Efficient Multi-Grained Wear Leveling for Inodes of Persistent Memory File Systems

Chaoshu Yang†‡, Duo Liu†‡,*, Runyu Zhang†‡, Xianzhang Chen†‡,*, Shun Nie†‡, Fengshun Wang†‡, Qingfeng Zhuge§, and Edwin H.-M. Sha§

† Key Laboratory of Dependable Service Computing in Cyber Physical Society (Chongqing University), Ministry of Education, China

‡ College of Computer Science, Chongqing University, Chongqing, China

§ School of Computer Science and Software Engineering, East China Normal University, Shanghai, China.

***Abstract*—Existing persistent memory file systems usually store inodes in fixed locations, which ignores the external and internal imbalanced wears of inodes on the persistent memory (PM). Therefore, the PM for storing inodes can be easily damaged. Existing solutions achieve low accuracy of wear-leveling with high-overhead data migrations. In this paper, we propose a Lightweight and Multi-grained Wear-leveling Mechanism, called LMWM, to solve these problems. We implement the proposed LMWM in Linux kernel based on NOVA, a typical persistent memory file system. Compared with MARCH, the state-of-the-art wear-leveling mechanism for inode table, experimental results show that LMWM can improve 2.5× lifetime of PM and 1.12× performance, respectively.**

***Index Terms*—Persistent memory, wear leveling, file system, inode management**

I. Introduction

Emerging Persistent Memories (PMs), such as Phase Change Memory (PCM) [1], [2] and 3D XPoint [3], are promised to revolutionize storage systems by providing non-volatility, byte-addressability, and low latency. Many persistent memory file systems, such as PMFS [4], SIMFS [5], NOVA [6], and and HiNFS [7], are developed to achieve high performance by exploiting the advanced features of PMs.

Unfortunately, the PMs have the problem of limited endurance [8]–[12]. Common file operations, such as $write$, $open$, $link$, $read$, and $unlink$, cause heavy writes on the PM for storing the file index node (i.e., inode) since all the operations modify the corresponding inodes. Furthermore, inodes are stored in fixed locations throughout their lifetime. As a consequence, the inode table is one of the most easily worn-out section in a persistent memory file system, which seriously damage the reliability of the file systems.

Unfortunately, existing wear-leveling mechanisms [10], [13], [14], such as MARCH [14] and VInode [10] do not fully exploit the features of inodes. Most persistent memory file systems, PMFS [4], SIMFS [5], and NOVA [6], store the variables of an inode by two 64B parts. However, we find that the total size of the frequently-updated variables in an inode is less than 64B. As a result, the asymmetric update between the two parts of an inode can lead to badly imbalanced wear inside the inodes. Therefore, these inode-sized wear-leveling mechanisms, such as MARCH and VInode, still lead to heavy overhead of data migration and low accuracy for achieving wear-leveling of the inode table. It is possible to alleviate the overhead and improve wear accuracy of the inode-sized wear-leveling mechanisms by achieving 64B-sized wear-leveling of the inode table. The higher wear accuracy means the more precise wear of PM cells. Moreover, the 64B wear-leveling mechanism will bring large overhead for metadata management. Therefore, the inode wear-leveling mechanism should handle the trade-off between the wear accuracy and the overhead of both data migration and metadata management.

In this paper, we propose a Lightweight and Multi-grained Wear-leveling Mechanism, called LMWM, to solve these problems. The main idea of LMWM is to adopt 64bytes-sized and page-sized (e.g., 4KB) data migration granularity for achieving wear-leveling of inode table in PM. LMWM consists of two techniques, Fine-grained Wear-leveling Mechanism (FWM) and Coarse-grained Wear-leveling Mechanism (CWM), which are used to achieve wear-leveling of all inodes within a page and among all inode pages of the persistent memory file system. It is worth noting that LMWM is the first solution to consider the imbalanced wear degree inside inodes.We implement the proposed LMWM in Linux kernel based on NOVA [6]. Experiments are conducted with two widely-used workloads OLTP and Webserver of Filebench [15]. Compared with MARCH, the state-of-the-art wear-leveling mechanism for the inode table, LMWM can improve 2.5× lifetime of PM and 1.12× performance on average, respectively.

The main contributions of this paper are as follows:

- We conduct in-depth investigations to reveal the serious imbalanced wear problem of PM for inodes.
- We present a new mechanism, LMWM, to achieve high performance and accuracy wear leveling for inodes.
- We implement and evaluate LMWM in the Linux Kernel. The experimental results show that LMWM outperforms existing solutions.

The rest of this paper is organized as follows. In Section II, we introduce the background and discuss the motivation. We present the design and implementation of LMWM in Section III. The experimental results are detailed in Section IV.

*Corresponding author: Xianzhang Chen (xzchen@cqu.edu.cn) and Duo Liu (liuduo@cqu.edu.cn).

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

Finally, we draw a conclusion in Section V.

II. Background And Motivation

A. Existing Persistent Memory File System

Emerging persistent memory file systems, such as PMFS [4], SIMFS [5], NOVA [6], and HiNFS [7], exploit the advanced characteristics of PMs to avoid overheads of block-oriented I/O stacks and thereby achieve higher performance. However, PMs suffer from limited endurance, e.g., a PCM cell can only sustain 10^8 writes [10]. The limited endurance problem of underlying PMs seriously damages the reliability of persistent memory file systems.

The file index node (i.e., inode) of file systems is used to record the basic information of a file. For example, property-related variables, such as $filesize$, $linkcount$, $ctime$, $mtime$, and $blockcount$, are used to record the total file size, the total number of links, inode modification time, file data index modification time, and the total number of blocks for a file, respectively. Common file operations need to update these variables. Thus, the inodes will be modified frequently. Moreover, whether the persistent memory file systems adopt an array (e.g., SIMFS) or a tree (e.g., PMFS and NOVA) to organize inode table, the inodes are stored in fixed locations throughout their lifetime. Therefore, the inode table is one of the most heavily worn sections in the underlying PM.

B. Wear-leveling Problem of Inodes

Since the inodes are frequently updated, many wear-leveling mechanisms, such as MARCH [14] and VInode [10], have been proposed to protect the underlying PMs for storing the inodes of persistent memory file systems. MARCH adopts a window-based marching strategy to change the mapping between logical inodes and inode slots in the physical persistent memory. VInode breaks the bind between logical inodes and inode slots by virtualizing the inode table and balances the wear of inode slots by migrating the heavily written inodes.

However, existing inode wear-leveling mechanisms all balance the wear among inodes, which are unaware of the imbalanced updates of variables within each inode. The size of an inode is 128B and the frequently-updated variables of an inode are concentrated in the first 64B. Assuming the size of an inode slot is 64B, each inode needs two slots to store its variables. Therefore, the first slot that keeps the frequently-updated variables is worn much heavier than the second slot.

C. Motivation

To reveal the problem of imbalanced wear inside inode, we measure the write differences between the two slots of each inode in NOVA [6]. The experiments are conducted with the widely-used workloads, $Webserver$ and $Oltp$, of Filebench [15]. We use the default setting of these workloads. We record the write counts in the first slots and the second slots for each inode.

As shown in Fig. 1, the write counts of the first slots are far more than the second slots. Concretely, the average write counts of the first slots is 351× and 7433× that of the second slots for $Webserver$ and $Oltp$, respectively. Accordingly, the first slots suffer heavy wear than the second slots of all inodes in persistent memory file systems.

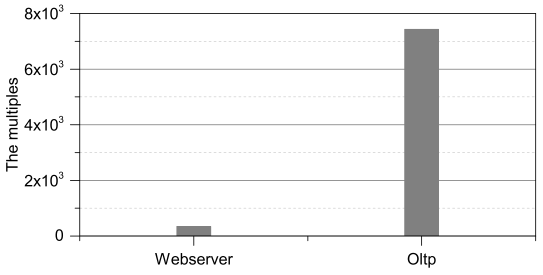


Fig. 1: Comparison of writes between 1st and 2nd slots.

In summary, existing wear-leveling mechanisms for inodes can lead to serious imbalanced wear inside inodes. It is necessary to revisit the wear-leveling mechanism in a finer level. On the other hand, a finer wear-leveling mechanism may bring large overhead for metadata management. Hence, there is a trade-off between the wear accuracy and the overhead both data migration and metadata management of inodes in the inode wear-leveling mechanism.

III. Design and Implementation

A. Overview

In this section, we propose a Lightweight and Multi-grained Wear-leveling Mechanism (LMWM), to effectively achieve accurate wear leveling of inodes. LMWM divides an inode into two 64B parts and adopts two 64B slots to store the variables of the inode. Specifically, the two slots of an inode are not required to be continuous. LMWM can achieve low-overhead and high-accuracy wear-leveling by migrating 64B-sized data instead of inode-sized data migration.

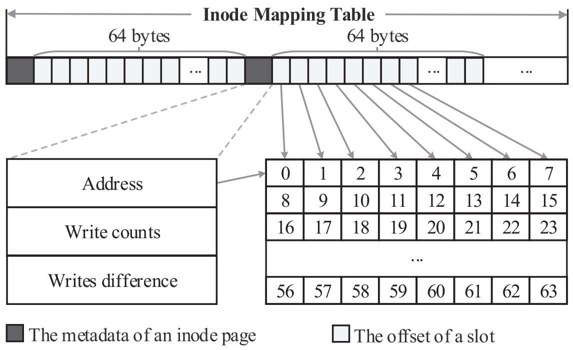


Fig. 2: The architecture of LMWM.

The inodes are orderly saved in the inode table following the inode numbers. Each inode page is divided into 64 slots where each inode occupies two slots. To achieve 64B-sized data migration in an inode page, LMWM adopts an *Inode Mapping Table* to record the offsets of the 64 slots, as shown in Fig. 2. LMWM can change the location of an inode by revising the corresponding offset. In detail, the *Inode Mapping Table* consists of a set of entries correspond to inode pages.

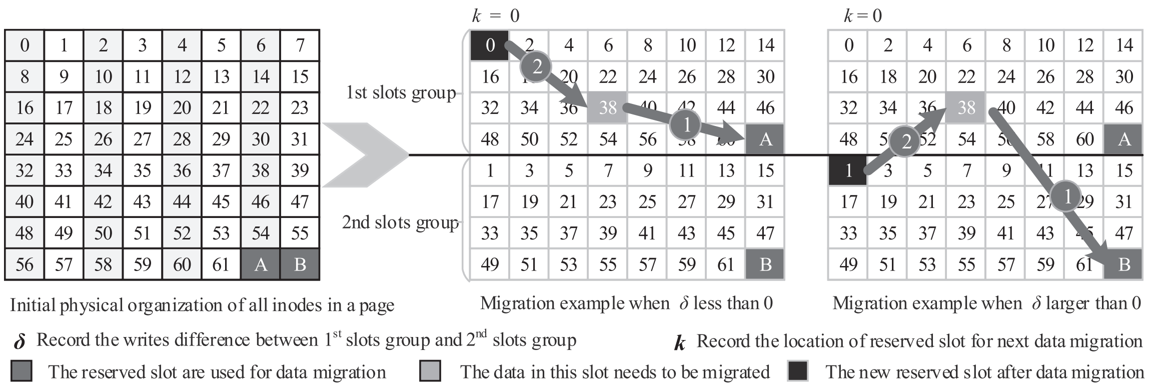

k = 0
1st slots group
2nd slots group
Initial physical organization of all inodes in a page
Migration example when δ less than 0
Migration example when δ larger than 0
δ Record the writes difference between 1st slots group and 2nd slots group
k Record the location of reserved slot for next data migration
The reserved slot are used for data migration
The data in this slot needs to be migrated
The new reserved slot after data migration

Fig. 3: The example of data migration.

Moreover, an entry also contains the metadata of an inode page. As Fig. 2 shows, the metadata consists of the $Address$, $Write\ counts$ (denoted by t), and $Writes\ difference$ (denoted by δ), which are used to record the address, total write counts, and the difference between the total write counts of the first and the second slots in an inode page.

The space overhead of LMWM is that, $Address$, t, δ, and offset array occupies 8 bytes, 4 bytes, 4 bytes, and 64 bytes for an inode page, respectively. Accordingly, an inode page only requires 80B-sized metadata, i.e., the size of an entry in *Inode Mapping Table* is 80 bytes. Moreover, two slots are reserved as the swapping slots for an inode page (mentioned in § III-B). Then, the space overhead of LMWM is only about 5% of the inode table. Furthermore, the storage area of variable t and δ suffer from heavy wear, since they will be updated each time the inode is being updated. To solve this issue, LMWM employs a copy of them in DRAM to reduce the writes to PM.

The proposed LMWM consists of two key techniques, Fine-grained Wear-leveling Mechanism (FWM) and Coarse-grained Wear-leveling Mechanism (CWM), to achieve 64B-sized wear-leveling inside inode page and page-sized wear-leveling between all inode pages, respectively. First, we will introduce the design and implementation of FWM (§ III-B) and CWM (§ III-C) in detail. Second, we discuss the wear-leveling of *Inode Mapping Table*.

B. Fine-grained Wear-leveling Mechanism (FWM)

All inodes are placed one after another in an inode page. To better illustrate the proposed FWM, we logically divide all slots of a page into two groups: the 1st group and the 2nd group consist of the even-number slots and odd-number slots, respectively. As shown in Fig. 3, during the initialization of file system, the 1st group and the 2nd group store the first or the second part of inodes, respectively. For the design of existing persistent memory file systems, the slots in the 1st group suffer from most heavy wear.

As shown in Fig. 3, FWM reserves two slots (denoted by A and B) as the swapping slots. The main idea of FWM is to move slot A or B ahead in 1st or 2nd group to achieve wear-leveling in an inode page. The range of moving A and B is limited in the 1st group and 2nd group, respectively. To determine the direction of migration, δ is used to quantify the difference of wears between the 1st and 2nd group of an inode page. We increase δ when write operations occur in the 1st group, and decrease it when the system only writes the 2nd group. If δ is larger than 0, it means that the wear degree of the 1st group is heavier than the 2nd group. Therefore, the frequently updated inodes in the 1st group will be migrated to the 2nd group.

To achieve effective and high accurate wear-leveling within inode pages, we need to answer two questions: (I) When to start a migration? (II) How to process the migration? For question (I), we set a threshold T to trigger the migrations. When t increases to a certain multiple of threshold T (that means $t \mod T = 0$), a migration is compulsively triggered to prevent excessive wears on a certain slot. For question (II), we record an offset k (i.e., $0 \leq k \leq 31$) to indicate the destination offset of the reserved slots (i.e., A and B) after a migration. The offset k increase progressively with the rounds of migrations.

Algorithm 1: Fine-grained Data Migration Algorithms

```
Input: t: The total write counts of an inode page; T: The write count
    threshold of migration; A and B: The reserved slot for swapping in 1st
    and 2nd group, respectively. S_i: The i_th slot that is being updated,
    assuming it belong to 1st group. k: Record the location of reserved
    slot for next data migration.
Output: NULL
1  if (0 ≠ (t mod T)) then
2      return;
3  else
4      if (δ > 0) then
5          Migrate the data of S_i into B;
6          Migrate the data of kst slot of 2nd group into S_i;
7      else
8          Migrate the data of S_i into A;
9          Migrate the data of kst slot of 1st group into S_i;
10     end
11     k ← (k +1) mod 32;
12     δ ← 0;
13     Flush the swapped data and metadata to PM;
14     Revise the corresponding offsets in Inode Mapping Table;
15 end
```

Algorithm 1 shows the fine-grained data migration algorithm where we assume the updated slot (denoted by S_i) is in the 1st group.

1) When t triggers a migration, we will check whether δ is positive or negative (line 4).

2) If δ is positive, which means there are more wears on 1st group, we firstly migrate the data of S_i into B (line 5). Secondly, we migrate the data of k_{th} slot in 2nd group into S_i (line 6).
3) If δ is negative or 0, which means there are more wears on the 2nd group or same wear between 1st and 2nd group, we migrate the data of S_i into A (line 8). Then, we migrate the data of the k_{th} slot in the 1st group into S_i (line 9).
4) After accomplishing the migration, we increase k by one to indicate the offset of A and B slot for the next round of migration (line 11). Moreover, we reset δ as zero for the next round of migration (line 12).
5) Flush the updated data to PM and revise the corresponding offsets in the *Inode Mapping Table* (lines 13-14).

To better understand the proposed FWM, as shown in Fig. 3, we take an example to illustrate the migration process of FWM. We assume that the data of the 38_{st} slot is being updated and k is 0. Meanwhile, the offsets of the A and B are 62 and 63, respectively. If δ is less than 0, the migration process is shown in the middle subfigure of Fig. 3. FWM moves the data of 38_{st} slot to the A in the 1st group. Otherwise, if δ is greater than 0, the migration process is shown in the right subfigure of Fig. 3. FWM moves the data of 38_{st} slot in the 1st group to B in the 2nd group. After that, k is set as 1 and the slot number of A is 0. Therefore, FWM uses δ and k to balance the wear inside an inode page.

C. Coarse-grained Wear-leveling Mechanism (CWM)

The CWM is used to resolve the imbalanced wears among inode pages. It is somehow simpler to achieve coarse-grained wear-leveling. As we have recorded the total number of wears t for each page, we can definitely distinguish the most-worn and least-worn pages. When the t of a certain page can be exactly divided by the threshold (which is multiple of P), we swap this page to the least-worn one. Then, the write activities are evenly distributed among pages.

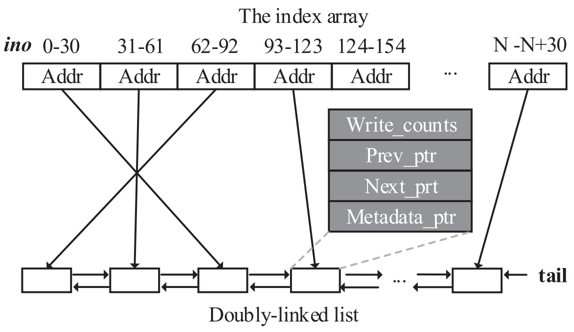


Fig. 4: The structure of sorted counters.

To efficiently select the least-worn inode page, we design a doubly-linked list in DRAM to store the copy of t of each page. We maintain a pointer to access its corresponding metadata of an inode page in PM. When the t in a node increases, CWM will compare its value with its previous t, denoted as t_{prev}. The node will be moved to the front of its previous node if the t is lager than t_{prev}. Consequently, the nodes in this list are sorted by the values of t. As shown in Fig. 4, we record the tail of the list to facilitate fetching the least-worn page in $O(1)$ complexity.

Additionally, we construct an array to save the addresses of these nodes in the doubly-linked list. Given an inode number i, we can access the corresponding node through the $(i+1)/31$ element in this array. When swapping two pages, CWM obtains the first inode of each page and locates the corresponding two nodes. After swapping the data within two pages, CWM modifies the *Metadata_ptr* to accomplish the swap routine.

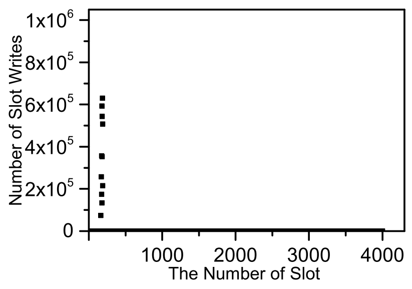
Discussions. In addition to the wear-leveling algorithms, we also give the proof that the wears on *Inode Mapping Table* will always be less than that on pages when $T \geq 64$ (e.g., an inode page contains 64 slots). First of all, the smaller T brings more wears on the *Inode Mapping Table*, since the *Inode Mapping Table* can be updated only when migration is triggered. we set T as 64, where the *Inode Mapping Table* is most likely to be worn-out. We denote N ($N \leq 64$) as the number of updated slots. The average wears of these slots are $64/N \geq 1$, which means the most-worn slot of this page always has larger wears than the *Inode Mapping Table*. Only when $N = 64$, where each slot is updated once, the wears on the page can be equal to that on the *Inode Mapping Table*. As for the coarse-grained wear-leveling routine, both the slots and *Inode Mapping Table* are updated once. Thus, the *Inode Mapping Table* is always less-worn than the corresponding inode page.

IV. Evaluation

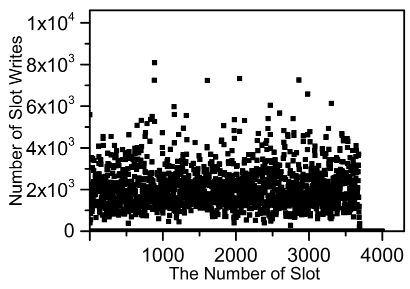
A. Experimental Setups

We implement LMWM and MARCH [14] on Linux 4.4.30 and integrate them into NOVA [6]. NOVA and MARCH are the state-of-the-art persistent memory file system and the inode-sized wear-leveling mechanism of inode table, respectively. LMWM is compared with the original NOVA and MARCH. The experiments are conducted on a machine equipped with a 2.40GHz Intel Xeon®E5-2640 v4 CPU and 256GB DRAM. Furthermore, we use the PMEM [16] emulation (/dev/pmem0) with 128GB of DRAM to emulate PM device and amount NOVA, MARCH and LMWM on this device with DAX [16] supported. The configuration of MARCH is as following. The threshold of sampling point for the change of the state of the sub-window T_t, the threshold of sampling point for the expansion of the size of the sliding window T_p, and the number of containers in each sub-window are set as 1024, 1024, and 3, respectively. For the proposed LMWM, the migration threshold of writes for the inode page and slot are set as 40960 (i.e., P) and 1024 (i.e., T), respectively.

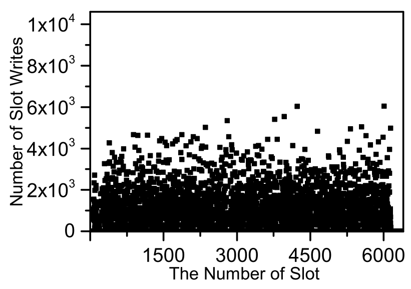
In these experiments, we first use two typical workloads, $OLTP$ and $Webserver$, of Filebench [15] to analyze the overall write distribution. OLTP is a database emulator. It performs file system operations using the Oracle 9i I/O model and tests the performance of small random reads and writes. Webserver emulates simple web-server I/O activity. The configurations



(a) NOVA.

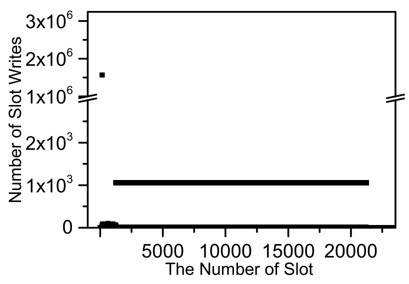


(b) MARCH.

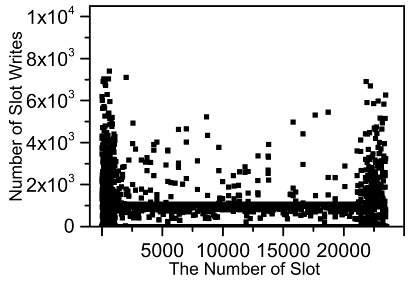


(c) LMWM.

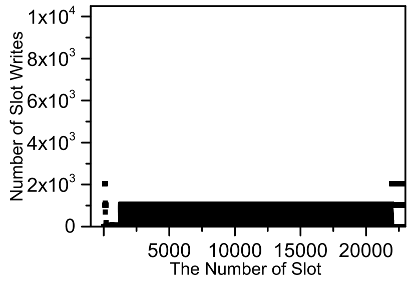
Fig. 5: The write distribution in $OLTP$.



(a) NOVA.



(b) MARCH.



(c) LMWM.

Fig. 6: The write distribution in $Webserver$.

of OLTP is set as default. The file number of $Webserver$ is 10 thousand, and the other is default setting. Second, we analyze the lifetime of PM according to the results of write distribution. Finally, we use $Webserver$ and $OLTP$ as macro workloads to evaluate the overhead of performance between MARCH and LMWM.

B. Impact on Lifetime of PM

We first evaluate the wear-leveling effect of NOVA, MARCH, and LMWM by evaluating the write counts of slots. Then, we analyze the write distribution of all slots and the lifetime of PM.

1) Overall Write Distribution: In order to get the write distribution of all slots for the inode table, we add codes into NOVA, MARCH, and LMWM to track the write counts of each slot. Firstly, we analyze the write distribution of the OLTP workload. For the default setting, the number of files and log files of OLTP are 10 and 1, respectively. Accordingly, the writes are concentrated in 11 inodes that occupy 22 slots. As shown in Fig. 5a, the writes are mainly concentrated in a few slots, inducing heavy wears. For example, the maximum number of writes is larger than 600 thousand in a slot. Conversely, other slots are updated zero times. This is because the inode management of NOVA is unaware of the limited endurance problem of PMs. NOVA causes severe imbalance of wear and damages the underlying PM quickly.

As shown in Fig. 5b, MARCH can distribute the concentrated writes of NOVA to other slots. However, MARCH still leads to severe imbalance write for each slot. For example, the writes difference between slots is larger than eight thousand. One of the main reasons is that MARCH only considers the wear between inodes. This is because that MARCH does not consider the characteristic that the frequently-updated variables of an inode are aggregated in the first part. Accordingly, MARCH still leads to severe wear imbalance of PM.

LMWM provides a multi-grained wear-leveling mechanism to achieve low-overhead and high-accuracy wear-leveling of the inode table. The writes distribution of LMWM is shown in Fig. 5c. Compared with MARCH, the writes of LMWM are dispersed in all slots of the inode table. Then, the writes difference between slots is much less than MARCH. The maximum number of writes is only 6048, which indicates that LMWM can provide better wear balance of PM than MARCH.

Secondly, we analyze the write distribution of the Webserver workload. The number of files in the Webserver is ten thousand. The writes distribution of NOVA is shown in Fig. 6a. We can observe that NOVA leads to severe wear imbalance of inode table. Especially, a slot suffers from larger than 1.5×10^6 writes, which cause heavily worn of PM. Moreover, the writes difference of other slots larger than one thousand on average. The writes distribution of MARCH is shown in Fig. 6b. Similar to the OLTP experiment, the writes difference between the maximum and minimum write is larger than 7 thousand. For LMWM, the writes distribution is shown in Fig. 6c. Compared with MARCH, the writes of LMWM are dispersed in all slots of the inode table. Then, the writes difference between slots is much less than MARCH. The maximum writes is only 2049, which still indicates that LMWM can provide better wear balance of PM than MARCH.

2) Lifetime Analysis: Fig. 7 shows the normalized lifetime of NOVA, MARCH, and LMWM. For all workloads, LMWM outperforms NOVA and MARCH in all cases.

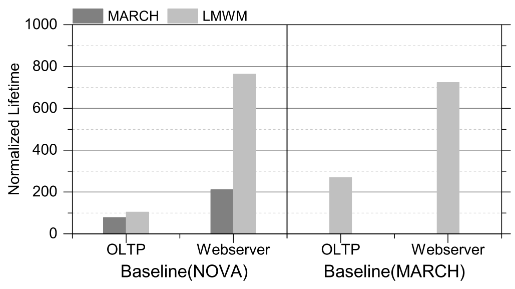


Fig. 7: Normalized Lifetime.

Furthermore, we use the multiple relationships of the maximum number of writes among NOVA, MARCH, and LMWM to evaluate PM lifetime in the tests of OLTP and Webserver workloads. The maximum writes of NOVA, MARCH, and LMWM are 630848/1565495, 8090/7410, 6048/2049 in the tests of OLTP and Webserver workloads, respectively. Accordingly, The lifetime improvement of LMWM against NOVA and MARCH is up to 487× and 2.5× on average, respectively.

C. Overhead Analysis

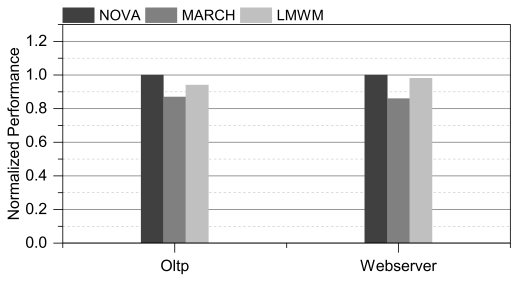


Fig. 8: Normalized Performance.

Finally, we evaluate the performance of LMWM, NOVA, and MARCH using $Webserver$ and $OLTP$ workloads. As shown in Fig. 8, the performance of LMWM and MARCH is about 96%, 86% of NOVA on average. The performance of LMWM is 1.12× than MARCH. The main reason that LMWM outperforms MARCH is that LMWM adopts 64 bytes as the migration granularity. MARCH adopts 128 bytes as the migration granularity, i.e., the migration overhead of MARCH is 2× that of LMWM. Furthermore, the migration times of MARCH are 5111 and 43298 in OLTP and Webserver, respectively. LMWM has only 3756 and 12538 migrations in OLTP and Webserver for fine-grained migrations, respectively. The number of coarse-grained migrations is only 93 and 38 using OLTP and Webserver. Hence, the total migration size of LMWM is less than MARCH.

V. Conclusion

In this paper, we investigated the asymmetric wear inside the inodes and among the inode pages of persistent memory file systems. We proposed a mechanism, called LMWM, to achieve both fine-grained and coarse-grained wear-leveling of the inodes table. Comprehensive experimental results showed that the proposed LMWM can significantly improve the accuracy and reduce the overhead of wear-leveling.

Acknowledgement

This work is partially supported by grants from the National Natural Science Foundation of China (61672116, 61601067 and 61802038), Chongqing High-Tech Research Key Program (cstc2019jscxmbdx0063), the Fundamental Research Funds for the Central Universities under Grant (0214005207005 and 2019CDJGFJSJ001), China Postdoctoral Science Foundation (2017M620412), Chongqing Postdoctoral Special Science Foundation (XmT2018003), Chongqing Youth Talent Support Program.

References

[1] P. M. Palangappa, J. Li, and K. Mohanram, "Wom-code solutions for low latency and high endurance in phase change memory," *IEEE Transactions on Computers.*, vol. 65, no. 4, pp. 1025–1040, Apr. 2016.

[2] 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)*, 2013, pp. 73–80.

[3] Intel, "3d xpoint unveiled, the next breakthrough in memory technology," in *http://www.intel.com/content/www/us/en/architecture-andtechnology/3d-xpoint-unveiled-video.html*, 2015.

[4] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*, 2014, pp. 15:1–15:15.

[5] E. Sha, X. Chen, Q. Zhuge, S. Liang, and W. Jiang, "A new design of in-memory file system based on file virtual address framework," *IEEE Transactions on Computers*, vol. 65, no. 10, pp. 2959–2972, 2016.

[6] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016, pp. 323–338.

[7] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proceedings of the Eleventh European Conference on Computer Systems (2016)*, C. Cadar, P. R. Pietzuch, K. Keeton, and R. Rodrigues, Eds., 2016, pp. 12:1–12:16.

[8] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *36th International Symposium on Computer Architecture (ISCA)*, 2009.

[9] F. Huang, D. Feng, Y. Hua, and W. Zhou, "A wear-leveling-aware counter mode for data encryption in non-volatile memories," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2017, pp. 910–913.

[10] X. Chen, E. H. . Sha, Y. Zeng, C. Yang, W. Jiang, and Q. Zhuge, "Efficient wear leveling for inodes of file systems on persistent memories," in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 1524–1527.

[11] D. Liu, Y. Lin, P. Huang, X. Zhu, and L. Liang, "Durable and energy efficient in-memory frequent-pattern mining," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 36, no. 12, pp. 2003–2016, 2017.

[12] C. Pan, S. Gu, M. Xie, Y. Liu, C. J. Xue, and J. Hu, "Wear-leveling aware page management for non-volatile main memory on embedded systems," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 2, no. 2, pp. 129–142, 2016.

[13] V. Gogte, W. Wang, S. Diestelhorst, A. Kolli, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Software wear management for persistent memories," in *17th USENIX Conference on File and Storage Technologies (FAST)*, 2019, pp. 45–63.

[14] H. S. Chang, Y.-H. Chang, P.-C. Hsiu, T.-W. Kuo, and H.-P. Li, "Marching-based wear-leveling for pcm-based storage systems," *Acm Transactions on Design Automation of Electronic Systems*, vol. 20, no. 2, pp. 1–22, 2015.

[15] E. Z. V. Tarasov and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *login*, vol. 41, no. 1, pp. 1–17, 2016.

[16] PMEM, "Persistent memory emulation, linux kernel," in *https://pmem.io/2016/02/22/pm-emulation.html*, 2016.