Efficient Storage Management for Aged File Systems on Persistent Memory

Kaisheng Zeng[†§], Youyou Lu[†§], Hu Wan[‡], and Jiwu Shu[†§*]
[†]Department of Computer Science and Technology, Tsinghua University, Beijing, China
[§]Tsinghua National Laboratory for Information Science and Technology, Beijing, China
[‡]College of Information Engineering, Capital Normal University, Beijing, China
cks14@mails.tsinghua.edu.cn, luyouyou@tsinghua.edu.cn, wanhu@cnu.edu.cn, shujw@tsinghua.edu.cn

***Abstract*—Emerging *persistent memories* (PMs) provide both byte addressability as DRAM and persistency as conventional storage technologies. Recent work on persistent memory file systems, such as BPFS, PMFS, have gained better performance by leveraging the dual characteristics. However, we observe that persistent memory file systems experience dramatic performance degradation over a long run. This phenomenon is referred to as file system aging. We find that the performance degradation is attributed to the inefficiency of storage management for both file space and dentry space. We also find that persistent memories wear out more quickly as file system ages.**

To address such issues, we propose SanGuo, a novel scatter-gather storage management mechanism for aged file systems on persistent memory. SanGuo consists of two key techniques. First, *Scatter-alloc* maximizes the efficiency and performance of file allocation while providing wear-leveling. Second, *Gather-free* accelerates the dentry operations, including dentry allocation, lookup and reclaim, especially for a directory file containing a large number of dentries. Experimental results show that SanGuo performs better wear-leveling while providing significant performance speed up (e.g., up to 10.33×, 8.9× respectively for Webproxy and Varmail workloads).

I. Introduction

Persistent memories (PMs), such as phase change memory (PCM), memristor, and 3D XPoint [1], promise to be orders of magnitude faster than existing storage technologies (e.g., disk and flash). PMs can be connected to CPUs via a direct memory access path, which offers the advantage of fast and fine-grained access. State-of-the-art file systems on persistent memory, such as BPFS [2], SCMFS [3], PMFS [4], HiNFS [5] and NOVA [6], exploit byte-addressability of persistent memory to avoid overheads of block-oriented storage.

We observe that some persistent memory file systems perform extremely well in a clean state. However, as files are created, modified, and deleted over a long run, they experience performance degradation. This problem is aggravated especially when running data-intensive applications, such as mail server applications that have millions of small (KB) files. We refer to such phenomenon as file system aging [7].

After careful analysis, we find that the performance degradation is attributed to two factors. First, both free and allocated space on the persistent memory become fragmented on a highly utilized file system. Allocation overhead of locating fragmented free space can have a noticeable impact on file system performance over the long run. Second, the latency of file operations, such as create and delete, increases dramatically when there is a large number of files in a directory, resulting in performance degradation. The root cause of this problem is the inefficiency of dentry (short for "directory entry") allocation, lookup and deallocation. A dentry is the glue that holds inodes and files together by relating inode numbers to file names. Most persistent memory file systems fail to efficiently manage dentries and free space in the directory file. As the number of files in a directory increases, it becomes increasingly difficult to find free space for new entries or give the in-use dentry space back to the directory file.

To make matters worse, persistent memories wear out more quickly as file systems age. Instead of allocating from other never used free space, we find that file systems frequently allocate file space from a previously freed area whenever possible. Though such allocation policies relieve fragmentation problem to some extent, it becomes a problem for persistent memories, which have low write-endurance in general. Since persistent memory is mostly directly managed by file systems, lack of wear-leveling in file systems leads to endurance problems.

In this paper, we present SanGuo[1], an aging-aware free space management mechanism on persistent memory-based storage system. It consists of two techniques, *Scatter-alloc* for coarse-grained free space management and *Gather-free* for fine-grained management. *Scatter-alloc* contains Flexible Bitmap Segment Tree to organize free space and Random Walk Algorithm to avoid localized writes. *Gather-free* contains Hashtable Mapped Free List to efficiently manage all directory file free space, and Dentry Index Tree to speed up directory entry lookups. We implement SanGuo in PMFS, a state-of-the-art persistent memory file system. The experimental results show that SanGuo performs better wear-leveling while providing significant performance speed up (e.g., up to 10.33×, 8.9× respectively for Webproxy and Varmail workloads).

II. Background and Motivation

A. Restructuring the I/O Stack for Persistent Memory

Recent studies [2], [4], [5], [8]–[10] show that running existing disk-based file systems on persistent memory fail to effectively exploit the persistent memory performance.

*Corresponding author

[1]**S**an**G**uo, pronounced "san-gwo", means the Three Kingdoms period in Chinese history. Our **S**catter-**G**ather storage management design is analogous to the process of division and reunification in the Three Kingdoms era.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

Because data I/O requests have to travel through a deep stack of software layers. To this end, state-of-the-art file systems on persistent memory bypass the OS page cache and the generic block layer, and access the byte-addressable persistent memory directly. However, the aging problem is mostly ignored in current file system research for persistent memory. Unfortunately, we find that some persistent memory file systems (e.g., PMFS) run in high performance at the beginning, but encounter a dramatic performance drop over the long run.

In addition to the performance exploration, the wear out problem of persistent memory is another issue that is better to be considered in file system designs. Most persistent memories endure limited number of writes. For example, PCM endures $10^8 - 10^9$ writes [11], which is limited for memory-level accesses. Unfortunately, the wear out issue has not been well studied in existing file systems.

We use PMFS as an example to analyze its aging problem on both performance and wear out issues.

B. Issues of Free Space Management in PMFS

Free space management is important for file systems on persistent memory. A file system must track which inodes and data blocks are free, and which are not, so that when a new file is created, the file system can find space for it. To keep track of free space, a file system maintains a free-space list. Frequently, the free-space list is implemented as a free list or in-use list.

PMFS uses a global in-use list to link all allocated blocks together. Each entry in the list records the starting and ending physical block number of a continuous space. This in-use list is sorted by physical block number in an increasing order. Therefore, it allows PMFS to track free blocks. When a block is needed, PMFS takes the first free block and adds it to the in-use list accordingly. When a block is reclaimed, PMFS traverses the in-use list to find the list entry containing this block, then removes it. As files are created and deleted, fragmentation of free space occurs. In-use list may grow larger, and result in poor lookup performance, especially for reclaiming data blocks. Meanwhile, in order to maintain data consistency, PMFS holds a mutex for updating the in-use list, resulting in scalability bottleneck. More seriously, with this free space management mechanism, PMFS frequently allocates file space from a previously freed area. There is potential for some workloads to result in "hot" locations. Although persistent memory, such as PCM, has much higher write endurance than NAND flash, it will still be worn out for memory-level accesses [2].

We use Filebench [12] to generate write-intensive workloads to illustrate the persistent memory wear out issue. We collect statistics of write operations for each block, as shown in Fig. 1. We make two observations. First, the overall distribution of write operations is imbalanced for all blocks. There are updates concentrated on some certain blocks. Second, there are some empty spaces in the rear section of persistent memory. This is mainly due to current free space management mechanism in PMFS fully reuse previously freed area for new files instead of allocating the never-used free space.

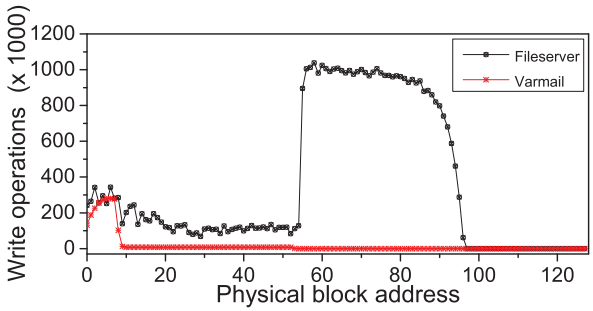


Fig. 1: Write distribution in PMFS.

C. Inefficiency of Dentry Management in PMFS

A directory entry, or dentry, is what the Linux kernel uses to keep track of the hierarchy of files in a directory. More specifically, a directory file contains an array of directory entries. Each directory entry consists of an inode number and a name of the corresponding file. Fig. 2 shows how directory entries are represented in PMFS. Of all the fields in a dentry, `name_len` is the length of the directory entry name string; `de_len` is an offset pointer indicating the number of bytes between the end of the `de_len` field and the end of the *next* `de_len` field. PMFS makes use of these two to manage free dentry space. To delete an entry from a directory, the `de_len` of the previous entry before the entry to be deleted is set to a value that points to the next entry after it. In turn, to add a new entry to the directory, PMFS scans entries in the directory one by one to calculate available free space according to `de_len` and `name_len`.

ino (inode number)	de_len	name_len	file_type	name							
	16	1	2	.	\0	\0	\0				
	16	2	2	.	.	\0	\0				
	40	5	7	l	i	n	u	x	\0	\0	\0
	20	6	1	r	m	f	i	l	e	\0	\0

Fig. 2: Representation of directory entries in PMFS.

With this dentry organization, however, after files or subdirectories in a directory are deleted, all reclaimed dentry space may scatter in the directory file space. For a directory containing a large number of files, the performance overhead of lookup free space may increase dramatically. Fig. 3 shows the time breakdown for Varmail workload in PMFS [4] as the file sets increases. We find that up to 83.5% time is spent on dentry related operations when the number of files is 32K.

D. Motivation

The above observations have intriguing implications on the design of file system to fully realize persistent memories' potential. On one hand, the persistent memory wear out issue due to free space management should be considered. On the other hand, scattered directory entries increase performance overhead. Hence, it is important to have a global view of free dentry space. The lack of global view of free dentry

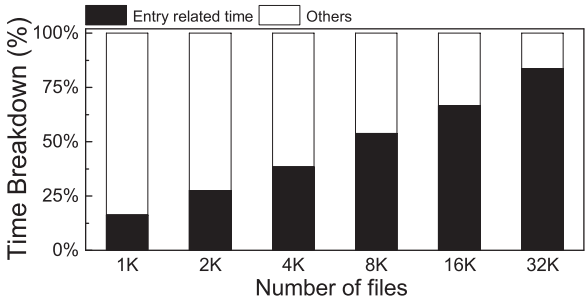


Fig. 3: Time breakdown for Varmail workload in PMFS.

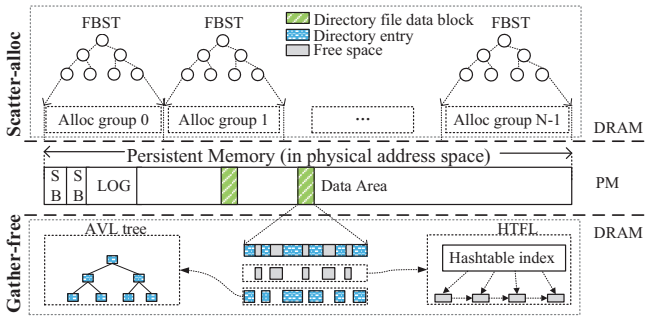


Fig. 4: SanGuo data structure layout.

space is the underlying cause of this inefficiency. Motivated by these observations, we propose SanGuo, a scatter-gather storage management mechanism to improve the equal repartition of write access to the memory and improve the storage management efficiency.

III. SanGuo Design and Implementation

A. Design Overview

SanGuo provides a hybrid coarse- and fine-grained free space management mechanism for aged file systems on persistent memory. Fig. 4 shows the high-level layout of SanGuo's data structure. For coarse-grained free space management, we use *Scatter-alloc* technique to manage the free space of persistent memory at a block granularity (e.g., 4KB). To make persistent memory allocation and reclaim faster, *Scatter-alloc* divides the entire persistent memory into multiple allocation groups to avoid the allocation contention and allow for parallel scanning in failure recovery. For example, we can set the number of allocation groups same as that of CPU cores. Each group is managed by a novel variant of segment tree, FBST (Flexible Bitmap Segment Tree), thus providing $O(\log n)$ allocation and reclaim. In addition, by enforcing random walk algorithm, *Scatter-alloc* ensures that all free blocks are given an equal chance to be allocated, thereby mitigating the stress problem on certain blocks. For fine-grained free space management, we propose *Gather-free* technique to manage free space of directory files at a byte granularity. In order to speed up free space allocation and reclaim in directory files, *Gather-free* employs hashtable mapped free list (HTFL) to efficiently manage all free space. To speed up the lookup and deletion operations of dentry, *Gather-free* keeps an AVL tree in DRAM for each directory file to index the hash value of dentry name string.

B. Scatter-alloc

Flexible Bitmap Segment Tree. We develop a new variant of segment tree, i.e., FBST, to efficiently manage a region of file system free space on persistent memory. FBST is a segment tree, and each leaf in this tree has a bitmap. Segment tree is a data structure which can be used to perform range queries and range updates. The underlying principle behind this data structure is to store certain values of ranges as a balanced binary tree, and hence queries and updates can be performed efficiently. We use segment tree to represent a range of physical persistent memory. Further, we use bitmaps to keep track of free space in each leaf node. By default, each bit represents a 4 KB block, but it can also be used to represent 2 MB, or 1 GB block to enable transparent large page support. Each block is represented by a single bit. If a block is free, the bit is 0; if a block is allocated, the bit is 1. Fig. 5 shows the overview of an FBST and its node layout.

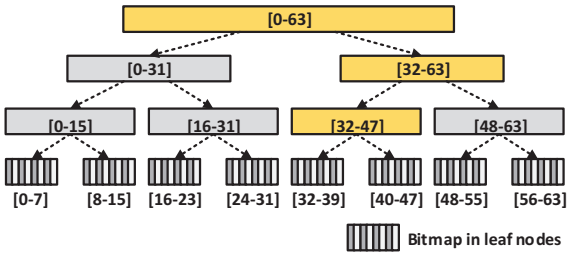


Fig. 5: FBST overview and node layout.

Consider an allocation group with N blocks (e.g., N equals 64 in Fig. 5), referred to as array A[N], and a corresponding FBST:

- The root of FBST represents the whole blocks in this allocation group, i.e., range from block index 0 to $N-1$.
- Each leaf in the FBST represents a single *atomic segment* of size M. The number of leaf nodes is $\lceil N/M \rceil$.
- The internal nodes in the FBST represent union of atomic segments.
- Each node also records the number of free blocks in this segment.

The root of the FBST represents the whole array A[0:N-1]. Then we break the segment into half, left son of the root represents the A[0:(N-1)/2] and right son represents A[(N-1)/2+1:(N-1)]. So in each step we divide the segment into half and the two children represent the two halves. So the height of the segment tree is $\log_2 \lceil N/M \rceil$.

Random Walk Algorithm. From the perspective of file system, frequently allocated data blocks are more prone to be worn out. To address such issue, a key idea is that each free block should have the same opportunity to be allocated. To this end, *Scatter-alloc* implements an efficient random walk policy to achieve wear-aware space allocation, as shown in Algorithm 1. Our random walk policy performs space allocation as follows. We first choose an allocation group which contains enough free blocks via a random method.

After that, it takes $\log_2 \lceil N/M \rceil$ steps to find a certain atomic segment. Our algorithm ensures a high degree of randomness for each step. Finally, a free block is allocated from the atomic segment bitmap randomly.

Algorithm 1: RANDOMWALKALLOCATION

```
Input: G: number of allocation groups; AG: allocation groups; N: size of AG; M:
       size of atomic interval; FBSTs: FBST for each allocation group
Output: Free block index
Algorithm RandomWalkAllocation()
    gid ← random()%G;
    find_flag ← false;
    for i ← 0 to G − 1 do
        if AG[gid].num_of_free >= 1 then
            find_flag ← true;
            break;
        gid ← (gid + 1)%G;
    if find_flag == false then
        return Out-of-Persistent-Memory;
    p ← FBSTs[gid].root;
    block_idx← FindFreeBlock(p);
Procedure FindFreeBlock(p)
    p.num_of_free ← p.num_of_free − 1;
    if p is a leaf node then
        location ← randomly find a 0 bit in atomic interval bitmap;
        return location;
    else
        R ← random();
        if R%2 == 0 then
            if p→leftson.num_of_free >= 1 then
                return FindFreeBlock(p→leftson);
            if p→rightson.num_of_free >= 1 then
                return FindFreeBlock(p→rightson);
        else
            if p→rightson.num_of_free >= 1 then
                return FindFreeBlock(p→rightson);
            if p→leftson.num_of_free >= 1 then
                return FindFreeBlock(p→leftson);
```

Allocation and Reclaim. When a block is needed, we first employ random walk algorithm to choose an allocation group. Then we traverse the FBST from root to the leaf to find a free block, and reflect the corresponding change, i.e., number of free blocks, in each level of the FBST. When a block is reclaimed, we can query on a segment and return the position on that particular segment, and mark it as free.

C. Gather-free

Hashtable Mapped Free List. We use a per-directory free list to keep track of free space for directories. Each free list is sorted by the offset to the beginning of the directory file. Each entry in the free list represents a range of free space in a certain directory file block. When a new dentry is needed, an adequate sized free space is easily acquired from the head of the free list. When reclaiming a dentry space, we traverse the free list and insert it into the free list at a desired position.

However, as dentries are allocated and reclaimed frequently, it is prone for a free list to grow dramatically, especially when a directory contains a large number of files. As a result, the time spent on finding a desired position when reclaiming a dentry increases significantly. For a large directory file, it always contains multiple blocks for storing dentries. In order to reclaim free space quickly, we maintain a hashtable and use block number as its key to accelerate the locating process. Each block is mapped to the entry which represents its first free space in the free list, as shown in the Fig. 6.

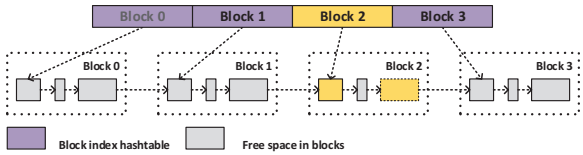


Fig. 6: Hashtable mapped free list overview.

For example, when one dentry in block 2 needs to be reclaimed, *Gather-free* first finds the first free space entry which has the same block number 2. Then, *Gather-free* can find the location according to its offset, denoted by dotted lines in Fig. 6, and simply add it to the free list. Alternatively, it is possible to examine the free list to establish whether there are adjacent free space to the current free space entry that has been freed. If there are, the adjacent free space entries can be merged into a single, larger free space entry.

Dentry Index Tree. In order to support quick dentry lookups, *Gather-free* maintains an AVL tree, a self-balancing binary search tree, in DRAM for each directory file. Each node stores dentry name as its key, and points to the corresponding dentry. The AVL tree makes search efficient even for large directories. It is worth noting that, differ from directory cache in Linux virtual file system (VFS) layer which caches dentry structures for better path lookup performance, *Gather-free* aims at improving dentry lookups in a directory file.

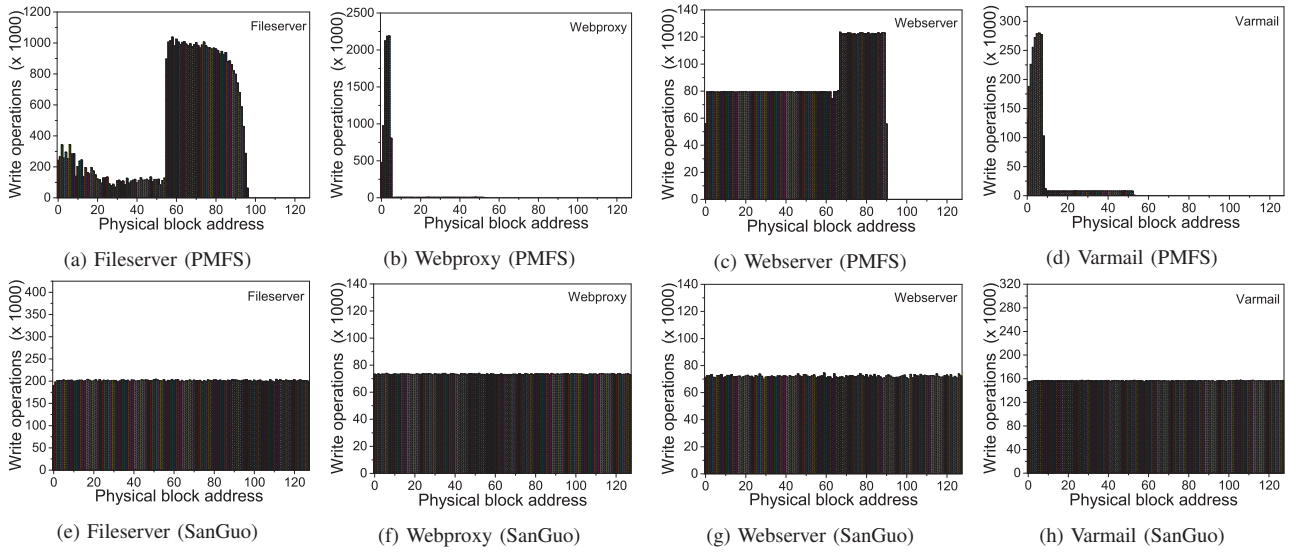
D. Discussion

It is crucial for a file system to survive arbitrary crashes or power failures. SanGuo adopts the same consistent and recoverable update mechanisms, i.e., transaction and fine-grained logging, used in PMFS for consistency. On a clean unmount, SanGuo stores the bitmap of FBST in the reserved log area and restores the allocator during the subsequent remount. In case of a failure, SanGuo rebuilds the allocator structures by walking the file system B-tree during recovery. By leveraging processor features to use atomic in-place updates whenever possible, SanGuo allows efficient and reliable updates to persistent state for dentries. Volatile data structures for dentry acceleration can be initialized from dentries on persistent memory at every boot with light rebuilding overhead.

IV. EVALUATION

A. Experimental Setup

We implement SanGuo storage management mechanism in PMFS [4], a state-of-the-art persistent memory file system. In this evaluation, we take PMFS as the baseline file system, and compare it with SanGuo to show the effectiveness of our proposed techniques. All experiments are conducted on an Intel 12-core machine with six Intel(R) Xeon E5-2620 processors. The operating system is RHEL 6.3, with Linux kernel 3.11.0. We take the DRAM to stand for the NVM as



(a) Fileserver (PMFS) (b) Webproxy (PMFS) (c) Webserver (PMFS) (d) Varmail (PMFS)

(e) Fileserver (SanGuo) (f) Webproxy (SanGuo) (g) Webserver (SanGuo) (h) Varmail (SanGuo)

Fig. 7: Comparision of write distribution in PMFS and SanGuo.

the persistent memory [2]. We plug in 16 GB DRAM where 4 GB is reserved for PMFS.

Workloads. We use Filebench [12], a widely used benchmark for file systems, to evaluate the application-level performance. We select four typical workloads: Fileserver, Webproxy, Webserver and Varmail. Table I summarizes the specific parameters of the four workloads. We pre-allocate file sets to half of the file system capacity (i.e., 2 GB) for each test to simulate aged file systems. We vary the number of threads from 1 to 8 for scalability evaluation. Unless otherwise specified, all the experiments are run with eight threads by default. Webproxy and Varmail workloads run for 1000 seconds, and Fileserver and Webserver workloads run for 60 seconds due to the space limitation. We run each test five times and report the average.

TABLE I: Filebench workload specification.

Workload	I/O size	R/W ratio	Threads	# of files	Dir operation intensive
Fileserver	512 B	1:2	1-8	8K∼32K	No
Webproxy	512 B	5:1	1-8	8K∼32K	Yes
Webserver	512 B	10:1	1-8	8K∼32K	No
Varmail	512 B	1:1	1-8	8K∼32K	Yes

B. Impact on Endurance

To understand the effectiveness of wear-leveling by adopting *Scatter-alloc* technique, we collect write access information of each Filebench workload at block granularity. For clarity purpose, we divide the persistent memory into 128 intervals, and calculate the total number of writes in each interval. We run workloads in single-thread mode to ensure the accuracy of result. Fig. 7 shows the distribution of block writes on PMFS and PMFS with SanGuo mechanism (denoted as SanGuo).

Fig. 7(a) - 7(d) show the results on PMFS. These four workloads exhibit different write access pattern. Fileserver workload's write accesses are usually in the *frontal and central* sections of the physical persistent memory, especially in the *central* section. Because the Fileserver workload performs a sequence of creates, writes, appends, and deletes operations on a directory tree, resulting in continual allocating new free space and leading to a broad access section. In contrast, Varmail workload has a strong spatial locality. This is because Varmail workload performs delete operations before performing a set of create-append-sync operations, thus providing more opportunities to reuse the freed space in the frontal section of the physical persistent memory. Due to similar reasons, read-intensive workloads Webproxy and Webserver show different write access patterns.

However, we find that PMFS is less resistant to persistent memory wear out for all workloads. First, all four workloads show the potential to result in "hot" locations due to concentrate updates. In particular, intervals from one to six of Webproxy workload seem to be more vulnerable to damage from persistent memory wear. Second, there are contiguous free zones in the *rear* section in PMFS for all workloads, for example, intervals starting from 97, 53, 91 and 54 to last interval 128 for Fileserver, Webproxy, Webserver, and Varmail, respectively. Because the file system is not completely full (we pre-allocate file sets to half of the total file system capacity), PMFS reuses freed space due to file deletion in the *frontal* section of persistent memory as much as possible. As a result, the frontal and central sections are prone to be wear-out.

Results using SanGuo mechanism are illustrated in Fig. 7(e) - 7(h). As we can see, all write accesses are well distributed over all intervals. The number of peak write operation of PMFS is higher than that of SanGuo by 5.08×, 29.63×, 1.65×, and 1.78× for Fileserver, Webproxy, Webserver, and Varmail, respectively. These results demonstrate the effectiveness of SanGuo's wear aware allocation technique. Note that, since we only show total write counts for all blocks in an interval other than independent write counts for each block in Fig. 7, blocks without any write access are hidden in the interval instead of

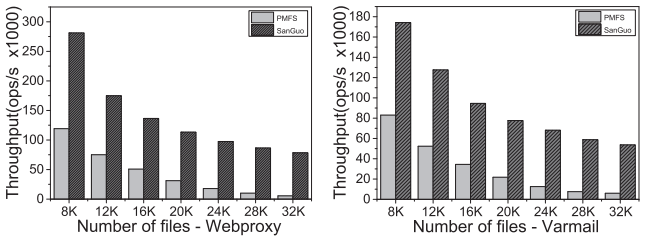
[2]We focus on storage management mechanism in this paper, and leave the evaluation on NVMs which have higher write latency relative to DRAM to our future work.

non-existent.

C. Impact on Performance

We choose Webproxy and Varmail workloads from Filebench to evaluate performance and scalability because they are sensitive to directory operations while Fileserver and Webserver are not. We run them on different structures of directories via varying the total number of files (directory width remain the same as the default). We use the number of operations per second, which is reported by the Filebench benchmark, as the performance metric.

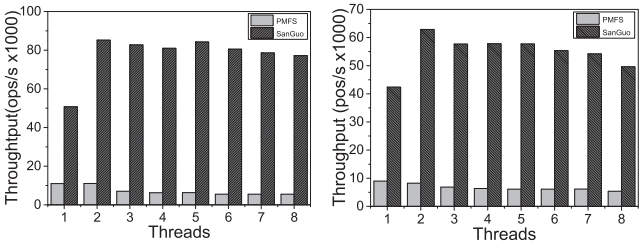
1) Overall Performance: We first evaluate the overall performance. Fig. 8 shows the IOPS results with various number of files for Webproxy and Varmail workloads. As shown in the figure, SanGuo achieves better performance for both evaluated workloads. For instance, SanGuo outperforms PMFS by 2.36× and 2.1× when the number of files is 8K for Webproxy and Varmail workload, respectively. This is attributed to the fact that SanGuo can get more performance benefits from the optimization of fast directory entry operations. We find that the performance of PMFS and SanGuo degrades as the number of files increases. This is because PMFS uses a flat and array-based structure to organize directory entries, and that reduces the performance when each directory holds large entries. However, we can observe that the performance benefits of SanGuo become more obvious with larger number of files. For example, SanGuo improves the performance by up to 14.33×, 8.9× over PMFS as the number of files increases to 32K for Webproxy and Varmail workloads, respectively. Because of using an efficient index structure to accelerate the array-based dentry operations, the performance degradation in SanGuo shows a nonlinear falling curve with the increasing number of files, compared to PMFS's near-linear performance degradation.



(a) Webproxy (b) Varmail

Fig. 8: Performance of PMFS and SanGuo.

2) System Scalability: We demonstrate the scalability of the SanGuo mechanism. Fig. 9 shows the throughput for the two Filebench workloads as we vary the number of threads. SanGuo scales better than PMFS as the thread number goes from one to two. SanGuo provides 70% performance improvement for Webproxy workload and 48.4% performance improvement for Varmail workload. In comparison, PMFS shows no performance improvement, and even 8.3% performance degradation for Webproxy and Varmail workloads, respectively. The performance of PMFS and SanGuo are gradually limited by the array-based directory entry organization when going from 2 to 8 threads. We intend to explore more efficient dentry organization to provide better scalability in the future.



(a) Webproxy (b) Varmail

Fig. 9: Throughput (operations per second) for 1-8 threads.

V. Conclusion

In this paper, we study the aging problem of persistent memory file systems, which hurts the long-term performance. Through analysis, we find that both free space and dentry management lead to fragmentations, which are even worse for persistent memories that provide all accesses in memory. Also, existing space allocation accelerates the wear-out process of persistent memory. To this end, we propose a scatter-gather storage management mechanism, SanGuo, with efficient data structures. Experiments show that SanGuo achieves effective wear-leveling while bringing significant performance improvements.

VI. Acknowledgments

This work is supported by National Natural Science Foundation of China (Grant No. 61433008, 61502266), Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), and China Postdoctoral Science Foundation (Grant No. 2016T90094, 2015M580098).

References

[1] Intel and Micron, "Intel and micron produce breakthrough memory technology," https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/, July 2015.

[2] J. Condit, E. B. Nightingale, C. Frost *et al.*, "Better I/O through byte-addressable, persistent memory," in *SOSP*. ACM, Oct. 2009.

[3] X. Wu and A. Reddy, "Scmfs: a file system for storage class memory," in *SC*. ACM, 2011.

[4] S. R. Dulloor, S. Kumar, A. Keshavamurthy *et al.*, "System software for persistent memory," in *EuroSys*. ACM, Apr. 2014.

[5] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *EuroSys*. ACM, 2016.

[6] J. Xu and S. Swanson, "Nova: a log-structured file system for hybrid volatile/non-volatile main memories," in *FAST 16*, 2016.

[7] K. A. Smith and M. I. Seltzer, "File system agingincreasing the relevance of file system benchmarks," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 25, no. 1. ACM, 1997.

[8] S. Swanson and A. M. Caulfield, "Refactor, reduce, recycle: Restructuring the i/o stack for the future of storage," *Computer*, vol. 46, no. 8, 2013.

[9] A. Badam, "How persistent memory will change software systems," *Computer*, vol. 46, no. 8, 2013.

[10] Y. Lu, J. Shu, and L. Sun, "Blurred persistence in transactional persistent memory," in *MSST*. IEEE, May 2015.

[11] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1537–1550, 2016.

[12] V. Tarasov, Z. Erez, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *;login:*, vol. 41, no. 1, March 2016.