Fine-grained Metadata Journaling on NVM

Cheng Chen, Jun Yang, Qingsong Wei*, Chundong Wang, and Mingdi Xue
Email:{CHEN_Cheng, yangju, WEI_Qingsong, wangc, XUE_Mingdi}@dsi.a-star.edu.sg
Data Storage Institute, A-STAR, Singapore

***Abstract*—Journaling file systems have been widely used where data consistency must be assured. However, we observed that the overhead of journaling can cause up to 48.2% performance drop under certain kinds of workloads. On the other hand, the emerging high-performance, byte-addressable Non-volatile Memory (NVM) has the potential to minimize such overhead by being used as the journal device. The traditional journaling mechanism based on block devices is nevertheless unsuitable for NVM due to the *write amplification* of metadata journal we observed. In this paper, we propose a fine-grained metadata journal mechanism to fully utilize the low-latency byte-addressable NVM so that the overhead of journaling can be significantly reduced. Based on the observation that conventional block-based metadata journal contains up to 90% clean metadata that is unnecessary to be journalled, we design a fine-grained journal format for byte-addressable NVM which contains only modified metadata. Moreover, we redesign the process of transaction committing, checkpointing and recovery in journaling file systems utilizing the new journal format. Therefore, thanks to the reduced amount of ordered writes to NVM, the overhead of journaling can be reduced without compromising the file system consistency. Experimental results show that our NVM-based fine-grained metadata journaling is up to 15.8x faster than the traditional approach under FileBench workloads.**

I. Introduction

Journaling file system has been de facto building brick for data management systems to guarantee data consistency and recoverability for decades. By recording the changes of data to a "journal" before in-place updating the data, such file systems can be restored from failures without corrupting the data. Because of the assumption that the underlying storage devices are block devices such as magnetic disks (HDDs), the design and implementation of modern journaling file systems always use a whole disk block as the basic unit of the journal. This causes a severe *write amplification* problem for file system metadata journaling in which we observed that up to 90% of metadata is clean and unnecessary to be journalled. The main objective of this paper is to remove such amplification so that the overhead of journaling can be significantly reduced.

Recently, the next generation of Non-Volatile Memory (NVM) technologies has attracted more and more attentions from both industrial and academic researchers. New types of memory such as phase-change memory (PCM) [1], spin-transfer torque memory (STT-RAM) [2], Memristor [3] and 3D-XPoint [4] have been under active development in the past few years. They are persistent just like HDDs and flash memory (flash-based SSDs), and more importantly, compared to their predecessor, flash memory, the next generation of NVM has DRAM-like performance and byte-addressability. Such characteristics suggest the best way of using it be connecting it through memory bus as shown in Figure 1. However, due to its current price and capacity, NVM cannot be used as a standalone large-capacity memory device in the near future. Meanwhile, using high performance external journal device has been proven to be effective to accelerate the journaling file systems [5]. In this paper, we propose to use the next generation of NVM, which is accessed through memory bus instead of I/O bus shown in Figure 1, as the external journal device for journaling file systems.

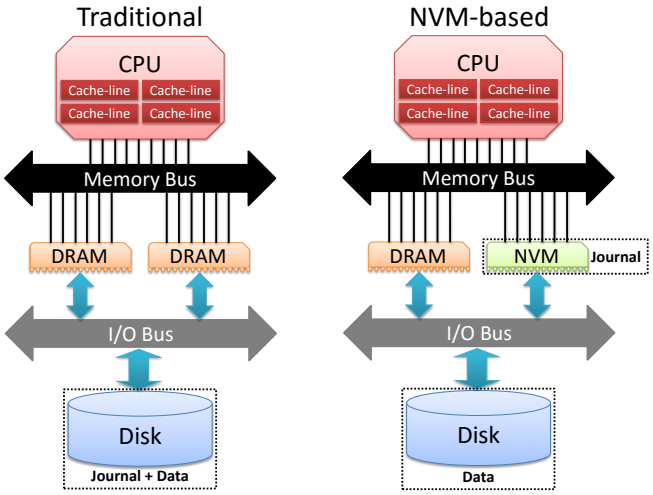


Fig. 1: Comparison of System Architectures with and without NVM

Current block-based journaling mechanism, even the state-of-art techniques that using flash-based SSDs as the external journal device, cannot fully utilize the byte-addressability of NVM due to the block interface constraints. Furthermore, to guarantee the data consistency, journal must be written before the data can be in-place updated. However, keeping the writes to NVM connecting through memory bus in a certain order has to involve a special set of CPU instructions such as memory fence (e.g., `MFENCE`) and cache-line flush (e.g., `CLFLUSH`) which have been proven to be extremely expensive [6], [7]. Since the overhead of these instructions is proportional to the amount of writes, the negative impact of the aforesaid write amplification in conventional journaling file systems gets worse with NVM being the journal device.

In this paper, we propose a fine-grained metadata journal mechanism for using NVM as the external journal device. In

*Corresponding author

particular, we develop a new journal format on NVM based on individual metadata instead of the whole on-disk metadata block so that the aforesaid write amplification is eliminated. The in-NVM journal is guaranteed to be consistently updated by ordered memory writes through CPU instructions of cache-line flushes and memory fences. Moreover, based on the new journal format, we redesign the process of transaction committing, checkpointing and recovery of the journaling file systems to achieve better performance by exploiting the byte-addressability and superior performance of NVM. Specifically, only modified metadata itself (e.g. inode in Ext4) instead of the entire metadata block (e.g. inode metadata blocks) is written in the journal when committing a transaction. On the other hand, the in-NVM journal is scanned during recovery and the modified inodes are merged to their corresponding on-disk metadata block if necessary.

Our contributions can be summarized as follows:

1) We quantify the overhead of existing block-based journaling, and present two insightful observations: (1) conventional block-based journaling on NVM severely underutilizes NVM (only around 10% faster than that journaling on HDD while NVM is orders of magnitude faster than HDD); (2) the write amplification of metadata journal (up to 90% of the metadata journal is for clean metadata which is unnecessary but inevitable given the block interface constraints on HDD).
2) Based on the observations, we present our **fine-grained metadata journaling**, which (1) utilizes byte-addressable NVM to minimize the aforesaid write amplification by adopting a new journal format based on individual metadata instead of metadata block, (2) redesigns the process of transaction committing, checkpointing and recovery based on the new journal format to reduce the amount of expensive ordered writes to NVM without compromising the file system consistency.
3) To validate fine-grained metadata journaling, we implement a journaling file system prototype based on Ext4 by modifying the JBD2[8] kernel module. We have evaluated our prototype on the real NVDIMM platform [9] under various workloads. The experimental results show that the performance of Ext4 can be improved by up to 15.8x with our fine-grained NVM-based metadata journaling compared to the conventional block-based approach under FileBench [10] workloads.

The rest of the paper is organized as follows: Section II discusses the background and related work of NVM and journaling file systems. Section III presents the detailed design and implementation of our journaling mechanism. The experimental results are shown and explained in Section IV. Finally, Section V concludes the paper.

II. Related Work and Motivation

A. Journaling File System

Many modern widely-used file systems can be categorized as journaling file systems, such as Ext3/Ext4 [11], [8], NTFS [12], XFS [13]. A typical journaling file system reserves a journal area on persistent storage devices for storing modified data temporarily before updating the data in-place. Specifically, when the data in DRAM buffer is dirty and needs to be written back to the persistent storage, e.g., HDD, directly updating it in-place on HDD may corrupt the data upon failure during the overwrite process. To keep the data consistency, journal file systems *commit* the changes of the data (usually the latest copy of the data) to the journal area on HDD. Later, the data is updated through *checkpointing* periodically. If a failure happens during a commit, the data on HDD is not affected. And even if the failure that occurs during checkpointing corrupts the data, it can still be recovered from the journal when the system restarts. However, such doubled writes for both journal and data itself are quite expensive. Typically, there are three modes of journaling, *writeback mode* only writes metadata changes to the journal, *ordered mode* provides higher consistency by writes metadata journal strictly after data (file content) is written (default) and *journaling mode* writes journal for both metadata and data. Most of the journaling file systems perfer journaling on the critical metadata by default so that the integrity of the whole file system can be maintained with reduced journaling overhead.

Although the data consistency can be guaranteed by using journal, a large portion of metadata when being committed to the journal is clean and unnecessary to be written. This is because a typical journaling file system always assumes the persistent storage to be a block device such as HDD. Therefore, the basic unit in the journal is the same as the disk block. However, since the size of an individual metadata structure is much smaller than the block size, metadata is usually stored on HDD in batch. For example, the size of the metadata in Ext4, *inode*, is 256 bytes, while a typcial size of a disk block is 4K bytes which means one metadata block stores 16 *inodes*. In consequence, when even one of them is needed to be written back to disk, all of them have to be written to the journal. We call this the **journal write amplification** which is main focus of this paper.

TABLE I: Characteristics of Different Types of Memory

Category	**Read Latency** (*ns*)	**Write Latency** (*ns*)	**Endurance** (*# of writes per bit*)
SRAM	2-3	2-3	∞
DRAM	15	15	10^{18}
STT-RAM	5-30	10-100	10^{15}
PCM	50-70	150-220	10^{8}-10^{12}
Flash	25,000	200,000-500,000	10^{5}
HDD	3,000,000	3,000,000	∞

B. Non-volatile Memory

Computer memory has been evolving rapidly in recent years. A new category of memory, NVM, has attracted more and more attention in both academia and industry. Early work [14], [15], [16], [17], [18], [19], [20], [21], [22] focused on flash memory to address the asymmetric I/O performance and write endurance issue. As shown in Table I, flash is faster

than a HDD but is still unsuitable to replace DRAM due to the much higher latency and limited endurance. Recent work [23], [24], [25], [26], [27] has focused on the next generation NVM, such as PCM [1], [28], STT-RAM [2] and 3D-XPoint [4], which (i) is byte addressable, (ii) has DRAM-like performance, and (iii) provides better endurance than flash. PCM is several times slower than DRAM and its write endurance is limited to as few as 10^8 times. However, PCM has larger density than DRAM and shows a promising potential for increasing the capacity of main memory. Although wear-leveling is necessary for PCM, it can be done by memory controller [29], [30]. STT-RAM has the advantages of lower power consumption over DRAM, unlimited write cycles over PCM, and lower read/write latency than PCM. Everspin has announced its commercial 64Mb STT-RAM chip with DDR3 interface [31]. In the rest of this paper, **NVM** is referred to the next generation of NVM with persistency and DRAM-like performance.

Due to the price and prematurity of NVM, mass production with large capacity is still impractical today. As an alternative, NVDIMM [32], which is commercially available [9], provides persistency and DRAM-like performance. NVDIMM is a combination of DRAM and NAND flash. During normal operations, NVDIMM is working as DRAM while flash is invisible to the host. However, upon power failure, NVDIMM saves all the data from DRAM to flash by using super-capacitor to make the data persistent. Since this process is transparent to other parts of the system, NVDIMM can be treated as NVM. In this paper, our fine-grained metadata journaling is implemented and evaluated on a NVDIMM platform.

C. Data Consistency with Ordered Writes in NVM

Data consistency on persistent storage guarantees that stored data can survive system failure. Since data is meaningful only if it is organized in a certain format, writing data consistently means that a system failure must not cause data loss or corruption. However, the atomicity of memory writes to NVM can only be supported with a very small granularity or no more than the memory bus width (8 bytes for 64-bit CPUs). As illustrated in the previous work [33], [34], [32], updating data larger than 8 bytes must adopt logging or copy-on-write (CoW) approaches which make data recoverable by writing a copy elsewhere before updating the data itself. The implementation of these approaches in NVM requires the memory writes to be in a certain order, e.g., the logs or backup copy of data must be written completely in NVM before updating the data itself.

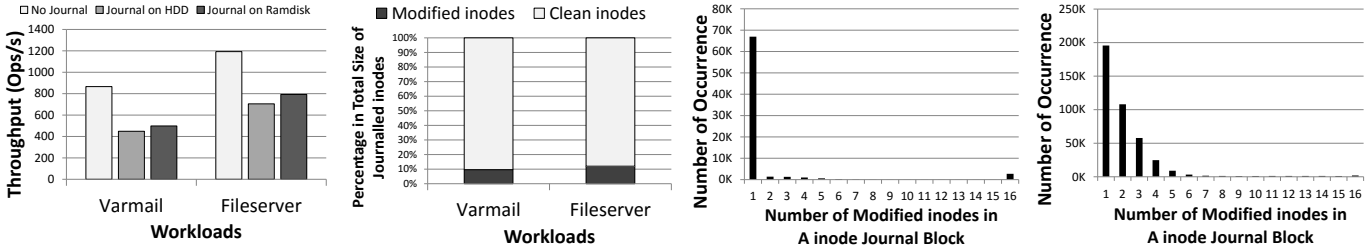
Unfortunately, ordering memory writes must involve expensive CPU instructions such as memory fence (e.g., `MFENCE`) and cache-line flush (e.g., `CLFLUSH`). For example, prior work [6] showed that directly using them in B^+-tree reduces the performance by up to 16x. The most effective way to minimize such overhead is reducing the amount of memory writes that require such ordering. However, due to the aforesaid *write amplification* problem, traditional journaling file systems further worsen the performance due to the ordered writes of unnecessary metadata in the journal. To fully utilize the superior performance of NVM, our fine-grained journaling is designed to write only necessary metadata to the journal by exploiting the byte-addressability of NVM so that the amount of ordered memory writes is minimized.

D. Comparison with Other Related Work

Journaling file systems has been extensively researched in past twenty years [11], [8], [12], [13], [35], [36], [37], [38], [39], [40]. Prabhakaran et. al. [41] proposed a model to investigate how journaling file systems should work under different journaling modes upon disk failures. Fryer et. al. proposed Recon [42] to verify the consistency of a file system at runtime. It not only protects the file system metadata from dangerous operations, but also takes advantage of the crash consistency provided by modern file systems using transactional updates. These work focus on the analysis and measurement of the journaling techniques that is required to have a deeper understanding of journaling file systems. However, they did not consider the influence of the new storage media other than HDDs can bring to the design of journaling file systems.

Motivated by the mismatch between block interface constraints and byte-addressability of NVM, a lot of research works have proposed NVM-optimized file systems [43], [44], [45], [46]. Condit et. al. proposed BPFS which is based on CoW approaches and optimized for small updates using CPU intrinsic 8-byte atomic writes [34]. SCMFS [47] was proposed to optimize the file system for NVM resides on the memory bus which can be accessed directly from CPU through the same address space of virtual memory system. Shortcut-JFS [48] assumes a PCM-only standalone storage device. Utilizing the byte-addressability of PCM, differential logging and in-place checkpointing techniques are proposed to reduce the journaling overhead for PCM devices. Recently, Dulloor et. al. proposed PMFS [49] for persistent memory to be used as normal storage device through existing POSIX style file system APIs. We are different from the above work in the way that we propose to use NVM only as external journaling device instead of the storage device for the entire file system, which is impractical at the current stage of NVM products. Most recently, Zeng et. al. [50] proposed SJM, a NVM-based journaling mechanism. However, SJM targeted the full journaling mode which still involves block-based journaling, and did not consider the cost of cache-line flush and memory fence when writing to NVM through memory bus.

The most related work to ours is UBJ, proposed by Lee et. al. [51] to union the buffer cache and the journal area so that modified data can be buffered in a persistent buffer to enable in-place commit. However, UBJ still involves block-based journaling which is inefficient both temporally and spatially. Furthermore, because UBJ has to maintain multiple versions of inode "blocks" in NVM, the aforesaid write amplification becomes more severe when there are only few inodes are modified in every version of those inode blocks. In contract, our design completely eliminates the block-based



(a) Performance with and without Journaling (b) Modified/clean Ratio of inodes in Journal (c) Modified inodes Count Distribution under Varmail Workloads (d) Modified inodes Count Distribution under FileServer Workloads

Fig. 2: Write Amplification in Traditional Journaling File Systems

journaling, focus on metadata journaling with minimized write amplification.

E. Motivation

To quantify the overhead of traditional journaling file systems, we measured the performance of a classic journaling file system, Ext4, with and without journaling under different workloads using FileBench [10], a widely used benchmark for file systems. As shown in Figure 2(a), the performance of Ext4 with *ordered* mode (default mode in Ext4) journaling on HDD is 48.2% and 40.9% slower than its non-journaling version under Varmail and FileServer workloads, respectively. Moreover, such large overhead of journaling does not vanish even if we switch the journal device from slow HDD to fast NVM. Specifically, the performance drop is still 42.5% and 33.6% when switching the journal device to NVM with traditional journaling mechanism. Although NVM can be more than five orders of magnitude faster than HDD in terms of latency, Journaling on NVM is only 11.1% and 12.4% faster than that on HDD under Varmail and FileServer workloads, respectively, which suggests NVM is severely underutilized.

To find out the reason of such small performance improvement when switching the journal device from HDD to NVM, we further studied the detailed content of the journal in the Ext4 file system under different workloads. Note that when one or some of the inodes in an inode block are updated, the entire inode block has to be written to the journal. Figure 2(b) shows the dirty/clean ratio of the inodes in all the inode blocks being written to the journal under two different workloads. We observed that up to 90% and 88% of the inodes are clean and unnecessary to be journaled under Varmail and FileServer workloads, respectively. In fact, as shown in Figure 2(c) and 2(d), up to 90% and 47.6% of the inode journal blocks only have one modified inode, about 96.3% and 97.5% of them have modified inodes less than half in one inode journal block under Varmail and FileServer workloads, respectively. Moreover, such **write amplification** of metadata journaling becomes a even bigger problem for NVM journal device because the expensive cost of cache-line flush and memory fence required to implement ordered writes to NVM is proportional to the amount of written data.

Fortunately, such **write amplification** problem is mainly caused by the block interface constraints of HDD. When byte-addressable NVM is being used as the journal device, we have the opportunity to eliminate the write amplification and optimize the metadata journaling on NVM. Motivated by this, we propose a fine-grained journaling mechanism to minimize the journaling overhead by exploiting the byte-addressability of NVM.

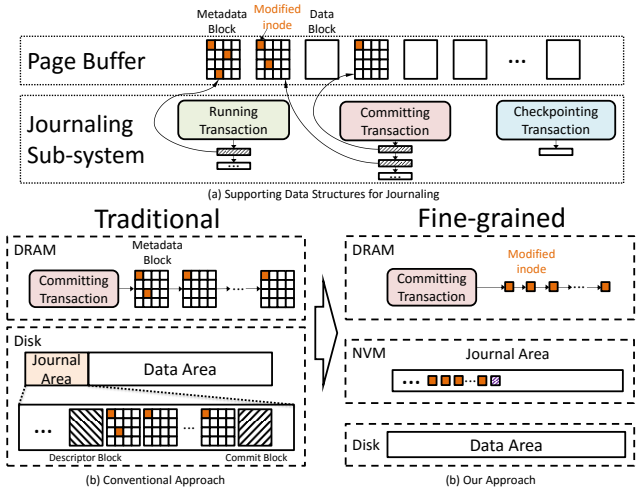


Fig. 3: Comparison of Traditional and Fine-grained Journaling on NVM

III. Design and Implementation

In this section, we present the detailed design and implementation of our fine-grained journaling on NVM. For better understanding, we take *Ext4* as an example of a traditional journaling file system. We use an *inode* as the representative for small metadata which is usually one or two orders of magnitude smaller than the disk I/O (block) size. In other words, our design is not only for inodes, it can be applied to any relatively small metadata such as *dentry*. Since we only focus on the metadata journaling, we consider the *ordered* mode of journaling as default.

A. Overview

In a conventional journaling file system such as Ext4, as shown in Figure 3, modified inodes are firstly in-place updated in the inode block in the page buffer. All the modified inode

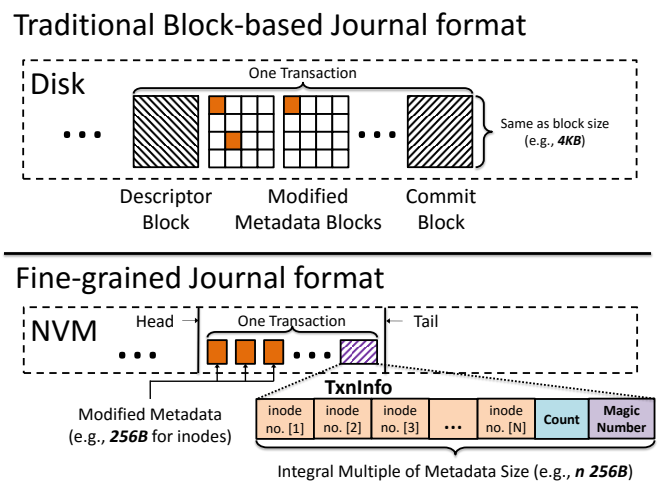


Fig. 4: Design of Fine-grained Journal Format

blocks are linked by *Running Transaction* list in DRAM page buffer. When transaction *commits*, the *Commiting Transaction* takes over the modified inode block list from *Running Transaction*. By traversing that list, the corresponding blocks are written to the journal area on the persistent storage. To distinguish the adjacent transactions, the inode journal blocks are surrounded by two special journal blocks, a *Descriptor Block* at the beginning and a *Commit Block* (or *Revoke Block* if the transaction is aborted) at the end.

Different from traditional journaling file systems which write every modified inode block to the journal, our fine-grained journaling only writes modified inodes to NVM. Specifically, instead of linking all the modified inode blocks, only the modified inodes are linked together. Moreover, when they are flushed to NVM, ordered memory writes through memory fences and cache-line flushes are used to guarantee the consistency. We further reduce the amount of writes by eliminating the *Descriptor Block* and *Commit Block* in the journal. Instead, we use a cache-friendly data structure called **TxnInfo** as the boundary of two adjacent transactions which is discussed in detail in Section III-B.

At the same time, the *checkpointing* process is not much changed since it essentially flushes all the modified blocks in the page buffer to the disk without touching the journal (except for the indication of a successful checkpointing). On the other hand, the recovery process is redesigned to utilize the fine-grained journal in NVM to ensure a consistent state for the entire file system. The modified workflow of committing, checkpointing and recovery is discussed in detailed in Section III-C.

B. Fine-grained Journal Format

As shown in Figure 4, the design of the fine-grained journal format differs from the traditional one in the following aspects:

- The basic unit of the journal in traditional journaling file system is a block which is, for example, 4KB in Ext4. Such design is to facilitate and maximize the disk I/O efficiency given the block interface constraints. However, when the byte-addressable NVM is in use, it is no longer efficient to write the entire inode block when there are only a few inodes in it are modified. Therefore, **we use the inode as the unit for the journal.**
- Another drawback of traditional journal format is the *Descriptor Block* and *Commit Block* (or *Revoke Block*) to indicate the beginning and end of a transaction, respectively. However, these two blocks occupy 8KB space in total while a typical size for inode is only 256B. To reduce such overhead, in our fine-grained journaling, **we design a new cache-friendly data structure called *TxnInfo* as the boundary of two adjacent transactions.**
- Traditional block-based Journal is guaranteed to be written to disk by calling `submit_bio` through legacy block I/O interface. However, when the journal device turns to be NVM which is connected through memory bus, to ensure the journal is written to NVM, the CPU instructions of flushing the corresponding CPU-caches and issuing a memory fence must be applied after the journal is `memcpy`ed from DRAM to NVM.

The **TxnInfo** (1) describes the inodes in the transaction and (2) is used to locates the boundary of each transaction to facilitate the recovery process and guarantee its correctness. Since the original inode structure does not contain any information of the inode number, **TxnInfo** includes this information so that the inodes in the journal can be located individually. On the other hand, since **TxnInfo** is the only data structure that works as the boundary of two adjacent transactions, it contains the information of locating the start and end position of the journal for each transaction. Therefore, we add two more fields, *Count* and *Magic Number*. The *Magic Number* is predefined to help the journal scan during recovery to identify the location of every **TxnInfo**. The *Count* is the number of inodes included in the transaction.

Theoretically, the length of **TxnInfo** can be elastic, but for performance consideration, we design its size to be an integral multiple of the inode size so that the journal for each transaction can be naturally aligned to CPU-cache which results in better performance for cache-line flushing. As the maximum number of inodes in each transaction is determined by the length of **TxnInfo**, it can be used to control the default *commit* frequency which can be used to optimize the overall performance for a certain workload, discussed in detail in Section IV-C.

The journal area in NVM works similar to a ring buffer whose access is controlled by the *head* and *tail* pointer. Specifically, the normal procedure of writing a journal is (1) the journal is `memcpy`ed from DRAM to NVM, (2) flush the corresponding cache-lines and issue a memory barrier, (3) use a 8-byte atomic write to update the *tail* pointer, flush its cache-line and issue a memory barrier. The detailed usage of the journal for file system transaction committing, checkpointing and recovery is discussed in the next section.

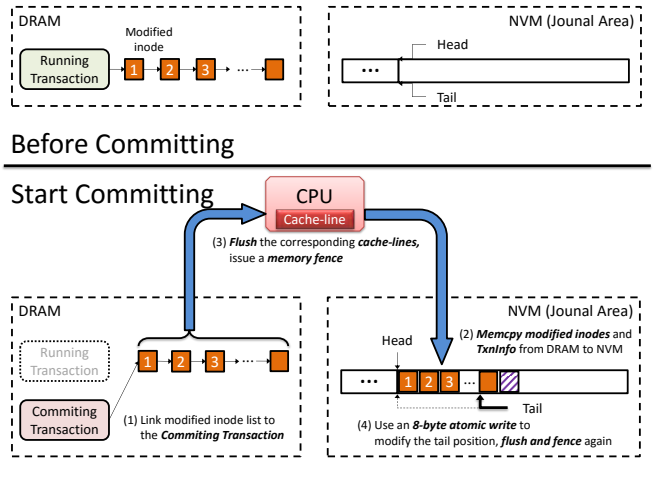


Fig. 5: Transaction Commit Workflow

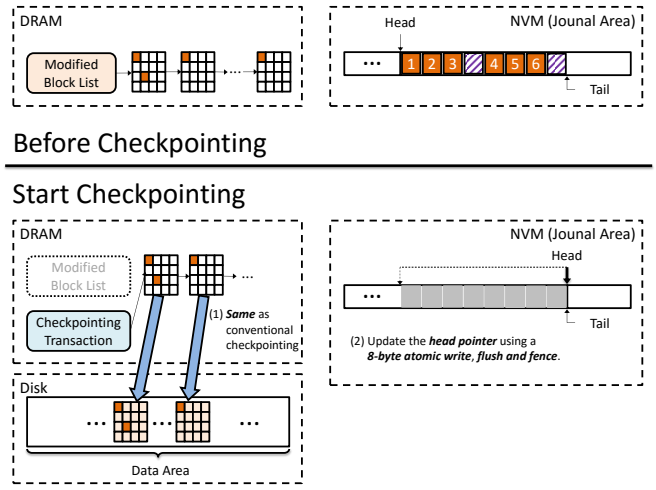


Fig. 6: Checkpointing Workflow

C. Workflow of Fine-grained Journaling

In this subsection, we present the modified procedure of committing, checkpointing and recovery in the journaling file system. We focus on how the fine-grained journal in NVM is utilized to guarantee the file system consistency.

1) Transaction Commit: In a traditional journaling file system, transaction *committing* is triggered either by a fixed timer (typically five seconds in Ext4) or a certain amount of inode blocks (around 8,000) has been modified. Our fine-grained journaling, on one hand, commits a transaction based on a pre-defined timer in a similar way to a traditional journaling file system. On the other hand, the commit frequency can also be controlled by the maximum size of **TxnInfo** because the number of inodes a TxnInfo can hold is limited. For example, for a 256-byte TxnInfo, if the length of *inode number*, *Count* and *Magic Number* are all 8 bytes, the maximum number of inodes is 30 which means *committing* must be performed when 30 different inodes are modified. By default, the maximum size of a TxnInfo is 8KB, which limits the maximum number of modified inodes in a transaction to 8,190. We show how this parameter affects the overall performance in Section IV-C.

Before committing, all the modified inodes are linked in the *Running Transaction* single-linked list as illustrated in the upper part of Figure 5. A committing process is executed according to the following steps as shown in the bottom part of Figure 5:

1) When the committing starts, all the modified inodes are linked to *Committing Transaction* list. At the same time, *Running Transaction* is initialized to accept new modified inodes.
2) By traversing the *Committing Transaction* list, all the modified inodes are retrieved one by one, then `memcpy`ed to the journal area in NVM starting from the *tail*. The TxnInfo is also constructed in DRAM when traversing the linked list. In particular, the *inode number* of each modified inodes is appended. After the last modified inode is `memcpy`ed, the actual length of TxnInfo can be calculated according to the number of modified inodes in the transaction. Before appending the final two fields, the *Count* and *Magic Number*, padding is added after the last modified inode to make sure the total length of the TxnInfo is integral multiple of the inode size.
3) To make sure the journal is physically written to NVM, the corresponding cache-lines are flushed and a memory fence is issued.
4) Finally, the *tail* is updated by an 8-byte atomic write followed by the cache-line flush and memory fence to indicate the transaction commits successfully.

The above steps guarantees that any failure which happens in the middle of the committing process never sabotage the file system consistency. In other words, the modification made by the committed transactions can survive any failure. The key of achieving data consistency is the *tail*, which controls the visibility of the written journal because the backwards scan always starts from the *tail* during recovery (discussed in Section III-C3). Consequently, all the written journal during committing is invisible to the system until the *tail* is updated, which itself is guaranteed to be consistent using an CPU-intrinsic 8-byte atomic write followed by the cache-line flush and memory fence.

By adopting a fine-grained journal format and replacing *Descriptor Block, Commit Block (Revoke Block)* with dynamic-sized TxnInfo, the total amount of journal writes when committing file system transactions can be significantly reduced (up to 99% as shown in Section IV).

2) Checkpointing: Checkpointing in conventional journaling file systems is performed periodically to keep the on-disk data up-to-date. Checkpointing is used to prevent the journal from growing too long. Since the file system is frozen during checkpointing, longer journal results in longer checkpointing and recovery which can cause unacceptably high latency.

When we use NVM as the journal device, performance-wise, longer time interval of checkpointing is preferred in our

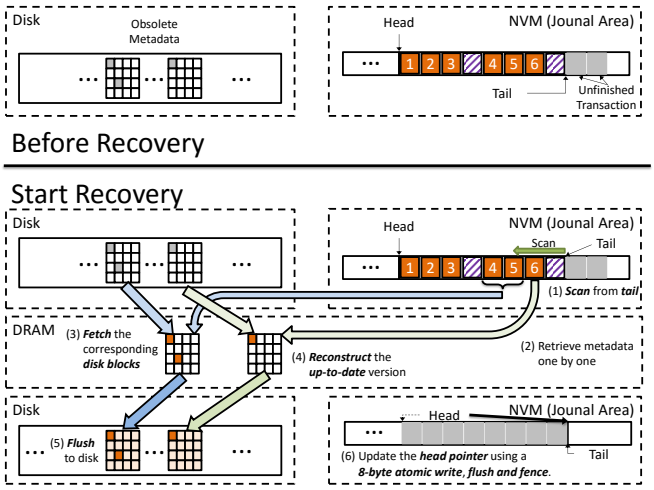


Fig. 7: Recovery Workflow

fine-grained journaling because more disk I/O for checkpointing can be saved. At the same time, the file system frozen time during checkpointing and recovery time upon failure must be controlled under an accepted range similar to conventional journaling file systems. By default, checkpointing is either triggered by a 10-minute timer or the utilization of the journal area in NVM being over 50%.

As illustrated at the upper part of Figure 6, similar to conventional checkpointing, a single-linked list which links all the modified inode blocks is maintained in DRAM. Before checkpointing, the journal area on NVM contains the modified inodes from multiple transactions between the *head* and *tail*. A checkpointing process is executed according to the following steps:

1) Similar to conventional journaling file systems, when a checkpointing starts, the *Checkpointing Transaction* takes over the modified inode block list and starts to write them to disk one by one.
2) After all the modified inode blocks are successfully flushed to disk, the *head* in the NVM journal is updated to point to the *tail* pointer by a 8-byte atomic write followed by a cache-line flush and a memory fence. Making the *head* equal to the *tail* indicates the checkpointing is done successfully and all the journal can be discarded to free the journal area.

Note that most of the checkpointing process for fine-grained journaling is the same as that in conventional journaling file systems except the 8-byte atomic write for updating the *head* pointer. This guarantees that any failure during the checkpointing process can be recovered by redoing the checkpointing since the *head* and *tail* used to identify the journal on NVM are always updated consistently.

3) Recovery: During recovery in a conventional journaling file system, the up-to-date inode blocks can be directly retrieved from the journal. However, our fine-grained journaling only writes modified inodes in the journal, the recovery process is quite different from that of conventional journaling file systems. During recovery in the fine-grained journaling, the up-to-date inode blocks must be reconstructed by merging the obsolete on-disk version with the up-to-date inodes from the journal on NVM.

As illustrated at the upper part of Figure 7, before recovery from a failure, some of the inode blocks on disk are obsolete while the journal area in NVM contains up-to-date inodes from multiple committed transactions before the failure happens. Note that the failure may happen during a transaction commit but thanks to the 8-byte atomic write for updating the *tail*, we can simply discard all the journal after the *tail* as it belongs to an unfinished transaction. A recovery process is executed according to the following steps:

1) Do a backward-scan in the NVM journal from the tail.
2) Retrieve the inodes in the journal one by one using the *Magic Number* and *Count* to identify each transaction and its corresponding inodes.
3) For each inode found in the journal, retrieve the corresponding (obsolete) inode block from the disk to DRAM using the *inode number* embedded in the TxnInfo.
4) Apply the up-to-date inode data in the journal to the obsolete inode block and reconstruct the up-to-date inode blocks in DRAM. Note that different inodes can reside in the same inode block, but for journal of the same inode, only the latest version is kept during the reconstruction.
5) When all the up-to-date inodes in the journal are merged to their inode blocks, flush all the up-to-date inode blocks to disk in batch.
6) After all the inode blocks are flushed, the *head* pointer for NVM journal is updated to point to the *tail* pointer using an 8-byte atomic write followed by a cacheline flush and memory fence. This final step is the same as the last step of checkpointing to indicate the journal space can be reused. In fact, after recovery, all the on-disk inode blocks are up-to-date just like those after checkpointing.

Note that the *head* and *tail* pointers are not modified until the last step of the recovery. Therefore, any failure during the recovery process does not affect the consistency of the entire file system because the recovery can always be redone as long as the journal can be retrieved correctly. If the file system is unmounted normally, no work needs to be done during recovery because a normal shutdown of the file system does a checkpoint to make sure all the inode blocks are up-to-date and empty the journal area.

IV. Evaluation

In this section, we present the experimental results of our fine-grained metadata journaling against the conventional block-based journaling under various workloads. We implement the prototype based on JBD2 in Ext4 [8] and measure the overall performance of journaling file system under four different kinds of workloads. We also quantify the reduction of the total amount of journal writes to explain how the write amplification problem is addressed. Last but not least, we

Fig. 8: NVM Space Layout

analyze the impact of the maximum size of TxnInfo on the frequency of transaction commit and show how to set the maximum size of TxnInfo to achieve the best performance.

A. Experimental Setup

1) Platform: All of our experiments are conducted on a Linux server (Kernel version 2.6.34-11) with an Intel Xeon E5-2650 2.4GHz CPU (512KB/2MB/20MB L1/L2/L3 cache), 4GB DRAM and 4GB NVDIMM [9] which has practically the same read/write latency as DRAM. To make use of NVDIMM as a persistent memory device, we modify the memory management of Linux kernel to add new functions (e.g., `malloc_NVDIMM`) to directly allocate memory space from NVDIMM.

2) Prototype and Implementation Effort: As shown in Figure 8, The NVDIMM space used as the journal area is guaranteed to be mapped to a continuous (virtual) memory space. The reserved space stores (1) the *start address* and *total size* to locate the boundary of journal space, and (2) the *head* and *tail* addresses for the current status of journal. The *head* or *tail* address is actually the memory offset to the start address of the mapped memory space. Therefore, even if the mapping is changed after reboot, the *head* and *tail* can always be located using the offset so the whole file system is practically recoverable after power down.

Implementation of Fine-grained metadata journaling involves modifications in file systems, memory management, and JBD2 journaling module. We choose Ext4 file system with kernel version 2.6.34-11 to implement our prototype (denoted as **Find-grained on NVM**). We modify function `ext4_mark_inode_dirty()` to link all modified inodes together. Meanwhile, we delete all the block-based journal writes to disk in function `jbd2_journal_commit_transaction()`, and replace them with fine-grained metadata journal writes to NVM. Checkpoint and recovery procedures are also modified accordingly. For a fair comparison, we also implement an NVM-based JBD2 (denoted as **JBD2 on NVM**) which writes block-based journal to NVM through `memcpy` with `CLFLUSH` and `MFENCE` instead of block I/O to disk. For a fair comparison, in **JBD2 on HDD** strategy, we also two disks for data and journaling separately.

3) Workloads: We use two widely used benchmark tools, FileBench [10] and Postmark [52] to evaluate the performance of fine-grained journaling against traditional block-based journaling.

1) **Varmail (FileBench)**
   Varmail workload in FileBench emulates I/O activity of a mail server that stores each e-mail in a separate file. The workload consists of a multi-threaded set of create-append-sync, read-append-sync, read and delete operations in a single directory on a large number of small files. The default setting we use is 960 thousand files with 12KB each (total size is around 12GB).
2) **FileServer (FileBench)**
   FileServer workload in FileBench emulates I/O activity of a file server. This workload performs a sequence of creates, deletes, appends, reads, writes and attribute operations on a directory tree. The default setting we use is 800 thousand files with 16KB each (total size is around 12GB).
3) **FileMicro_writefsync (FileBench)**
   This is a micro-benchmark which emulates I/O activity of a database logging system. It appends to a big file with a certain size per request and performs *fsync* every pre-defined number of requests. By default, the original file for appending is 1GB and *fsync* is performed every 32 appends.
4) **Postmark**
   PostMark is a single-threaded benchmark which emulates synthetic combined workload of email, usenet, and web-based commerce transactions. The workload includes a mix of data and metadata-intensive operations. By default, we use write-only workloads with 50%/50% create/delete ratio.

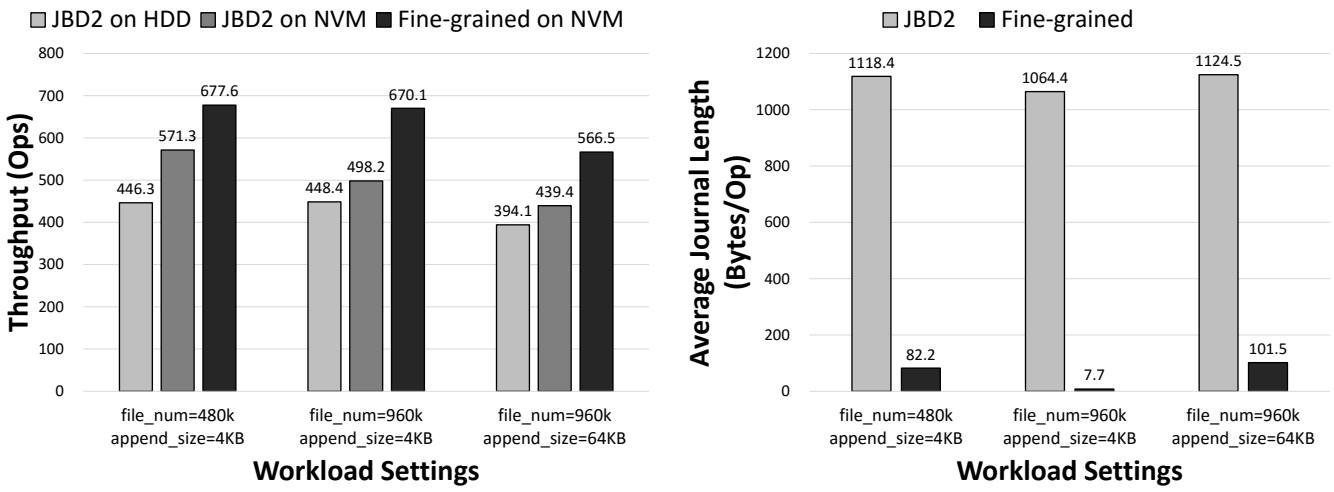
The default mode of journaling is *ordered* because we mainly optimize for the the performance of metadata journaling. Note that all results shown in this section are produced by running application on NVDIMM server instead of simulation. All the numbers shown are the average of five independent runs.

B. Performance of Fine-grained Journaling

1) Varmail Performance: Figure 9(a) shows the file system performance with three journaling strategies under Varmail workloads. With different workload settings, our fine-grained journaling on NVM is consistently better than JBD2 on either disk or NVM, having up to 51.8% and 34.5% higher throughput, respectively. Note that the higher performance improvement of fine-grained journaling over JBD2 on NVM is obtained with larger number of files and smaller append size which is the most metadata write-intensive workload.

To further investigate the performance gain with fine-grained journaling, we quantify the reduction of journal writes by measuring the average length of journal each operation writes (total length of journal divided by the total number operations). As shown in Figure 9(b), compared to the block-based journaling, fine-grained journaling saves 92.6%, 99.3% and 91.0% journal writes with different workload settings. The workload setting that leads to the highest reduction of journal writes is identical to that with highest performance improvement in Figure 9(a).

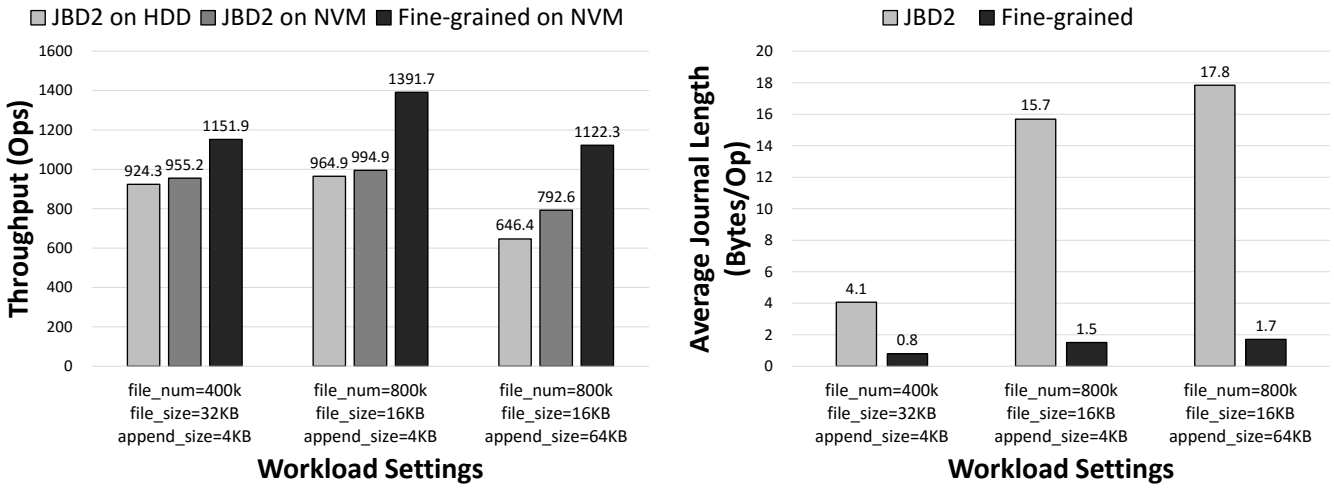
2) FileServer Performance: Figure 10(a) shows the file system performance with three journaling strategies under FileServer workloads. With different workload settings, our



(a) Throughput

(b) Journal Writes

Fig. 9: Performance under Varmail Workloads



(a) Throughput

(b) Journal Writes

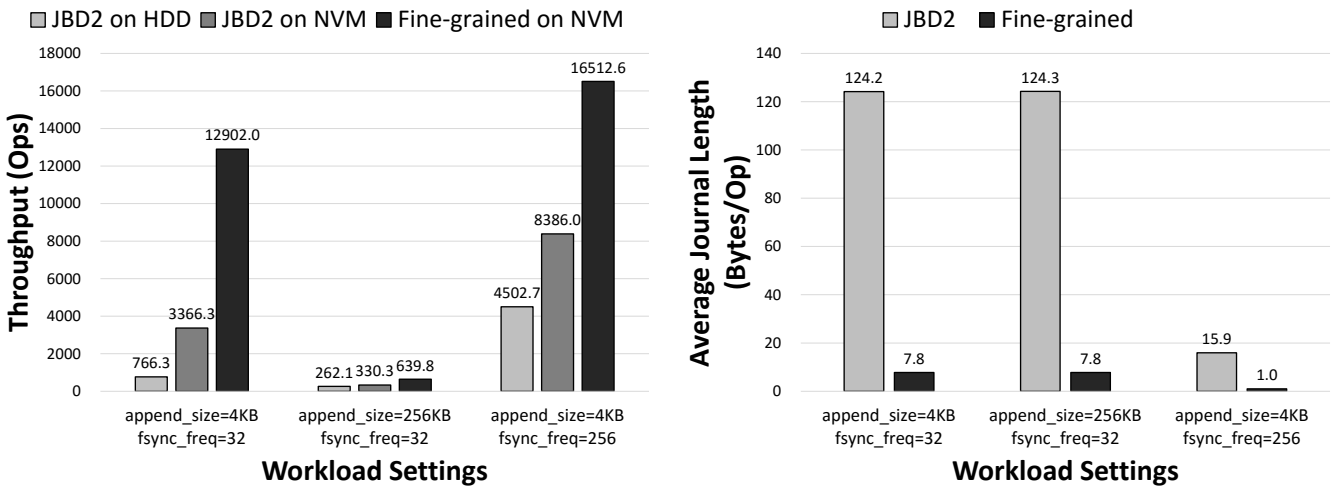
Fig. 10: Performance under FileServer Workloads

fine-grained journaling on NVM performs up to 73.6% and 41.6% better than JBD2 on disk and NVM, respectively.

Compared to Varmail, FileServer is less metadata-intensive because it does not contain *fsync* operations. As shown in Figure 10(b), compared to JBD2, fine-grained journaling reduces the average journal length per operation up to 90.4%. Similar to Varmail, the workload settings for the highest reduction of journal writes is identical to that with the highest performance improvement.

3) FileMicro_writefsync Performance: Figure 11(a) shows the file system performance under FileMicro_writefsync workload with different journaling mechanisms. The performance improvement with fine-grained journaling on NVM is much higher than that under previous two workloads because the transaction commits more frequently under FileMicro_writefsync workloads. Remember that the longer time interval between two consecutive commits leads to higher chance to save the multiple I/O on the same metadata block for block-based journaling. Therefore, the performance improvement of fine-grained journaling over block-based journaling can be up to 15.8x and 2.8x under frequent-commit workload settings (*fsync* every 32 4KB-appends). On the other hand, the performance improvement drops when the append size increases because the data I/O to disk becomes dominant.

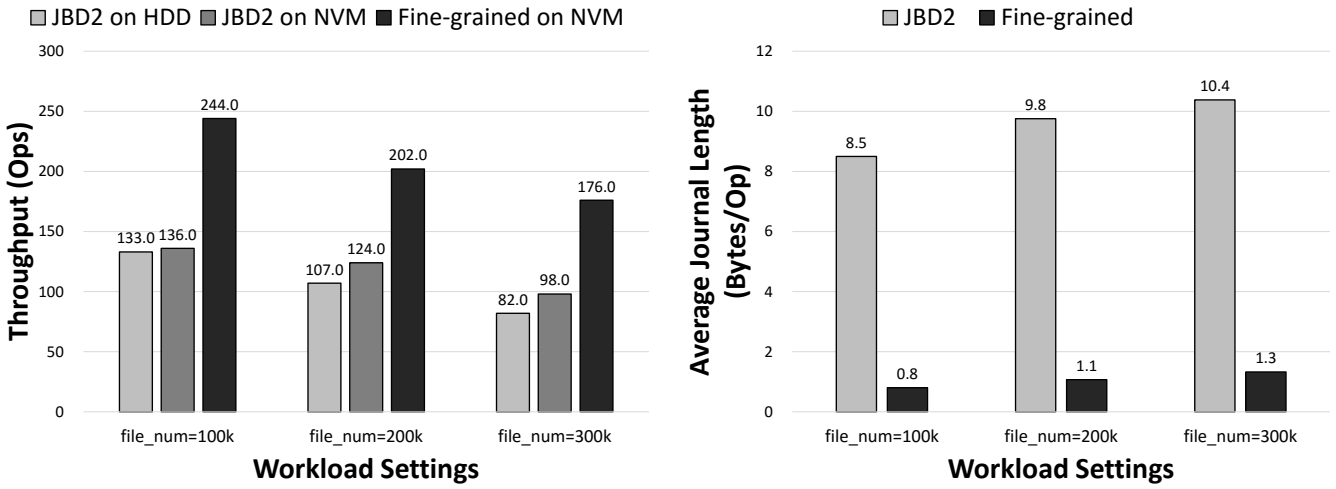
Moreover, as shown in Figure 11(b), the reduction of journal writes is around 93.7%, almost the same under different workload settings. This is because there is only one file, which means the amount of journal writes is fixed, i.e., one block for blocked-based JBD2, one metadata (inode) for fine-grained journaling, which is unrelated to the *fsync* frequency.



(a) Throughput

(b) Journal Writes

Fig. 11: Performance under FileMicro_writefsync Workloads



(a) Throughput

(b) Journal Writes

Fig. 12: Performance under Postmark Workloads

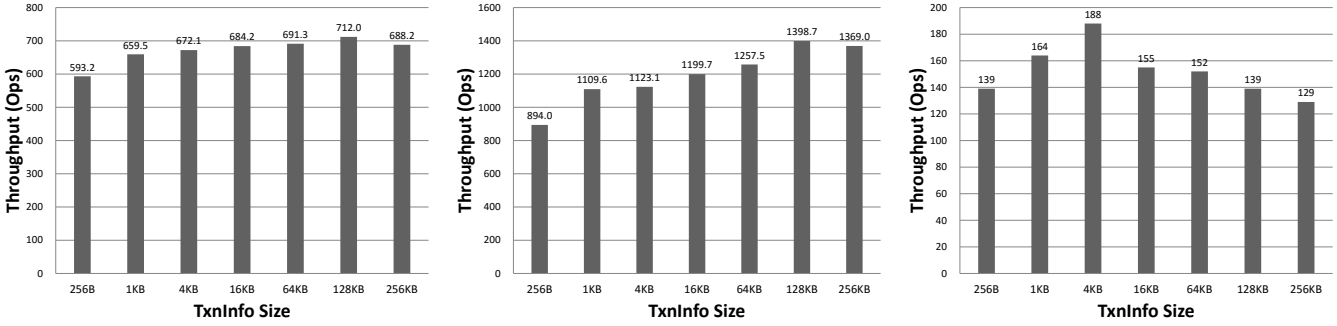
4) Postmark Performance: Last but not least, as shown in Figure 12(a), fine-grained journaling outperforms JBD2 on HDD and NVM up to 114% and 79.6% under Postmark workloads. The highest performance improvement is obtained with the largest number of files. This is because more files to operate leads to more severe write amplification problems, i.e., more inode journal blocks contains only one modified inode.

As shown in Figure 12(b), the average journal length per operation is reduced around 90% consistently because the I/O amount of each operation is unrelated to the number of files.

C. Impact of TxnInfo Size

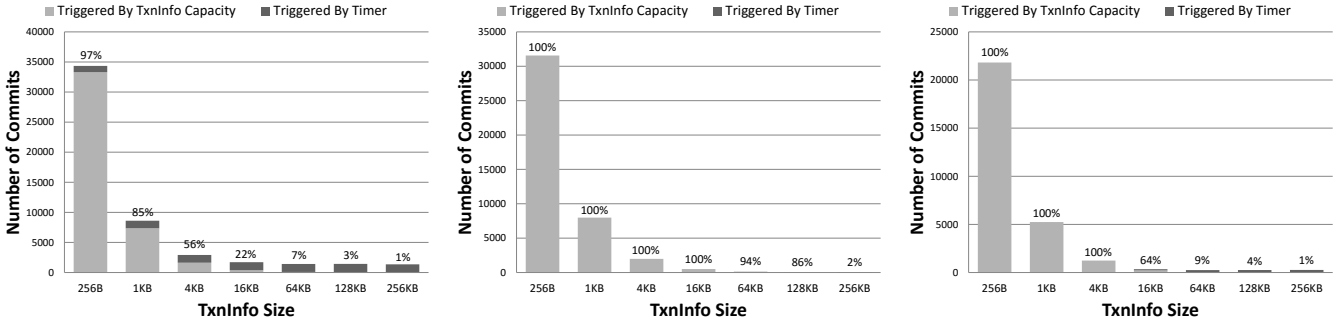
As mentioned in Section III-B, a larger TxnInfo can reduce the commit frequency which results in optimizing the overall performance. To quantify the impact of TxnInfo on the overall performance, we measure the throughput of fine-grained journaling file system under three workloads with different maximum sizes of TxnInfo.

As shown in Figure 13(a), the throughput under Varmail workloads with 128KB TxnInfo can be 20% higher than that with 256B TxnInfo. Similarly, as shown in Figure 13(b), the best performance under FileServer workload is also obtained with 128KB TxnInfo, which is 56.5% better than that with 256B Txninfo. The performance under Postmark workload is also affected by the TxnInfo size. As shown in Figure 13(c), the highest throughput is achieved with 4KB TxnInfo, which is up to 35% better than that with smaller TxnInfo. The main reason of such performance improvement with larger TxnInfo is the reduction of the number of transaction commits. For instance, as shown in Figure 14(a) and 14(b), the total number of commits reduces 95.8% and 99.8% when the maximum size



(a) Varmail Throughput (b) FileServer Throughput (c) Postmark Throughput

Fig. 13: Impact of Txninfo Size on Overall Performance



(a) Number of Commits Distribution under Varmail (b) Number of Commits Distribution under FileServer (c) Number of Commits Distribution under Postmark

Fig. 14: Impact of Txninfo Size on Number of Commits

of TxnInfo changes from 256B to 128KB under Varmail and FileServer workloads, respectively.

However, we also observe that when the maximum size of TxnInfo continues to grow, the overall performance remains the same or even drops under all three workloads. The reason is two-folded. On one hand, a larger TxnInfo introduces more spatial overhead in fine-grained journaling. On the other hand, transaction committing can also be triggered by the timer (e.g., five seconds). In fact, the portion of the transaction commits triggered by full TxnInfo (the percentage number above each bar in Figure 14) decreases when the maximum size of TxnInfo increases. For instance, as shown in 13(a), with 128KB TxnInfo, only 3% of the transaction commits are triggered by the full TxnInfo.

V. Conclusion

In this paper, we present a fine-grained journaling mechanism for file systems using NVM as the journal device. Motivated by the observation that more than 90% of the metadata journal writes in block-based journaling is for clean metadata, we propose to use NVM as the journal device, utilize its byte-addressability to minimize the write amplification problem by using metadata itself instead of the entire metadata block as the basic unit for journaling. We develop a new journal format for journaling on NVM. Based on that, we redesign the process of transaction committing, checkpointing and recovery accordingly in journaling file systems. As a result, we reduce the amount of ordered writes to NVM so that the overhead of journaling on NVM can be minimized. We implement our fine-grained metadata journaling based on JBD2 in Ext4, and evaluate it on a real NVDIMM platform. The experimental results show that the fine-grained journaling can save up to 99.3% of journal writes under FileBench workloads. The overall performance of Ext4 can be improved by up to 15.8x under FileBench workloads.

References

[1] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung *et al.*, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[2] T. Kawahara, "Scalable spin-transfer torque ram technology for normally-off computing," *IEEE Design & Test of Computers*, vol. 28, no. 1, pp. 0052–63, 2011.

[3] L. Chua, "Resistance switching memories are memristors," *Applied Physics A*, vol. 102, no. 4, pp. 765–783, 2011.

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

[5] P. E. Rocha and L. C. Bona, "Analyzing the performance of an externally journaled filesystem," in *IEEE Brazilian Symposium on Computing System Engineering (SBESC)*. IEEE, 2012, pp. 93–98.

[6] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, "Nv-tree: Reducing consistency cost for nvm-based single level systems," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, Santa Clara, CA, Feb. 2015, pp. 167–181.

[7] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[8] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," in *Proceedings of the Linux symposium*, vol. 2. Citeseer, 2007, pp. 21–33.

[9] VikingTechnology, "Arxcis-nv (tm) non-volatile dimm," *http://www.vikingtechnology.com/arxcis-nv*, 2014.

[10] R. McDougall, "Filebench: Application level file system benchmark," 2014.

[11] S. Tweedie, "Ext3, journaling filesystem," in *Ottawa Linux Symposium*, 2000, pp. 24–29.

[12] H. Custer, *Inside windows NT*. Microcomputer Applications, 1992.

[13] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system." in *USENIX Annual Technical Conference*, vol. 15, 1996.

[14] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. Brandt, "Flash on rails: consistent flash performance through redundancy," in *2014 USENIX Annual Technical Conference (ATC)*, 2014.

[15] F. Margaglia, G. Yadgar, E. Yaakobi, Y. Li, A. Schuster, and A. Brinkmann, "The devil is in the details: Implementing flash page reuse with wom codes," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, Santa Clara, CA, Feb. 2016.

[16] J. Lee, Y. Kim, J. Kim, and G. M. Shipman, "Synchronous i/o scheduling of independent write caches for an array of ssds," *Computer Architecture Letters*, vol. 14, no. 1, pp. 79–82, 2015.

[17] D. Jeong, Y. Lee, and J.-S. Kim, "Boosting quasi-asynchronous i/o for better responsiveness in mobile devices," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 191–202.

[18] J. D. Strunk, "Hybrid aggregates: Combining ssds and hdds in a single storage pool," *ACM SIGOPS Operating Systems Review*, vol. 46, no. 3, pp. 50–56, 2012.

[19] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 45–58.

[20] S. Boboila, Y. Kim, S. S. Vazhkudai, P. Desnoyers, and G. M. Shipman, "Active flash: Out-of-core data analytics on flash storage," in *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.

[21] L. Mármol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan, "Nvmkv: A scalable and lightweight flash aware key-value store," in *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[22] Y. Zhang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Removing the costs and retaining the benefits of flash-based ssd virtualization with fsdv," in *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[23] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Anvil: advanced virtualization for modern non-volatile memory devices," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 111–118.

[24] Z. Li, S. Zhang, J. Liu, W. Tong, Y. Hua, D. Feng, and C. Yu, "A software-defined fusion storage system for pcm and nand flash," in *IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, 2015.

[25] S. Li, P. Chi, J. Zhao, K.-T. Cheng, and Y. Xie, "Leveraging nonvolatility for architecture design with emerging nvm," in *IEEE Non-Volatile Memory System and Applications Symposium (NVMSA)*, 2015.

[26] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[27] D. Kang, S. Baek, J. Choi, D. Lee, S. H. Noh, and O. Mutlu, "Amnesic cache management for non-volatile memory," in *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[28] S. Gao, J. Xu, B. He, B. Choi, and H. Hu, "Pcmlogging: Reducing transaction logging overhead with pcm," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, ser. CIKM '11, New York, USA, 2011, pp. 2401–2404.

[29] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Computer Architecture News*, 2009.

[30] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ACM SIGARCH Computer Architecture News*, vol. 37. ACM, 2009, pp. 14–23.

[31] Everspin, "Second generation mram: Spin torque technology," *http://www.everspin.com/products/second-generation-st-mram.html*, 2004.

[32] D. Narayanan and O. Hodson, "Whole-system persistence," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012, pp. 401–410.

[33] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39. ACM, 2011, pp. 91–104.

[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*, 2009, pp. 133–146.

[35] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh *et al.*, "Betrfs: A right-optimized write-optimized file system," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 301–315.

[36] D. Fryer, M. Qin, J. Sun, K. W. Lee, A. D. Brown, and A. Goel, "Checking the integrity of transactional mechanisms," *ACM Transactions on Storage (TOS)*, vol. 10, no. 4, p. 17, 2014.

[37] G. Amvrosiadis, A. D. Brown, and A. Goel, "Opportunistic storage maintenance," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[38] R. Santana, R. Rangaswami, V. Tarasov, and D. Hildebrand, "A fast and slippery slope for file systems," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 2, pp. 27–34, 2016.

[39] J. Bent, G. Grider, B. Kettering, A. Manzanares, M. McClelland, A. Torres, and A. Torrez, "Storage challenges at los alamos national lab," in *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.

[40] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter, "How to get more value from your file system directory cache," in *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[41] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Model-based failure analysis of journaling file systems," in *IEEE International Conference on Dependable Systems and Networks (DSN)*, 2005.

[42] D. Fryer, K. Sun, R. Mahmood, T. Cheng, S. Benjamin, A. Goel, and A. D. Brown, "Recon: Verifying file system consistency at runtime," *ACM Transactions on Storage (TOS)*, vol. 8, no. 4, p. 15, 2012.

[43] J. Chen, Q. Wei, C. Chen, and L. Wu, "Fsmac: A file system metadata accelerator with non-volatile memory," in *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, 2013.

[44] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on nvm," in *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[45] C. Wang, Q. Wei, J. Yang, C. Chen, and M. Xue, "How to be consistent with persistent memory? an evaluation approach," in *10th IEEE International Conference on Networking, Architecture and Storage (NAS)*, 2015, pp. 186–194.

[46] Q. Wei, J. Chen, and C. Chen, "Accelerating file system metadata access with byte-addressable nonvolatile memory," *ACM Transactions on Storage (TOS)*, vol. 11, no. 3, p. 12, 2015.

[47] X. Wu and A. Reddy, "Scmfs: a file system for storage class memory," in *Proceedings of 2011 International ACM Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[48] E. Lee, S. Yoo, J.-E. Jang, and H. Bahn, "Shortcut-jfs: A write efficient journaling file system for phase change memory," in *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.

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

[50] 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*, 2015, p. 1.

[51] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory." in *FAST*, 2013, pp. 73–80.

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