Reducing Write Amplification for Inodes of Journaling File System using Persistent Memory

Chaoshu Yang†, *Duo Liu†, *Xianzhang Chen†‡, Runyu Zhang†, Wenbin Wang†, Moming Duan† and Yujuan Tan†
† College of Computer Science, Chongqing University
‡ College of Communication Engineering, Chongqing University

***Abstract*—Conventional journaling file systems, such as Ext4, guarantee data consistency by writing in-memory dirty inodes to block devices twice. The write back of inodes may contain up to 80% clean inode that is unnecessary to be written back, which caused severe write amplification problem and largely reduce performance since the size of an inode is several times less than the size of a basic unit for updating the block device. Emerging persistent memories (PMs), such as phase change memory, provide the possibility for storing the offset of inodes in memory persistently. In this paper, we propose an efficient scheme, Updating Frequency based Inode Aggregation (UFIA), to reduce the write amplification of dirty inodes using PM. The main idea of UFIA is to identify the frequently-updated inodes and reorganize them in adjacent physical locations on block device. Firstly, UFIA adopts PM as an inode mapping table for remapping logical inodes to any physical inodes. Secondly, we design an efficient algorithm for UFIA to identify and reorganize the frequently-updated inodes. We implement UFIA and integrate it into Ext4 (denoted by UFIA-Ext4) in Linux kernel 4.4.4. The experiments are conducted with widely-used benchmark Filebench. Compared with original Ext4, the experimental results show that UFIA significantly reduces the write amplification of inodes and improves 54% of the performance on average.**

I. Introduction

In recent years, emerging big data [1] and real-time applications [2] impose demands on data processing performance. File systems, as the basic facility for storing data, have a vital impact on the performance of data processing. The performance of conventional journaling file systems, such as Ext4 [3] and XFS [4], are limited by the slow block I/O, especially for the frequently-updated and fine-grained inodes. Therefore, the performance of existing journaling file systems cannot meet the growing requirements of applications gradually. How to improve the performance of journaling file systems becomes an important topic nowadays.

Journaling file systems have been widely used where data consistency must be assured. It can recover the file system to a consistent state after unexpected system crash or power failure during system reboots. It employs journaling techniques [5]–[7] to guarantee data consistency that causes the dirty inodes in memory to be written back to block device twice. Moreover, the design and implementation of journaling file systems always adopt a block as the basic unit for writing back inode to the block device. We observed that the inodes of existing journaling file systems are fixed on physical locations throughout their lifetime. If the frequently updated inode was distributed among different blocks, then all the inodes in a block containing dirty inode will be written back. These blocks may contain up to 80% clean inodes that are unnecessary to be written back. Thus, it can cause severe write amplification problem [8]–[10] and largely reduce performance for the size of an inode is several times less than the size of a block.

Emerging persistent memories (PMs), such as STT-RAM [11] and phase change memory(PCM) [7], [12]–[14], are promised to revolutionize storage systems by providing the characteristic of non-volatility, byte-addressability and DRAM-like performance. PM can be linked to the memory bus directly and share the address space with DRAM. In the light of the advantages of PMs, PMs have the potential capacity to reduce the write amplification of inodes and improve the performance of journaling file systems.

In this paper, we focus on solving the write amplification of inode for journaling file systems. We propose an efficient scheme, Updating Frequency based Inode Aggregation (UFIA), to reduce the write amplification of dirty inodes using PM. The main idea of UFIA is to identify the frequently-updated inodes and reorganize them in adjacent physical locations on block device. Firstly, UFIA adopts PM as inode mapping table for recording the offset of physical inode in the inode table. Then, UFIA can remap logical inodes to any physical inodes. Secondly, we design an efficient algorithm for UFIA to identify and reorganize the frequently-updated inodes. We implement UFIA in Linux kernel based on Ext4. The experimental results show that UFIA significantly reduces the write amplification of inodes and improves 54% performance on average in comparison with the original Ext4. Our contributions can be summarized as follows:

- We propose a new mechanism UFIA to reduce the write amplification of dirty inodes using PM. UFIA can efficiently identify and reorganize the frequently-updated inodes via the proposed inode mapping table and migration algorithm.
- We implement a functional version of Ext4 in Linux kernel based on the proposed UFIA mechanism.
- Extensive experiments are conducted to evaluate UFIA. The experimental results show that UFIA outperforms exiting solutions.

The remainder of this paper is organized as follows. In Section II, we introduce the background and motivation. We

Corresponding author: Xianzhang Chen, email: xzchen109@gmail.com
Duo Liu, email: liuduo@cqu.edu.cn

978-3-9819263-2-3/DATE19/©2019 EDAA

provide details of the proposed UFIA in Section III. The experimental results are analyzed in Section IV. Section V concludes the paper.

II. Background And Motivation

A. Organization of Inodes

Journaling file systems, such as Ext4 [3] and BtrFS [15], organize the inodes by tree or array. For example, BtrFS uses B-tree to organize the inodes and Ext4 organizes inodes by an array, namely, *inode table*. The inodes are stored on fixed physical locations throughout their lifetime in spite of the organizing structure of inodes [16].

The file systems use a unique *i-number* to represent an inode logically. For the array structure shown in Figure 1, an i-number is the index of an element in the *Inode Table*, which inherently points to a physical inode. The inodes are all the same size. For example, each inode occupies 256 bytes in Ext4 by default. The inode table for storing the mapping between i-numbers and physical inodes is fixed since the file system is mounted. This permanent mapping restricts the physical storage locations of inodes.

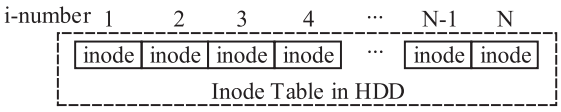


Fig. 1: The conventional inode section

For all the journaling modes, a dirty inode is written to "journal area" before written back to its storing location in Inode Table. The doubled writes of inodes in journaling file systems are quite expensive due to the high latency of block I/O. Furthermore, a large portion of inodes in a write operation are clean and unnecessary to be written back to the block device. This is because journaling file systems always adopt "block" as the basic unit for updating the block device and a block generally contains multiple inodes.

In summary, the update of inode in journaling file systems brings many additional writes to the block device, i.e., write amplification, and causes large overhead. In this paper, we focus on reducing write amplifications by using persistent memory as an inode mapping table.

B. Write Amplification of Inodes

The widely-used journaling file systems, such as Ext4 [3], generally reserve a journal area on block device to store the affected inodes and new data before modifying the original files. The file can be recovered to consistent state using the information stored in the journal area in case of system crash or power failure. The journaling file systems use three modes of journaling [17]:

1) **Write-back mode**, which only writes the modified metadata to the journal area.
2) **Ordered mode**. Similar to write-back mode, ordered mode also writes the modified metadata to journal area. But it provides higher consistency insurance than write-back mode by strictly writing journals after file data is written.
3) **Journal mode**. In this mode, both metadata and data are written to the journal area first. The performance of journal mode is worse than the previous two modes for all information are written twice. As shown in Figure 2, the file system first writes the dirty inode in memory to the journal area and then update the physical inode in the inode table.

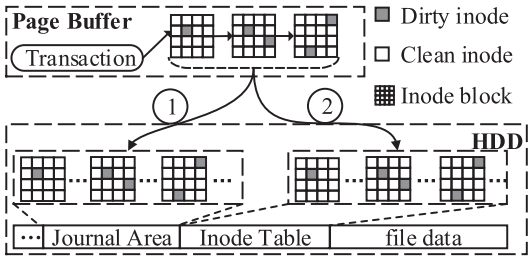


Fig. 2: Illustration of writing back dirty inodes.

Most file operations, such as create, write, delete, and link, invoke the journaling of inodes. After a long run, the frequently-updated inodes are scattered among the blocks in the inode table. This causes large write amplification for a block is the basic unit of write back, while the size of an inode is several times less than the size of block. For example, the size of inode in Ext4 is 256B and the default block size is 4KB. It means that one block can contain 16 inodes [8]. Thus, suppose a dirty inode in the block needs to be written back to the block device, all the other 15 inodes in the block will also be written back, no matter they are dirty or clean.

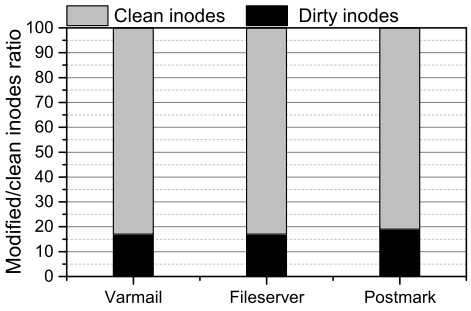


Fig. 3: Dirty/clean ratio of write back inodes

We show the write amplification of inodes by an experiment. We run two typical workloads, Varmail and Fileserver, of the Filebench [18] and Postmark [19] on Ext4. Then, we calculate the total number of dirty and clean inodes in the written-back inode blocks of all the operations. Figure 3 shows the ratio of dirty inodes over clean inodes, i.e., dirty/clean ratio. It shows that up to 83%, 83% and 81% of the inodes are clean in the write-back inode blocks of Varmail, FileServer, and Postmark, respectively. Therefore, it is important to reduce the write amplification of inodes for improving the file system performance.

III. Design and Implementation

As aforementioned, the fixed bound of i-number and physical inode result in write amplification of inodes and degraded

system performance in updating the scattered dirty inodes. Suppose we can converge the frequently-updated inodes storing in several blocks, then we can greatly cut down the number of write backs of dirty inodes and reduce the write amplification. Thus, we propose a mechanism, called Updating Frequency based Inode Aggregation (UFIA), to reduce write amplification of inodes and improve system performance. First, we design an inode mapping table in PM to detach logical inodes from physical inodes. Then, we present an efficient approach to converge the frequently-updated inodes by an inode migration algorithm.

A. Inode mapping table

In order to reorganize the frequently-updated inodes in adjacent physical locations of block device to reduce the clean inodes write back to block device. We need to change the mapping between logical inodes (represented by i-number) and physical inodes. In UFIA, we propose to use PM as an *inode mapping table* for remapping a logical inode to any physical inode. The inode mapping table maintains the offset of inodes in the inode table. As shown in Fig. 4, each i-number is bound with a physical inode in the inode table. Each i-number has a unique offset in inode mapping table.

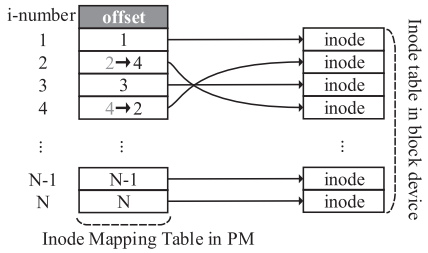


Fig. 4: The proposed inode section

On the other hand, logical inodes are mapped to physical inodes via the inode mapping table. Each i-number corresponds to an entry in the inode mapping table that stores the offset of the corresponding physical inode. The inode mapping table is updated only when an inode is migrated. As a consequence, an i-number can be mapped to a different location in the inode table dynamically by changing its offset. For example, as shown in Fig. 4, the inode that its i-number is 2 was stored in the 4th location of the inode table. Similar to array-structured inode sections, tree-structured inode sections can also be reorganized by constructing an inode mapping table for the inodes in the leaf nodes. Therefore, to access an inode, the file system first get an offset on the inode mapping table via i-number. Then, the file system calculates the location of physical inode by adding the offset with the beginning inode table. Thus, the overhead for locating inodes is negligible.

We use the design parameters of Ext4 to calculate the space overhead of the proposed inode mapping table. Suppose the size of HDD is 1TB and one percent of the HDD is used as inode table, then the size of the inode table is 10.24GB. Thus, there are 42,949,672 inodes in the file system for the size of each inode is 256B in Ext4. In the inode mapping table, each inode needs 8B for storing its offset. Therefore, the required PM space is 327.68MB, which is acceptable for many embedded systems.

B. Converging Frequently-Updated Dirty Inodes

To reorganize the frequently-updated inodes in adjacent physical locations on block device, UFIA employs a window-based marching strategy to gather the frequently-updated inodes. The window-based marching strategy operates in DRAM with indirect pointers pointing to the inode mapping table of UFIA. We collect the inode writes into the window by migrating the frequently-updated inode into the window. The initial position of window is aligned with the inode table and the default size is *N* inodes. The left side of the window is fixed, and the right side can be extended forward when the number of frequently-updated inodes is larger than the size of the window. The items involved in the metadata of the window are shown in Table I.

TABLE I: Notations used in this paper.

Notation	Definition
L_{free}	The list of free inodes in the window
L_{closed}	The list of in-use inodes and the corresponding file is closed
L_{lazy}	The list of in-use inodes and stay unmodified for a long time
WI_i	The write count of inode I_i in hash table
P_n	The write count threshold of inode migration

When the journaling file system is mounted, the blocks belonging to the window are loaded into *Page Buffer*, the L_{free} list is initialized by the unused inodes according to the bitmap of inodes. All the inodes that are involved in the window and the corresponding file was closed are inserted into L_{lazy}. UFIA deletes an inode from L_{lazy} when its corresponding file is opened. An inode is inserted into L_{closed} when its corresponding file is closed.

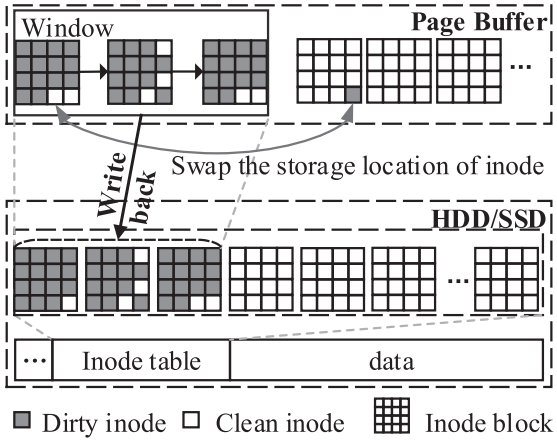


Fig. 5: The example of inode migration

To identify a frequently-updated inode outside of the window, UFIA uses a hash table to store the i-numbers of the inodes whose corresponding files are opened in writable mode. The key and value of the hash table are the i-number and the write count, respectively. The i-number is inserted into the hash table when the corresponding file was opened in writable mode. On contrary, an i-number is deleted from hash table if

its corresponding file is closed. The write count in the hash table is increased when its corresponding inode is modified. UFIA creates an *inode migration thread* in the file system and awakes it up periodically. When the thread is awaken, then the thread will query the hash table to check if the write counts of the inodes reaches the migration threshold P_n. If the write count of an inode reaches P_n, the thread will move the inode to the window.

An example is illustrated in Fig. 5, the window is composed of three blocks. The first block in the window contains two unused inodes. Meanwhile, there is a frequently-accessed inode outside of the window whereas the other inodes in the same block are clean. For this case, UFIA migrates the frequently-accessed inode into the window in two steps: 1) UFIA copies the inode into to an unused inode in the window; 2) UFIA updates the corresponding offset in the inode mapping table.

Algorithm 1: Inode migration algorithms

Input: WI_i: write counts of inode I_i in hash table; P_n: the write count threshold of inode migration;
Output: The new storing location O_i of I_i;

```
if WI_i ≤ P_n then
    return;
else
    while (true) do
        if there are items in L_free or L_lazy or L_closed then
            I_candidate ← the inode in Window belong to L_free or L_lazy or L_closed;
            break;
        else
            ExpandWindow;
            continue;
        end
    end
    if I_candidate is free then
        Set the location of I_i to unused;
        Set the location of I_candidate to in-use;
        Copy I_i to I_candidate;
    else
        Swap I_i with I_candidate;
    end
    O_i ← the storing location of I_candidate;
    Exchange the offset of I_i and I_candidate maintained in the inode mapping table in PM;
    Set the write count of inode in hash table to zero;
end
```

The details of the inode migration is shown in Algorithm 1. The inode migration thread checks the write counts of the inodes in the hash table one-by-one. If the write count of inode I_i that is outside of the window is larger than or equal to the threshold P_n, then I_i will be migrated into the window. During migration, the file system first checks whether the L_{free}, L_{lazy} and L_{closed} are all empty or not. If these lists are not empty, then UFIA finds an inode and set it as $I_{candidate}$. Otherwise, the inode migration thread needs to expand the window to enlarge the search space of proper inodes. Then, the UFIA traverses above three lists iteratively until an exchangeable inode is found.

When the candidate inode is found, UFIA checks whether I_i is in use or not. If inode I_i is free, UFIA sets the location of I_i to unused and that of $I_{candidate}$ to in-use. Then, UFIA copies I_i to the location of $I_{candidate}$. Otherwise, i.e., I_i is in-use, UFIA swaps the content of I_i and $i_{candidate}$. Finally, UFIA exchanges the offsets of I_i and $I_{candidate}$ in the inode mapping table in PM. The write counts of the affected inodes are set to zero in the hash table.

IV. Evaluation

A. Experimental Setups

We implement UFIA in Linux 2.6.34 and integrate it into the typical journaling file system Ext4 [3], denoted by UFIA-Ext4. UFIA-Ext4 is compared with the original Ext4, denoted by Original-Ext4. In the original Ext4, the mapping between *i-number* and physical inode is fixed. The experiments are conducted on a machine equipped with a 3.40GHz Intel®core i5-7500 CPU, 8GB DRAM, and 1TB HDD. To better analyze the performance differences between Original-Ext4 and UFIA-Ext4, we partition 100GB HDD space to mount Ext4 and set its journaling mode as $Ordered$. Since DIMM-based persistent memory (PM) is not yet commercially available, we configure 500MB DRAM to stand for PM.

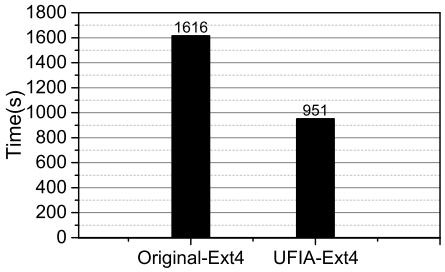
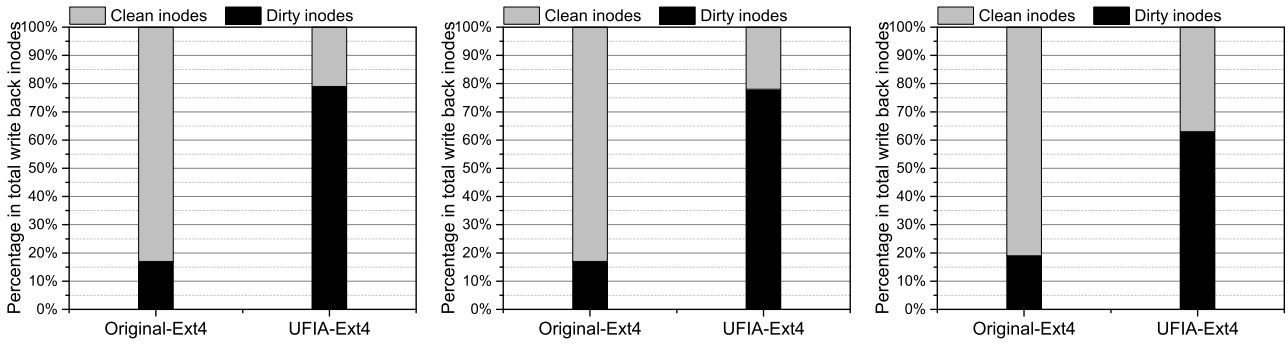


Fig. 6: Comparison of the total elapsed time.

In the experiments, we first evaluate the overhead of writing back inodes by a dedicated test case. Then, we use Postmark [19] and two typical workloads, $varmail$ and $fileserver$, of Filebench [18] as test cases. Postmark simulates a server that frequently accesses lots of small files. It commits a series of create, delete, read, and append operations to access the preset file pool. In Postmark, we use write-only workloads with 50%/50% create/delete ratio. We record the times of create/delete operations per second to demonstrate the throughput of file system. $fileserver$ emulates a server that hosts home directories of multiple users, which is represented by 50 threads. Each thread performs a sequence of create, read, write, delete, append and stat operations on their own home directories. $varmail$ emulates a mail server with three kinds of behaviors including reading mails (open, read, and close), composing (open/create, append and close), and deleting mails. 16 threads are used to mimic such behaviors. For all the workloads, the I/O size and the mean append size are both set to 4KB. The number of files involved in the workloads is set to 100 thousand, 200 thousand, and 300 thousand.

B. Overhead of writing back inodes

Here we evaluate the overhead of writing back inodes. We create ten thousand files and perform one thousand append operation to write each file. The file size field in inode is dirty in each append operation and needs to be constantly written back to block devices. The size of append is set to 30 bytes to reduce the impact of writing file data. Then, we compare the total elapsed time of Original-Ext4 and UFIA-Ext4 for accomplishing all the append operations.



(a) Dirty and clean inodes ratio on $varmail$ (b) Dirty and clean inodes ratio on $fileserver$ (c) Dirty and clean inodes ratio on Postmark

Fig. 7: Comparison of write amplification of different workloads.

The experimental results are shown in Fig. 6. Compared with UFIA-Ext4, the total elapsed time of Original-Ext4 is 3.4 times higher than UFIA-Ext4. It is because that the modified inode is distributed in different blocks in Original-Ext4. For example, the size of inode is 256 bytes and the default block size is 4KB, then each block contains 16 inodes. In extreme case, the 16 dirty inodes are scattered in 16 blocks, which will cause the write back operations of 16 blocks in Original-Ext4. On the contrary, UFIA-Ext4 can gather the 16 frequently-updated inodes in the adjacent locations in the inode table and only write one block back to the underlying block device. Compared with Original-Ext4, UFIA-Ext4 can reduce lots of block I/Os. Thus, UFIA-Ext4 can reduce the write amplification and significantly improve the performance of block-base journaling file system.

C. Impact on Write Amplification

Next, we evaluate the write amplification of inodes by measuring the ratio of dirty inodes over clean inodes in write back operations. Fig. 7 shows the dirty and clean ratio of Original-Ext4 and UFIA-Ext4 using different workloads. Fig. 7a shows that for $varmail$, the write-back operations of UFIA-Ext4 contain only 21% clean inodes whereas the ratio of clean inodes in the write-back operations of Original-Ext4 is 82%. In other words, the write amplification of inodes of UFIA-Ext4 and Original-Ext4 is 1.27 (i.e., 1/0.79) and 5.56 (i.e., 1/0.18), respectively. The write amplification of inodes of Original-Ext4 is 1.7 times higher than that of UFIA-Ext4. It is because that UFIA-Ext4 dynamically converges the frequently modified inodes to the adjacent physical locations and writes back many dirty inodes in a batch.

Fig. 7b and Fig. 7c shows similar results. For the $fileserver$ workload, the write amplification of inodes of Original-Ext4 and UFIA-Ext4 is 5.56 and 1.28, respectively. For the Postmark workload, the write amplification of inodes of Original-Ext4 and UFIA-Ext4 is 5.06 and 1.59, respectively. Therefore, UFIA-Ext4 achieves 3.34 and 2.31 times write reduction over Original-Ext4 for $fileserver$ and Postmark, respectively. The results proved the effectiveness of the proposed UFIA mechanism on the reduction of write amplification.

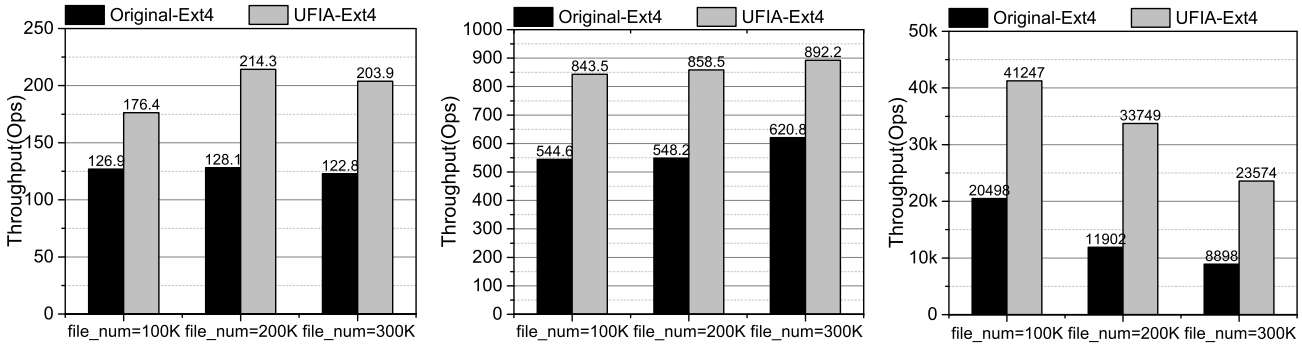
Besides, UFIA-Ext4 shows higher write amplification on the Postmark workload than on the other two workloads. It is because that the Postmark workload has lower locality on the files. As mentioned in SectionIV-A, Postmark frequently create and delete files for we use write-only workloads with 50%/50% create/delete ratio. It is not easy to gather frequently updated inodes. On the contrary, $fileserver$ shows high locality for each thread simulates a user that only accesses the files under its own home directory. As a result, UFIA-Ext4 reduces more writes on $fileserver$ than Postmark for UFIA gets more frequently accessed dirty inodes together in $fileserver$.

D. Impact on Performance

Now, we evaluate the impact of the proposed UFIA mechanism on the performance of file system. Fig. 8 shows the throughput (operations per second) of Original-Ext4 and UFIA-Ext4 using $varmail$, $fileserver$, and Postmark workloads. The experimental results show that UFIA-Ext4 constantly outperforms Original-Ext4.

Fig. 8a shows the performance of Original-Ext4 and UFIA-Ext4 under $varmail$ workloads. For different number of files, the throughput of UFIA-Ext4 showing up to 39%, 67%, and 66% higher throughput than Original-Ext4, respectively. Fig. 8b and Fig. 8c show similar improvement of UFIA-Ext4 over Original-Ext4 on $fileserver$ and Postmark. For Postmark workload, UFIA-Ext4 achieves up to 2.0×, 2.8×, and 2.6× higher throughput than Original-Ext4 when the number of files is 100K, 200K, and 300K, respectively. It is because that UFIA-Ext4 reduces the number of additional writes caused by the write-back of dirty inodes.

For all the workloads, the performance improvement of UFIA-Ext4 over Original-Ext4 mismatches the reduction of write amplifications, as shown in Fig. 7. Especially for $varmail$ and $fileserver$, they have larger reduction of write amplifications but gain lower performance improvement than Postmark. In the experiments, the Ops of the workloads include not only updating operations, such as write and create, but also include the operations that do not incur writes of inodes or file data, such as read and stat. Thus, the impact of updating operations on performance improvement is weakened when the ratio of updating operations is lowered. Furthermore, even for Postmark, the scale of performance improvement is



(a) Throughput on $varmail$ (b) Throughput on $fileserver$ (c) Throughput on Postmark

Fig. 8: Comparison of Throughput of different workloads.

also less than that of the reduction of write amplification. It is because that a write operation updates both the corresponding inode and the file data. In our experiments, the cost for updating an inode and the file data is generally fifty-fifty since we set the append size as 4KB, which is the same to the size of a block for storing inodes.

V. Conclusion

In this paper, we studied the write amplification problem of inodes for block-based journaling file systems, which is generally the most frequently updated data in a file system. We observed that the inodes of existing journaling file systems are fixed on physical locations throughout their lifetime. To this end, we proposed a mechanism called UFIA to identify the frequently-updated inodes and reorganize them in adjacent physical locations on block device. Experimental results show that the proposed UFIA mechanism can largely reduce the write amplification of inode and efficiently improve the performance of block-based journaling file system.

VI. Acknowledgement

This work is partially supported by grants from the National Natural Science Foundation of China (61402061, 61672116 and 61802038), Chongqing High-Tech Research Program (cstc2016jcyjA0274 and cstc2016jcyjA0332), China Postdoctoral Science Foundation (2017M620412), Chongqing Postdoctoral Special Science Foundation (XmT2018003), Fundamental Research Funds for the Central Universities (2018CDXYJSJ0026).

References

[1] S. J. Walker, "Big data: A revolution that will transform how we live, work, and think," *Mathematics & Computer Education*, vol. 47, no. 17, pp. 181–183, 2014.

[2] D. Han, X. Chen, Y. Lei, Y. Dai, and X. Zhang, "Real-time data analysis system based on spark streaming and its application," *Journal of Computer Applications*, 2017.

[3] A. Mathur, M. Cao, and A. Dilger, "Ext4: the next generation of the ext3 file system," *the magazine of USENIX & SAGE*, vol. 32, pp. 25–30, 2007.

[4] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system," in *Conference on Usenix Technical Conference*, 1996, pp. 1–1.

[5] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *Acm Transactions on Database Systems*, vol. 17, no. 1, pp. 94–162, 1992.

[6] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," *Acm Sigarch Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.

[7] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *Acm Sigarch Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

[8] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue, "Fine-grained metadata journaling on nvm," in *MASS Storage Systems and Technologies*, 2017.

[9] X. Zhang, D. Feng, Y. Hua, and J. Chen, "A cost-efficient nvm-based journaling scheme for file systems," in *IEEE International Conference on Computer Design*, 2017, pp. 57–64.

[10] Z. Qin, Y. Wang, D. Liu, Z. Shao, and Y. Guan, "MNFTL: an efficient flash translation layer for MLC NAND flash memory storage systems," in *Proceedings of the 48th Design Automation Conference (DAC)*, 2011, pp. 17–22.

[11] T. Kawahara, "Scalable spin-transfer torque ram technology for normally-off computing," *IEEE Design & Test*, vol. 28, no. 1, pp. 52–63, 2011.

[12] D. Liu, X. Li, P. Huang, Y. Gu, Y. Ling, K. Zhong, R. Liu, X. Chen, and L. Liang, "Downsizing without downgrading: Approximated dynamic time warping on non-volatile memories," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. Available online, pp. 1–1, 2018.

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

[14] D. Liu, K. Zhong, T. Wang, Y. Wang, Z. Shao, E. Sha, and J. Xue, "Durable address translation in pcm-based flash storage systems," *IEEE Transactions on Parallel & Distributed Systems*, vol. 28, no. 2, pp. 475–490, 2017.

[15] O. Rodeh, J. Bacik, and C. Mason, "Btrfs:the linux b-tree filesystem," *Acm Transactions on Storage*, vol. 9, no. 3, pp. 1–32, 2013.

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

[17] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems." in *Usenix Technical Conference, April 10-15, 2005, Anaheim, Ca, Usa*, 2005, pp. 105–120.

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

[19] J. Katcher, "Postmark: A new file system benchmark," *Technical Report TR3022, Network Appliance, Tech. Rep.*, 1997.