

Wear-aware Memory Management Scheme for Balancing Lifetime and Performance of Multiple NVM Slots

Chunhua Xiao*

*School of Computer Science
Chongqing University
Chongqing, China*

Corresponding author: xiaochunhua@cqu.edu.cn

Linfeng Cheng

*School of Computer Science
Chongqing University
Chongqing, China*

linfeng_cqu@cqu.edu.cn

Lei Zhang

*School of Computer Science
Chongqing University
Chongqing, China*

zhanglei51@cqu.edu.cn

Duo Liu

*School of Computer Science
Chongqing University
Chongqing, China
liuduo@cqu.edu.cn*

Weichen Liu

*School of Computer Science and Engineering
Nanyang Technological University
Singapore
wchliu@gmail.com*

Abstract—Emerging Non-Volatile Memory (NVM) has many advantages, such as near-DRAM speed, byte-addressability, and persistence. Modern computer systems contain many memory slots, which are exposed as a unified storage interface by shared address space. Since NVM has limited write endurance, many wear-leveling techniques are implemented in hardware. However, existing hardware techniques can only be effective in a single NVM slot, which cannot ensure wear-leveling among multiple NVM slots.

This paper explores how to optimize a storage system with multiple NVM slots in terms of performance and lifetime. We show that simple integration of multiple NVMs in traditional memory policies results in poor reliability. We also reveal that existing hardware wear-leveling technologies are ineffective for a system with multiple NVM slots.

In this paper, we propose a common wear-aware memory management scheme for in-memory file system. The proposed memory scheme enables wear-aware control of NVM slot use which minimizes the cost of performance and lifetime. We implemented the proposed memory management scheme and evaluated their effectiveness. The experiments show that the proposed wear-aware memory management scheme can outperform wear-leveling effect by more than 2600x, and the lifetime of NVM can be prolonged by 2.5x, the write performance can be improved by up to 15%.

Index Terms—Wear-aware, Memory Management, Lifetime, Performance, Multiple NVM slots

I. INTRODUCTION

For decades, emerging Non-Volatile memory (NVM) storage technologies, such as phase change memory [1] [34],

spin-torque transfer [35], resistive memories [36] and Intel 3D XPoint [2], are radically new and highly promising. NVM combines performance and byte addressability of DRAM with the persistence of traditional storage devices like flash (SSD). There are some advantages of NVM, that is high density, low cost, persistence and byte addressable. While these properties make NVM highly promising, many approaches have been proposed to integrate NVM into computer systems [3] [4] [5] [38] [39] [40] [41] [42].

With the rapid development of NVM, the new in-memory file systems designed for NVM have emerged, such as SIMFS [6], PMFS [7], NOVA [8]. As the middle layer between user application and storage device, the file system provides a unified and fast storage interface. Traditional file system is designed for block devices. It mainly improves the performance through data structure and I/O scheduling, and ensures data consistency through logging. The performance of these emerging file systems based on NVM is greatly improved compared to the traditional disk file system.

Unfortunately, NVM has limited write endurance and it can quickly worn out by unprotected writes. For example, a unit of PCM can tolerate $10^8 - 10^{12}$ writes at most. To address this problem, it is common practice to use wear-leveling technology to ensure a single NVM lifetime. The existing researches on wear leveling technology for NVM are implemented in hardware [9] [10] [11] [12]. Hardware wear leveling usually utilize an additional controller to achieve wear leveling. However, the capacity limiting of single NVM calls for a storage system consisting of multiple NVM slots, the most advanced wear leveling techniques are only effective in ensuring single NVM slot lifetime. It is not effective in increasing the overall lifetime of a storage system with

This work is supported by National Natural Science Foundation of China: No.61502061, Chongqing application foundation and research in cutting-edge technologies: No.cstc2015jcyjA40016, the fundamental research funds for the central universities: 2019CDXYJSJ0021, and also the financial support from the program of China Scholarships Council (No.201706055029).

multiple NVM slots.

In this paper, we propose a general memory management scheme for in-memory file systems based on multiple NVM slots, called Wear-aware Selection Allocation (WASA). The goal of this scheme is providing an effective and reliable software wear-leveling technology to guarantee performance and lifetime. The essential idea of WASA is to allocate NVM space from the least used slot. To achieve this goal, each NVM slot is assigned a wear counter, which is increased every time when a write operation is issued to the corresponding slot. The wear counter aware the wear of each slot and be used to adjust the use of each NVM slot. And to improve the performance in concurrent workload pattern, we propose the private Counters-Group and Freelists-Group of each CPU. We achieve the global wear-leveling among multiple NVM slots through the local wear-leveling of each CPU. And each CPU only access their own private Counters-Group and Freelists-Group, thus improve the performance in concurrent workload pattern.

In summary, this paper makes the following contributions:

- We reveal the problem of existing hardware wear leveling technologies that are ineffective for a system with multiple NVM slots. Based on our experimental results, we find that the existing in-memory file system based on NVM can potentially cause significant skewness in terms of writes to each NVM slot.
- To the best of our knowledge, the proposed dynamic wear-aware memory management scheme, Wear-aware Selection Allocation (WASA), is the first wear-leveling policy that focuses on multiple NVM slots from the perspective of upper-layer file system, which can provide software wear-leveling and ensure concurrent pattern performance by eliminating the overhead of performance caused by updating wear counter in allocation and write processes.
- The proposed Wear-aware Selection Allocation (WASA) including wear counter and allocation strategy can dynamically adjust the use of each NVM slot, which achieves the global wear-leveling through the local wear-leveling of each CPU, meanwhile has a good performance in concurrent workload pattern through exclusive resources of each CPU. To validate WASA, we implemented and evaluated our memory management scheme under various workloads. The experimental results show that the proposed wear-leveling technology can outperform wear-leveling effect by more than 2600x, and the lifetime of NVM can be prolonged by 2.5x, the write performance can be improved by up to 15%.

The remainder of this paper is organized as follows. Section II discusses the background and challenges for in-memory file system based on NVM. Section III presents the design of WASA scheme. Section IV presents the experimental evaluation results. Section V presents related work. This paper is concluded in Section VI.

TABLE I: Characteristics of Different Types of Memory

Category	Read Latency (ns)	Write Latency (ns)	Endurance (# of writes per bit)
SRAM	2-3	2-3	∞
DRAM	15	15	10^{18}
STT-RAM	5-30	10-100	10^{15}
PCM	50-70	150-220	$10^8 - 10^{12}$
Flash	25000	200000-500000	10^5
HDD	3000000	3000000	

II. BACKGROUND AND MOTIVATION

A. Background

Emerging Non-Volatile Memory (NVM) has the advantages of high density, low cost, persistent and byte addressability, which can be used to build storage systems to improve I/O performance. Table. I [13] shows the characteristics of different types of memory. Since NVM can be directly connected to the memory bus and accessed through read/store instructions, it provides a completely different way of storing data [37]. Therefore, many research works have proposed a variety of ways to integrate NVM into the computer storage system. In general, there are three alternative ways to use NVM in a storage system [14]. The first is to replace the DRAM with NVM to increase storage capacity. The second is to use NVM as another cache level for HDD or SSD. The last is to use DRAM and NVM in a common address space.

With the development of NVM, recent research mainly integrates NVM in the flowing two aspects: 1) using NVM as persistence storage of the file system; 2) using NVM as a traditional memory. For example, SIMFS [6], NOVA [7], PMFS [8], BPFS [29], SCMFS [30] are new persistent in-memory file system designed on NVM. These file systems take advantage of NVM and have significant improvements in I/O performance. SIMFS organizes file structures by virtual file page table and uses MMU hardware to accelerate read/write data. PMFS organizes file structures through B-tree. NOVA is designed on PMFS, and utilize multiple linked lists to manage free pages, thus improving concurrent I/O performance. Compared with the traditional disk file system, such as EXT4 [15], BTFS [16], ZFS [17], the in-memory file system has the following advantages: 1) Breaking the traditional files system I/O performance bottleneck and has a significant improvement in read and write file performance. 2) Access file data through CPU read/store instructions and has lower latency. These emerging in-memory file systems have enabled user applications to get more reliable and fast storage services from file system level.

There are two main approaches to manage NVM. The first, we can use traditional memory management strategies to manage NVM, such as PTMalloc [18], TCMalloc [19], Hoard [20]. These memory management strategies are widely used in user applications, such as Glibc, MySQL and NoSQL databases. It is optimized for concurrent environments and small data block's allocation and free. For example, PTMalloc is based on Doug's Malloc, which adds parallel allocation and free. TCMalloc is a memory management library developed

by Google, which use a multi-layered cache to efficiently allocate and free. Hoard is optimized for high concurrency environments with multiple threads and processors. But these traditional memory management are not suitable for in-memory storage system because they change access patterns and use memory inefficiently [43]. With the rapid development of NVM, the new memory management scheme and data structures for NVM has emerged in recent years [44] [45] [46]. Makalu [21] uses NVM as a complement to DRAM, considering the consistency of NVM failures and providing an integrated allocator and garbage collector. NVM_Malloc [22] is designed on Jemalloc [24], which combine NVM and DRAM to provide high-performance memory allocator. Walloc [23] considers the wear leveling of NVM and provides a small grained memory allocator. Chameleon [33] is an adaptive wear balancer for flash clusters with multiple SSD devices. Since SSD does not have the byte addressable feature of NVM and the non-distributed storage system with multiple NVM slots studied in this paper is different from the distributed system for Chameleon. In-memory file system such as SIMFS, PMFS, NOVA has their own way to organize NVM free blocks. SIMFS, PMFS organize the available free blocks as a single linked-list. NOVA is designed based on PMFS, divides the free blocks into multiple sub-lists, one per CPU to improve performance in high concurrent workload pattern.

Unfortunately, NVM has limited write endurance and it can quickly wear out by unprotected writes. As shown in Table. I, PCM can only tolerate $10^8 - 10^{12}$ writes per bit. Recently researches usually utilize hardware wear leveling technology to increase the lifetime of single NVM, such as Start-Gap [10], Flip-N-Write [11], Bloom-Filter [12] and so on. Star-Gap proposes a simple, novel, and effective wear-leveling techniques by combining two registers with simple address-space randomization techniques. Flip-N-Write is a simple micro-architectural technique to replace a phase-change random access memory (PRAM) write operation with a more efficient read-modify-write operation. Flip-N-Write reduces the PRAM write time by half, and achieves commensurate savings in write energy under the same instantaneous write power constraint. Bloom-Filter is a new dynamic wear-leveling method that reduces unnecessary data migrations by adopting a hot/cold swapping-based dynamic method. Compared with the conventional hot/cold swapping-based dynamic method, the proposed method requires only a small amount of space overhead by applying Bloom filters to the identification of hot and cold data.

B. Motivation

In this section, we briefly describe the system architecture for existing in-memory file systems. Then we discuss the challenges in the design for NVM management of in-memory file systems.

The modern architecture of storage system usually has multi-processor and multiple storage devices. These storage devices share the address bus and the processor can access

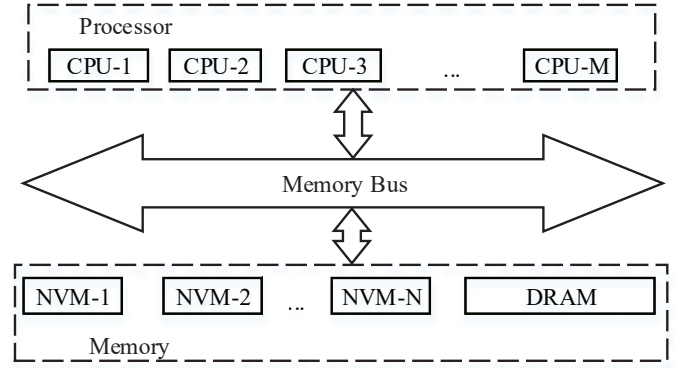


Fig. 1: Architecture of Modern Storage System.

the storage device in parallel. Fig. 1 shows the architectural overview of the modern storage system.

In Fig. 1, the storage system consists of m CPUs, n NVM slots and one DRAM. An application usually runs on one or more CPUs and accesses the NVM through an in-memory file system, such as SIMFS, PMFS, NOVA, HINFS [25], EXT4-DAX [26]. These in-memory file systems have two goals. First, achieve high file I/O performance by eliminating the overhead of page cache. Second, ensure file data and metadata consistency to prevent system crashed. Unfortunately, these advanced in-memory file system does not solve a fundamental problem: how to increase the lifetime of NVM. An NVM units can only tolerate a limited number of write, thus the lifetime of NVM can be quickly worn out due to uncontrolled write operations. For this reason, recently research utilize the hardware wear leveling techniques to ensure a single NVM lifetime.

In order to present the wear intensity among multiple NVM slots after a large number of writes, we mount SIMFS, PMFS, NOVA on eight NVM slots, and show the write times of each NVM slot. We assume that the single NVM slot lifetime can be guaranteed by hardware wear-leveling technologies. We show comparisons of writes distribution to all the NVM slots after running Postmark benchmark in PMFS, NOVA and SIMFS as shown in Figure 2, 3 and 4, respectively.

Fig. 2, 3, 4 show comparison of writes distribution to all the NVM slots in PMFS, NOVA and SIMFS. From Fig. 2, 3, 4, we can see clearly that the write times among NVM slots have significant differences. PMFS has more write operators on NVM-1 and NVM-6. NVM-1 and NVM-6 write times are 5x to 35x of others NVM slots. NOVA has fewer write times on NVM-7, the write times of NVM-7 is nearly 30x less than other NVM slots. SIMFS concentrates on NVM-1 for writing, which is 10x to 20x than others NVM slots.

The results show that the existing in-memory file system will result in a large write gap among multiple NVM slots. Therefore, recent research has proposed hardware wear leveling technology to ensure a single NVM lifetime, but different workloads will cause different write times on NVM slots and reduce the lifetime of whole system.

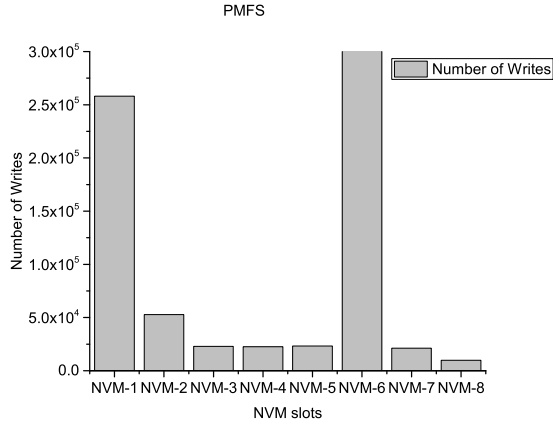


Fig. 2: Comparison of writes distribution to all the NVM slots for PMFS.

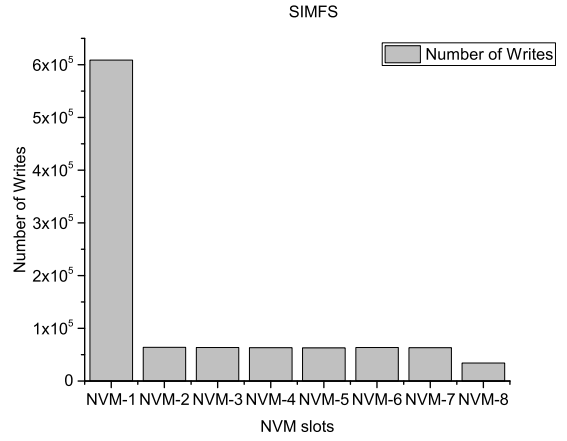


Fig. 4: Comparison of writes distribution to all the NVM slots for SIMFS.

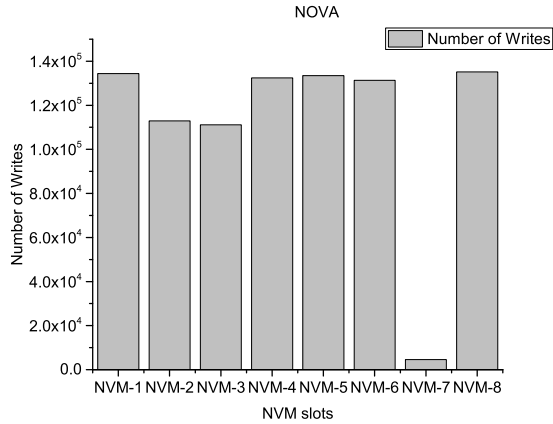


Fig. 3: Comparison of writes distribution to all the NVM slots for NOVA.

Based on the above analysis, since the most advantage hardware wear-leveling cannot effective in the lifetime of a storage with multiple NVM slots, we focus on NVM management in file system and propose a general NVM management scheme called Wear-aware Selection Allocation (WASA) to achieve the wear-leveling among multiple NVM slots. We assume that the single NVM already able to guarantee wear-leveling at the hardware level. WASA achieves software wear-leveling among multiple NVM slots through controls the writes on each NVM slot at the upper system level. Because WASA dynamic aware the wear of each NVM slot, we insert statistical code to count the writes time in write process. And the write time of each slot is used to adjust the use of each NVM slot in allocation process. Therefore, it brings the additional performance overhead. So we considers how to eliminate these overheads. In summary, WASA provides software-level wear leveling among multiple NVM slots and ensures concurrent workload pattern performance by eliminating the overhead of performance due to updating wear counter in allocation and write process.

III. DESIGN

In this section, we present the design of Wear-aware Selection Allocation scheme.

A. Overview

We propose a general NVM management scheme called Wear-aware Selection Allocation (WASA). WASA is a wear-aware memory management scheme for file system with multiple NVM slots while ensuring the performance in concurrent workload pattern. WASA consists of two parts: 1) wear counter; 2) allocation strategy. The wear counter dynamic aware the wear of each slot and be used to adjust the use of each NVM slot. The allocation strategy considers concurrent workload pattern and effectively enhance lifetime and minimizes the cost of performance. We propose Per-CPU private NVM free-list least use first out (PCFLLUF) for concurrent workload pattern. Fig. 5 shows the architecture overview of WASA.

In Section IV, we show the impact of the allocation strategies on file system performance. Fig. 5 shows the overall architecture of a system with m CPUs and n NVM slots. Applications run on one or more CPUs. An Application utilize in-memory file system to read and write data to NVM via Virtual File System (VFS) layer. WASA is located between in-memory file system layer and NVMs layer. For an in-memory file system mount on multiple NVM slots, the allocation strategy is combined with the wear counter to control the usage of each NVM slot. WASA utilizes wear counter to sense the wear of each NVM and use allocation strategy to reach wear leveling among multiple NVM slots.

Although WASA is a software-level implementation, in order to effectively achieve wear-leveling among multiple NVM slots it must combine the underlying hardware wear-leveling technology. The method proposed in this paper controls the use of the underlying NVMs at the software layer, thereby avoiding huge gaps in the amount of data written in multiple NVM slots. The underlying hardware wear-leveling technology prevents a single device from being quickly written out.

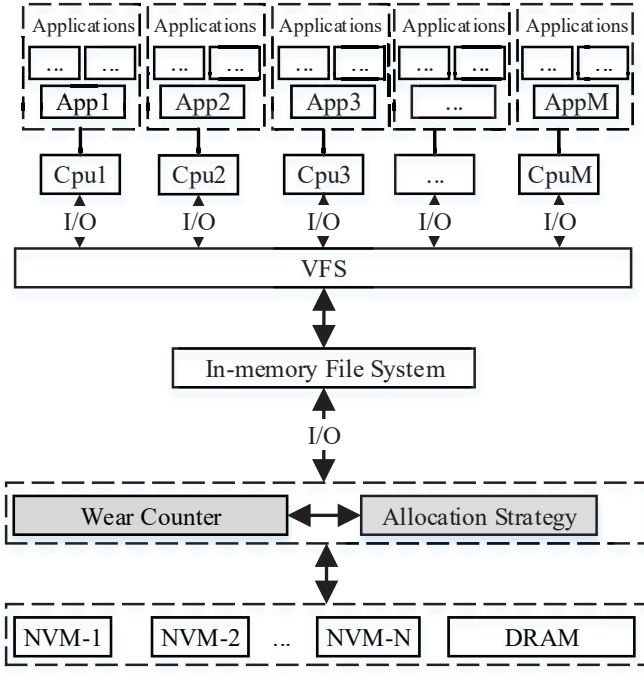


Fig. 5: Architecture Overview of a System with m CPUs and n NVM slots.

Through the access control of multiple NVM slots at software-level and the life guaranteeing of single NVM slot at hardware-levels, the extra overhead is minimized and the overall life of the system is guaranteed. Therefore, WASA must cooperate with hardware wear-leveling technology to meeting desirable objectives of extending the lifetime of multiple NVM slots and improving I/O performance.

B. Wear Counter

In this section, we describe the design of wear counter. Wear counter is used to sense the wear of NVM, so how to record the wear is very important. Usually, an NVM unit has limited write operations, such as a PCM unit only can tolerate $10^8 - 10^{12}$ write operations. We insert code in the file write function and store the total write times on each NVM slot. The write times of NVM represent the wear intensity. We reserve a space on each NVM slot to persisting the total number of writes on the current NVM slot.

Since the NVM-Counter on each NVM device persists the total number of writes, we can get the initial state of all NVM devices by reading the NVM-Counter in the file system initialization process. In the initial mount of the file system, we read the value from NVM-Counter from each NVM device, then initialize Cached-NVM-Counter of Counters-Group of in the Counters-Group with the value of NVM-Counter. For a system with m CPUs and n NVM slots, there are n counters in a Counters-Group. We initialize the value of each counter in each Counters-Group to $\text{NVM-Counter}/m$. In this way, WASA can obtain the initialization status of all NVM devices, thus

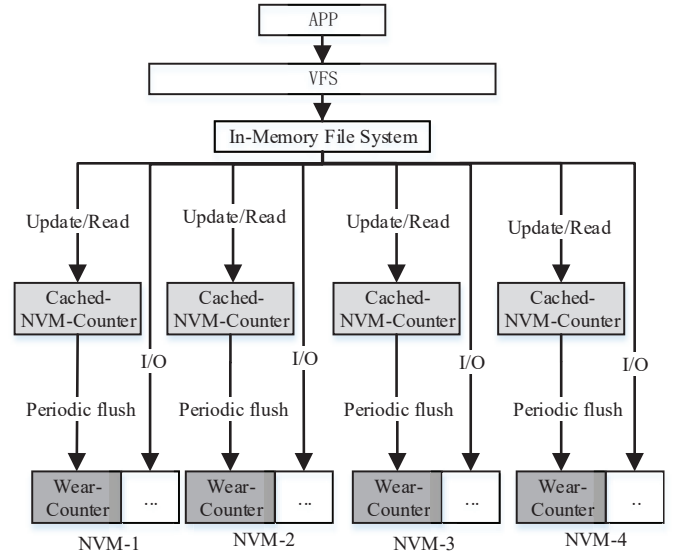


Fig. 6: Architecture overview of Wear Counter with WB strategy.

avoiding some NVM devices being quickly worn out due to different initialization states.

Considering the latency between NVM and DRAM, we use two strategies to update wear counter: 1) write-through (WT); 2) write-back (WB). WT strategy directly accesses and updates the value of wear counter on NVM slot. WT strategy can accurately record the number of writes of NVM slot but have performance loss. WB strategy creates a cache of wear counter in DRAM and each write only updates the cache of wear counter. For persistent store the wear counter, WB periodically flushes the cache of wear counter. WB has the advantage of performance but has the disadvantage of inaccuracy of wear record of NVM when system crash. We call the persistent wear counter as NVM-Counter and the cache wear counter as Cache-NVM-Counter. Fig. 6 shows the design of wear counter using WB strategy.

The overhead of wear counter is mainly in two aspects: 1) the space overhead of wear counter; 2) the performance overhead of wear counter. The space overhead of wear counter because we need persistent store the write times on each NVM slot. The persistent wear counter on NVM can be read after the system restart or crash. We use a page to store the value of wear counter, so the space cost of wear counter is 4KB on each NVM slot. The performance overhead of wear counter due to we need update wear counter. Because we need to record the number of write, so we update wear counter after each write. We propose two update mode write through and write back for updating the wear counter. In Section IV, experimental results show the performance overhead difference between write through and write back.

C. Allocation strategy

In this section, we first present the most usual way of NVM management in the in-memory file system and its allocation

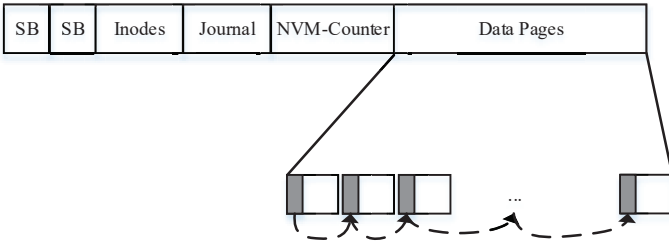


Fig. 7: Layout of the in-memory File Systems with NVM-Counter.

strategy. Then we present its defect for in-memory file system with multiple NVM slots. Finally, we describe the design of PCFLLUF strategy.

In Section II, experimental results of motivation show the existing NVM management of in-memory file system cannot ensure the wear leveling among NVM slots. And SIMFS and PMFS use a single linked list to organize the free block of NVM, there are additional performance overheads when multiple processors select the same NVM slot in allocation.

Fig. 7 shows the layout of the in-memory file systems with NVM-Counter. NVM space is divided into five areas, superblock area and superblock backup area, inodes area, journal area, NVM-Counter area and free blocks area. The in-memory File system usually organizes free blocks as the single linked list, such as PMFS and SIMFS. Each node of the linked list is a 4KB NVM block. SB stores the head and tail of the linked list. The first 8 bytes of each NVM block stores the next address of free NVM block, thereby forming a persistent single linked list. When allocating a new NVM block, we take the free NVM block from the free list, and then we update the list head information of SB. When free an NVM block, the NVM block is inserted into the free list in a suitable way. NOVA is slightly different from PMFS and SIMFS, which divides the available free blocks into multiple sub-lists, one per CPU.

The design of our allocation strategy is to ensure wear leveling among multiple NVM slots and performance in high concurrent workload pattern. To achieve this goal, we propose an allocation strategy called Per-CPU Free List Least Use First (PCFLLUF). PCFLLUF can ensure the wear leveling among NVM slots and eliminate the overhead of performance bring by updating wear counter in the allocation and write process.

Fig. 8 presents the architecture overview of PCFLLUF in a system with m processors and n NVM slots. PCFLLUF creates a persistent NVM-Counter in each NVM slot, so there are n NVM-Counters corresponding to these n NVM slots. Moreover, each processor has a private Counters-Group and a private FreeLists-Group. In the Counters-Group of one processor, there are n private NVM-Counters corresponding to these n NVM slots. Each private NVM-Counter records the number of writes made by the application running on this processor to the corresponding NVM slot. Similarly, in the FreeLists-Group of one processor, there are n private NVM-FreeLists

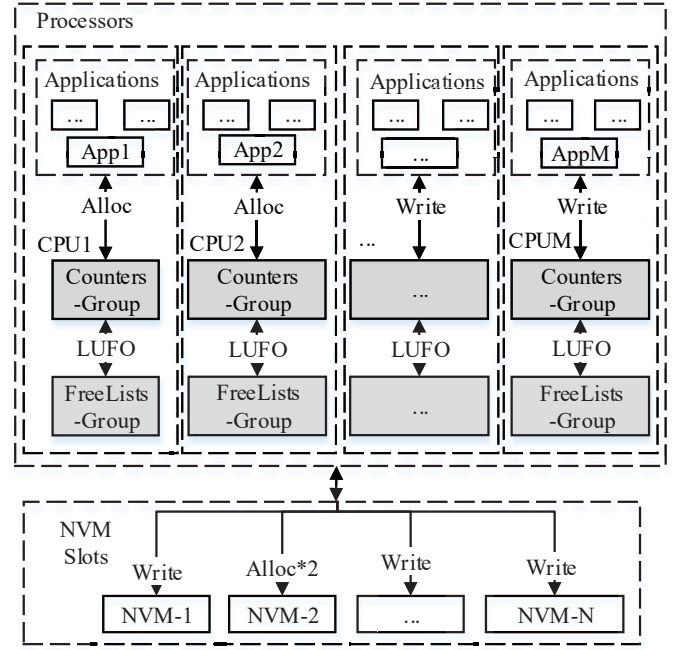


Fig. 8: Architecture overview of the Per-CPU FreeList Least Use First.

corresponding to these n NVM slots. For one processor, each private NVM-FreeList records its private NVM free blocks of the corresponding NVM slot. The processor only allocates and frees the NVM free blocks from own private FreeLists-Group. In Fig. 8, CPU1 and CPU2 are simultaneously allocating new NVM blocks for their respective applications. CPU1 and CPU2 access each private Counters-Group to find the local NVM free list with minimal write times. Processors use the Least Use First Out (LUFO) strategy to allocate new NVM block. In Fig. 8, CPU1 and CPU2 both find that the minimal-write NVM free list is NVM2, then they allocate new NVM block from the private NVM2 free list of its FreeLists-Group.

Fig. 9 shows the architecture overview of Counters-Group in a system with m processors and n NVM slots. We reserve a place on each NVM slot as the persistent NVM-Counter. Each processor has private Counters-Group. Counters-Group of each processor has n cache NVM-Counter. For example, CPU1 has CPU1-NVM1-Counter, CPU1-NVM2-Counter, and so on. CPU1-NVM1-Counter represent the number of writes on NVM1 by CPU1. PCFLLUF strategy utilize Counters-Group to find which NVM is minimal write times on current processor. And then PCFLLUF strategy do allocation from the Freelists-Group. Equation (1) shows the relation between persistent NVM-Counter and cached Counters-Group. When we update the persistent NVM-Counter of NVM slots. First we lock the all Counters-Group of each processor. Then we add the CPU-NVM-Counter of each Counters-Group according to NVM slot. Finally we unlock the Counters-Group to release the lock.

Fig. 10 shows the architecture overview of FreeLists-Group.

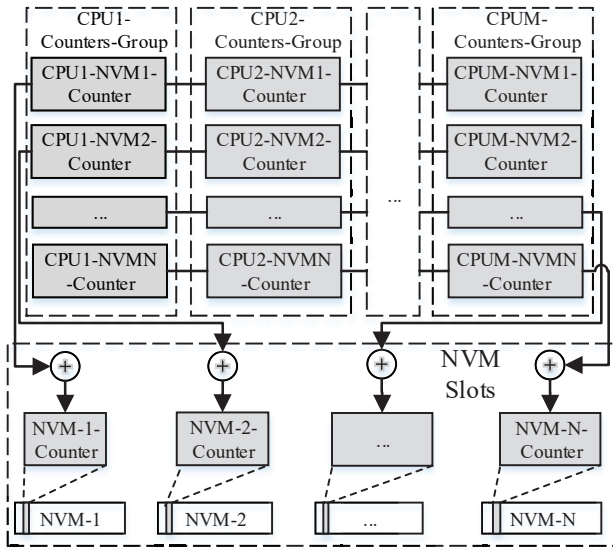


Fig. 9: Architecture overview of Counters-Group.

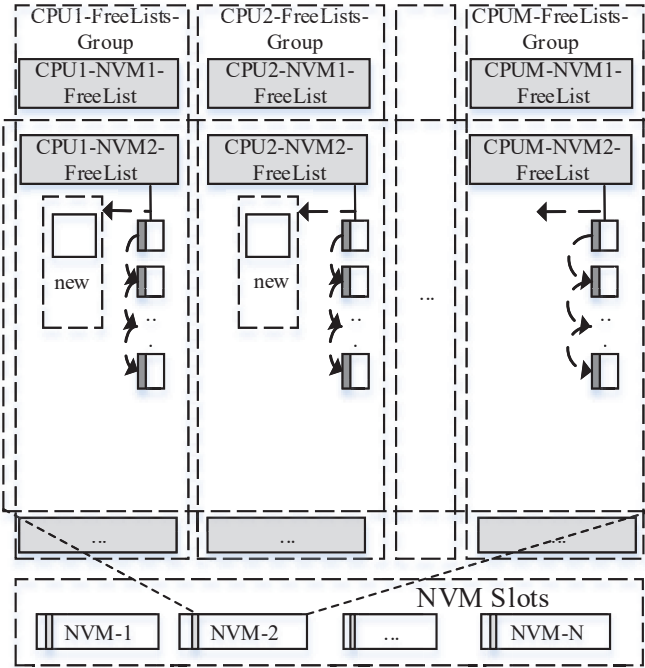


Fig. 10: Architecture overview of FreeLists-Group.

Each FreeLists-Group records the private NVM free blocks of the processor. All free blocks of each NVM slot are divided into multiple FreeLists according to the number of processors. In Fig. 10, all the free blocks of NVM-2 slot are divided into m FreeLists. And these m FreeLists are evenly assigned to m processors. We called the FreeLists owned by processor CPU-NVM-FreeList. In Fig. 10, CPU1 has CPU1-NVM2-FreeList, CPU2 has CPU2-NVM2-FreeList, and so on. Each CPU-NVM-FreeList is organized as a single linked list. The first 8 bytes of each NVM block stores the next address of free

Algorithm 1 The Write Process of PCFLLUF

INPUT:

filp: the file needs to write

offset: the offset of written data

size: the size of written data

buf: the written data

counters_group: the counters-group of all processor

OUTPUT:

NONE

- 1: DO_FS_WRITE(*filp*, *offset*, *size*, *buf*)
- 2: $cpu_i \leftarrow \text{CURRENT_CPU}()$
- 3: $write_nvms \leftarrow \text{GET_WRITE_NVM}(filp, offset, size)$
- 4: **for** each $nvm \in write_nvms$ **do**
- 5: $use_cpu_nvm_counters \leftarrow counters_group.cpu_i$
- 6: LOCK($use_cpu_nvm_counters.nvm$)
- 7: INC($use_cpu_nvm_counters.nvm$)
- 8: UNLOCK($use_cpu_nvm_counters.nvm$)
- 9: **end for**

NVM block, thereby forming a persistent single linked list. In Fig. 10, CPU1 and CPU2 are allocating at the same time. CPU1 and CPU2 both find that the local minimal write times NVM slot is NVM-2 slot. Then CPU1 and CPU2 both do the allocation operations. CPU1 does the allocation operation on CPU1-NVM2-FreeList. CPU2 does the allocation operation on CPU2-NVM2-FreeList. Thus, CPU1 and CPU2 can do the allocation operation in the meantime. Therefore PCFLLUF can achieve the high performance in the high concurrent workload pattern.

In the allocation process, Firstly PCFLLUF acquires the currently used processor, thereby get the currently Counters-Group. Then PCFLLUF get the CPU-NVM-Counter with the minimal value. Then PCFLLUF select a CPU-Free-List with the minimal value from FreeLists-Group. Finally, we take the first free block of the CPU-Free-List selected. In the write process, PCFLLUF only read and update CPU-NVM-Counter of the Counters-Group. In the write process, PCFLLUF first finds which NVM is written by offset and size. Then, according to the currently used processor, we update the CPU-NVM-Counter. And PCFLLUF periodically updates the CPU-NVM-Counter to NVM-Counter.

Algorithm 1 shows the write process in PCFLLUF strategy. In Line 1, the write function of file system needs to be executed. In Line 2, we find which processor is being used for write. In Line 3, we get which NVM are written by the offset and size. In Lines 4 to 9, we update these CPU-NVM-Counter selected from Line 3. Since only records the write time of currently used processor, PCFLLUF only update and read its own private wear counters. Therefore, PCFLLUF can avoid the overhead of writing simultaneously by different processors.

Algorithm 2 shows the allocation process in PCFLLUF strategy. In Line 1, we find which processor is being used for allocation. In Line 2, we find the private wear counter of currently used processor. In Line 3, we only lock CPU-

Algorithm 2 The Allocation Process of PCFLLUF

INPUT:

counters_group: the counters-group of all processor*freelists_group*: the freelists-group of all processor

OUTPUT:

page: the new page

```
1:  $cpu_i \leftarrow \text{CURRENT\_CPU}()$ 
2:  $use\_cpu\_nvm\_counters \leftarrow counters\_group.cpu_i$ 
3:  $\text{LOCK}(use\_cpu\_nvm\_counters)$ 
4: for each  $c \in use\_cpu\_nvm\_counters$  do
5:   if  $c < use\_cpu\_nvm\_counters.min\_nvm$  then
6:      $min\_nvm \leftarrow c.nvm$ 
7:   end if
8: end for
9:  $\text{UNLOCK}(use\_cpu\_nvm\_counters)$ 
10:  $\text{LOCK}(freelists\_group.cpu_i.min\_nvm)$ 
11:  $page \leftarrow \text{NEW\_PAGE}(freelists\_group.cpu_i.min\_nvm)$ 
12:  $\text{UNLOCK}(use\_cpu\_nvm\_counters)$ 
13: return  $page$ 
```

NVM-counter of currently used processor. In Lines 4 to 8, we find the CPU-Free-List with minimum wear counter from the FreeLists-Group. In Line 9, we unlock currently processor's private counter. In Line 10, we lock the CPU-NVM-FreeList selected by Lines 4 to 8. In Line 11, we do allocate free block from CPU-NVM-FreeList. In Line 12, we unlock CPU-NVM-FreeList. In Line 13, return the free block and finish the allocation.

Local wear leveling among NVM slots of each processor:

The key PCFLLUF is to find the NVM with the minimum wear counter in currently used processor. Therefore, PCFLLUF accurately counts the write times of each NVM on each processor in the writing process. Since PCFLLUF strategy select an NVM with the minimal wear counter in current processor, so it can achieve wear leveling among NVM slots of each processor.

Global wear leveling among NVM slots: From the above, we know that PCFLLUF can achieve local wear leveling among NVM slots of each processor. In a system with m processors and n NVM slots, the total number of writes on an NVM is equal to the sum of the writes of all processors on this NVM. Equation (1) shows the relationship between the total write times on an NVM and the total number of writes per CPU on an NVM. If we can guarantee all private wear counters of each CPU is a similar value, then we can reach global wear leveling between NVM because the sum of all CPU's private wear counters is a similar value.

$$NVM_j - Counter = \sum_{i=0}^m CPU_i - NVM_j - Counter \quad (1)$$

PCFLLUF use the private wear counter of each processor to select the least used NVM slot. PCFLLUF strategy can eliminate the overhead of performance bring by updating wear

counter in the concurrent allocation and write process. Because a processor only accesses its own private wear counter, the application read and update wear counter on different processors are independent of each other. And PCFLLUF use the private free list of each processor to allocate. PCFLLUF divide each NVM free blocks into multiple free lists according to the number of processors. Each processor only takes free blocks from its own private free list. The NVM free block allocation can be executed in parallel on different processors. Thereby PCFLLUF eliminates the overhead of parallel allocation. PCFLLUF is more suitable for parallel updating, appending and creating files in a high-concurrency environment. However, PCFLLUF reduces the available free blocks for a processor because it divides the total free block of NVM into multiple free lists according to the number of processors. This will result in a processor being able to use all free blocks of an NVM, but now only $1/m$ free blocks of an NVM can be used. If an application has been allocating free blocks on the same processor, then the available free blocks of this processor will be quickly exhausted. For this case, we choose to re-allocate free blocks from other processors until finds free blocks. If all processors have no available free blocks, the allocation fails.

D. Overhead analysis

The overhead caused by allocation strategies is in two aspects: 1) the overhead caused by finding the NVM with the minimum write times in the allocation process; 2) the overhead caused by updating the wear counters in the write process. In the allocation process, each processor needs to wait the previous allocation operation to release the lock of free list. For a system with n processors and m NVM devices. PCFLLUF strategy select the NVM with the least write times from n CPU-NVM-Counter. The time complexity of this process is $O(n)$. PCFLLUF only read the CPU-NVM-Counter of current processor, thus PCFLLUF eliminate the lock overhead of reading wear counter. PCFLLUF only allocate the free blocks from CPU-NVM-FreeList, thus eliminating the allocation lock overhead of multiple processors simultaneously allocating.

In the write process, each processor needs to wait the previous update operation to release the lock of wear counter. PCFLLUF eliminate the overhead of lock of wear counter by using CPU-NVM-Counter. PCFLLUF only update the private wear counter of current processor, thus eliminating the overhead of multiple processors simultaneously updating.

In conclusion, multiple applications may run one or more CPUs in real world scenarios. There are some applications concurrently reading or writing NVM-Counter in the file writing process. To ensure accuracy of NVM-Counter, each application needs to lock the NVM-Counter to a mutually exclusive access. In this way, if an application occupies the NVM-Counter, the others applications cannot access unless it is unlocked. If Counters-Group is used, each CPU only locks the Counters-Group it owns. If an application on one CPU occupies the Counters-Group, it only affects application access on the same CPU, and does not affect application access

on the other CPUs. Similarly, multiple applications may be allocated in the same NVM device during the writing process. In order to ensure the correctness of the free and allocation operations on the free NVM blocks list, each application needs to lock and mutually exclusive access the free list in the allocation process. If FreeLists-Group is used, the application occupies the FreeLists-Group on its running CPU, so it only affects application access on the same CPU and does not affect application access on the other CPUs. In general, if the original scheme is used, the extra overhead for locking and unlocking will be in the write and allocation process for each application. However, when Counters-Group and FreeLists-Group are used, the extra overhead for locking and unlocking is reduced when applications concurrently allocates and writes on different CPUs.

IV. EVALUATION

We have implemented Wear-aware Selection Allocation (WASA) scheme for PMFS [7]. The experiments are built on the operating system Ubuntu 16.04, using the Linux kernel version 4.4.4. The experiments are conducted on an 4-Core platform with Intel Core i5-6500 CPU, running at 3.20 GHz. This system is equipped with 16 GB 2133 MHz memory.

Since NVM devices are not yet available, we reserve 8 GB DRAM from 16 GB memory to emulate a persistent memory device pmem0 [27]. The space of the pmem0 device is divided into 4 areas according to the size of 2 GB, which is used to simulate 4 NVM slots. We mount PMFS on pmem0. The first NVM slot stores the super block information and file metadata information of the file system. The size of 4 KB is reserved in the header of each NVM slot as the wear counter NVM-Counter.

A. Wear Leveling

We have implemented Wear-aware Selection Allocation (WASA) scheme for PMFS. The experiments are built on the operating system Ubuntu 16.04, using the Linux kernel version 4.4.4. The experiments are conducted on an 4-Core platform with Intel Core i5-6500 CPU, running at 3.20 GHz. This system is equipped with 16 GB 2133 MHz memory.

We use Postmark benchmark [28] to simulate file system write operations to test the impact of WASA on the storage system life with multiple NVM devices. First, we use Postmark to create a large number of files in the file system. The file size is between 1 KB and 10 MB. Then, we use Postmark to generate a large crowd of I/O operations, such as read, write, create and delete, in the created file. The I/O basic unit for read and write operations is 1 KB. Finally, we set the Postmark read/write ratio to 5:5, the create/delete ratio to 9:1, and the number of transactions from 100 to 5K. We use this Postmark configuration to evaluate the original PMFS and PMFS with WASA memory management scheme respectively.

We first present the standard deviation of the number of writes for NVM slots that use Postmark to perform 100 to 5K transactions.

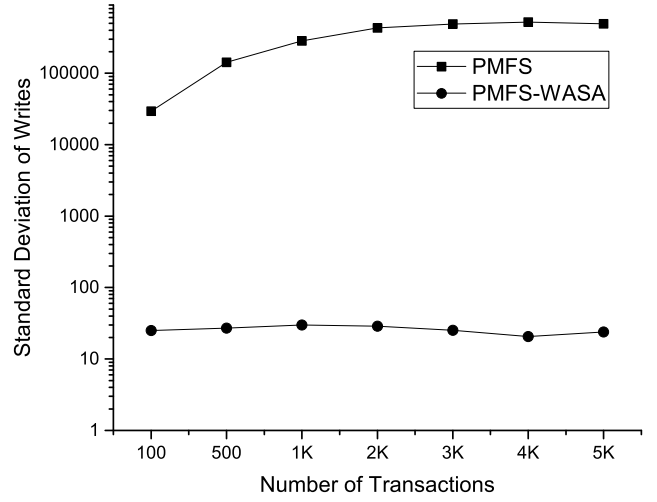


Fig. 11: The standard deviation of writes of 4 NVM slots for different memory management schemes with different number of Postmark transactions.

Fig. 11 shows the standard deviation of writes for above four memory management schemes when 100 to 5K Postmark transactions are issued to the file system. As shown Fig. 11, the standard deviation of the number of writes between NVM slots using the original PMFS allocation method is the largest, which is 2600 to 11000 times of PMFS-WASA. The original PMFS allocation method uses a single linked list to organize the free pages of all NVM slots, and adopts a strategy of taking out from the head of the list and puts the freed pages to the tail. Therefore, when there are too many writes on a certain NVM slot, it will continue to be used.

As the total number of transactions increases, the standard deviation of writes between NVM slots using the original allocation of PMFS is getting more larger, while the standard deviation of write of WASA can be kept basically stable.

This means that under the continuous use of PMFS, the difference in the number of writes between NVM slots will become larger and larger, and the wear leveling of each NVM slot will be more uneven, thereby resulting in a decrease in the life of system. However, WASA can keep the number of writes between NVM slots stable under continuous use. That is, WASA can use each NVM slot on average to ensure the wear leveling among NVM slots.

In the system with multiple NVM devices, the lifetime of system can be expressed by (2):

$$Lifetime = W_{MAX} * S / B_{MAX} \quad (2)$$

In (2), W_{MAX} represents the maximum number of writes an NVM unit can stand, S is the size of an NVM slot, and B_{MAX} represents the maximum amount of data that writes on an NVM slot every day.

Assuming that the size of data written each time is fixed to W bytes, then the maximum amount of data that writes on an NVM slot B_{MAX} can be expressed by (3):

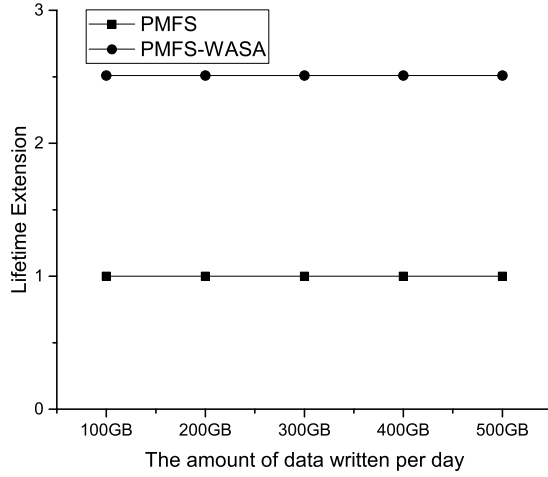


Fig. 12: The improvement of lifetime for different memory management schemes.

$$B_{MAX} = NVM - Counter_{MAX} * W \quad (3)$$

In (3), $NVM - Counter_{MAX}$ represents the maximum number of writes on an NVM slot. We use Postmark to simulate file system writes to demonstrate NVM slot life usage.

Assuming an NVM unit can be written 10^8 times. An NVM slot is 2 GB, the system has a total of 4 NVM slots. The amount of data written per day in the system is 100 GB to 500 GB, and the size of data per write is 4 KB which is equal to the size of one page. Thus, according to (2) and (3), we can get the lifetime of system for different allocation strategies. Fig. 11 shows the improvement of PMFS-WASA compared with the origin PMFS in the case of the total number of different postmark transactions. As shown in Figure 12, WASA can extend the life of system by more than 2.5 times. Meanwhile, as the total number of transactions increases, the life extension is basically stable.

B. Overall Performance

We evaluate the performance of WASA using FIO benchmark. We compare the performance of the PMFS with WASA scheme and PMFS with origin scheme. Since the parts of wear counter and allocation strategy proposed in this paper do not affect any original read process of the file system, the memory management scheme proposed in this paper will only impose extra overhead on file writing, so we only compare the write performance in this section.

In this experiment, the system has 4 processors and 4 NVM slots. So PCFLUF have 4 Counters-Groups and 4 FreeLists-Groups, each processor has a Counters-Group and a FreeLists-Group. Each Counters-Group have 4 CPU-NVM-Counters. Each FreeLists-Group have 4 CPU-NVM-FreeLists.

Fig. 13 to Fig. 17 show the performance of sequential write for PMFS file system with different allocation strategies in 1, 2, 4, 8, and 16 threads and the data blocks are from 1 KB

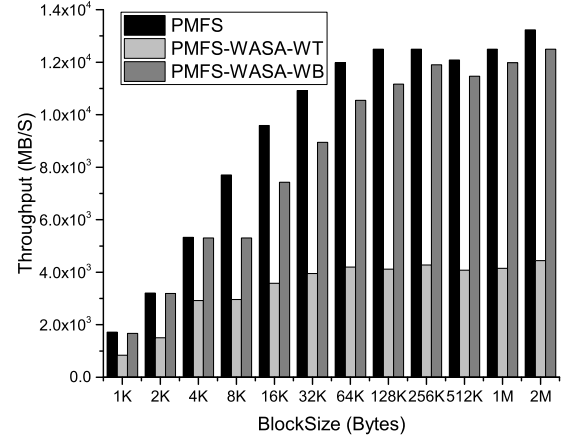


Fig. 13: The performance of single-threaded sequential write for each allocation strategy.

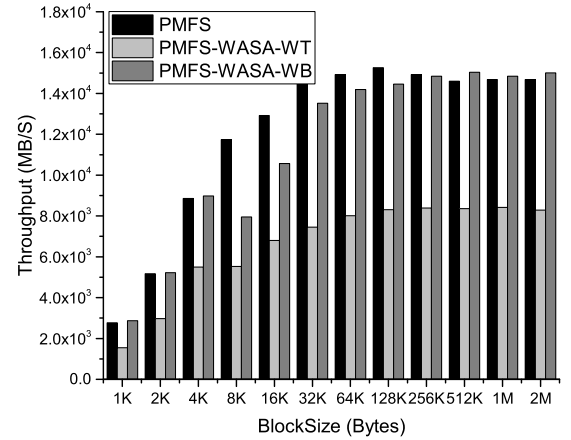


Fig. 14: The performance of two-threaded sequential write for each allocation strategy.

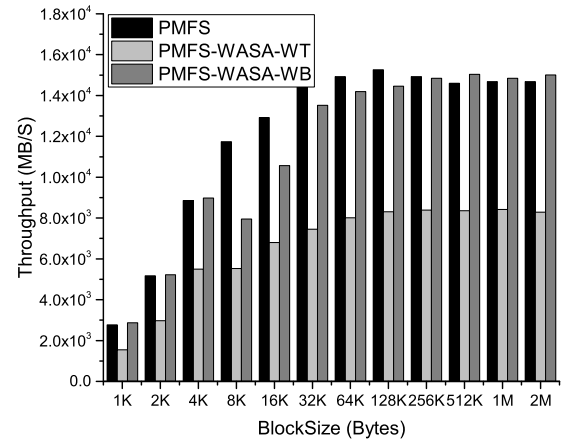


Fig. 15: The performance of four-threaded sequential write for each allocation strategy.

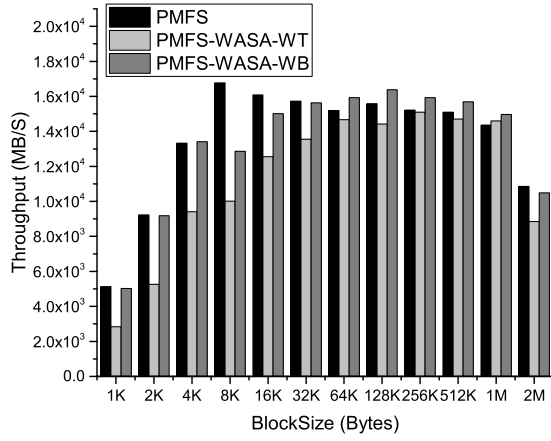


Fig. 16: The performance of eight-threaded sequential write for each allocation strategy.

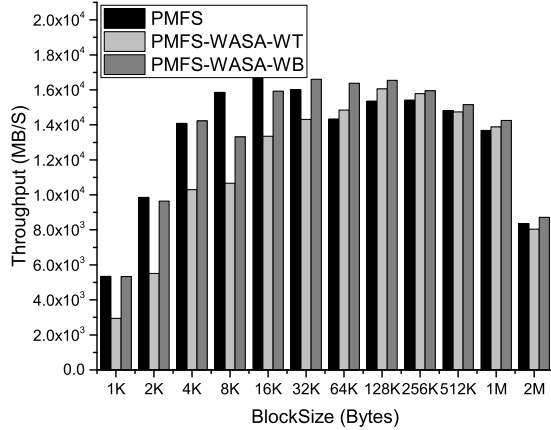


Fig. 17: The performance of sixteen-threaded sequential write for each allocation strategy.

to 2 MB. The number of threads in the experimental test is specified by FIO's numjobs parameter. In the fio test process, there may be multiple threads or single thread on a CPU, so there is no certain mapping relationship between test threads and CPU. This test environment is also more in line with the real world scenarios.

Fig. 13 to Fig. 17 show the performance of sequential write for PMFS file system with different allocation strategies in 1, 2, 4, 8, and 16 threads and the data blocks are from 1 KB to 2 MB. PMFS represent PMFS with original allocation strategy. PMFS-PCFLLUF-WT and PMFS-PCFLLUF-WB both represent PMFS with PCFLLUF allocation strategy. PMFS-PCFLLUF-WT updates NVM-Counter in write-through mode. Write-through mode flushes the value of CPU-NVM-Counter to NVM-counter in each writing process. PMFS-PCFLLUF-WB updates the NVM-Counter in write-back mode. WB mode periodic flushes all the value of CPU-NVM-Counters to NVM-Counters.

From the results shown in Fig. 13 to Fig. 17, PMFS-PCFLLUF-WT has the worst performance. Because PMFS-PCFLLUF-WT updates the wear counter in write-through

mode. The latency of NVM is lower than the latency of DRAM. PMFS-PCFLLUF-WT updates the persistence NVM-Counter on NVM slot immediately after each write operation. So PMFS-PCFLLUF-WT has extra performance overhead. However, PMFS-PCFLLUF-WB updates the wear counter in a write-back mode. It only updates the CPU-NVM-Counter on DRAM after each write operation. PMFS-PCFLLUF-WB has extra performance overhead because it periodic flushes CPU-NVM-Counter to NVM-Counter. The experimental results shows that the write-back mode has only the performance overhead in low concurrent workload.

As shown in Fig. 13 to Fig. 17, The performance gap between PMFS-PCFLLUF-WB and the original PMFS is very small which can be negligible with block sizes from 1KB to 4KB. The reason is that small block size has a shorter I/O duration which makes a lower chance of lock conflict. As shown in Fig. 13 and Fig. 14, when the test uses a lower number of threads, only 1 or 2 threads, PMFS-PCFLLUF-WB has low performance compared to the original PMFS with block sizes from 64KB to 2MB. The reason is that the possibility of lock conflict between multiple threads is high due to the I/O operation lasts for a long time for large block size.

As shown in Fig. 15 and Fig. 17, when the test uses a higher number of threads, 4 or 8 threads, PMFS-PCFLLUF-WB and the original PMFS have basically the same performance with small block sizes from 1 KB to 4 KB. And the PMFS-PCFLLUF-WB has a better performance than the original PMFS with large block sizes from 64 KB to 2 MB. In particular, PMFS-PCFLLUF-WB has the best performance advantage when the block size is among 64KB-1M, and the performance is improved up to 15% when the test with 16 threads and a block size of 64KB.

In conclusion, PMFS-PCFLLUF-WB basically shows the best performance in high concurrent workload. It not only uses Counters-Group to avoiding performance loss when writing operations simultaneously but also uses FreeLists-Group to improving the performance in the simultaneous allocation. From the performance experiment results, we verified the write performance improvement Counters-Group and FreeLists-Group in concurrent workloads.

V. RELATED WORK

In this section, we present some research works that are most closely related to ours.

A. Memory Management For In-Memory File System

The optimization of most existing in-memory file systems is based on the following aspects: 1) Improve the I/O performance by utilizing the performance advantages of NVM. 2) Optimize the traditional software I/O stack by utilizing the byte-addressable of NVM. 3) NVM provides a completely different programming model for consistency from the block device interfaces. For this reason, BPFS [29] utilizes shadow pages to provide fast consistency. SCMFS [30] utilizes virtual memory management to provide fast access to file data. SIMFS

utilizes file page table to organize files and use MMU hardware to speed up file access. But the free NVM blocks are organized as a simple single linked list. NOVA, HINFS are based on PMFS. The principle of their memory management is the same: merge the adjacent free blocks to create large contiguous free blocks. These file systems are designed to optimize the performance and consistency and not considering the wear leveling.

B. Wear Leveling Technologies For NVM

The existing NVM wear leveling techniques are mainly divided into two way: 1) Hardware wear leveling. 2) Software wear leveling. Hardware wear leveling techniques achieve wear leveling by utilizing extra controllers on hardware. For example, Start-Gap periodically migrates data between adjacent lines to achieve uniform writes at each bank. Start-Gap adds additional start and gap registers to record the lines and achieve wear leveling by migrating data from more written address to less written address. Flip-N-Write determines whether to flip the data before writing by comparing the difference between written data and existing data. Reducing the total amount of data written by flipping different bits. Bloom-Filter finds the hot address and cold address, and achieve wear leveling by swapping the hot address and cold address. Software wear leveling techniques achieve wear leveling by control the allocation and free of NVM blocks. For example, Walloc achieves wear leveling by utilizing metadata write counter (MWC) and data write counter (DWC). Yu-Ming Chang [31] utilize an adaptive sliding window to achieve wear leveling for metadata space, and utilize virtual mapping to achieve wear leveling for data space. DWARM [32] considers that each NVM page does not require accurate records and achieve wear leveling by utilizing multiple linked lists. However, these wear-leveling technologies are only suitable for a single NVM and not suitable for multiple processors and NVM in concurrent environments. Chameleon [33] employs data redundancy techniques such as replication and erasure-coding, coupled with endurance-aware write offloading, to mitigate wear level imbalance in distributed SSD-based storage. Since SSD does not have the byte addressable feature of NVM, the optimization methods of Counters-Group and FreeLists-Group proposed in this paper are not applicable to SSD. And Chameleon aims to balance the wear among different flash servers while WASA aims to balance the wear among system based on multiple NVM slots.

VI. CONCLUSIONS

In a storage system with multiple NVM slots and multiple processors, one fundamental challenge is how to avoid wear-out of an NVM slot and ensure performance. In this paper, we research how to ensure lifetime and performance of a storage system with multiple NVM slots and multiple processors. We propose a software-leveling wear leveling technology called Wear-aware Select Allocation (WASA). WASA is a common memory management scheme for in-memory file systems based on multiple NVMs. WASA extends the lifetime of

system and minimizes the cost of performance caused by achieving wear leveling. WASA consists of wear counter and allocation strategy. Wear counter record the number of writes on each NVM and used to aware the wear level of an NVM. We propose PCFLLUF allocation strategy to improve the performance in concurrent workload pattern. Extensive evaluations show that Wear-aware Select Allocation memory management scheme can outperform wear-leveling effect by more than 2600x, and the lifetime of NVM can be prolonged by 2.5x, the write performance can be improved by up to 15%.

REFERENCES

- [1] Burr G W, Brightsky M J, Sebastian A, et al. Recent Progress in Phase-Change Memory Technology[J]. IEEE Journal on Emerging & Selected Topics in Circuits & Systems, 2016, 6(2):146-162.
- [2] Intel Optane Technology, <http://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html>.
- [3] J. Arulraj, A. Pavlo, S. R. Dulloor, Let's talk about storage recovery meth-ods for non-volatile memory database systems[C], in: Proceedings of the 201530ACM SIGMOD International Conference on Management of Data, SIG-MOD '15, ACM, New York, NY, USA, 2015, pp. 707-722.
- [4] M. Caul_eld, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, S. Swanson, Moneta: A high-performance storage array architecture for next-generation, non-volatile memories[C], in: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '43, IEEE Computer Society, Washington, DC, USA, 2010, pp. 385-395.
- [5] Gotze P, van Renen A, Lersch L, et al. Data Management on Non-Volatile Memory: A Perspective[J]. Datenbank-Spektrum, 2018: 1-12..
- [6] E. H. M. Sha, X. Chen, Q. Zhuge, L. Shi, W. Jiang, A new design of in-memory file system based on file virtual address framework[J], IEEE Trans-actions on Computers 65 (10) (2016) 2959-2972.
- [7] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, J. Jackson, System software for persistent memory[C], in: Pro-ceedings of the Ninth European Conference on Computer Systems, EuroSys'14, ACM, New York, NY, USA, 2014, pp. 15:1-15:15.
- [8] J. Xu, S. Swanson, Nova: A log-structured file system for hybrid volatile/non-volatile main memories[C], in: 14th USENIX Conference on File and Storage Technologies (FAST 16), USENIX Association, Santa Clara, CA, 2016, pp. 323-338.
- [9] Palangappa P M, Mohanram K. CompEx++: Compression-expansion coding for energy, latency, and lifetime improvements in MLC/TLC NVMs[J]. ACM Transactions on Architecture and Code Optimization (TACO), 2017, 14(1): 10.
- [10] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, B. A-bali, Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling[C], in: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2009, pp. 14-23.
- [11] Cho S, Lee H. Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance[C]//Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2009: 347-357.
- [12] Yun J, Lee S, Yoo S. Bloom filter-based dynamic wear leveling for phase-change RAM[C]//Proceedings of the Conference on Design, Automation and Test in Europe. EDA Consortium, 2012: 1513-1518.
- [13] Chen C, Yang J, Wei Q, et al. Fine-grained metadata journaling on NVM[C]//Mass Storage Systems and Technologies (MSST), 2016 32nd Symposium on. IEEE, 2016: 1-13.
- [14] Wu, G. Zhang, K. Li, Rethinking computer architectures and software systems for phase-change memory[J], J. Emerg. Technol. Comput. Syst. 12 (4) (2016) 33:1-33:40.
- [15] Introduction to EXT4, <https://kernelnewbies.org/Ext4>.
- [16] Introduction to BTRFS, <https://en.wikipedia.org/wiki/Btrfs>.
- [17] Sun Microsystems file system-ZFS, <https://en.wikipedia.org/wiki/ZFS>.
- [18] PTmalloc, <https://github.com/emeryberger/Mallocc>
- [19] TCmalloc, <http://goog-perftools.sourceforge.net/doc/tcmalloc.html>.

- [20] E. D. Berger, K. S. McKinley, R. D. Blumofe, P. R. Wilson, Hoard: A scalable memory allocator for multithreaded applications[C], in: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IX, ACM, New York, NY, USA, 2000, pp. 117-128.
- [21] K. Bhandari, D. R. Chakrabarti, H.-J. Boehm, Makalu: Fast recoverable allocation of non-volatile memory[C], in: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, ACM, New York, NY, USA, 2016, pp. 677-694.
- [22] M. F. M. D. H. P. David Schwalb, Tim Berning, nvm malloc: Memory allocation for nvram[C], in: International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS), in Conjunction with VLDB, 2015.
- [23] S. Yu, N. Xiao, M. Deng, Y. Xing, F. Liu, Z. Cai, W. Chen, Walloc: An efficient wear-aware allocator for non-volatile main memory[C], in: 2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC), 2015, pp. 1-8.
- [24] Jemalloc, <http://jemalloc.net/>.
- [25] J. Ou, J. Shu, Y. Lu, A high performance file system for non-volatile main memory[C], in: Proceedings of the Eleventh European Conference on Computer Systems, EuroSys '16, ACM, New York, NY, USA, 2016, pp. 12:1-12:16.
- [26] M. Wilcox, Add support for nv-dimms to ext4. URL <https://lwn.net/Articles/613384/>.
- [27] Persistent Memory Programming, <https://pmem.io>.
- [28] J. Katcher, Postmark: A new file system benchmark, technical Report TR3022. Network Appliance Inc. October 1997 (1997).
- [29] Condit J, Nightingale E B, Frost C, et al. Better I/O through byte-addressable, persistent memory[C]//Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. ACM, 2009: 133-146.
- [30] Wu X, Reddy A L. SCMFS: a file system for storage class memory[C]//Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, 2011: 39.
- [31] Zeng K, Lu Y, Wan H, et al. Efficient storage management for aged file systems on persistent memory[C]//Proceedings of the Conference on Design, Automation & Test in Europe. European Design and Automation Association, 2017: 1773-1778.
- [32] Wu, L., Zhuge, Q., Sha, E. H. M., Chen, X., & Cheng, L. (2018). DWARM: A wear-aware memory management scheme for in-memory file systems. *Future Generation Computer Systems*.
- [33] Zhao N, Anware A, Cheng Y, et al. Chameleon: An adaptive wear balancer for flash clusters[C]//2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2018: 1163-1172.
- [34] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A prototype phase change memory storage array. In Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems, HotStorage'11, pages 2-2, Berkeley, CA, USA, 2011. USENIX Association.
- [35] Kawahara T. Scalable spin-transfer torque ram technology for normally-off computing[J]. *IEEE Design & Test of Computers*, 2010 (1): 52-63.
- [36] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L. Lung, and C. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465-479, July 2008.
- [37] M. Wu and W. Zwaenepoel. eNVy: A Non-volatile, Main 338 14th USENIX Conference on File and Storage Technologies (FAST'16) USENIX Association Memory Storage System. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-VI, pages 86-97, New York, NY, USA, 1994. ACM.
- [38] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In Presented as part of the 11th USENIX Conference on File and Storage Technologies, FAST'13, pages 73-80, San Jose, CA, 2013. USENIX.
- [39] D. Narayanan and O. Hodson. Whole-system Persistence with Non-volatile Memories. In Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012). ACM, March 2012.
- [40] D. E. Lowell and P. M. Chen. Free Transactions with Rio Vista. In SOSP'97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles, pages 92-101, New York, NY, USA, 1997. ACM.
- [41] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In SOSP'93: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, pages 146-160, New York, NY, USA, 1993. ACM.
- [42] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In ASPLOS'11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, 2011. ACM.
- [43] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In Proceedings of the 12th USENIX Conference on File and Storage Technologies, FAST'14, pages 1-16, Santa Clara, CA, 2014. USENIX.
- [44] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems, TRIOS'13, pages 1:1-1:17, New York, NY, USA, 2013. ACM.
- [45] S. Venkataraman, N. Tolia, P. Ranganathan, and R. Campbell. Consistent and durable data structures for non-volatile byteaddressable memory. In Proceedings of the 9th USENIX Conference on File and Storage Technologies, FAST'11, San Jose, CA, USA, February 2011.
- [46] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In 13th USENIX Conference on File and Storage Technologies, FAST'15, pages 167-181, Santa Clara, CA, Feb. 2015. USENIX Association.