A Cost-efficient NVM-based Journaling Scheme for File Systems

Xiaoyi Zhang, Dan Feng✉, Yu Hua and Jianxi Chen
Wuhan National Lab for Optoelectronics, Key Laboratory of Information Storage System (School of Computer Science and Technology, Huazhong University of Science and Technology), Ministry of Education of China, Wuhan, China
✉Corresponding author: dfeng@hust.edu.cn
{zhangxiaoyi, dfeng, csyhua, chenjx}@hust.edu.cn

***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 relatively slow write and endurance limitation of NVM. Our NJS consists of three major contributions: (i) In order to minimize 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 relatively slow write and endurance limitation of NVM. (ii) We propose a novel journaling update scheme in which the journaling 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. (iii) 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 original Ext4 with a ramdisk-based journaling device, the throughput improvement of Ext4 with our NJS is up to 137.1%.**

I. 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 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], [8]. However, due to expensive price and limited capacity, NVMs co-exist with HDDs and SSDs in storage systems, and in the meantime, 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 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 [9]. Second, NVMs have relatively long write latency and limited 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 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 [10], [11].

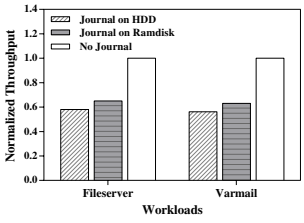


Fig. 1: Overhead of Traditional Journaling Schemes

In order to quantify the overhead of traditional journaling schemes, we measured the throughput of Ext4, with and without journaling under different workloads in Filebench [12]. Details about the experimental environment are described in Section IV-A. As shown in Figure 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, which indicates that traditional journaling schemes fail to perform efficiently in NVMs. Therefore, we consider highly-efficient journaling techniques for NVMs.

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. NJS proposes to use three optimization techniques:

(1) In order to alleviate the relatively slow write and endurance limitation of NVM, NJS minimizes the amount of journal writes. In the process of transaction committing, only the metadata of file systems and over-write data are needed to be written to NVM as write-ahead logging, and the append-write data blocks are directly issued to the file system.

(2) 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 different 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.

(3) 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 II provides the background of NVM, existing consistency and journaling techniques. Section III presents the design and detailed implementations. Experiment results are presented in Section IV. Related work is discussed Section V and we conclude the paper in Section VI.

II. Background

A. Consistency and Journaling for File Systems

File system consistency can be categorized into three levels, including *metadata consistency*, *data consistency*, and *version consistency* [13]. 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. The writeback mode only provides the 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, the version consistency is guaranteed. But for over-writes, the original data are modified. Thus, the ordered mode only provides data consistency.

The 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 for performance reasons. Hence, in order to meet the needs of data integrity in storage systems, it is important to obtain the version consistency in a cost-efficient manner.

B. 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 [14]. Among current NVM technologies, PCM is mature and more promising for volume production [15]. In 2012, Samsung announced in volume production of a 8 Gbit PCM device [16].

Different NVMs have similar limitations. First, NVMs have the read/write performance asymmetry. Write latency is much higher (i.e., 3-8X) than read latency [4], [5]. Second, NVMs generally have the limited write cycles, e.g., 10^8 times for PCM [4], 10^{10} times for ReRAM [5]. To extend the lifetime of NVMs, wear-leveling techniques have been proposed [17]. Since most of the proposed wear-leveling techniques are built on device level, we assume such wear leveling is present and do not address it in our NJS work. Actually, our design of reducing journal writes can be combined with wear-leveling techniques to further lengthen NVM's lifetime.

C. 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. In general, the atomicity of writes to memory is very small (8 bytes for 64-bit CPUs) [18]. The updates with larger sizes must adopt logging or copy-on-write mechanisms which require the memory writes to be in a correct order [19]. 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 [20]. In order to address the data consistency problem in NVM, 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 [21]. Thus, we should reduce the amount of journal writes to NVM.

III. Design and Implementations

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

A. Overview

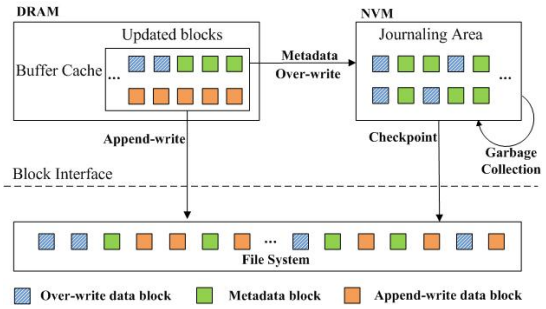


Fig. 2: Overall architecture of NJS

Figure 2 illustrates the workflow overview of a journal write request among the file system, buffer cache, and 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, the 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 checkpoint time interval arrives, the latest version of valid journal blocks in the journaling area are written back to the file system. When the system is rebooted due to unexpected crashes, the information in the journaling area is validated and the valid journal blocks are restored.

B. Journaling Space Management

Figure 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*.

Superblock is used to record the global information of the journaling space. 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 the logical areas described later in this Section.

Journal header acts as the metadata of *journal block* and all the accesses to *journal block* are handled via *journal header*. *Journal header* tracks the address of the matched *journal block*, the corresponding file system block number and some state bits for the attributes of the *journal block* (e.g., valid/invalid, old/latest). Each 16-byte *journal header* matches a 4KB *journal block*. We reserve 256 journal headers, the total size of which is equal to a journal block. These reserved journal headers will be used to store persistent copies of valid journal headers during garbage collection step (described in Section III-E).

Journal block is used to store journal data blocks.

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

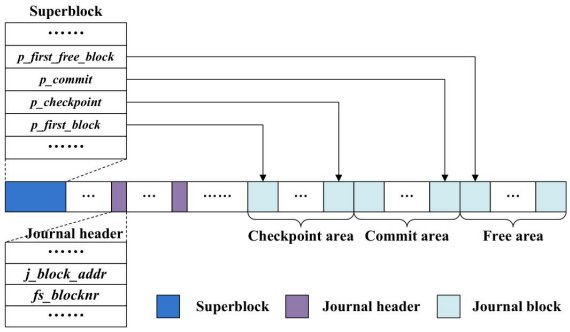


Fig. 3: Space management and corresponding data structures

Free area: The blocks in this area are used to store journal data blocks. The pointer *p_first_free_block* points to the first free block. When the journal blocks begin to be committed to the journaling area, they are written from the block *p_first_free_block* pointed to.

Checkpoint area: In this area, all the journal blocks belong to previous transactions which have been committed successfully, and these journal blocks are in the consistent state. The pointer ***p_checkpoint*** points to the last block of checkpoint area. When the checkpointing or system recovery process starts, only the blocks in this area are needed to be searched and recovered.

Commit area: The journal blocks in this area belong to the current committing transaction. The pointer *p_commit* always points to the last block written to the NVM-based journal area, which is the block just before *p_first_free_block* pointed to. During the system recovery process (described in Section III-F), the journal blocks in this area should be discarded.

C. Transaction Commit Process in NJS

In order to minimize the amount of journal writes to NVM, we redesign the process of transaction committing, in which only the metadata and over-write data are needed to be written to NVM as write-ahead logging, and the append-write data blocks are directly issued to the file system. This technique of differentiating over-writes from append-writes first appears in [22], 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 relatively slow write and endurance limitation of NVM.

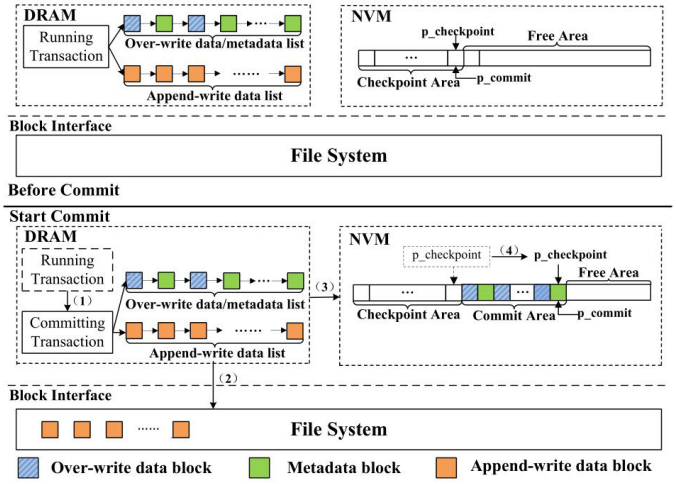


Fig. 4: Transaction commit process in NJS

As illustrated in Figure 4, 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 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*. And the pointer *p_commit* always points to the last block written to the journal space. When a *memcpy* finishes, the corresponding cachelines should be flushed (*clflush*) and then a memory fence (*mfence*) is issued. (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* to indicate the committing transaction commits successfully.

D. Byte Level Journal Update Scheme

Since over-write data and metadata blocks are stored in the NVM-based journaling area, these blocks may be updated again in a very short time due to workload locality. Specifically, metadata blocks are accessed and updated more frequently than data blocks [23]. Thus different versions of the same block possibly exist in the NVM-based journaling area, and the difference (i.e. delta) between different versions can be as small as several bytes according to the content locality [24], 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 can be partially updated and damaged. 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 the search efficiency, the version information of the journal blocks is kept in a hash table, as shown in Figure 5. Note that the hash table is only used to improve the search efficiency of journal block version information. And the version information is also maintained in the corresponding journal header (e.g. valid/invalid, old/latest), thus the hash table is kept 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.

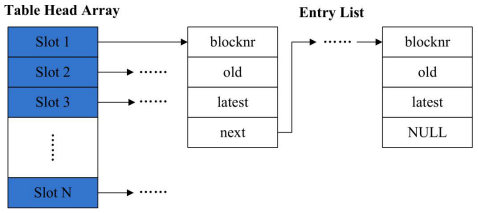


Fig. 5: Hash table for improving search efficiency

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.

(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 the checkpoint area due to being committed to the journaling area earlier in previous transactions. In our design, the commit process of a transaction is performed in the commit area, and all the journal blocks in the checkpoint area should be kept in a consistent state. To avoid directly modifying journal block in the checkpoint area, we swap out the old version of journal block with the currently first free block before writing the target journal block to the NVM-based journaling area. For example, as shown in Figure 6, 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 the journal space, the old version A_1 is swapped out with the currently first free block, and the free block exchanged to the 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 needed to be 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.

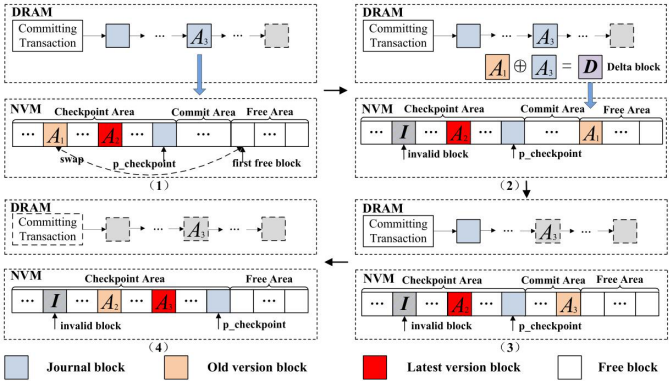


Fig. 6: The process of journal block update

The swap operation is implemented by updating the corresponding journal block address maintained in the journal header instead of simply copying the contents of the journal blocks. Note that the latest version block A_2 and its journal header are not modified, the consistency of the NVM-based journaling area is not decreased.

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 IV-C prove that these overheads are negligible compared with the reduction of journal writes.

E. Garbage Collection and Checkpointing in NJS

In traditional journaling schemes, checkpointing is performed when the free journal space is lower than a predefined threshold. In our NJS, as invalid blocks exist in the NVM-based journaling area, we propose a garbage collection mechanism to recycle the invalid blocks and delay the checkpointing process. The garbage collection step is shown in Figure 7.

Before garbage collection begins, the pointer *first* points to the first block in the checkpoint area and *last* points to the last block in the checkpoint area. When garbage collection starts, the state of each journal block should be examined, if the block *first* pointed to is invalid and meanwhile the block *last* pointed to is valid, the two blocks will be swapped. Then pointer *first* advances to the next journal block and *last* retreats to the previous journal block. When *first* and *last* point to the same block, the garbage collection process finishes. Then the pointer *p_checkpoint* is updated to the last valid block by an 8-byte atomic write. In the garbage collection process, as the journal headers of latest journal blocks are modified during the swap operation, the modifications of the updated journal headers are performed in the copy-on-write manner by using the reserved journal headers (described in Section III-B). Specifically, before updating a journal header of the valid journal block, it is copied to the first journal header of the reserved journal headers. After the swap operation completes, the current persistent copy in the reserved journal header is cleared to store the next persistent copy. Under the proposed garbage collection mechanism, the invalid journal blocks can be recycled, more journal blocks can be stored in the journaling area, and the checkpointing process is delayed.

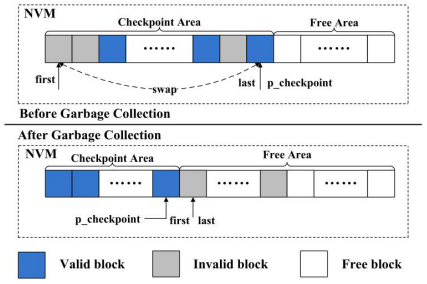


Fig. 7: Garbage collection

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 checkpoint area are then checkpointed to the file system. After that, all of the journal blocks in the journal space are marked invalid and another round of garbage collection process is performed. And all of the corresponding block entries in the hash table are deleted.

By using the proposed journal update and garbage collection schemes, the redundant journal updates are absorbed, the process of checkpointing is delayed, the writes to the file system are reduced, and some random writes can be merged into sequential ones. In this way, the file system performance is further improved.

F. System Recovery

When unexpected system crashes or power losses occur, the file system must recover to the last consistent state. 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. The current commit transaction is possible not to completely committed, including the append-write data to the file system and journal blocks to the NVM-based journal space. As an append-write does not modify any original data, we simply discard the append-write data blocks and journal blocks in the current commit transaction. In order to restore to the last consistent state, NJS scans the whole checkpoint area (the data blocks between *p_first_block* and *p_checkpoint*). Note that system crashes possibly occur during the garbage collection step, the data blocks of the reserved journal headers also need to be scanned. The latest version of valid journal blocks should be written to their home locations in the file system. After all the latest version of journal blocks have been updated to their home locations, the file system recovers to the last consistent state that the last transaction is committed successfully.

In the second scenario, a system crash occurs during a checkpoint operation. The journal blocks in the NVM-based journal space are partially reflected to the file system. However, the journal blocks still remain in the journaling area 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.

IV. Performance Evaluation

A. Experimental Setup

We implement a prototype of NJS on Linux 3.12 and integrate it into Ext4 file system. Since NVM is not yet commercially available, we develop a simple NVM simulator based on the simulator used in Mnemosyne [19] to evaluate our NJS's performance. We prototype NJS with the characteristics of PCM because it is mature and more promising for commercial deployment [15], [16], and our simulator can be also used in other NVMs. Like previous research works on NVM [18], [19], we introduce extra latency in our NVM simulator to account for NVM's slower writes. In our experiments, we set NVM latency to 300ns by default [8], [18]. Besides the write latency, we set the write bandwidth of NVM to 1GB/s, about 1/8 of the DRAM's bandwidth [8], [18]. The capacity of NVM we use in NJS is 128MB as it is the default journal size value in Ext4 file system.

For fairness, we compare our work (NJS) with Ext4 file system that uses the same capacity ramdisk with the above NVM performance model as its journaling device (Journal on Ramdisk). The journaling mode of Ext4 is set to journal, which logs both data and metadata, to provide version consistency like NJS. And we also add the non-journaling mode of Ext4 (No Journal) 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.

The used server is configured with an Intel Xeon E5-2620 CPU, 8GB DRAM, and WD 1TB HDD. Three well-known and popular storage benchmarks are used to measure the performance of our NJS work: IOzone [25], Postmark [26], and Filebench [12]. The main parameters of the workloads used in our experiments are shown in Table I.

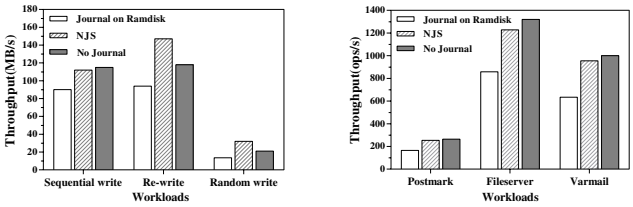
TABLE I: Parameters of Different Workloads

Benchmark	Workload	R/W Ratio	# of Files	File Size	Write Type
IOzone	Sequential Write	Write	1	8GB	Append-write
IOzone	Re-write	Write	1	8GB	Over-write
IOzone	Random Write	Write	1	8GB	Over-write
Postmark	Postmark	1:1	10K	1KB∼1MB	Append-write
Filebench	Fileserver	1:2	50K	128KB	Append-write
Filebench	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.

B. Overall Performance

In the synthetic workloads, we use Sequential Write, Re-write and Random Write scenarios in IOzone benchmark to evaluate the throughput performance. As shown in Figure 8 (a), NJS outperforms Journal on Ramdisk by 24.4%, 56.3% and 137.1% 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 the NVM-based journal area. The proportion of metadata is very small as IOzone benchmark creates a single large file in the evaluation. Thus the function of journaling reduction and delayed checkpointing in NJS can not be fully leveraged. In Re-write and Random Write, all writes are over-write, both metadata and data blocks are written to the NVM-based journal area, and NJS even performs better than No Journal. In this case, the delayed checkpointing plays an important role for NJS, especially in Random Write due to the long seek latencies of HDDs.



(a) Synthetic Workloads (b) Macro Benchmarks

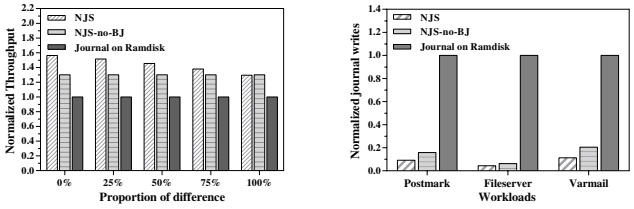
Fig. 8: The Throughput

In the macro workloads, we use Postmark, Fileserver and Varmail to evaluate the throughput performance. Figure 8 (b) shows the throughput performance comparison of Journal on Ramdisk, NJS and No Journal. As shown in the figure, NJS exhibits better than Journal on Ramdisk by 53.9%, 43.1% and 50.6% 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 Journal on Ramdisk, although all the updates are written to the ramdisk-based journal device, the software overheads are still non-trivial.

C. 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. Figure 9 (a) shows the results normalized to the throughput of Journal on Ramdisk. NJS achieves the performance improvement of up to 20.1%, and 10.8% 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 III-D, 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 Journal on Ramdisk by 30.1%. This is because the delayed checkpointing plays an important role in improving the performance.



(a) Sensitive to the Differences (b) Journal Write Reduction

Fig. 9: 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. Figure 9 (b) shows the results normalized to Journal on Ramdisk. It is clear that both NJS-no-BJ and NJS gain significantly journal write reductions compared with Journal on Ramdisk. 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.

D. Effect of the Delayed Checkpointing

To evaluate performance gains from the delayed checkpointing function, we test the throughput of NJS, NJS without the function of delayed checkpointing (NJS-no-DC) and Journal on Ramdisk under the aforementioned macro-benchmarks. Figure 10 shows the results normalized to the throughput of Journal on Ramdisk. NJS performs better than NJS-no-DC by 31.5%, 20.3% and 32.1% in Postmark, Fileserver and Varmail respectively. The results indicate that the function of delayed checkpointing 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.

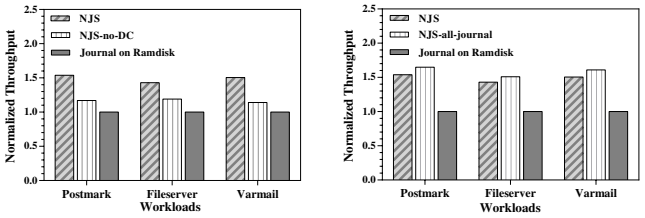


Fig. 10: Effect of the Delayed Checkpointing

Fig. 11: Effect of Eliminating Append-write to NVM

E. Effect of Eliminating Append-write to NVM

At last we 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 (NJS-all-journal) and Journal on Ramdisk under the aforesaid macro-benchmarks. Figure 11 shows the results normalized to the throughput of Journal on Ramdisk. In three workloads, NJS-all-journal outperforms NJS. But the improvements in Postmark, Filebench and Varmail are only 7.2%, 5.5% and 6.9%, 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. But as shown in Figure 9 (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.

V. Related Work

NVM-based file systems. BPFS [27] uses short-circuit shadow paging technique and 8-byte atomic write to provide consistency. SCMFS [9] 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 [28] 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 [18] is a lightweight POSIX file system designed for persistent memory, it bypasses the page cache and eliminates the copy overheads to improve performance. NOVA [8] is a

recently proposed NVM file system that adapts conventional log-structured file system techniques to guarantee strong consistency. 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.

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.* [29] proposed an NVM-based journaling mechanism SJM with write reduction. Kim *et al.* [30] 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 does not consider NVM delays checkpointing. Chen *et al.* [11] proposed a fine-grained metadata journaling technique on NVM, and this is the most related work 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.

VI. 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 minimize 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. Furthermore, we design a byte-level journal update scheme in which journal block can be updated in the byte-granularity based on the difference of the old and new versions of journal blocks so as to exploit the unique byte-accessibility characteristic of NVM. NJS also includes 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 NJS outperforms Ext4 with a ramdisk-based journaling device by 60.9% on average in different workloads.

Acknowledgment

This work was supported by the National High Technology Research and Development Program (863 Program) No.2015AA015301, NSFC No.61472153, No.61772222, No.61772212, No.61502191; the National Key Research and Development Program of China under Grant 2016YFB1000202; State Key Laboratory of Computer Architecture, No.CARCH201505; This work was also supported by Engineering Research Center of data storage systems and Technology, Ministry of Education, China.

References

[1] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems." in *Proc. ATC*, 2005.
[2] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory." in *Proc. 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 *Proc. ICCD*, 2015.
[4] J. Yue and Y. Zhu, "Accelerating write by exploiting PCM asymmetries," in *Proc. HPCA*, 2013.
[5] M. Mao, Y. Cao, S. Yu, and C. Chakrabarti, "Optimizing latency, energy, and reliability of 1t1r reram through appropriate voltage settings," in *Proc. ICCD*, 2015.
[6] Intel and Micron, "Intel and micron produce breakthrough memory technology," https://newsroom.intel.com/news-releases/, 2015.
[7] S. Bock, B. R. Childers, R. Melhem, and D. Mossé, "Concurrent migration of multiple pages in software-managed hybrid main memory," in *Proc. ICCD*, 2016.
[8] J. Xu and S. Swanson, "NOVA: a log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. FAST*, 2016.
[9] X. Wu and A. Reddy, "SCMFS: a file system for storage class memory," in *Proc. SC*, 2011.
[10] A. Hatzieleftheriou and S. V. Anastasiadis, "Okeanos: Wasteless journaling for fast and reliable multistream storage," in *Proc. ATC*, 2011.
[11] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue, "Fine-grained metadata journaling on NVM," in *Proc. MSST*, 2016.
[12] R. McDougall, "Filebench: Application level file system benchmark," 2014.
[13] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Consistency without ordering." in *Proc. FAST*, 2012.
[14] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 439–447, 2008.
[15] 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," in *Proc. FAST*, 2014.
[16] C. Youngdon, S. Ickhyun *et al.*, "A 20nm 1.8 v 8gb pram with 40mb/s program bandwidth," in *Proc. ISSCC*, 2012.
[17] F. Huang, D. Feng, Y. Hua, and W. Zhou, "A wear-leveling-aware counter mode for data encryption in non-volatile memories," in *Proc. DATE*, 2017.
[18] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proc. EuroSys*, 2014.
[19] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc. ASPLOS*, 2011.
[20] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Proc. ICCD*, 2014.
[21] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proc. MSST*, 2015.
[22] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proc. SOSP*, 2013.
[23] J. Chen, Q. Wei, C. Chen, and L. Wu, "FSMAC: A file system metadata accelerator with non-volatile memory," in *Proc. MSST*, 2013.
[24] G. Wu and X. He, "Delta-FTL: improving SSD lifetime via exploiting content locality," in *Proc. EuroSys*, 2012.
[25] W. D. Norcott and D. Capps, "Iozone filesystem benchmark," *URL: www. iozone. org*, vol. 55, 2003.
[26] N. Appliance, "Postmark: A new file system benchmark," 2004.
[27] 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 *Proc. SOSP*, 2009.
[28] E. Lee, S. Yoo, J.-E. Jang, and H. Bahn, "Shortcut-JFS: A write efficient journaling file system for phase change memory," in *Proc. MSST*, 2012.
[29] L. Zeng, B. Hou, D. Feng, and K. B. Kent, "SJM: an SCM-based journaling mechanism with write reduction for file systems," in *Proc. DISCS*, 2015.
[30] J. Kim, C. Min, and Y. Eom, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," *Consumer Electronics, IEEE Transactions on*, vol. 60, no. 2, pp. 217–224, 2014.