Optimizing File Systems with a Write-efficient Journaling Scheme on Non-volatile Memory

Xiaoyi Zhang, Dan Feng, *Member, IEEE,* Yu Hua, *Senior Member, IEEE,* and Jianxi Chen

Abstract—Modern file systems employ journaling techniques to guarantee data consistency in case of unexpected system crashes or power failures. However, journaling file systems usually suffer from performance decrease due to the extra journal writes. Moreover, the emerging non-volatile memory technologies (NVMs) have the potential capability to improve the performance of journaling file systems by being deployed as the journaling storage devices. However, traditional journaling techniques, which are designed for hard disks, fail to perform efficiently in NVMs. In order to address this problem, we propose an NVM-based journaling scheme, called NJS. The basic idea behind NJS is to reduce the journaling overhead of traditional file systems while fully exploiting the byte-accessibility characteristic, and alleviating the slow write and endurance limitation of NVM. Our NJS consists of four major contributions: (1) In order to decrease the amount of journal writes, NJS only needs to write the metadata of file systems and over-write data to NVM as write-ahead logging, thus alleviating the slow write and endurance limitation of NVM. (2) NJS adopts a wear aware strategy for NVM journal block allocation to provide wear-leveling, thus further extending the lifetime of NVM. (3) We propose a novel journaling update scheme in which journal data blocks can be updated in the byte-granularity based on the difference of the old and new versions of journal blocks, thus fully exploiting the unique byte-accessibility characteristic of NVM. (4) NJS includes a garbage collection mechanism that absorbs the redundant journal updates, and actively delays the checkpointing to the file system. Evaluation results show the efficiency and efficacy of NJS. For example, compared with Ext4 with a ramdisk-based journaling device, the throughput improvement of Ext4 with our NJS is up to 131.4%.

Index Terms—Non-volatile memory, journaling, consistency, file system.

1 Introduction

Journaling techniques have been widely used in modern file systems due to offering data consistency for unexpected system crashes or power losses [1]. In general, the basic idea of a journaling technique is that, a file system first logs updates to a dedicated journaling area, called *write-ahead logging*, and then writes back the updates to the original data area, called *checkpointing*. If a system crash occurs, the consistent data are kept either in the journaling area or in the original file system. However, the performance of a journaling file system deteriorates significantly due to the extra journal writes. For example, the write traffic of Ext4 with journaling is about 2.7 times more than that without journaling [2]. Therefore, how to reduce the journaling overhead is an important problem to improve the file system performance.

Recently, the emerging Non-Volatile Memory (NVM) technologies have been under active development, such as Spin-Transfer Torque Magnetic RAM (STT-MRAM) [3], Phase Change Memory (PCM) [4], ReRAM [5] and 3D-XPoint [6]. NVMs synergize the characteristics of non-volatility as magnetic disks, and high random access speed and byte-accessibility as DRAM. Such characteristics allow NVMs to be placed on the processor's memory bus along with conventional DRAM, i.e., hybrid main memory systems [7]. NVM offers an exciting benefit to applications: applications can directly access persistent data in main memory via fast *load/store* instructions [8]. It is predicted that NVMs will have the potential to play an important role in computer memory and storage systems [9], [10]. However, due to expensive price and limited capacity, NVMs can not be used as a standalone persistent storage medium and will co-exist with HDDs and SSDs in storage systems in the near future. For instance, some previous research works use NVMs to store hot data (e.g., data cache [2], [11]) or critical data (e.g., file system metadata [12]) of storage systems. In this way, NVMs play an important role in improving system performance in a cost-effective way.

NVMs provide a new opportunity to reduce the journaling overhead of traditional file systems. However, simply replacing SSDs or HDDs with NVMs in building journaling storage device needs to address new challenges. First, for traditional journaling schemes, the software overheads caused by the generic block layer will become the performance bottleneck because these overheads are not ignorable compared with the low access latency of NVMs [13]. Second, NVMs have relatively long write latency and limited write endurance [4], [5]. If NVMs are simply used as journaling storage devices in traditional journaling schemes, the write performance and endurance of NVMs will be inefficient due to the heavy journal write traffic. Third, the characteristic of byte-accessibility of NVM is not well explored and exploited in traditional journaling schemes. Traditional journaling schemes need to write an entire block to the journaling area even though only a few bytes of the block are modified, which causes a write amplification problem and further degrades the overall system performance [14].

In order to quantify the overhead of traditional jour-

• *X. Zhang, D. Feng, Y. Hua and J. Chen are with Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {zhangxiaoyi, dfeng, csyhua, chenjx}@hust.edu.cn*

The preliminary version appears in the Proceedings of the 35th IEEE International Conference on Computer Design (ICCD), 2017, pages: 57-64.

naling schemes, we measured the throughput of Ext4 with and without journaling scheme under different workloads in Filebench [15]. Details about the experimental environment are described in Section 4.1. As shown in Fig. 1, the throughput of Ext4 with *data journal* mode on HDD is 41.9% and 43.8% slower than its non-journaling mode under Filesever and Varmail workloads, respectively. Even if we switch the journaling storage device from slow HDD to fast NVM (Ramdisk), the throughput of Ext4 with journaling on ramdisk is only 11.7% and 12.1% faster than that on HDD. These results indicate that traditional journaling schemes fail to perform efficiently in NVMs. Therefore, we consider highly-efficient journaling schemes for NVMs.

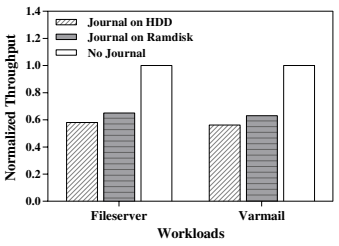


Fig. 1: Overhead of Traditional Journaling Schemes.

To this end, we present an NVM-based journaling scheme, called NJS, which can be used in traditional file systems to guarantee strict consistency and reduce the journaling overhead. The main contributions of our NJS can be summarized as follows:

- In order to alleviate the slow write and endurance limitation of NVM, NJS decreases the amount of journal writes. In the process of transaction committing, only the metadata of file systems and over-write data are written to NVM as write-ahead logging, and the append-write data blocks are directly issued to the file system.
- To avoid localized writes to the NVM-based journaling area, NJS adopts a wear aware strategy for NVM journal block allocation. NJS keeps a list of free NVM journal blocks in DRAM. The block with the least number of write counts in the free list is assigned during the process of transaction committing. The proposed block allocation strategy allows each NVM journal block to be evenly worn out and further extends the lifetime of NVM.
- In our NJS, a byte-level journaling update scheme is proposed to allow a journal block to be updated at the byte granularity by computing the difference between the old and new versions of the same journal block. In order to protect the latest version of journal block from being modified, we maintain an extra previous version (the one just before the latest) for each journal block. When a journal block is written to NVM, if the previous version exists, instead of writing another entire block, NJS only writes the differentiated bytes (i.e. delta) between the updating and the previous version to the previous version block. Thus, NJS exploits the unique byte-accessibility characteristic of NVM and further reduces the journal writes to NVM.
- When the NVM-based journal space is nearly full, we propose a garbage collection scheme, which recycles the redundant versions of the same journal block and delays the checkpointing to the file system. The redundant journal updates are absorbed, thus the writes to the file system can be reduced. In this way, the file system performance is improved.

The remainder of this paper is organized as follows. Section 2 provides the background of NVM, existing consistency and journaling techniques. Section 3 presents the design and detailed implementations. Experiment results are presented in Section 4. Related work is discussed Section 5 and we conclude the paper in Section 6.

2 Background

2.1 Consistency and Journaling for File Systems

File system consistency can be categorized into three levels, including *metadata consistency*, *data consistency*, and *version consistency* [16]. Specifically, *metadata consistency* guarantees that the metadata structures of file systems are entirely consistent with each other. It provides minimal consistency. *Data consistency* has the stronger requirement than *metadata consistency*. In *data consistency*, all data that are read by a file should belong to this file. However, the data possibly belong to an older version of this file. *Version consistency* requires the metadata version to match the version of the referred data compared with *data consistency*. *Version consistency* is the highest level of file system consistency.

Journaling techniques provide the consistency for file systems. According to the contents written to the journaling area, there are three journaling modes [1]:

The Writeback Mode: In this mode, only metadata blocks are written to the journaling area without order constraints. This mode only provides *metadata consistency*.

The Ordered Mode: Like the writeback mode, only metadata blocks are written to the journaling area in this mode. However, data blocks written to their original areas are strictly ordered before metadata blocks are written to the journaling area. Since append-write does not modify any original data, *version consistency* can be guaranteed. But for over-writes, the original data may be partially modified after system reboots. Thus, the ordered mode only provides *data consistency*.

The Data Journal Mode: In this mode, both metadata and data are written to the journaling area and *version consistency* is guaranteed. However, this mode suffers from significant performance degradation since all the data are written twice.

In fact, the ordered mode is the default journaling mode in most journaling file systems (e.g., Ext3, Ext4) for performance reasons. Hence, in order to meet the needs of data integrity in storage systems, it is important to obtain *version consistency* in a cost-efficient manner.

2.2 Non-volatile Memory Technologies

In recent years, computer memory technologies have evolved rapidly. The emerging non-volatile memory technologies, e.g., PCM, STT-MRAM, ReRAM and 3D-XPoint, have attracted more and more attentions in both academia

and industry [17], [18]. The key attributes and features of these NVM technologies are listed in Table 1[1] [9], [10], [17], [18]. Among current NVM technologies, PCM is mature and more promising for volume production [19] and attracts lots of research efforts from devices to architectures and systems [4], [20], [21].

TABLE 1: Characteristics of Different Memory Techniques

Techniques	DRAM	PCM	RRAM	STT-MRAM
Read speed(ns)	10	20∼85	10∼20	5∼15
Write speed(ns)	10	150∼1000	100	10∼30
Scalability(nm)	40	5	11	32
Endurance	10^{18}	10^{8}	10^{10}	10^{12}∼10^{15}

From the table, we observe that different NVMs have similar limitations. First, NVMs have the read/write performance asymmetry. Specifically, write speed is much slower (i.e., 3-8X) than read speed [4], [5]. Second, NVMs generally have limited write cycles, e.g., 10^8 times for PCM [4], 10^{10} times for ReRAM [5]. Actually, our design of reducing journal writes can alleviate the aforementioned two limitations simultaneously. To extend the lifetime of NVMs, wear-leveling techniques have been proposed [22]. Although our NJS does not employ an explicit wear-leveling scheme, we adopt a wear aware strategy for block allocation, which can be combined with the design of reducing journal writes to further lengthen NVM's lifetime.

2.3 Data Consistency in NVM

When NVM is directly attached to the processor's memory bus, the consistency of data updates to NVM must be ensured during the memory operations [23]. In general, the atomicity unit of NVM is very small (8 bytes for 64-bit CPUs) [24]. The updates with larger sizes must adopt logging or copy-on-write mechanisms which require the memory writes to be in a correct order [25]. Unfortunately, in order to improve the memory performance, modern processors and their caching hierarchies usually reorder write operations on memory. And the reordering writes possibly induce data inconsistency in NVM when a power failure occurs [8]. In order to address the data consistency problem, today's processors provide instructions such as *mfence* and *clflush* to enforce write ordering and explicitly flush a CPU cacheline. However, these instructions have been proved to be significantly expensive, and the overhead of these instructions is proportional to the amount of writes [26]. Thus, NVM systems should be designed to reduce writes.

3 Design and Implementations

In this section, we present the design and implementation details of our NJS.

3.1 Overview

Fig. 2 illustrates the workflow overview of a journal write request among the file system, buffer cache, and NVM-based journaling area in NJS. For each updated data block in the write request, if the update is an append-write, the data block is directly written to the file system. Moreover, if the update is an over-write, the data block is written to the NVM-based journaling area. The updated metadata blocks are also written to the NVM-based journaling area. When the free space size of the journaling area is low, garbage collection is invoked to free invalid journal blocks. If the free journal space is still lower than a predefined threshold after garbage collection or the periodical checkpointing time interval arrives, the latest version of valid journal blocks in the NVM-based journaling area are written back to the file system. When the system is rebooted due to unexpected crashes, the information in the NVM-based journaling area is validated and the valid journal blocks are restored.

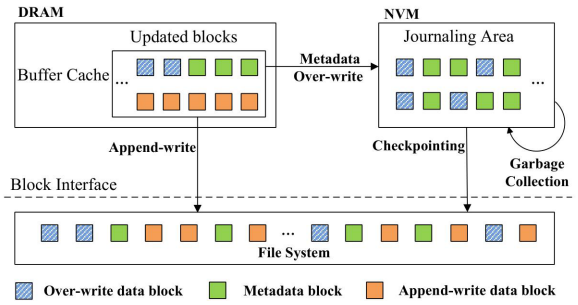


Fig. 2: The overall architecture of NJS.

3.2 NVM Data Structures and Space Management

Fig. 3 shows the space management and corresponding data structures used in the NVM-based journaling area. There are three types of structures: *superblock*, *journal header* and *journal block*. The superblock is followed by static journal headers and dynamically allocated journal blocks.

Superblock records the global information of the NVM-based journaling area. Two kinds of global information are kept in *superblock*: (1) the statistical information of the journaling space, e.g., the total amount of journal blocks; and (2) the pointers that mark the transactions and define the boundaries of logical areas described later in this section.

Journal header acts as the metadata of *journal block*. Each 16-byte *journal header* matches a 4KB *journal block*. As illustrated in Fig. 3, the leftmost 8 bits contain 3 flag bits, which are used to label the attributes of *journal block*. The f bit is used to label whether the matched *journal block* is free ('0') or occupied ('1'). The v bit is used to label whether the matched *journal block* is invalid ('0') or valid ('1'). The l bit is used to label whether the matched *journal block* is the old version ('0') or latest version ('1'). The remaining 120 bits are divided into four fields equally, which contain the journal header number (*jh_number*), the write count of the matched journal block (*write_count*), the journal block number (*j_blocknr*) and the corresponding file system block number (*fs_blocknr*).

To improve the search efficiency, the version information (old/latest) of the *journal block* is kept in a hash table in DRAM (described in Section 3.5). We reserve 256 journal headers, the total size equals to a journal block. The journal headers in this region do not match any journal blocks and are used to store persistent journal header copies during journal block swap operation (described in Section 3.3).

1. The parameters in the table are not strict, but give an overall concept of the characteristics.

Journal block is used to store journal data blocks. All the accesses to a *journal block* are handled via the journal block number (i.e., *j_blocknr*) kept in its corresponding *journal header*. NJS maintains a list of free blocks in DRAM for journal block allocation.

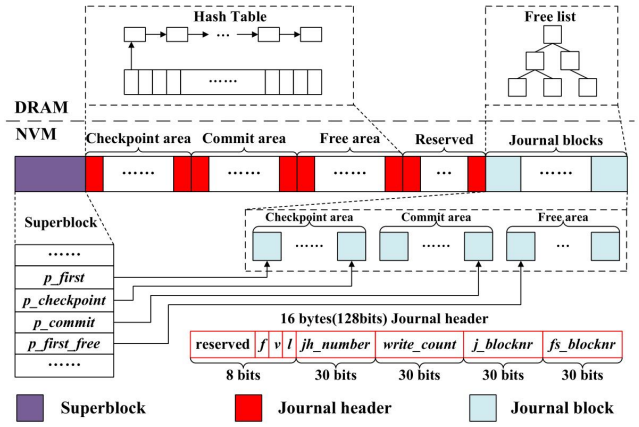


Fig. 3: NJS data structures layout and space management.

Logically, we divide the journal blocks into three areas: checkpoint area, commit area, and free area, as shown in Fig. 3. The pointers (e.g., *p_commit*, *p_checkpoint*) in the *superblock* are used to define the boundaries of these areas. The journal headers in each area are contiguous.

Checkpoint area: In this area, all the journal blocks belong to previous transactions which have been committed successfully. These journal blocks can be checkpointed to their original locations and they are in the consistent state. Pointer *p_first* points to the first block of this area. Pointer *p_checkpoint* points to the last block of this area. Since all the blocks in this area are needed to be searched and checked during checkpointing or recovery process, NJS keeps the blocks in this area contiguous to improve the search efficiency. Otherwise, extra data structures (e.g., linked list) are needed to be maintained in NVM while traversing the journal blocks in this area. However, the journal blocks in this area may be not contiguous due to the block swap operations (described in Section 3.3). As all the accesses to a journal block are handled via its corresponding journal header, NJS keeps the journal headers contiguous instead of the journal blocks in this area, as shown in Fig. 3.

Commit area: The journal blocks in this area belong to the current committing transaction. Pointer *p_commit* always points to the last block written to the NVM-based journal area. After the current committing transaction has been committed, the journal blocks in this area enter into checkpoint area. During the system recovery process (described in Section 3.7), the journal blocks in this area should be discarded. Like checkpoint area, the journal headers of the journal blocks in this area are contiguous.

Free area: The blocks in this area are used to store journal data blocks. NJS maintains a list of free journal blocks in DRAM for allocation. Pointer *p_first_free* points to the first block of this area, which is the block just after *p_commit* points to. When committing a journal block to the NVM-based journaling area, the journal block is written to the first block of free area.

3.3 Wear Aware Strategy for Journal Block Allocation

Write endurance is a crucial problem in NVM-based storage systems. Besides reducing the write traffic to NVM, allowing all the blocks in NVM to be worn out evenly is also important.

In order to avoid localized writes to the NVM-based journaling area, NJS adopts a wear aware strategy for NVM journal block allocation. NJS keeps a list of free journal blocks in DRAM, and uses a red-black tree to keep the free list sorted by write count. We choose red-black tree because it has fast search, insert and delete speeds. Before committing a journal block to the NVM-based journaling area, NJS assigns a block with the least number of write counts among the journal blocks in the free list. Actually, the least write count block is possibly not the first block of free area, NJS swaps the least write count block with the first block of free area. To better present the swap operation during allocation, we give an illustration as shown in Fig. 4. The number in the block represents the write count. The write count of the current first block of free area is 6, which is not the least write count. Thus we swap the first block with the least write count block whose write count is 2. After the swap operation, pointer *p_first_free* points to the least write count block 2, and the committed journal block can be written to this block. After committing, the write count is increased from 2 to 3. The least write count block 2 is removed from the red-black tree. By using the wear aware allocation strategy, each block in the NVM-based journaling area can be evenly worn out, and the contiguities of commit area and checkpoint area are maintained.

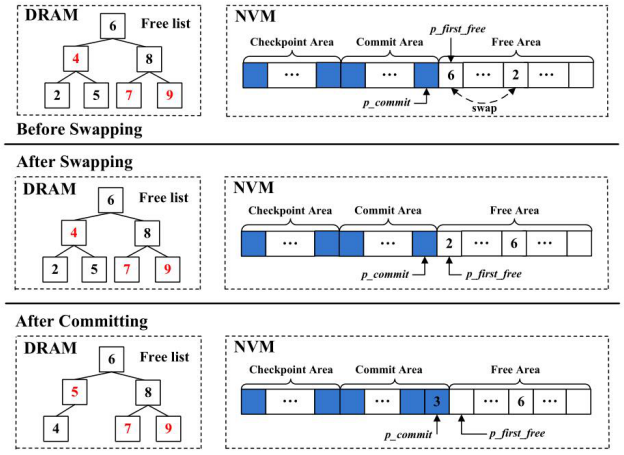


Fig. 4: An illustration of the swap operation during allocation. The number in the block represents the write count.

As each journal block is matched with a journal header, the swap operation is implemented by switching the match information in the corresponding journal headers instead of copying the contents of the journal blocks. For example, journal header *A* matches journal block *A* and journal header *B* matches journal block *B* before the swap operation. After the swap operation, journal header *A* matches journal block *B* and journal header *B* matches journal block *A*. Besides the journal block number, the write count and the corresponding file system block number are also switched during the swap operation. Since free journal blocks have

the same attributes, it is not necessary to switch the flag bits. Note that the journal header numbers are not switched. In order to guarantee the consistency of journal headers, the modifications of journal headers are performed in the copy-on-write manner by storing the persistent copies in the reserved journal header region (described in Section 3.2).

3.4 Transaction Commit Process in NJS

In order to decrease the amount of journal writes to NVM, we redesign the process of transaction committing, in which only the metadata and over-write data blocks are written to NVM as write-ahead logging, while the append-write data blocks are directly issued to the file system. This technique of differentiating over-writes from append-writes first appears in [27], but it is used in the traditional journaling scheme. We synergize this technique in an NVM-based journaling scheme. In our NJS, eliminating append-writes to NVM can alleviate the slow write and the endurance limitation of NVM.

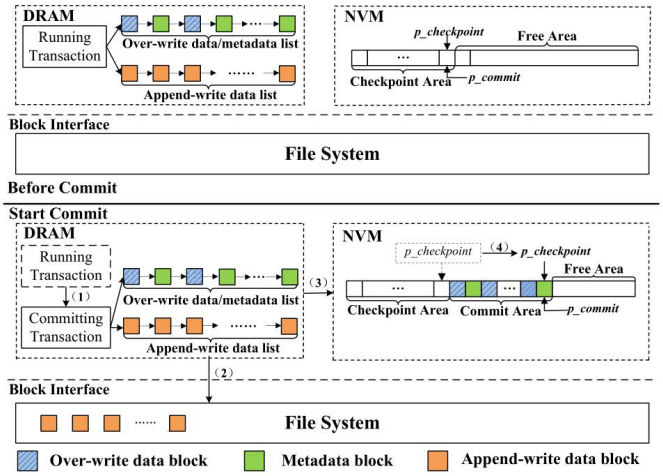


Fig. 5: Transaction commit process in NJS.

As illustrated in Fig. 5, each transaction in NJS contains two linked lists: one for append-write data blocks, and the other for over-write and metadata blocks. The transaction commit process is executed according to the following steps: (1) When the transaction commit process starts, Running Transaction is converted to Committing Transaction. Meanwhile, Running Transaction is initialized to accept newly updated blocks. (2) The data blocks in the append-write list of Committing Transaction are directly written to the file system through block interface. (3) The over-write data blocks and the modified file system metadata blocks are written to NVM-based journaling area through *memcpy*. Pointer *p_commit* always points to the last block written to the journal space. In order to guarantee the persistence of the journal writes to NVM, when a *memcpy* finishes, the corresponding cachelines should be flushed (*clflush*) along with memory fence (*mfence*) instruction. (4) After all the append-write data blocks and over-write data/metadata blocks have been persisted to the file system and NVM-based journaling area, *p_checkpoint* is updated to the block *p_commit* pointed to by an 8-byte atomic write followed by *clflush* and *mfence* instructions to indicate Committing Transaction commits successfully.

3.5 Byte-level Journal Update Scheme

As mentioned before, we need to fully consider the byte-accessibility characteristic of NVM while reducing the journaling overhead. Thus, NJS includes a byte-level journal update scheme that a journal block can be in-place updated based on the delta, which is computed based on the difference of the old and new versions of the journal block.

Since over-write data and metadata blocks store in the NVM-based journaling area, these blocks may be updated again in a very short time due to the workload locality. Particularly, metadata blocks are accessed and updated more frequently than data blocks [12]. Thus different versions of the same block possibly exist in NVM, and the difference (i.e. delta) between the different versions can be as small as several bytes according to the content locality [28], e.g., an update to an inode or a bitmap. Based on this observation, when a journal block has to be updated, for the frequently updated journal blocks in NJS, we only write the modified bytes (i.e. delta) to the existing journal block instead of writing another entire block. This update scheme not only leverages the byte-accessibility characteristic of NVM, but also further reduces the journal writes to NVM.

However, we can not directly write the delta to the latest version of journal block. If a system crash occurs during the update process, the latest version of journal block may be partially updated. In this case, the data consistency is compromised. Therefore, an old version is maintained for each journal block in addition to the latest version. We hence write the delta to the old version of journal block to complete the update. In order to improve search efficiency, the version information of journal block is kept in a hash table, as shown in Fig. 6. As the hash table is only used to improve the search efficiency, and the version information is also maintained in the corresponding journal headers (e.g., valid/invalid, old/latest), we keep the hash table in DRAM, instead of persistent NVM. The hash table consists of a table head array and some entry lists. The unit of the table head array is called hash slot. In each hash slot, the content is an address that points to an entry list. Each entry has the following items:

blocknr: the logical block number of the file system.

old: the address of the journal header of the old version journal block.

latest: the address of the journal header of the latest version journal block.

next: the pointer to the next item in the entry list.

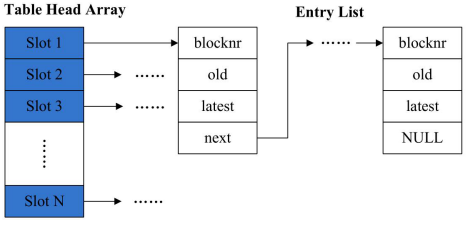


Fig. 6: Hash table for improving search efficiency.

When committing a journal block to the NVM-based journaling area, we first search the corresponding hash slot by hashing the logical block number of that journal block, then search the entry list in the slot. There are three cases:

(1) Neither the old version nor the latest version exists. It means that this journal block is a new block. In this case, the committing journal block is directly written to the first block of free area. We allocate a new entry in the entry list, and add the journal header of this block to *latest* item. Then the journal block is tagged as the latest version.

(2) We find this block in the entry list and only *latest* item contains the journal header of this block. It implies that the latest version exists but the old version does not exist. In this case, the committing journal block is still written to the first block of free area directly. The existing latest version of the block is tagged as the old version and the journal header is added to *old* item. The committing journal block is tagged as the latest version and the journal header of this block is added to *latest* item.

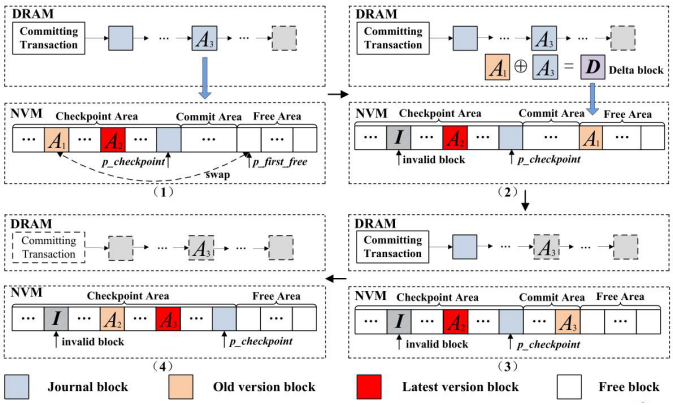


Fig. 7: An example of byte level update scheme.

(3) We find this block in the entry list and both *latest* item and *old* item contain journal headers. This implies that we can write the delta to the existing old version of journal block instead of writing another entire block. In this case, the old version of journal block is possibly in checkpoint area due to being committed to NVM earlier in previous transactions. In our design, the transaction commit process is performed in commit area, and all the journal blocks in checkpoint area should be kept in the consistent state. To avoid directly modifying journal block in checkpoint area, we swap out the old version of journal block with the currently least write count free block (*p_first_free*) before writing the target journal block to the NVM-based journaling area. For example, as shown in Fig. 7, block A is contained in the committing transaction. The committing version is A_3. The latest version A_2 and the old version A_1 exist. The update process of A is executed according to the following steps: 1) Before writing A_3 to NVM, the old version A_1 is swapped out with the currently least write count free block, and the free block exchanged to checkpoint area is tagged invalid. 2) The difference (block D) between A_3 and A_1 is calculated. D(i) represents the i-th byte of block D. If D(i) is not all zero bytes, the i-th byte of A_3 and A_1 is different. 3) A_1 is in-place updated to A_3 with D. Only the different bytes between A_3 and A_1 are written. 4) A_3 is tagged to the latest version, A_2 is tagged to old version, and the corresponding version information in the hash table is updated.

Even though there are extra overheads in the proposed journal update technique, such as search on the hash table, a read operation on journal block before writing and XOR calculation overhead. The experimental results in Section 4.3 prove that these overheads are negligible compared with the reduction of journal writes.

3.6 Garbage Collection and Checkpointing in NJS

In traditional journaling schemes, checkpointing is performed when the free journal space is lower than a predefined threshold. Due to writing journal blocks to slow HDDs during checkpointing, frequently checkpointing operations cause significantly high latency.

In our NJS, as invalid blocks exist in the NVM-based journaling area, we propose a garbage collection mechanism to recycle the invalid blocks. When the free journal space is lower than a predefined threshold, garbage collection is performed before checkpointing.

As shown in Fig. 8, the garbage collection process is executed as follows: (1) Before garbage collection begins, pointer *first* points to the first block of checkpoint area and *last* points to the last block of checkpoint area. (2) When garbage collection starts, the state of each journal block is examined, 1) if the block *first* pointed to is valid, *first* advances to the next journal block; 2) if the block *last* pointed to is invalid, *last* retreats to the previous journal block; 3) if the block *first* pointed to is invalid and meanwhile the block *last* pointed to is valid, the two blocks are swapped. Then pointer *first* advances to the next journal block and *last* retreats to the previous journal block. (3) When *first* and *last* point to the same block, all the invalid journal blocks have been swapped out successfully. Pointer *p_checkpoint* is updated to the last valid block by an 8-byte atomic write followed by *clflush* and *mfence* instructions, which indicate the garbage collection process finishes. (4) After the garbage collection process finishes, the recycled invalid blocks are inserted into the free list.

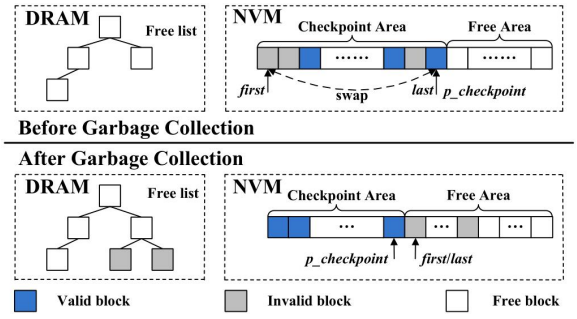


Fig. 8: Garbage collection process in NJS.

If the free journal space is still lower than a predefined threshold after garbage collection or the periodical checkpointing time interval arrives, the latest version of valid journal blocks in checkpoint area are written back to the file system. In traditional journaling file systems, journal data blocks are stored in slow HDDs. It is necessary to maintain checkpointing transactions in DRAM. Checkpointing must be performed periodically to keep the data in the file system up-to-date and prevent the journal from growing too large. In our NJS work, journal blocks are stored in fast NVM which co-exists with DRAM at memory level. The up-to-date journal data blocks can be checkpointed directly from

NVM to their original locations. Thus, it is not necessary to maintain checkpointing transaction in DRAM. As append-write data blocks have been synchronized to the file system, only the over-write data blocks and metadata blocks are needed to be written to their original locations.

As illustrated in Fig. 9, the checkpointing process is executed as follows: (1) From the first block (pointer *p_first*) to the last block (pointer *p_checkpoint*), all of the journal blocks in checkpoint area are checked, only the latest valid journal blocks are synchronized to the file system. (2) After that, all of the journal blocks in the NVM-based journal area are marked invalid and another round of the garbage collection process is performed. Then all of the corresponding block entries in the hash table are deleted, and the recycled blocks are inserted into the free list.

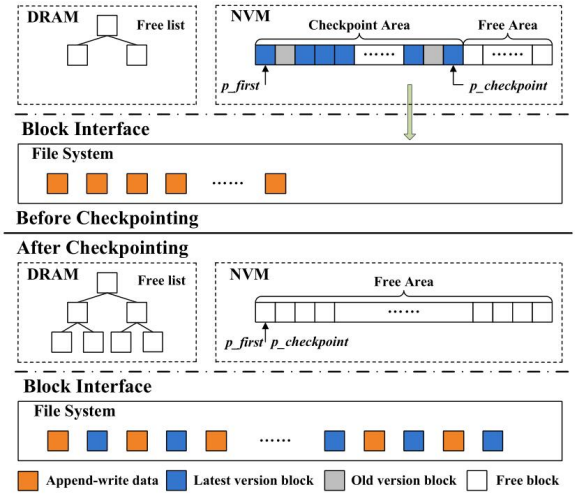


Fig. 9: Checkpointing process in NJS.

By using the proposed journal update and garbage collection schemes, the redundant journal updates can be absorbed, the NVM-based journaling area can store more journal blocks, the process of checkpointing is delayed. Therefore, checkpointing is triggered less frequent and the writes to the file system are reduced. Moreover, some random writes can be merged into sequential ones during garbage collection and checkpointing process. In this way, the file system performance can be further improved.

3.7 System Recovery

As file systems are possible to be in an inconsistent state only when system crashes occur during data updating, we classify the possible cases into two scenarios.

First, a system crash occurs during a commit operation. In this case, the current committing transaction is possible not to completely committed, including the append-write data to the file system and journal blocks to NVM. As append-writes do not modify any original data, they will not compromise the file system consistency. In order to restore to the last consistent state, NJS scans the whole NVM-based journaling area. While scanning the journaling area, NJS reconstructs the hash table and the free block list by using the version information and write count kept in journal headers. The journal blocks in commit area are discarded due to partially committed. The latest version of valid journal blocks in checkpoint area are written to their home locations in the file system. After all the latest version of valid journal blocks have been synchronized to their home locations, the file system recovers to the last consistent state that the last transaction is committed successfully.

Note that system crashes possibly occur during the swap operation (e.g., byte-level update, garbage collection). As the modifications of journal headers are performed in the copy-on-write manner by storing the persistent copies in the reserved journal header region, the journal headers in the reserved region also need to be scanned while scanning the whole journaling area. If the reserved region is not empty, the persistent journal header copies should be recovered to their home location by using their journal header numbers. After that, NJS continues to synchronize the latest version of valid journal blocks in checkpoint area to their home locations to finish the recovery process.

In the second scenario, a system crash occurs during a checkpoint operation. In this case, the journal blocks in the NVM-based journal space are partially reflected to the file system. However, the journal blocks still remain in NVM and are in consistent state. NJS restores the file system to the consistent state by simply replaying the journal blocks to their original locations in the file system again.

4 Performance Evaluation

In this section, we evaluate the performance of our NJS and answer the following questions:

(1) *How does NJS perform against traditional journaling schemes and other journaling schemes on NVM?*

(2) *What are the benefits of the byte-level journal update scheme, the garbage collection mechanism and eliminating append-write to NVM?*

(3) *How is NJS sensitive to the variation of the NVM write latency size and the NVM-based journaling space size?*

(4) *How does NJS address the endurance problem of NVM?*

We first describe the experimental setup and then evaluate NJS with different benchmarks.

4.1 Experimental Setup

We implement a prototype of NJS on Linux 3.12 and integrate it into Ext4 file system. The configurations of the server are listed in Table 2. Since NVM is not yet commercially available, we develop a simple NVM simulator based on the simulator used in the open source project Mnemosyne [29] to evaluate our NJS's performance. Like previous research works on NVM [7], [24], [29], we introduce extra latency in our NVM simulator to account for NVM's slower writes than DRAM. We prototype NJS with the characteristics of PCM because it is mature and more promising for commercial deployment [19], and our simulator can be also used in other NVMs. In our experiments, we set NVM latency to 300ns by default [7], [24]. Besides the write latency, we set the write bandwidth of NVM to 1GB/s, about 1/8 of the DRAM's bandwidth [7], [24]. The default capacity of NVM we use in NJS is 128MB as it is the default journal size value in Ext4 file system.

We evaluate the performance of our NJS against three existing journaling schemes listed in Table 3. For fairness,

the ramdisk used in JBD2 has the same parameters with the above NVM performance model. The journaling mode of Ext4 is set to *data journal*, which logs both data and metadata, to provide version consistency like NJS. For the recently proposed work FGM [14], although it focus on metadata journaling and only provide data consistency, it is closely related with our work and we add it into the evaluation comparison. The commit interval is set to 5 seconds according to the conventional configurations. In NJS, garbage collection is performed when three fourths of journal space is filled, and checkpoint is either triggered by a 5-minute timer or the utilization of the journaling area being over 75% after garbage collection.

TABLE 2: Server Configurations

CPU	Intel Xeon E5-2620, 2.0 GHz
CPU cores	12
Processor cache	384KB/1.5MB/15MB L1/L2/L3 cache
DRAM	8GB
NVM	Emulated with slowdown, the write latency is 300 ns, the write bandwidth is 1 GB/s
HDD	WD 1TB, 7200-RPM
Operating system	CentOS 6.5, kernel version 3.12.0

TABLE 3: Existing Journaling Schemes for Comparison

JBD2 on NVM	Traditional journaling scheme in Ext4 file system built on ramdisk
FGM [14]	Fine-grained metadata journaling on NVM, a recently proposed journaling scheme on NVM
No Journal	Ext4 file system without journaling

Three well-known and popular storage benchmarks are used to measure the performance of our NJS work: IOzone [30], Postmark [31], and Filebench [15]. The main parameters of the workloads used in our experiments are shown in Table 4.

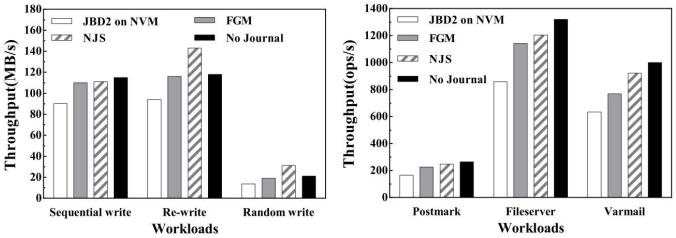
TABLE 4: Parameters of Different Workloads

Workloads	R/W Ratio	# of Files	File Size	Write Type
Sequential write	Write	1	8GB	Append-write
Re-write	Write	1	8GB	Over-write
Random write	Write	1	8GB	Over-write
Postmark	1:1	10K	1KB∼1MB	Append-write
Fileserver	1:2	50K	128KB	Append-write
Varmail	1:1	400K	16KB	Append-write

The purpose of NJS is to reduce the journaling overhead of traditional file systems, we only choose Fileserver and Varmail in Filebench, because these two workloads contain a large proportion of write operations.

4.2 Overall Performance

In synthetic workloads, we use Sequential write, Re-write and Random write scenarios in IOzone benchmark to evaluate the throughput performance. As shown in Fig. 10 (a), NJS outperforms JBD2 on NVM by 23.2%, 52.1% and 131.4% in Sequential write, Re-write and Random write scenarios respectively. In Sequential write, all writes are append-write, and only metadata blocks are written to NVM. The proportion of metadata is very small because IOzone benchmark only has a single large file in the evaluation. Thus the function of journaling reduction and delayed checkpointing in NJS can not be fully leveraged. In Sequential write, the throughput of NJS is nearly equal to FGM. Similarly, the proposed journaling scheme in FGM can not be well leveraged in this case. In Re-write and Random write scenarios, all writes are over-write, both metadata and data blocks are written to NVM, and NJS even performs better than No Journal. Although NJS has one more copy compared with No Journal, the extra copy is on fast NVM. Moreover, under the proposed journal update and garbage collection schemes in our NJS, checkpointing is delayed and can be triggered less frequent. The disk writes to the file system are reduced, and some random writes can be merged into sequential ones, which are preferred in disk I/O. Therefore, NJS performs better than No Journal, especially in Random write due to the long seek latencies of HDDs. In Re-write and Random write scenarios, our NJS performs better than FGM consistently. In these two scenarios, FGM still only writes metadata blocks to NVM as write-ahead logging. In our NJS, although both metadata and data blocks are written to NVM, the proposed garbage collection scheme can reduce the frequency of checkpointing and merge some random writes into sequential ones.



(a) Throughput of Synthetic Workloads (b) Throughput of Macro Benchmarks

Fig. 10: Overall Throughput of Different Workloads.

In macro workloads, we use Postmark, Fileserver and Varmail to evaluate the throughput performance. Fig. 10 (b) shows the throughput performance comparison of JBD2 on NVM, FGM, NJS and No Journal. As shown in the figure, NJS exhibits better than JBD2 on NVM by 50.3%, 40.2% and 45.3% in Postmark, Fileserver and Varmail respectively. In these benchmarks, all the writes are append-write, and for NJS, only metadata blocks need to be logged to NVM. Large amounts of journal writes can be eliminated. For JBD2 on NVM, although all the updates are written to the ramdisk-based journal device, the software overheads caused by the generic block layer are still non-trivial. We notice that NJS still outperforms FGM consistently. This is because our NJS leverages a garbage collection mechanism which can reduce the frequency of checkpointing and the writes to the file system.

4.3 Effect Analysis in NJS

In this section, we evaluate the effect of the byte-level journal update scheme, the garbage collection mechanism and eliminating append-write to NVM on our NJS.

Effect of the Byte-level Journal Update Scheme

The proportion of the difference between the old and new versions of journal blocks can affect the performance. We add another run of Re-write test while changing the proportions of differences from 0% to 100%. And we add NJS without the byte-level journal update scheme (referred

as NJS-no-BJ) into the comparison. Fig. 11 (a) shows the results normalized to the throughput of JBD2 on NVM. NJS achieves the performance improvement of up to 19.4%, and 10.2% on average compared with NJS-no-BJ. It is clear that the more similar the contents of the two blocks are, the more performance improvement can be achieved. In 100% case, which is the worst case, the throughput of NJS is a bit (under 1%) less than that of NJS-no-BJ. The reason is that, in this case, NJS has to update the entire block like NJS-no-BJ with some extra overheads mentioned in Section 3.5, thus these overheads are negligible. Actually, the differences between new and old versions are usually very small due to workload locality. The worst case hardly appears. Furthermore, we observe that even in 100% case, NJS still performs better than JBD2 on NVM by 27.8%. This is because garbage collection plays an important role in improving the performance.

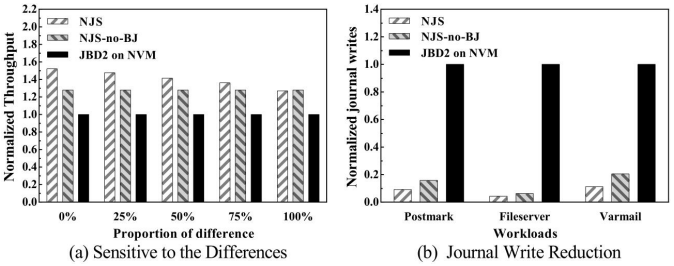


Fig. 11: Effect of the Byte-level Update Scheme.

In order to evaluate the journal write reductions from the byte-level journal update scheme, we measure the journal write amounts of Postmark, Fileserver and Varmail. Fig. 11 (b) shows the results normalized to JBD2 on NVM. It is clear that both NJS-no-BJ and NJS gain significantly journal write reductions compared with JBD2 on NVM. This is because all the writes in Postmark, Fileserver and Varmail are append-write and the data journal writes can be eliminated. Specifically, NJS reduces the journal write amount by 41.7%, 30.3% and 45.4% in Postmark, Fileserver and Varmail respectively compared with NJS-no-BJ. The byte-level journal update scheme further reduces the journal writes. We observe that the amount of journal write reduction in Fileserver is less than that in Postmark and Varmail. The reason is that Postmark and Varmail are metadata intensive workloads, the proportion of metadata in Postmark and Varmail are higher than that in Fileserver, thus the byte-level journal update scheme reduces more journal writes.

Effect of the Garbage Collection Scheme

To evaluate performance gains from the garbage collection mechanism, we test the throughput of NJS, NJS without the function of garbage collection (referred as NJS-no-GC) and JBD2 on NVM under the aforementioned macro-benchmarks. Fig. 12 (a) shows the results normalized to the throughput of JBD2 on NVM. NJS performs better than NJS-no-GC by 30.7%, 19.8% and 28.6% in Postmark, Fileserver and Varmail respectively. The results indicate that the function of garbage collection plays an important role in improving the performance. Note that the improvement in Fileserver is less than that in Postmark and Varmail. The reason is that Postmark and Varmail are metadata intensive workloads, the proportion of metadata in Fileserver is lower than that in Postmark and Varmail, garbage collection can be better leveraged in Postmark and Varmail than in Fileserver.

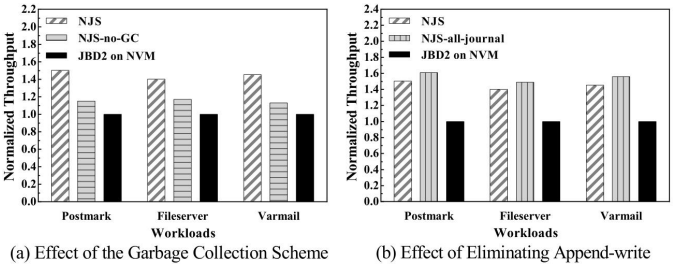


Fig. 12: Effect of Garbage Collection and Eliminating Append-write to NVM.

Effect of Eliminating Append-write to NVM

To examine the effect of eliminating append-write to NVM, we evaluate the throughput of NJS, NJS with all of the data blocks logged to NVM (referred as NJS-all-journal) and JBD2 on NVM under the aforesaid macro-benchmarks. Fig. 12(b) shows the results normalized to the throughput of JBD2 on NVM. In three workloads, NJS-all-journal outperforms NJS. But the improvements in Postmark, Filebench and Varmail are only 7.1%, 6.3% and 7.4%, respectively. The reason is that, for NJS-all-journal, even though all of the data are logged to NVM and written back to the file system with delay, the file system is frozen during garbage collection and checkpointing step, larger journal results in longer garbage collection and checkpointing latency. But as shown in Fig. 11 (b), the amount of journal writes in NJS-all-journal is much more than that in NJS. Therefore, it is not necessary to log append-write data to NVM.

4.4 Sensitivity Analysis

As the NVM latency and the NVM-based journaling area size can affect the system performance, we evaluate their effects on our NJS in this section.

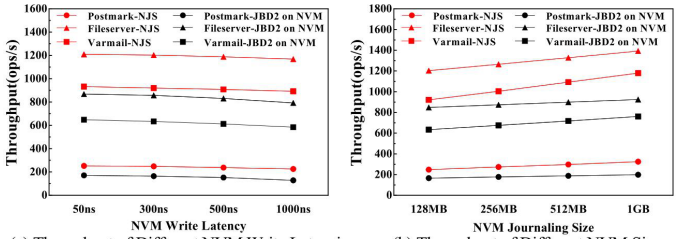
Sensitivity to the NVM Write Latency

As described in Section 2.2, different NVMs have different write latencies. The NVM write latency can affect the system performance. Fig. 13 (a) shows the throughput performance of NJS and JBD2 on NVM under the aforesaid macro-benchmarks as the NVM latency varies from 50ns to 1000ns. As shown in the figure, the throughputs of NJS and JBD2 on NVM decrease as the NVM latency increases in all of the three macro-benchmarks. And the throughput of JBD2 on NVM decreases more severely than NJS in all of the three macro-benchmarks. Moreover, compared with JBD2 on NVM, the performance improvements of NJS increase as NVM has longer write latency. For instance, NJS exhibits better than JBD2 on NVM by 43.9%, 45.3%, 48.1% and 52.7% when the NVM write latency is 50ns, 300ns, 500ns and 1000ns in Varmail. This is because the byte-level journal update scheme in NJS can reduce journal writes to NVM. When the NVM write latency increases, the throughput of NJS deceases slower than JBD2 on NVM. Therefore, NJS gains more performance benefits as the NVM write latency increases.

Sensitivity to the NVM Size

The NVM-based journaling area size also has a strong impact on the system performance. Fig. 13 (b) shows the

throughput performance of NJS and JBD2 on NVM under the aforesaid macro-benchmarks as we vary the NVM size from 128MB to 1GB. From the figure we observe that the throughputs of NJS and JBD2 on NVM increase as the NVM size grows in all of the three macro-benchmarks. This is because when the journal space size becomes larger, the frequency of checkpointing decreases. Moreover, the throughput of NJS increases faster than JBD2 on NVM in all of the three macro-benchmarks. Compared with JBD2 on NVM, the performance improvements of NJS increase as the NVM-based journaling area size grows. For instance, NJS performs better than JBD2 on NVM by 40.2%, 44.7%, 47.7% and 50.6% when the NVM size is 128MB, 256MB, 512MB and 1GB in Fileserver. The reason is that, as the NVM size grows, more journal blocks can be stored in NVM, the garbage collection scheme can be leveraged better. Hence NJS gains more performance improvements as the NVM size grows.



(a) Throughput of Different NVM Write Latencies (b) Throughput of Different NVM Sizes

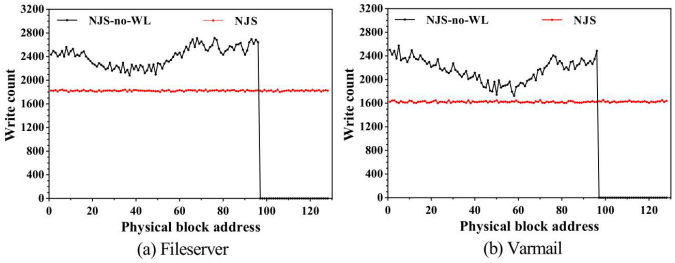
Fig. 13: Throughput of Different NVM Write Latencies and NVM sizes.

4.5 Endurance Analysis

Limited write endurance is an important weakness for NVMs. A straightforward way to expand the lifetime of NVM is reducing the write amount. In order to reduce the write traffic to NVM, deploying a DRAM buffer in front of NVM is a solution [32]. However, most writes incurred by journaling are synchronous writes which must be flushed to persistent storage device immediately. If journal writes are buffered at volatile DRAM for too long, the file system consistency will be compromised. In our NJS, eliminating append-write journal data writes to NVM can reduce the write traffic to NVM and effectively alleviate the endurance limitation.

To further extend the lifetime of NVM, wear-leveling techniques [22] are also important. In journaling file systems, checkpointing is performed periodically to keep the data in the file system up-to-date. In general, the updates are committed to the journaling area sequentially from the first block, checkpointing is triggered when the free journal space is lower than a predefined threshold or the periodical checkpoint time interval arrives. In this way, the remaining free journaling space behind the predefined threshold will be never accessed. Therefore, a wear-leveling technique is necessary when NVM is used as journaling storage device.

In order to avoid localized writes to the NVM-based journaling area, NJS includes a wear aware strategy for journal block allocation which assigns the least write count block in the free area while committing a block to NVM. To understand the effectiveness of NJS's wear aware allocation strategy, we collect the write count information of Fileserver and Varmail at block granularity, like the previous work [33]. We divide the NVM journal blocks into 128 intervals, and calculate the total write count in each interval.



(a) Fileserver (b) Varmail

Fig. 14: Write Count Distribution of NJS and NJS without Wear Leveling Scheme.

Fig. 14 shows the write count distribution of NJS and NJS without wear leveling scheme (referred as NJS-no-WL) under Fileserver and Varmail. As checkpointing in NJS is triggered when the free journal space is lower than 25%, the write counts of the blocks in the last 25% space are 0. Moreover, the blocks in the front 75% are not worn evenly. This is because NJS adopts a byte-level journal update scheme in which NJS only needs to write modified bytes to the existing journal block. The write counts of the frequently updated journal blocks are more than that of infrequently updated journal blocks. As shown in the figures, the write counts of NJS in Fileserver and Varmail are almost evenly distributed over all intervals. Specifically, the maximum write count of NJS-no-WL is higher than that of NJS by 47.2% and 55.6% in Fileserver and Varmail, respectively. These results can prove the effectiveness of NJS's wear aware allocation strategy.

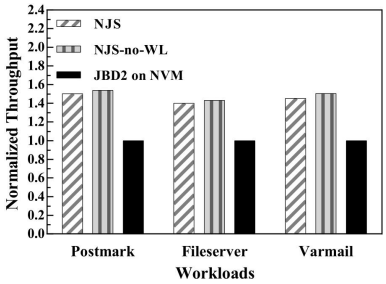


Fig. 15: Effect of the Wear Aware Allocation Strategy.

At last we examine the performance impact of the wear aware allocation strategy. We evaluate the throughput of NJS, NJS-no-WL and JBD2 on NVM under the aforesaid macro-benchmarks. Fig. 15 shows the results normalized to the throughput of JBD2 on NVM. In three workloads, NJS-no-WL performs better than NJS. Specifically, the improvements in Postmark, Fileserver and Varmail are only 2.4%, 2.1% and 3.6%, respectively. These results indicate that the overhead of the wear aware allocation strategy is acceptable. The reason is that, the block swap operation during the wear aware allocation strategy is implemented by updating the information in the journal headers of the target journal blocks instead of simply copying the contents of the journal blocks. Considering the effectiveness of providing wear-leveling (the results in Fig. 14), it is necessary for NJS to adopt the wear aware allocation strategy.

5 Related Work

5.1 NVM-based File Systems

BPFS [34] uses short-circuit shadow paging technique and 8-byte atomic write to provide consistency. SCMFS [13] utilizes the existing OS VMM module and maps files to a large contiguous virtual address, thus reducing the complexity of the file system. Shortcut-JFS [21] is a journaling file system that assumes PCM as its standalone storage device, it proposes differential logging and in-place checkpointing techniques to reduce the journaling overhead. PMFS [24] is a lightweight POSIX file system designed for persistent memory, it bypasses the page cache and eliminates the copy overheads to improve performance. NOVA [7] is a NVM file system that adopts conventional log-structured file system techniques to guarantee strong consistency. HiNFS [32] uses a DRAM buffer to buffer the lazy-persistent file writes persists them to NVM lazily to hide the long write latency of NVM in order to improve file system performance. SoupFS [8] is a recently work that introduces the idea of soft updates into NVM-based file system, it significantly shortens the critical path latency by delaying most synchronous metadata updates. NOVA-Fortis [35] is another recently proposed work that adds fault tolerance to an NVM file system, and it is resilient in the face of corruption due to media errors and software bugs.

Different from the above mentioned NVM-based file systems, our NJS deploys NVM as journaling storage device to reduce the journaling overhead of traditional file systems. In the meanwhile, HDDs/SSDs can be used as major storage devices, thus NVM can be used to improve the performance of storage systems in a cost-effective way.

5.2 NVM-based journaling schemes

Lee *et al.* [2] proposed a buffer cache architecture UBJ that subsumes the functionality of caching and journaling with NVM. However, copy-on-write is used in journal block updating which does not exploit the byte-accessibility characteristic of NVM. Moreover, UBJ does not consider reducing journal writes to NVM. Zeng *et al.* [36] proposed an NVM-based journaling mechanism SJM with write reduction. Kim *et al.* [37] proposed a journaling technique that uses a small NVM to store a journal block as a compressed delta for mobile devices. However, these two works only use NVM as journal storage device and focus on reducing journal write traffic to NVM, but does not consider NVM delaying the checkpointing process of journaling file systems. Wei *et al.* [11] proposed a transactional NVM disk cache called Tinca that guarantees consistency of both disk cache and file system while reducing file system journaling overhead. However, Tinca deploys NVM as external disk cache to HDD or SSD with a lightweight transaction scheme. Different from Tinca, our NJS is an NVM-based journaling scheme. Chen *et al.* [14] proposed a fine-grained metadata journaling technique on NVM, and it is the work most related to ours. However, the proposed journaling technique only uses NVM to store the file system metadata, and provides data consistency. In contrast, our NJS logs the file system metadata and over-write data to NVM as write-ahead logging, and provides version consistency, which is a higher consistency level compared with data consistency. Moreover, our NJS leverages a garbage collection mechanism that absorbs the redundant journal updates and delays the checkpointing to the file system.

6 Conclusion

In this paper, we present an NVM-based journaling scheme, called NJS, to reduce the journaling overhead for traditional file systems. In order to decrease the amount of write to NVM due to its relatively long write latency and limited write cycles, NJS only logs the file system metadata and over-write data to NVM as write-ahead logging, and directly issues the append-write data to the file system. To avoid localized writes to NVM and further extend the lifetime of NVM, NJS adopts a wear aware allocation strategy in which all the NVM journal blocks can be evenly worn out. Furthermore, we design a byte-level journal update scheme in which a journal block can be updated in the byte-granularity based on the difference of the old and new versions of the journal block so as to exploit the unique byte-accessibility characteristic of NVM. NJS also proposes a garbage collection mechanism that absorbs the redundant journal updates, and actively delays the checkpointing to the file system. Thus, the journaling overhead can be reduced significantly. Evaluation results show that Ext4 with our NJS outperforms Ext4 with a ramdisk-based journaling device by 57.1% on average in different workloads.

References

[1] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems." in *Proceedings of the 2005 USENIX Annual Technical Conference (ATC)*, 2005, pp. 196–215.

[2] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory." in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[3] A. Ahari, M. Ebrahimi, F. Oboril, and M. Tahoori, "Improving reliability, performance, and energy efficiency of STT-MRAM with dynamic write latency," in *Proceedings of the IEEE 33rd International Conference on Computer Design (ICCD)*, 2015, pp. 109–116.

[4] Z. Li, F. Wang, D. Feng, Y. Hua, J. Liu, W. Tong, Y. Chen, and S. S. Harb, "Time and space-efficient write parallelism in PCM by exploiting data patterns," *IEEE Transactions on Computers*, vol. 66, no. 9, pp. 1629–1644, 2017.

[5] M. Mao, Y. Cao, S. Yu, and C. Chakrabarti, "Optimizing latency, energy, and reliability of 1T1R ReRAM through appropriate voltage settings," in *Proceedings of the IEEE 33rd International Conference on Computer Design (ICCD)*, 2015, pp. 359–366.

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

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

[8] M. Dong and H. Chen, "Soft updates made simple and fast on non-volatile memory," in *2017 USENIX Annual Technical Conference (ATC)*, 2017, pp. 719–731.

[9] S. Yu and P.-Y. Chen, "Emerging memory technologies: Recent trends and prospects," *IEEE Solid-State Circuits Magazine*, vol. 8, no. 2, pp. 43–56, 2016.

[10] J. Boukhobza, S. Rubini, R. Chen, and Z. Shao, "Emerging NVM: A survey on architectural integration and research challenges," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 2, p. 14, 2017.

[11] Q. Wei, C. Wang, C. Chen, Y. Yang, J. Yang, and M. Xue, "Transactional NVM cache with high performance and crash consistency," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis(SC)*. ACM, 2017.

[12] J. Chen, Q. Wei, C. Chen, and L. Wu, "FSMAC: A file system metadata accelerator with non-volatile memory," in *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, 2013.
[13] X. Wu and A. Reddy, "SCMFS: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.
[14] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue, "Fine-grained metadata journaling on NVM," in *Proceedings of the IEEE 32nd Symposium on Mass Storage Systems and Technologies (MSST)*, 2016.
[15] R. McDougall, "Filebench: Application level file system benchmark," 2014.
[16] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering." in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.
[17] S. Mittal, J. S. Vetter, and D. Li, "A survey of architectural approaches for managing embedded DRAM and non-volatile on-chip caches," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1524–1537, 2015.
[18] 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.
[19] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," *ACM Transactions on Storage (TOS)*, vol. 10, no. 4, p. 15, 2014.
[20] F. Xia, D.-J. Jiang, J. Xiong, and N.-H. Sun, "A survey of phase change memory systems," *Journal of Computer Science and Technology*, vol. 30, no. 1, pp. 121–144, 2015.
[21] E. Lee, S. Hoon Yoo, and H. Bahn, "Design and implementation of a journaling file system for phase-change memory," *IEEE Transactions on Computers*, vol. 64, no. 5, pp. 1349–1360, 2015.
[22] F. Huang, D. Feng, Y. Hua, and W. Zhou, "A wear-leveling-aware counter mode for data encryption in non-volatile memories," in *Proceedings of the IEEE 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017, pp. 910–913.
[23] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Proceeding of the 41st International Symposium on Computer Architecuture (ISCA)*, 2014, pp. 265–276.
[24] 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 9th European Conference on Computer Systems (EuroSys)*, 2014.
[25] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.
[26] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proceedings of the IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.
[27] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013, pp. 228–243.
[28] G. Wu and X. He, "Delta-FTL: improving SSD lifetime via exploiting content locality," in *Proceedings of the 7th ACM european conference on Computer Systems (EuroSys)*, 2012, pp. 253–266.
[29] H. Volos and M. M. Swift, "Mnemosyne: Lightweight persistent memory," http://research.cs.wisc.edu/sonar/projects/mnemosyne/index.html.
[30] W. D. Norcott and D. Capps, "Iozone filesystem benchmark," *URL: www. iozone. org*, vol. 55, 2003.
[31] N. Appliance, "Postmark: A new file system benchmark," 2004.
[32] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, 2016.
[33] K. Zeng, Y. Lu, H. Wan, and J. Shu, "Efficient storage management for aged file systems on persistent memory," in *Proceedings of the IEEE 2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2017, pp. 1773–1778.
[34] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 133–146.
[35] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff, "NOVA-Fortis: a fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, 2017, pp. 478–496.
[36] L. Zeng, B. Hou, D. Feng, and K. B. Kent, "SJM: an SCM-based journaling mechanism with write reduction for file systems," in *Proceedings of the 2015 International Workshop on Data-Intensive Scalable Computing Systems (DISCS)*, 2015.
[37] J. Kim, C. Min, and Y. Eom, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," *IEEE Transactions on Consumer Electronics*, vol. 60, no. 2, pp. 217–224, 2014.

Xiaoyi Zhang received the B.E degree in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 2011. He is currently working toward the PhD degree majoring in computer science and technology at HUST. His current research interests include NVM-based storage systems, NVM-based file systems, NVM-based data structures, and file system consistency.

Dan Feng received her B.E, M.E. and Ph.D. degrees in Computer Science and Technology from Huazhong University of Science and Technology (HUST), China, in 1991, 1994 and 1997 respectively. She is a Professor and the Dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems and parallel file systems. She has over 100 publications in journals and international conferences. She is the distinguished member of CCF, member of IEEE and ACM.

Yu Hua received the BE and PhD degrees in computer science from Wuhan University, China, in 2001 and 2005, respectively. He is currently a professor at Huazhong University of Science and Technology (HUST), China. His research interests include computer architecture, cloud computing, and network storage. He has more than 80 papers in major journals and international conferences. He has been on the organizing and program committees of multiple major international conferences. He is the distinguished member of CCF, senior member of ACM and IEEE, and the member of USENIX.

Jianxi Chen received his B.E degrees in Nanjing University, China, in 1999, M.S. in Computer Architecture from Huazhong University of Science and Technology(HUST), Wuhan, China in 2002 and Ph.D. in Computer Architecture, from HUST in 2006. He is currently with Wuhan National Lab for Optoelectronics, and School of Computer, HUST, as an Associate Professor. He publishes more than 20 papers in major journals and conferences. He is a member of CCF and IEEE.