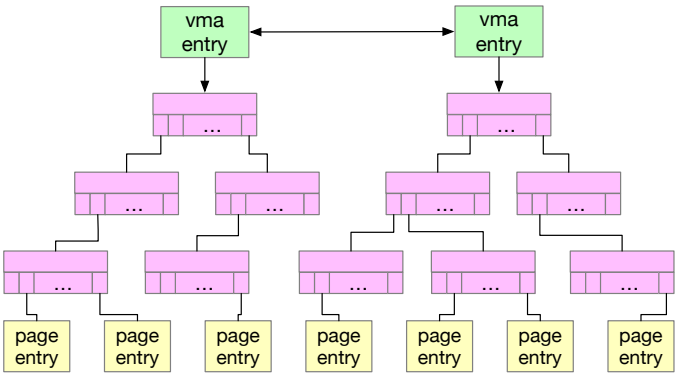


Fig. 4. Mapping Data Structure

and a CoWed data block which belongs to current version exclusively will be memory-mapped. Initiative remapping does not alter the page table, it creates a copy of the data block, and swap the pointers in corresponding direct node blocks so that the old data block which belongs to the last version now belongs to the current version, and the newly allocated data block is reassigned to the last version.

B. Block Locater

To establish the mapping between the memory mapped virtual memory area and the file data blocks in NVM, we must be able to find the page frame number of a data block given its page offset in the file. So we implemente a procedure called *get_xip_block()* which accepts a inode pointer and a page offset as its arguments and returns the PFN of the corresponding data block.

get_xip_block() first tries to locate the data block by following the links of node blocks. A typical searching path would be: inode $\rightarrow$ indirect node $\rightarrow$ direct node $\rightarrow$ data. As we mentioned before, node blocks are indexed by NIDs instead of physical block address. So traversing the links would require the translation between NIDs and the physical block address, which is done by looking up NAT.

After the physical address of the data block is obtained, we check the version number of the page which is stored in Block Information Table. If the version number of the page is equal to the current version number, then the PFN of this data block is returned. If not, which means the data block is shared among previous versions, we copy the page to a new location and modify the pointer in its parent direct node block to point to the new page and the PFN of the newly allocated data block will be returned. This ensures that the data block obtained by *get_xip_block()* is exclusively held by current version.

C. Mapping Information Recorder

Mapping information are recorded during handling page faults to provide useful information to do remapping when creating a snapshot. Initiative remapping needs to find the direct node blocks, so the inode number of the file to which the memory-mapped data block belongs and the page offset in the file are needed. Meanwhile, passive remapping needs to find the PTE which is identified by a page table and a virtual memory address in that page table. So there are four pieces of information we need to record in handling page faults:

- inode of the memory-mapped file
- page offset of accessed file block
- page table pointer
- virtual memory address in the page table

To efficiently record these information, we designed a data structure which is shown in figure 4. There are two key components in our Mapping Data Structure: `vma_entry` and `page_entry`. `vma_entry` holds a pointer that points to the memory-mapped virtual memory area (VMA), thus indicates a memory-mapped file. `page_entry` represents the memory-mapped data blocks in the file. The definition of `page_entry` is given as follow:

```
typedef enum {INITIATIVE, PASSIVE} remapping_method;
struct hmvfs_mmap_page_entry {
      unsigned long pgoff;
      int access_cnt;
      remapping_method rm_method;
};
```

- **pgoff** is the page offset of this data block within the file.
- **access_cnt** denotes how frequently the page is accessed by memory-mapped I/O among snapshots. `access_cnt` is increased by 2 on every page fault, and decrease by 1 when a snapshot is taken. So if a page fault happens in the last snapshot, `access_cnt` will be increase by 1, otherwise decrease by 1. The larger the `access_cnt` is, the more frequently the page is accessed among snapshots.
- **rm_method** is the remapping method indicator. Its value is either INITIATIVE which indicates the page will be remapped with initiative remapping method or PASSIVE which indicates the page will be remapped with passive remapping method.

`page_entry`s belonging to the same memory-mapped file are indexed using a radix tree to enable fast lookup. The tree root is stored in the corresponding `vma_entry`, and `vma_entry`s are organized in a doubly linked list.

Now we describe how required information is fetched from Mapping Data Structure. Given a `vma_entry` and a `page_entry`, page table can be found by following references:

```
struct mm_struct* mm = vma->vm_mm;
pgd_t* pgd = mm->pgd;
```

the virtual memory address can be calculated as follow:

```
offset = (pgoff - vma->pgoff) * PAGE_SIZE;
vaddr = vma->vm_start + offset;
```

the inode of the memory-mapped file and the page offset in the file can be obtained by:

```
inode =vma_entry->vma->vm_file->f_mapping->host;
page_offset = page_entry->pgoff;
```

Algorithm 1 Page Fault Handling Routine

```
1: function HMVFSDOPAGEFAULT(vma, vmf)
2:     pgoff = vmf→pgoff
3:     inode = vma→vm_file→f_mapping→host
4:     pfn = get_xip_block(inode, pgoff)
5:     vaddr = vmf→virtual_address
6:     if vma is mapped with MAP_SHARED flag then
7:         AddMappingEntry(vma, pgoff)
8:     end if
9:     add mapping (vaddr, pfn) to page table
10: end function
11: function ADDMAPPINGENTRY(vma, pgoff)
12:     vma_entry = find vma entry in vma entry list
13:     if vma is not found in the vma entry list then
14:         vma_entry = create a new vma entry
15:     end if
16:     page_entry = find page entry in radix tree
17:     if page entry not found then
18:         page_entry = create a new page entry
19:         page_entry→rm_method = PASSIVE
20:         page_entry→access_cnt = 0
21:     end if
22:     page_entry→access_cnt += 2
23: end function
```

D. Mapping

Linux uses demand paging to manage virtual memory. So the mapping between a memory-mapped virtual memory area and NVM address space will not be created until accessed. When an application tries to operate on a page in memory-mapped virtual memory area for the first time, a page fault will be triggered and the handling of the page fault will be delegated to the *fault()* method defined in VMA operations (*struct vm_operation_struct*) which is registered to VMA during handling of *mmap* system call. So the core logic of *mapping* module is implemented in *fault()* method.

Our page fault handling procedure, illustrated in Algorithm 1, starts by getting the PFN of the target data block with `get_xip_block()`. It then checks if the map is shared by testing if VM_SHARED flag is set in `vma->vm_flags`. If the file is privately mapped, all pages in the VMA will be marked as CoW page, which means the application will hold an exclusive copy of the mapped content, thus no consistency issue will occur. Therefore we will not record the mapping information if the file is mapped privately. However, if a file is mapped with MAP_SHARED flag, consistency issue would arise as we discussed before, so the mapping information will be recorded. The recording job is done by `AddMappingEntry()`.

`AddMappingEntry()` first iterates the `vma_entry` list to find the `vma_entry` that points to the given VMA. If it is not found, a new entry will be created and initialized. It then searches the radix tree with page offset as the key to find the corresponding `page_entry`. If not found, a new `page_entry` will be created. `rm_method` will be initialized to be PASSIVE for newly created `page_entry` which means the default remapping method is passive remapping. Finally, it increases the `access_cnt` by 2.

After all the above job is done, we create a page table entry in current application's address space with `vm_insert_mixed()`. Since the PFNs of the mapped data blocks are obtained by `get_xip_block()` and `get_xip_block()` copies data blocks that are shared with previous versions to new locations, memory-mapped I/O and file system read/write operations are guaranteed to be performed on same data blocks in current version.

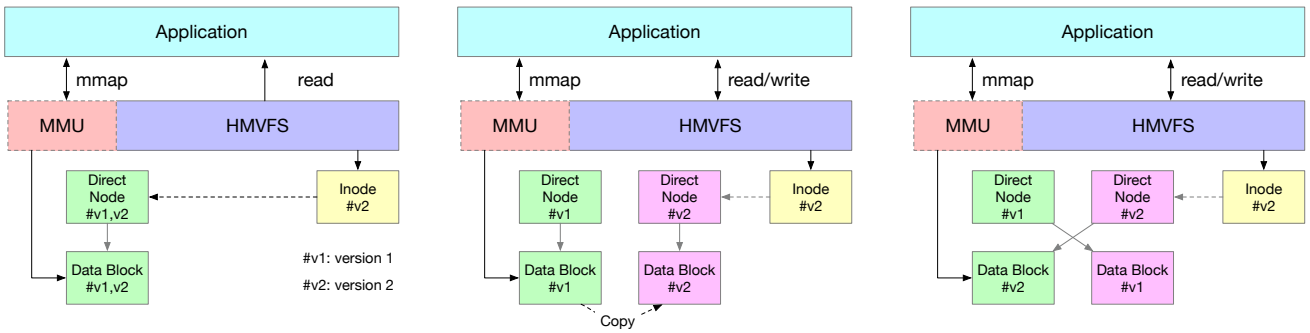
E. Adaptive Remapping

Algorithm 2 Adaptive Remapping Procedure

```
1: for each vma_entry and page_entry do
2:     if page_entry→access_cnt > 0 then
3:         page_entry→access_cnt -= 1
4:     end if
5:     if page_entry→access_cnt ≥ I_THRESHOLD then
6:         page_entry→rm_method = INITIATIVE
7:     end if
8:     if page_entry→access_cnt ≤ P_THRESHOLD then
9:         page_entry→rm_method = PASSIVE
10:    end if
11:    if page_entry→method == INITIATIVE then
12:        InitiativeRemapping(vma_entry, page_entry)
13:    else
14:        PassiveRemapping(vma_entry, page_entry)
15:    end if
16: end for
```

In order to maintain consistency between snapshots for memory-mapped I/O, a remap job must be done when creating a snapshot. We propose two remapping methods: passive remapping and initiative remapping.

passive remapping is done by invalidating the PTE established in the last version. Given a `vma_entry` and a `page_entry`, the page table and the virtual memory address can be obtained as described in subsection B. By traversing the page table, we can then find and delete the target page table entry. Once the snapshot is created, passive remapping ensures that all page table entries related to shared memory-mapped I/O have been destroyed. So operate on the memory mapped virtual memory area for the first time in current version will always trigger a page fault. The page fault handling routine, described in the last section, will either copy the data block to a new location and map the newly allocated data block to application's address space if the page is not yet been written in the current version or map the data block directly to application's address space if the block has already been written in the current version. Either of that will ensure both memory mapped I/O and file system read/write operations operate on the same data block. Therefore the consistency is guaranteed.



(a) Initial state (b) Create a new copy (c) Swap pointers in direct node block

Fig. 5. Initiative Remapping

Initiative remapping is done by copying the memory-mapped data block to a new location to form a new version of the data block which is exclusively held by current version and swap the pointers in the direct node blocks of the two version. Figure 5 gives a demonstration of initiative remapping process. In figure 5a, the data block is memory-mapped and accessed in the last version, and the snapshot is created. At this time, the data block is shared by last and current version. If the snapshot creation process stops here, write operation to the data block through HMVFS will trigger a CoW, which results in the load/store operation through memory-mapped I/O and read/write operation via HMVFS to be performed on different data blocks. To prevent this from happening, initiative remapping creates a copy of the memory-mapped data block, as shown in figure 5b, and exchange the data block pointers in the direct node blocks of the two version, shown in figure 5c. This ensures that the memory-mapped data block is always exclusively owned by the current version. Therefore no CoW will be triggered on memory-mapped data blocks. The consistency issue is thereby eliminated.

Passive remapping will degrade the performance of memory-mapped I/O because more page faults will be triggered, and handling page faults is quite time-consuming. Meanwhile, the snapshotting performance will drop if initiative remapping is adopted. To achieve a balance between memory-mapped I/O and snapshot creation performance, we propose an adaptive remapping method, which dynamically chooses whether passive or initiative remapping method should be used based on how frequently the memory-mapped page is accessed among snapshots. If a page is frequently accessed, it is very likely to be accessed in next version, so initiative remapping is adopted to avoid page fault. If a page is not frequently accessed, we can assume the page will unlikely be accessed in next version, so passive remapping is used to avoid unnecessary block copying.

The adaptive remapping procedure is further described in Algorithm 2. As we mentioned in subsection B, `access_cnt` indicates how frequently the memory-mapped page is accessed. The value of `access_cnt` is increased by 2 on every page fault and decreased by 1 when a snapshot is created. The default remapping method is passive remapping and is switched to initiative remapping when `access_cnt` reaches a threshold I_THRESHOLD. When the remapping method of a page is set to initiative remapping, access to the page after the creation of a snapshot will not trigger page fault and `access_cnt` will drop by 1 on every snapshot. Once the `access_cnt` drops to a threshold P_THRESHOLD, the remapping method is then reset to passive mapping. The reason of the adoption of P_THRESHOLD is, if the page is considered as frequently accessed, we assume that in the next I_THIRESHOLD − P_THRESHOLD versions, the page will likely be accessed.

IV. Evaluation

In this section, we evaluate our mmap mechanism based on our implementation in HMVFS. We first briefly introduce our experimental setup. Then we compare the performance of proposed mechanism against several state of art NVM-based file systems. Finally, we study the impact of I_THRESHOLD.

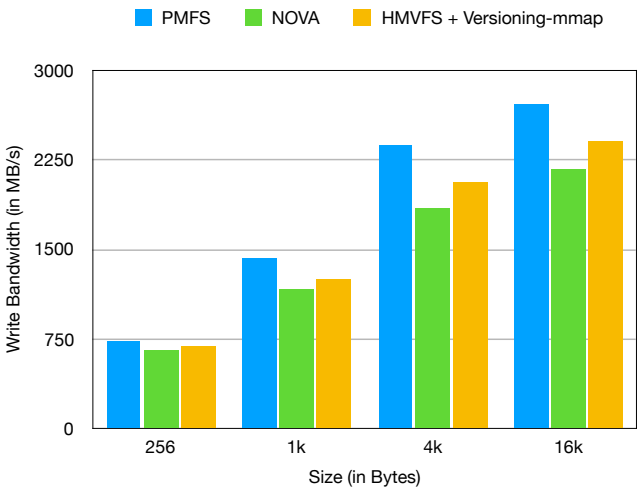
A. Experimental Setup

Our experiments are conducted on a Supermicro X10DRH-I-O server with NVDIMM support. The server has two Intel Xeon E5 processors, each processor runs at 3.4GHz, has six cores and supports 8 DDR4 Channels. The total memory capacity is 32GB, with 16GB RDIMM DRAM and 16GB NVDIMM. The entire NVDIMM is reserved for file systems by using the grub option `memmap`.

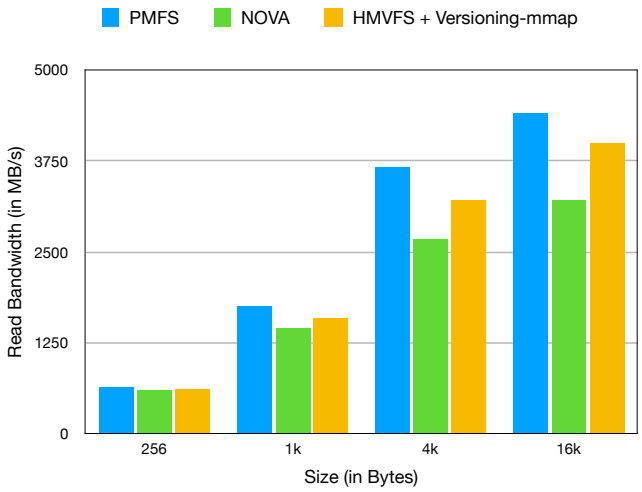
B. Memory-mapped I/O performance

We use fio[16] benchmark to compare the performance of memory-mapped I/O in our implementation against PMFS and NOVA. We test the random read and write performance of memory-mapped I/O with different size.

The results can be found in figure 6. As is illustrated in the figure, both HMVFS and PMFS outperform NOVA. This is reasonable because both HMVFS and PMFS enable applications to directly access NVM without going through the page cache layer, while NOVA uses `atomic-mmap` to support strong consistency. `atomic-mmap` allocates replica NVM pages and copies the file data to replicated pages when

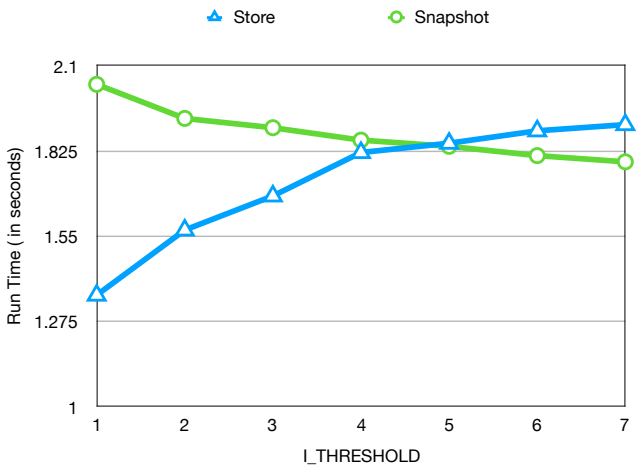


(a) Random Write

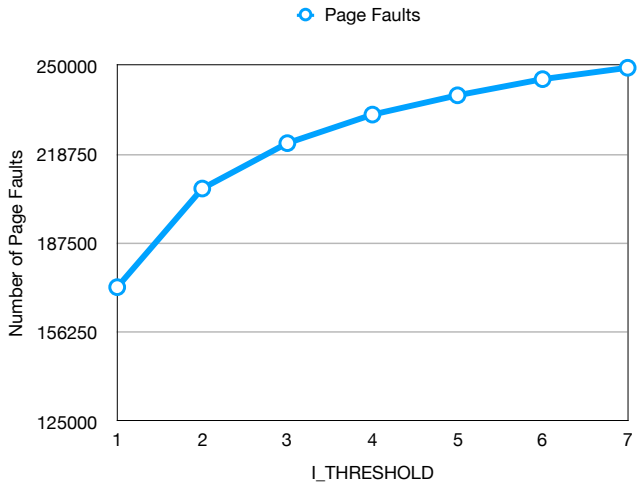


(b) Random Read

Fig. 6. Memory-mapped I/O performance



(a) Store and Snapshot Time



(b) Number of Page Faults

Fig. 7. Impact of I_THRESHOLD

a file is memory-mapped, read and write operations will be performed on replicated pages instead of original pages, and dirty replicated pages will be written back when `msync` is called. Compared to PMFS, memory-mapped I/O in HMVFS is about 10% slower, this overhead is mainly caused by recording mapping information in handling page faults.

C. Impact of I_THRESHOLD

I_THRESHOLD is the threshold on which we switch remapping method to initiative remapping. When the remapping method of a page is switched to initiative remapping, all remapping job will be done during snapshotting and no page fault will happen on this page in the next I_THRESHOLD−P_THRESHOLD versions. To study how I_THRESHOLD would affect the performance of memory-mapped I/O and snapshotting, we map a file with the size of 64MB and perform random writes on the mapped file. For a fixed I_THRESHOLD, we perform 30 rounds of writing with a snapshot created for each round, time spent on store operations and snapshotting are recorded as well as the number of page faults during the whole process. Figure 7a gives a demonstration of our result. Note that P_THRESHOLD is set to I_THRESHOLD − 1 in our experiment.

As is shown in figure 7(a), the time consumed by store operations to the memory-mapped file is increasing as I_THRESHOLD goes up while time spent on snapshotting is dropping. The reason for this is, with the increasing of I_THRESHOLD, memory-mapped pages need to be accessed more frequently among snapshots in order to switch remapping method to initiative remapping, which result in more pages to be remapped with passive remapping method and more page faults to be triggered, as shown in figure 7b. Both store and snapshot time approaches a fixed value as I_THRESHOLD in-

creased because our adaptive remapping method will degrade to passive remapping if a large enough I_THRESHOLD is applied.

V. Related Work

PMFS[14] is a light-weight POSIX compliant file system optimized for NVM. It enables applications to directly access NVM without going through the block layer. The implementation of memory-mapped I/O in PMFS is quite different from those in traditional file systems. In traditional file systems, memory-mapped I/O would first copy the mapped content of the file into DRAM(page cache) on a page fault, and then map the DRAM pages that hold the copy of the mapped content into application's address space. However, in PMFS, the NVM pages are directly mapped into application's address space without copying them into DRAM. PMFS uses copy-on-write to guarantee the consistency of file data. Since file data will be copied to another location on every write and memory-mapped I/O operate directly on NVM pages, PMFS will face a similar problem we mentioned in this paper if a file is written by memory-mapped I/O and file system write operation at the same time.

NOVA[15] is a log-structured file system designed for hybrid memory systems. The main goal of NOVA is to provide strong consistent guarantee while preserving the performance. To achieve consistency for memory-mapped I/O, NOVA proposed so called `atomic-mmap` mechanism, which is similar to the method implemented in traditional file systems. When applications try to map a file into its address space, NOVA first makes a copy of the mapped content in NVM, and then maps the copied NVM pages to applications address space. When `msync` is called, NOVA then writes the copied NVM pages back to the memory-mapped file. In our implementation, consistency of memory mapped I/O is also guaranteed. Since HMVFS is periodically checkpointed, and memory-mapped file is guaranteed to be consistent between checkpoints by our proposed versioning-mmap mechanism. We just need to mount the last checkpoint on recovery from a sudden system failure and remap the file. Whether file system should provide failure-atomic mmap is still in debate, some researchers think that memory-mapped I/O is too low level, hence they proposed some higher level libraries, such as NV-Heap[21], Mnemosyne[22], and NVML[20], to manage NVM by memory-mapped I/O.

DAX[18] is a set of programming interfaces provided by Linux kernel to support direct access to NVMs. It is supposed to be the standard way of doing direct accessing to NVMs in Linux since kernel verison 4.0. Several state of art file systems, such as ext2, ext4, and xfs have already supported DAX. Since our implementation relies on HMVFS which is built on kernel version 3.10, we don't use the standard DAX interface, but the idea behind is similar.

VI. Conclusion

DAX is a feature enabled in most NVM-based file systems to bypass the page cache. However, DAX-style mmap may cause serious consistency issue in versioning file systems that utilize CoW to solve block sharing problem. In this paper, we design a mmap mechanism that aims to solve this consistency issue. We propose an adaptive remapping method to remap the memory-mapped pages during the creation of a snapshot. We implement our proposed mmap mechanism in HMVFS and conduct a series of experiments based on our implementation. Experiment results show that our mechanism can achieve consistency with little performance overhead.

VII. Acknowlegement

This work is supported by the National High Technology Research and Development Program of China (No.2015AA015303) and the National Natural Science Foundation of China (No.61472241).

References

[1] Wong, H. S. P., Raoux, S., Kim, S., Liang, J., Reifenberg, J. P., Rajendran, B., ... & Goodson, K. E. (2010). Phase change memory. Proceedings of the IEEE, 98(12), 2201-2227.

[2] Chen, E., et al. "Advances and future prospects of spin-transfer torque random access memory." IEEE Transactions on Magnetics 46.6 (2010): 1873-1878.

[3] Strukov, Dmitri B., et al. "The missing memristor found." nature 453.7191 (2008): 80-83.

[4] Liu, Ren-Shuo, et al. "NVM Duet: Unified working memory and persistent store architecture." ACM SIGARCH Computer Architecture News 42.1 (2014): 455-470.

[5] Ren, Jinglei, et al. "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems." Microarchitecture (MICRO), 2015 48th Annual IEEE/ACM International Symposium on. IEEE, 2015.

[6] Lee, Changman, et al. "F2FS: A New File System for Flash Storage." FAST. 2015.

[7] Snyder, Peter. "tmpfs: A virtual memory file system." Proceedings of the autumn 1990 EUUG Conference. 1990.

[8] McKusick, Marshall K., Michael J. Karels, and Keith Bostic. "A Pageable Memory Based Filesystem." USENIX Summer. Vol. 52. 1990.

[9] Wu, Xiaojian, and A. L. Reddy. "SCMFS: a file system for storage class memory." Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011.

[10] Condit, Jeremy, et al. "Better I/O through byte-addressable, persistent memory." Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009.

[11] Rodeh, Ohad, Josef Bacik, and Chris Mason. "BTRFS: The Linux B-tree filesystem." ACM Transactions on Storage (TOS) 9.3 (2013): 9.

[12] Konishi, Ryusuke, et al. "The Linux implementation of a log-structured file system." ACM SIGOPS Operating Systems Review 40.3 (2006): 102-107.

[13] Rodeh, Ohad. "B-trees, shadowing, and clones." ACM Transactions on Storage (TOS) 3.4 (2008): 2.

[14] Dulloor, Subramanya R., et al. "System software for persistent memory." Proceedings of the Ninth European Conference on Computer Systems. ACM, 2014.

[15] Xu, Jian, and Steven Swanson. "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories." FAST. 2016.

[16] Axboe, Jens. "Flexible io tester." https://github. comlaxboe/fio (2015).

[17] Zheng, Shengan, et al. "Hmvfs: A hybrid memory versioning file system." Mass Storage Systems and Technologies (MSST), 2016 32nd Symposium on. IEEE, 2016.

[18] DAX: https://www.kernel.org/doc/Documentation/filesystems/dax.txt

[19] XIP: https://lwn.net/Articles/135472/

[20] nvml: https://github.com/pmem/nvml/

[21] Coburn, Joel, et al. "NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories." ACM Sigplan Notices 46.3 (2011): 105-118.

[22] Volos, Haris, Andres Jaan Tack, and Michael M. Swift. "Mnemosyne: Lightweight persistent memory." ACM SIGARCH Computer Architecture News. Vol. 39. No. 1. ACM, 2011.