SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

March 2022, Vol. 65 132102:1–132102:15
https://doi.org/10.1007/s11432-019-2808-x

Stabilizing and boosting I/O performance for file systems with journaling on NVMe SSD

Lin QIAN[1,2], Bin TANG[1*], Baoliu YE[1*], Jianyu WU[1], Xiaoliang WANG[1] & Sanglu LU[1]

[1]*National Key Laboratory for Novel Software Technology, Nanjing* 210023*, China;*
[2]*State Grid Electric Power Research Institute, Nanjing* 211000*, China*

Received 16 October 2019/Revised 11 December 2019/Accepted 16 February 2020/Published online 22 February 2021

Abstract Many journaling file systems currently use non-volatile memory-express (NVMe) solid-state drives (SSDs) as external journal devices to improve the input and output (I/O) performance. However, when facing microwrite workloads, which are typical of many applications, they suffer from severe I/O fluctuations and the NVMe SSD utilization is extremely low. The experimental results indicate that this phenomenon arises mainly because writing back data to backend file systems on hard disk drives is much slower than journal writing, causing journal writing to frequently freeze because of the two-phase mechanism. We, therefore, propose a merging-in-memory (MIM) acceleration architecture to stabilize and boost the I/O performance for such journaling file systems. MIM employs an efficient data structure of hash-table-based multiple linked lists in memory, which not only merges random microwrites into sequential large blocks to speed up writebacks but also provides additional gains in terms of reducing the frequency of write addressing and object opening and closing. Using a prototype implementation in Ceph FileStore, we experimentally show that MIM not only eliminates severe fluctuations but also improves the I/O operations per second by roughly 1×–12× and reduces the write latency by 75%–98%.

Keywords journaling file systems, NVMe SSD, microwrite merging, hash table, Ceph

Citation Qian L, Tang B, Ye B L, et al. Stabilizing and boosting I/O performance for file systems with journaling on NVMe SSD. Sci China Inf Sci, 2022, 65(3): 132102, https://doi.org/10.1007/s11432-019-2808-x

1 Introduction

In order to guarantee the consistency and durability of data in case of system crash and power failure, many popular file systems, both local (e.g., EXT3 [1] and EXT4 [2]) and distributed (e.g., Ceph [3]), employ a journaling mechanism — each write transaction is first committed to an append-only journal, and then written back to the backend file system. When a system crash or power failure occurs, a recovery process will scan the journal, and then redo the write transactions that have not completed successfully.

Modern journaling file systems mainly use magnetic hard disk drives (HDDs) as underlying storage devices for both journal and data. The development of non-volatile memory (NVM) technologies, including NVM-express (NVMe) solid-state disks (SSD) as external journal devices, has recently drawn much attention from both academic and industrial researchers [4–6]. NVMe SSDs are orders of magnitude faster than HDDs. However, as indicated by a recent study [6] and our own experiments, their input and output (I/O) performance improvement in current journaling file systems is rather limited.

We conduct experiments on a prototype that implements the FileStore default storage engine, produced by Ceph [7], which includes an NVMe SSD as an external journal device and three object storage devices (OSDs), each corresponding to a different HDD. Microwrites having block sizes of 4, 8, and 64 kB are typically considered in many applications, such as database online transactional processing (OLTP) and internet-of-thing (IoT) data collection [8–11]. We observe that the improvement in I/O operations per second (IOPS) caused by the use of NVMe SSD is only about 83%. Moreover, as shown in Figure 1(a), IOPS fluctuates severely, resulting in large deviations in both throughput and latency (c.f., Section 6).

* Corresponding author (email: tb@nju.edu.cn, yebl@nju.edu.cn)

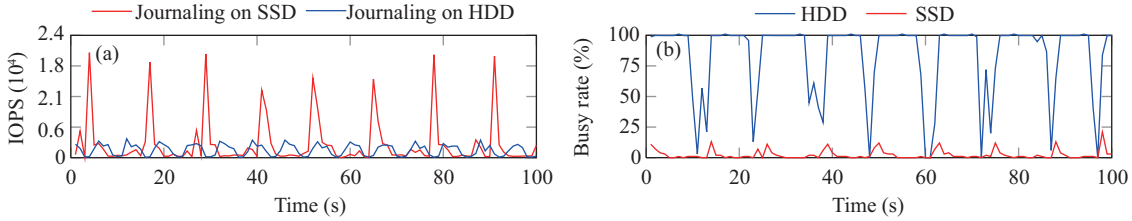


Figure 1 (Color online) Journaling on NVMe SSD causes severe I/O performance fluctuations, whereas NVMe SSD is busy only for very short periods. (a) IOPS performance; (b) utilization of disks.

This can be very harmful for related applications [12–15]. For example, with I/O-intensive applications, severe IOPS fluctuations lead to very low quality of experience for users, and with time-sensitive applications, the stringent requirements on latency are not always satisfied. Figure 1(b) shows that NVMe SSD is busy only for very short periods. The underlying reason for this phenomenon is that it is extremely fast when recording microwrites in the journal using NVMe SSD. However, HDD performs poorly for microwrites, making writebacks much slower than journaling. Hence, the writeback queue for flushing is often fulfilled. Owing to the two-phase journaling mechanism, this causes journaling to frequently freeze, and subsequently, leads to severe performance fluctuations.

The key to resolving the above issue is improving the speed of writebacks. One approach is using NVMe SSDs as backend storage devices. However, this incurs significant expenses, owing to the high price of large-volume NVMe SSDs. Because the magnetic heads of HDDs spin less frequently in cases of sequential writes than that during random writes, HDDs perform much better for sequential writes than for random microwrites [16]. Inspired by this property, our main idea is to merge multiple microwrites into sequential large writes in the memory, which are then flushed to the backend storage devices. Because only microwrites having the same address (i.e., the same directory and object) can be merged, the realization of this concept faces several challenges. First, identifying whether multiple microwrites have the same address should be performed very quickly, so that this additional process does not become a new limitation for performance improvement. Second, to gain more merging chances, each microwrite is expected to stay in the memory for a relatively long time prior to flushing. However, this will delay the writeback of the microwrite. This increases the possibility that the journal queue becomes full, resulting in new waiting times for new microwrites. Third, owing to the reordering of microwrites caused by merging operations, the original checkpointing mechanism will lose its functionality.

This paper proposes the merging-in-memory (MIM) acceleration architecture to support merging microwrites into sequential large writes in the memory. MIM addresses the first challenge by introducing a novel open-address hash table-based multiple linked-list (HTMLL) data structure having certain properties. This can not only support the merging operations very efficiently but also bring additional benefits for flushing. MIM deals with the second challenge by employing a novel flushing scheme that accounts for both the occupancy ratio of HTMLL and time. Checkpointing refers to redesigned in MIM to solve the third challenge. Moreover, MIM can provide additional gains in terms of reducing the frequency of write addressing and object opening and closing, which are very beneficial for speeding up the writeback process. Based on these properties, MIM can stabilize and boost the I/O performance of journaling file systems, and it can be easily incorporated with existing journaling file systems.

This paper makes the following contributions.

• We observe and analyze the I/O fluctuation phenomenon in typical file systems (Ceph FileStore and EXT4) using journaling on NVMe SSD, where the utilization of NVMe SSD is extremely low. The underlying reason for this is that the writeback of microwrites to the backend file system based on HDDs is too slow to catch up with microwriting in journals based on NVMe SSD, causing the journaling to frequently freeze.

• Based on the observations, we introduce a memory acceleration architecture (i.e., MIM) for file systems with journaling on NVMe SSD, which introduces a data-structure HTMLL in the memory that supports merging microwrites into sequential large writes and provides some additional gains. We design a novel flushing scheme to fully exploit the benefits of merging while preventing the journal queue from growing too long. Furthermore, we redesign the checkpointing process to guarantee data durability.

• We implement MIM on a prototype Ceph FileStore and conduct extensive evaluations, wherein various workloads are considered. The experimental results show that MIM can not only eliminate severe fluctuations but also improve IOPS by roughly 1×–12× and reduce write latency by 75%–98%.

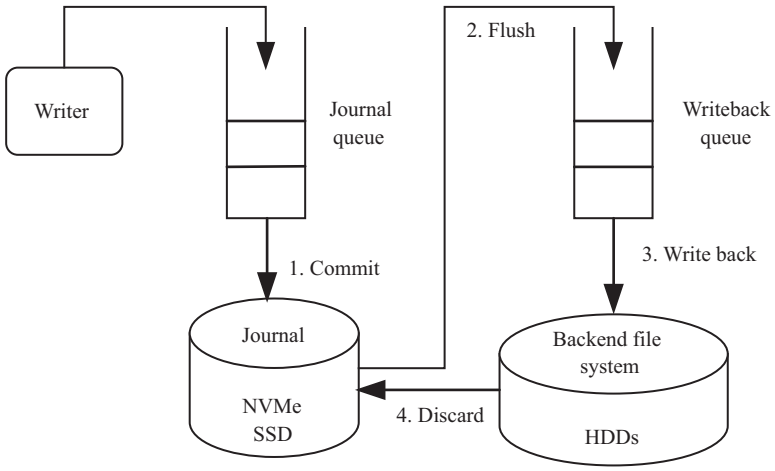


Figure 2 Conventional file system with journaling on NVMe SSD.

The remainder of the paper is organized as follows. In Section 2, we introduce some background and related work. In Section 3, we introduce and analyze the performance fluctuation phenomenon of current journaling file systems. We present the detailed design of MIM in Section 4 and the implementation of MIM in Section 5. The experimental results are shown and explained in Section 6. Finally, this paper is concluded in Section 7.

2 Background and related work

In this section, we first introduce how a typical journaling mechanism works. Then, we discuss the related studies.

2.1 Journaling mechanism

As shown in Figure 2, we consider a file system that supports journaling on external NVMe SSDs. Each write transaction is first committed to the journal via journal queuing. Write operations are then flushed to the writeback queue on a batch basis, and are further written back to the backend file system on HDDs. Once the writeback succeeds, the data become permanent, and the related journal item is discarded from the journal based on checkpointing. If a system crash or power failure occurs, the HDD data can be recovered to the latest consistency status, using the redo log and journaling checkpoint mechanism. To reduce the burden of journaling on the entire data, some file systems only adopt journaling on metadata. Because they cannot guarantee the durability of all data, they are only suitable for specific applications. In this paper, we consider that journaling is applied to the entire data.

2.2 Related work

Many modern file systems adopt the journaling mechanism to provide durability [17–22]. At the same time, the two-phase committing scheme of journaling brings extra overheads, which affect the I/O performance in journaling [23].

In a journaling file system where an HDD is used as a physical device store for both metadata and data, the write performance degrades because of the I/O competitions between the journal and the backend. Aghayev et al. [24] proposed Ext4-lazy, which writes each metadata block approximately once to the journal and inserts a mapping to an in-memory map so that the number of microwrites on the HDD is reduced. The Z file system (ZFS) [25] and the B-tree file system (BTRFS) [26] are widely used in Linux to apply copy-on-write update policies to transform the access patterns from a large amount of small random writes to a single large sequential write [27]. However, these studies are not really applicable to file systems providing SSD journaling.

Recently, several journaling file systems have employed SSDs as external journal devices, and great efforts have been made to exploit the potential of various SSDs [4–6]. Choi et al. [28] proposed an efficient flash transaction layer based on a journal remapping technique, to eliminate redundant duplicates during

the writing process. Lee et al. [29] focused on the byte accessibility of phase-change memory to reduce the write amount of journaling. Chen et al. [30] designed byte-enabled journaling using a hardware encoder to reduce the write amount and managed a compress-enabled journaling strategy to avoid write amplification on non-volatile random-access memory journaling file systems. These studies only focused on the optimization of the first commit phase of the journaling mechanism. However, the second writeback phase can also deeply impact the overall performance, which can result in low and fluctuated performance. This issue is observed and addressed in this study.

Jannen et al. [16] introduced the microwrite optimization, which affects the second-phase write performance, and adopted a log-structured merge (LSM) tree [31] to merge microwrites in the memory. Since then, many data structures based on the LSM tree have been proposed to provide more efficient write performance with backend file systems. Shetty et al. [32] extended the LSM tree and proposed an expanded form tree to efficiently handle sequential and file-system workloads. Wu et al. [33] introduced a design based on an out-of-memory overspilling index, to speed up the performance of small writes. Lu et al. [34] separated keys from values in KVstore based on the LSM tree, to leverage write performance. However, the LSM tree limits the speed of reads and writes, and the compaction operation in it incurs severe performance degradation. Thus, Jannen et al. [16] introduced a B^{ε}-tree that can provide the same insert performance as the LSM tree but without compaction. However, it is still not suitable for frequency opening and closing workloads. In contrast, our study does not create an index of data to be written to the file system. Specifically, we introduce a data structure of hash-table-based multiple linked lists that can not only perform write merging efficiently but also provide additional gains in terms of reducing the frequency of writing addressing and object opening and closing.

3 Motivation

In this section, we present some experimental results to show the I/O performance fluctuation phenomenon, which motivates the MIM design.

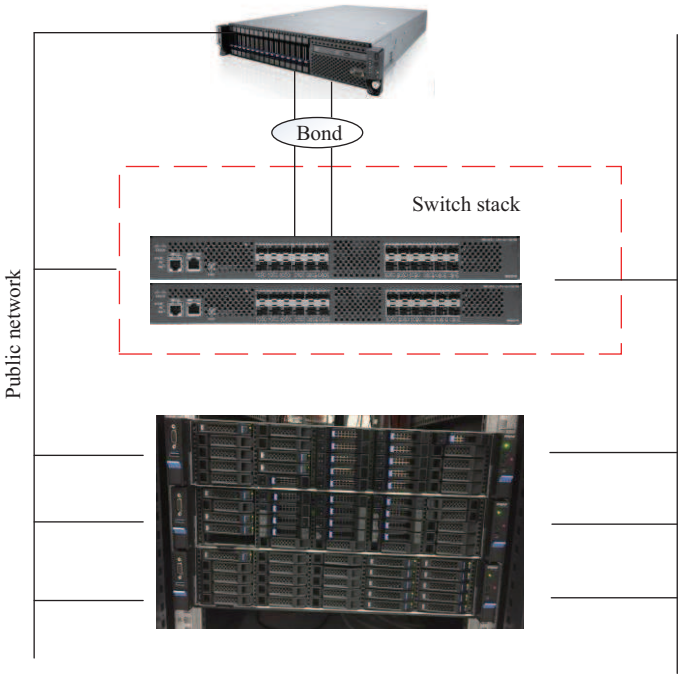
3.1 Experimental setup

Testbed. We deploy Ceph FileStore, a typical journal-based store engine of the Ceph file system, on a cluster with four X86 servers, as illustrated in Figure 3. We then perform some experiments[1)]. In the Ceph file system, each file is divided into several objects in some directory, and when a write operation approaches, it is first written to an interface (a Rados block device (RBD)), which converts file writing to object writing. One of the X86 servers is used to deploy RBD, while the others are used as storage nodes deployed with Ceph FileStore, which connect to each other via the transport control protocol internet protocol on 10-Gbps Ethernet.

Configurations. To avoid network disturbances, we separate the storage network from the public network. For the hardware, each X86 server is equipped with two Intel E5-2620 v4 2.1 GHz central processing units (CPUs). We use 32 GB error-correcting code memory to avoid other errors originating from the operating system (OS) and hardware itself. We choose three 1.2 TB, 10000 RPM serial-attached small-computer-system-interface HDDs as data storage on a redundant array of independent disk card without cache and three 100 GB partitions of a 1.6 TB Intel P3600 NVMe SSD as an external journal device. We set the CPUs to the mode of performance instead of the default eco-mode, to eliminate the hardware influence. A redundancy scheme based on 2-replication is applied. For the software, we choose FileStore from open-source Ceph 0.94.6[2)]. The block size of the file system is set to 4 MB. Other parameters are set to default.

Workload. We adopt a widely used benchmark tool, flexible I/O (FIO) (version 2.1.10)[3)] tester, for performance evaluation. FIO can generate workloads of different microwrite sizes (4, 8, and 64 kB) and different numbers of concurrent write threads (8, 16, and 64), which are typical in applications including database OLTP and IoT data collection [8, 9]. Both the IOPS and disk busy rate are collected using an Nmon tool [35]. For each experiment, the system is run for 360 s, and only the performance results during the middle 100 s are collected for use.

1) We also conduct these experiments with default EXT4 in Ubuntu 14.04 and observe similar results that are omitted.
2) https://github.com/ceph/ceph/.
3) http://freshmeat.sourceforge.net/projects/fio.



Application server/client (×1)	
Processer	E5-2620 v4
Memory	32 GB
Network	10 Gbps (bond ×2)

Switches (×2)	
Public network	10 Gbps (×5)
Storage network	10 GB (×3)

Data server (3)	
Processer	E5-2620 v4
Memory	32 GB
Network	10 Gbps (×2)
Disk	HDD (×3), NVMe (×1)

Figure 3 (Color online) Testbed for experiments.

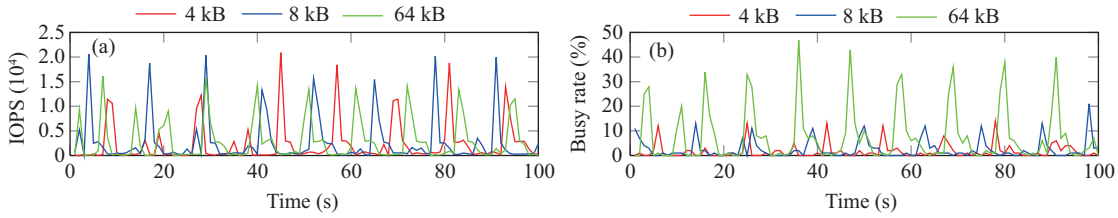


Figure 4 (Color online) IOPS performance of Ceph FileStore with journaling on NVMe SSD fluctuates, and the journaling on NVMe SSD is frozen frequently. Here, the number of threads is set to eight.

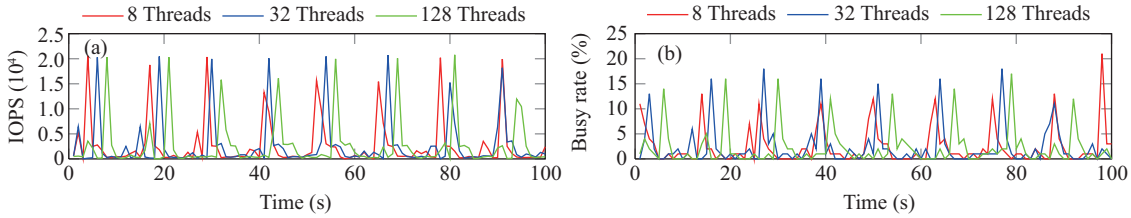


Figure 5 (Color online) IOPS performance of Ceph FileStore with journaling on NVMe SSD fluctuates, and the journaling on NVMe SSD is frozen frequently. Here, the microwrite size is set to 8 kB. (a) IOPS performance; (b) utilization of NVMe SSD.

3.2 Observations and analysis

Our main experimental results are given in Figures 4 and 5. Specifically, Figures 4(a) and (b) show the IOPS performance of the system and the busy rate of NVMe SSD, respectively, where the number of threads is eight and the microwrite size varies. Figures 5(a) and (b) show similar results for the case where the microwrite size is 8 kB and the number of threads varies. From the figures, we collect the following observations.

• **Observation 1.** For all these parameter settings, the IOPS performance fluctuates severely and the NVMe SSD is frozen most of the time.

• **Observation 2.** The stability of IOPS performance is closely related to the microwrite size, but has little relevance with the number of threads. Specifically, when the microwrite size is smaller, the fluctuation of the IOPS performance is more severe.

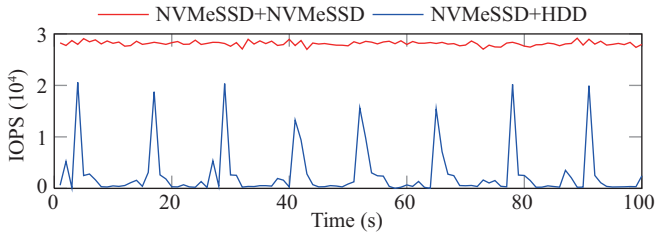


Figure 6 (Color online) When replacing the backend HDD with NVMe SSD, the IOPS performance becomes stable.

The underlying reason for this performance fluctuation phenomenon is as follows. It is extremely fast to record random microwrites in the journal based on NVMe SSD, but HDD performs poorly for random microwrites, making the speed of microwrite much lower than that of journaling. Hence, the writeback queue for flushing is often fulfilled. This causes the journaling capability to freeze frequently, and subsequently, leads to severe performance fluctuations. This analysis also provides an explanation for Observation 2. For random microwrites of larger sizes, the HDD performs better [16], which leads to a faster writeback process. Thus, the performance fluctuation decreases. On the other hand, although the number of write threads has some effect on the journaling, because of the lock contention [36], it is hardly relevant with the writeback speed. Thus, the fluctuation for each number of threads is similar.

To further validate the underlying reason for the performance fluctuation phenomenon, we conduct another experiment in which we use the 1.6 TB Intel P3600 NVMe SSD as the device for both journal and data. The microwrite size is set to 8 kB, and the number of write threads is set to eight. The IOPS performance of this system compared with the original one is depicted in Figure 6, where the IOPS performance becomes very stable and is much higher. This is because the writeback speed based on NVM SSDs can now catch up with the journaling speed, which significantly reduces the frozen time of journaling.

Although replacing backend HDDs with NVMe SSD can solve the performance fluctuation problem, owing to the current high price of large-volume NVMe SSDs, this approach is impractical, at least in the near future.

4 Design

In this section, we first outline the design goals, and then introduce a detailed design of MIM.

4.1 Goals

We aim to support the following properties in the proposed file system with journaling on NVMe SSD.

(i) Performance. The overall I/O performance in terms of IOPS is significantly improved compared to the original journaling file system.

(ii) Stability. The I/O performance is relatively stable over time.

(iii) Durability. Once a write transaction is committed to the journal successfully, it will survive permanently.

(iv) Low cost. The additional resource consumption incurred by the new design is maintained at a low level.

(v) Compatibility. The design is relatively simple and can be easily incorporated into the existing journaling file systems.

4.2 Overview of MIM

According to the results shown in Section 3, if we can speed up the writeback to the backend file system, the I/O performance can be stabilized and improved. Inspired by the well-known fact that HDDs perform much better for large sequential writes than random microwrites [37], our main idea is to efficiently merge multiple random microwrites into sequential large writes in the memory. We design MIM, a novel memory acceleration architecture, as illustrated in Figure 7, where an HTMLL data structure is introduced in the memory to support microwrite merging. MIM also adopts a novel flushing scheme, so that the benefit of merging can be fully exploited while preventing the journal from growing too large. Furthermore, the checkpointing process is redesigned to guarantee data durability.

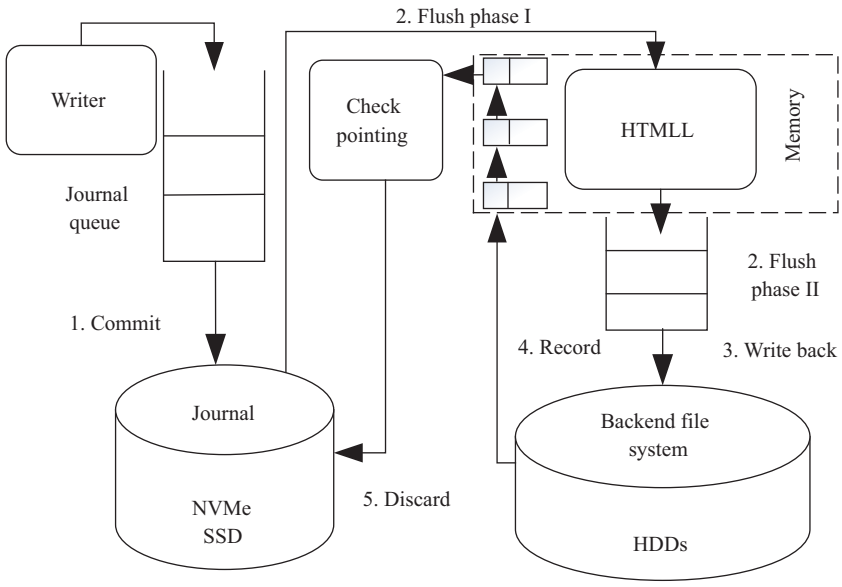


Figure 7 Illustration of the MIM architecture.

As illustrated in Figure 7, the entire write process based on MIM is slightly different from the original one. The flush operation is mainly split into two phases. The first phase is for flushing each microwrite operation into HTMLL, whereas the second phase is for flushing multiple merged microwrite operations into the writeback queue. Another difference is that the microwrite operations having been successfully written to the backend file system must be recorded in some manner for the new checkpointing scheme.

4.3 HTMLL-based merging

Consider the writeback procedure executed by the write thread. Each microwrite operation can be represented as a quadruple: $\langle \mathsf{cid}, \mathsf{oid}, \mathsf{sn}, \mathsf{data} \rangle$, where

- cid denotes the identity of the object group (i.e., the directory) where the microwrite operation is to be performed;
- oid denotes the identity of an object in the objective group of cid where the data are to be written, and in particular, different object groups share the same set of values as oid;
- sn denotes the sequence number for marking the microwrite operations; and
- data is the data to be written to the file system.

Generally, the number of object groups can be very large when the number of objects in an object group is usually small (e.g., 512); thus, the time required for locating an object is short. In other words, cid can vary over a very large range, whereas the number of possible values of oid is limited. Based on this observation, we introduce the following data-structure HTMLL to support the merging of microwrite operations.

The data-structure HTMLL, as illustrated in Figure 8, is initialized in the memory, which is a combination of a hash table containing N slots and N linked lists, where each slot acts as the starting pointer of a linked list. Specifically,

- The hash table takes oid as the key and applies open addressing for collision avoidance, such that each value of oid is mapped into a different slot when there is an empty slot in the hash table.
- Each linked list comprises M linked blocks. The size of each block is equal to that of an object, which is specified by the file system. The blocks located at the same position in the linked lists (depicted in Figure 8) are associated with the same cid. The values of the cids corresponding to the blocks are assigned to the most frequently used ones, which are known by the write thread. They are updated after an entire flushing operation (see Subsection 4.4) is triggered.

Clearly, the memory consumption of HTMLL is determined by the parameters M and N, as well as the object size. Hence, the memory consumption is controllable because we can choose proper values for M and N.

Next, suppose that a new microwrite operation, $\langle \mathsf{cid}, \mathsf{oid}, \mathsf{sn}, \mathsf{data} \rangle$, arrives at the HTMLL during the flush phase 1. Based on its oid, the write thread will try to map it into a certain slot of the hash table. If

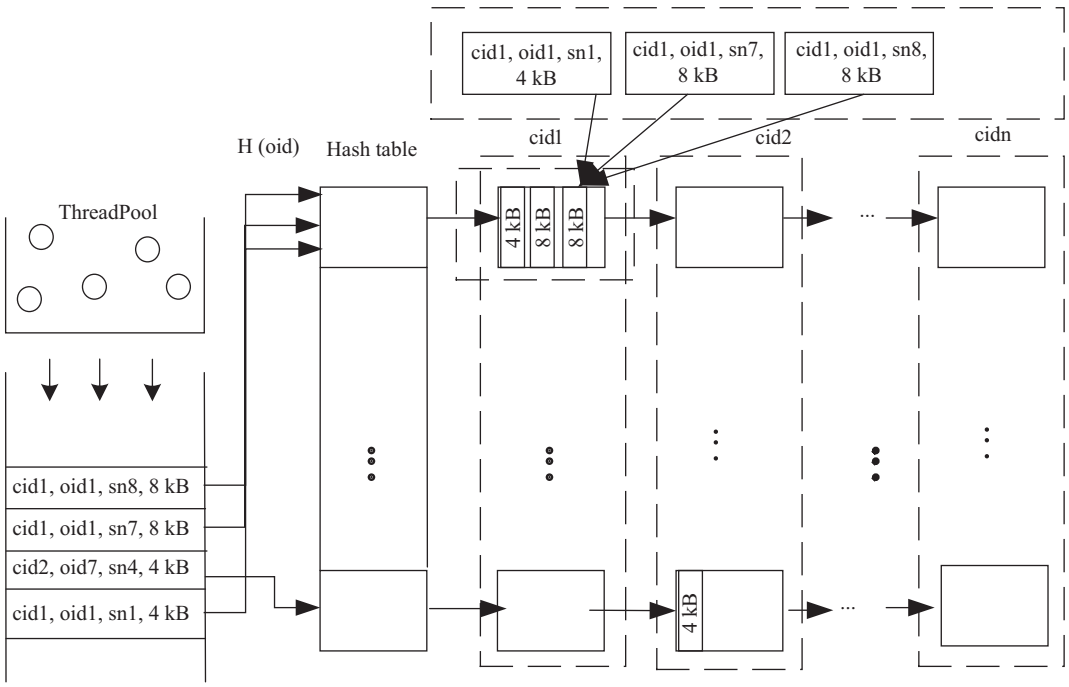


Figure 8 Illustration of HTMLL.

it does not succeed (i.e., there is no empty slot in the hash table and its oid is different from the existing ones), the operation will be immediately flushed to the writeback queue. If it succeeds, the write thread will check whether there exists a block associated with the cid in the corresponding linked list. If there is no such block, the microwrite operation will be directly flushed to the writeback queue. Otherwise, it will be merged into the block in a data-appended manner.

4.4 Flushing phase II

A microwrite operation is expected to remain in the HTMLL for a relatively long time to have more chances to merge with other microwrite operations. However, if the occupancy ratio of the HTMLL, or the percentage of blocks in HTMLL used by microwrite operations, is very high, then many new arriving microwrite operations will be flushed to the writeback queue directly according to the merging procedure, which decreases the benefit gained from merging. Moreover, if there is some microwrite operation remaining in the HTMLL other than that written back to the file system, the checkpoint-based journal will grow too large, causing severe performance degradation.

To prevent the above situations, we propose a novel flushing scheme by introducing two thresholds: the space threshold, based on the occupany ratio of the HTMLL, and the time threshold, based on the interval length between two consecutive flushing operations performed on HTMLL. Specifically, the space threshold is defined as

$$\mathsf{sp\_thresh} = \frac{\mathsf{journalthroughput} \times \mathsf{max\ flushing\ intvl}}{\mathsf{blocksize}},$$

where journalthroughput denotes the inherent maximum throughput of the NVMe SSD for appending microwrites, max flushing intvl denotes the maximum interval time between two consecutive journal-flushing (phase 1 in MIM) operations, specified by the file system, and blocksize is the size of an object. Thus, sp_thresh is roughly the maximum number of blocks in the memory that can be fully filled in a journal-flushing interval.

We define the time threshold depending on the configuration of the file system. For a distributed file system, we relate it to the scale of the cluster and the number of replications for considering the latest write success. Because journal writing cannot start until the latest write is successful, replication is written one-by-one or across all nodes in the cluster in the worst case. Thus, the time threshold is defined as

$$\mathsf{time\_thresh} = \max\{\mathsf{rep}, \mathsf{scale}\} \times \mathsf{max\ flushing\ intvl},$$

where rep is the number of replicas of the data to be written to the backend (distributed) file system and scale is the number of nodes in the file system.

When the occupancy ratio of HTMLL exceeds sp_thresh, or the time passed since the last flushing (phase 2) operation performed on HTMLL exceeds time_thresh, the writer thread will stop inserting new microwrite operations into HTMLL, and then perform an entire flushing operation (i.e., all microwrite operations in the HTMLL are flushed to the writeback queue immediately on the basis of a block). In particular, the blocks corresponding to the same cid are flushed consecutively to the writeback queue. In this manner, multiple microwrite operations are flushed together to the writeback queue, and hence, the frequency of write addressings and object openings and closings can be significantly reduced.

4.5 Checkpointing and recovery

In journaling file systems, microwrite operations are appended to the journal file. There exists a checkpoint in the journal file, which is updated periodically (i.e., checkpointing) and denotes the first microwrite operation that has not been written back to the file system at the time of the last checkpointing. In conventional journal file systems, the microwrite operations are written back to the file system in the same order in which they are appended to the journal file. Thus, only the sn of the last microwrite operation that was successfully written back to the file system is required for checkpointing. However, in MIM, owing to the merging operations, a microwrite operation that is behind another one in the journal file might be written back earlier. Hence, the sn of the last microwrite operation that was successfully written back to the file system is insufficient for checkpointing.

To overcome this, we record all sn of the microwrite operations that are written back successfully since the last checkpoint. We then find the microwrite operation with the minimum sn that fails to be written to the backend file system. Although this scheme can be performed using a sorting-based algorithm, we can perform it more efficiently. Specifically, we use an auxiliary linked list to record sn, and for each new microwrite operation that is written back successfully, its sn is inserted into the linked list such that all sn in the linked list are sorted according to the orders of these microwrite operations in the journal. Thus, the checkpointing process is executed as follows. Compare the sn of the microwrite operation at the checkpoint having the sn value of the first node in the linked list. If equal, then we move the checkpoint backward by one microwrite operation and delete the first node in the linked list. Then, we repeat this procedure. Otherwise, the procedure terminates.

Based on this new checkpointing, the recovery process is executed as with a conventional journaling file system. It is straightforward to see that the data durability property preserves.

5 Implementation

Our prototype implementation modifies Ceph filestore.cc and filestore.hh in the \src\os\ directory from Ceph version 0.94.6. The FileStore engine is independent of other stores and interconnected with placement groups according to ObjectStore. Thus, other components of Ceph do not require modification, and it is convenient to deploy MIM by using a patch that is auto-executed.

We define several data structures that are used for filestore.cc.

(i) cache_obj defines an object in the memory with cid, oid, and a data structure, called bufferlist, is used to merge microwrites, especially the random ones, which can take advantage of this data structure to convert random physical addresses to sequential logical bulks in the memory.

(ii) HTMLL includes ⟨uint64_t, cached_obj*⟩ key-value pairs. The key comes from the get_filestore_key method, and the value is cache_obj, which is defined as a pointer, *obj, and a list, ⟨uint64_t⟩.

(iii) Thread pool defines ThreadPool::WorkQueue and cached_obj_writeback_tp method for cache_obj to insert and write back transactions having multiple-thread management and mutex lock on cached_obj.

We implement MIM in the main program, file-named filestore.cc. It comprises the following multiple components.

Merging. For adding MIM in filestore.cc, we modify the FileStore::_write() function to merge multiple microwrites in the bufferlist memory space with a hash function unordered_map ⟨uint64_t, cached_obj*⟩. We obtain the key from the original filestore as oid and search the hash table and linked list in HTMLL as if the hash address has existed. When the address is in HTMLL and object size as obj_bl.length() does not exceed the recent length of data in the bufferlist and the offset of this writing, then the mircowrite can be merged in the existing bufferlist to sequentially write a function, obj_bl.copy_in(), performed to

iteratively append the new arriving data offset to the rear pointer of bufferlist. If the length of the data is over the size of the bufferlist, the pointer moves to cached_obj with the next cid to rewrite.

Flushing. When a write transaction arrives, we expect it to hit each cached_obj in HTMLL. However, there are always some microwrites that are very difficult to merge with the bufferlist. We take the original write method, which we redefine as function _write_raw(), to directly write to the object in the backend file system. When the write transaction hits the HTMLL and the flushing conditions based on thresholds are satisfied, it calls function do_cold_obj_insert() to insert the objects into the writeback queue first and conduct writeback processing. In function do_cold_obj_insert(), we adopt last_update_time to select cold-cached_obj to insert into the writeback queue. Using this method, we can attempt to merge microwrites in MIM instead of frequent open real objects in the backend file systems.

Sync and crash recovery. We modify FileStore::sync_entry() by recording all write operations, including MIM and written back ones, in the new linked list and recall JournalingObjectStore::ApplyManager::commit_start() to give the first node value in linked list to committing_seq, which is the checkpoint in the journal. During the recovery phase, if the function of commit_finish() returns success and the transactions are inserted into MIM, failure occurs. The recovery thread is then initialized by Thread-Pool::TPHandle and scans log items until the checkpoint and redo failure transactions with journal items to provide consistency and durability.

6 Performance evaluation

6.1 Experiment setup

The experiments are conducted in the testbed, as described in Section 3. For performance evaluation, we consider three types of workloads generated by two widely used benchmark tools: FIO and FileBench [38].

• FIO workload, generated by an FIO tool, which emulates continuous heave microwrite operations on a single large file.

• Varmail workload, generated by FileBench tool, emulates a mail server that stores each email in a separated file. This workload comprises multiple threads to create, read, and delete small email files in a single directory. For this workload, we create 12000 email files, with a capacity of 128 kB each.

• FileServer workload, also generated by the FileBench tool, emulates complicated file operations, including creates, deletes, appends, reads, writes, and attribute operations on a directory tree. We also set the file number as 12000 and the file size as 128 kB. There are a total of 20 directories, each containing 600 files.

In particular, the ratios of read operations to write operations under the Varmail workload and the FileServer workload are 1:1 and 1:2, respectively. For each type of workload, we consider different sizes of microwrites (4, 8, and 64 kB) and different numbers of concurrent write threads (8, 16, 64), as used in Section 3.

6.2 Performance improvement by MIM

Because MIM is implemented in Ceph FileStore, we first conduct experiments to see how the performance of Ceph FileStore can be improved by MIM under different workloads. The performance results of the average IOPS and the average latency under FIO, Varmail, and Fileserver workloads are plotted in Figures 9–11, respectively.

For the FIO workload, we observe the following from Figure 9.

• The IOPS performance improvement of MIM over the original Ceph FileStore is 9.0×–12.3×, 8.1×–10.0×, and 4.6×–5.2× for microwrites when the microwrite size is 4, 8, and 64 kB, respectively.

• The performance improvement increases when the microwrite size is decreased. When the microwrite size is smaller, there are more chances of merging operations that can be leveraged by MIM, leading to a higher performance improvement.

• When the number of threads changes from 8 to 16, the performance of MIM degrades. Under the FIO workload, all microwrite operations are executed on a single file. Thus, when the number of threads grows from 8 to 16, the lock contention becomes severe. However, when the number of threads changes from 16 to 64, the performance of MIM remains almost the same, because the lock contention reaches a very high level.

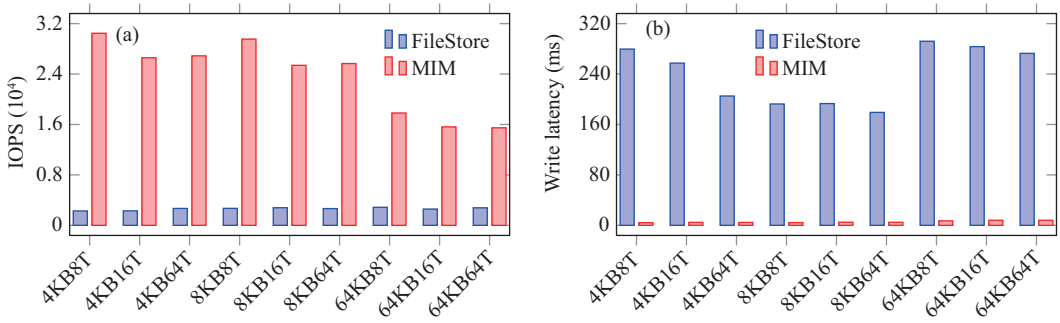


Figure 9 (Color online) Performance comparison of MIM and the original Ceph FileStore under the FIO workload. (a) IOPS; (b) latency.

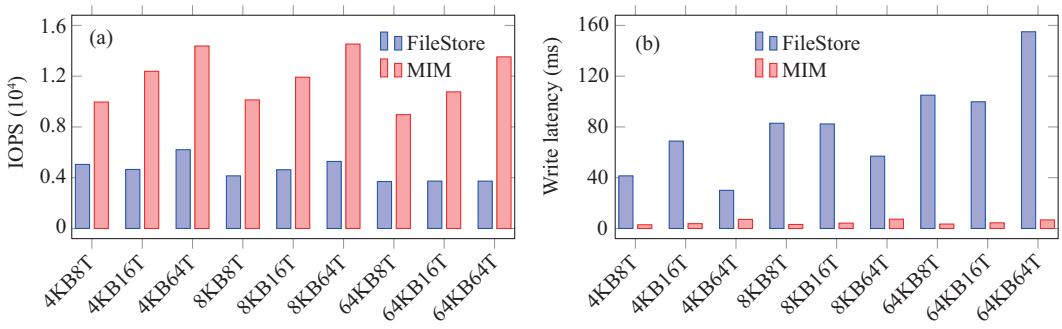


Figure 10 (Color online) Performance comparison of MIM and the original Ceph FileStore under the Varmail workload. (a) IOPS; (b) latency.

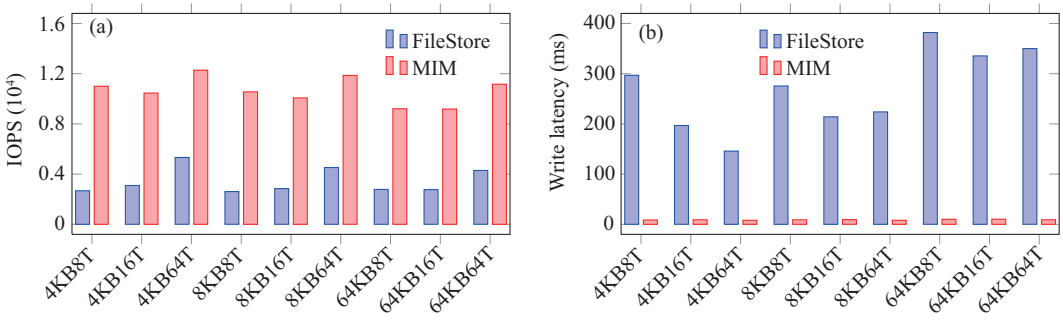


Figure 11 (Color online) Performance comparison of MIM and the original Ceph FileStore under the FileServer workload. (a) IOPS; (b) latency.

• MIM can achieve very low write latencies, less than 10 ms for different settings of microwrite sizes and numbers of threads. The latency reduction of MIM over the original Ceph FileStore is between 97% and 98.5%.

For the Varmail workload, we observe the following from Figure 10.

• The IOPS performance improvement of MIM over the original Ceph FileStore is 1.0×–1.7×, 1.4×–1.8×, and 1.4×–2.9× for microwrites when the microwrite size is 4, 8, and 64 kB, respectively, which is much less than that in the FIO workload case, because there are many operations, such as read and close, besides just write in the Varmail workload, for which MIM has little gain.

• MIM performs better when the microwrite size is smaller, which is similar to the FIO workload case. However, it is less significant because of the complex operations in the Varmail workload.

• MIM performs better when the number of threads increases, which is different from the FIO workload case. This is because the files are in a single directory in the Varmail workload, and the possibility of lock contention is low, such that more threads are affordable for operations at the same time.

• As with the FIO workload case, MIM can achieve less than 10-ms latencies under different settings. The latency reduction of MIM over the original Ceph FileStore is between 76.4% and 96.7%.

For the FileServer workload, we note the following from Figure 11.

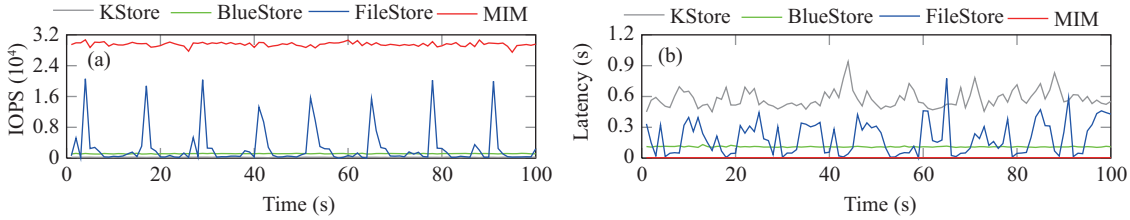


Figure 12 (Color online) Instantaneous performance of IOPS and write latency of the systems under the FIO workload. (a) IOPS performance; (b) write latency.

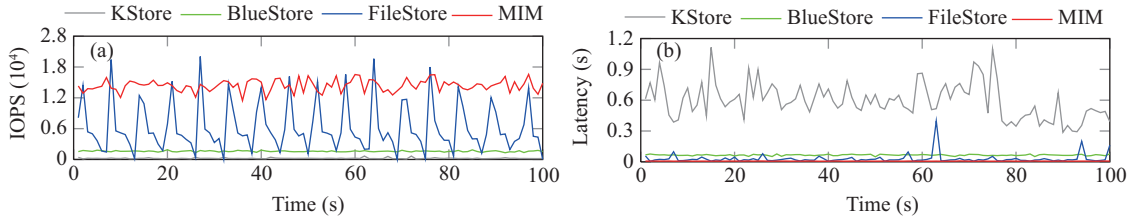


Figure 13 (Color online) Instantaneous performance of IOPS and write latency of the systems under the Varmail workload. (a) IOPS performance; (b) write latency.

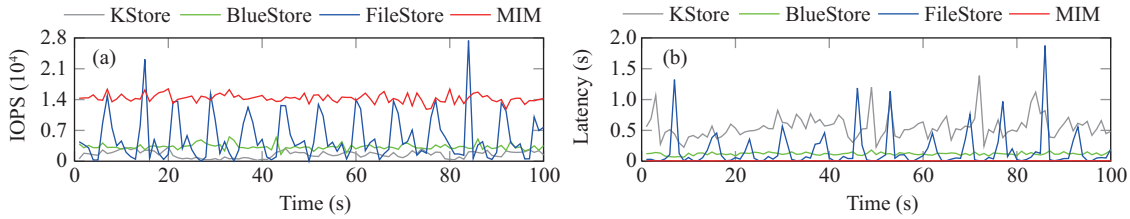


Figure 14 (Color online) Instantaneous performance of IOPS and write latency of the systems under the FileServer workload. (a) IOPS performance; (b) write latency.

• The IOPS performance improvement of MIM over the original Ceph FileStore is 2.3×–3.1×, 2.3×–3.5×, and 1.3×–1.6× for microwrites when the microwrite size is 4, 8, and 64 kB, respectively.

• The IOPS performance of MIM degrades slightly when the microwrite size increases, which is similar to both the FIO workload case and the Varmail workload case.

• The performance of MIM with 8 and 16 threads is similar, but is less than that of 64 threads.

• Same as the previous two workload cases, MIM can achieve less than 10 ms latencies under different settings. The latency reduction of MIM over the original Ceph FileStore is between 94.5% and 97.4%.

Comparing the IOPS performance results under different workloads, we can further see that the improvement in MIM under the FIO workload is higher than that under the FileServer workload, which is higher than that under the Varmail workload. In other words, when the ratio of read-to-write operations is lower, the performance improvement of MIM is higher. This is because MIM does not change the reading process, and thus, has little influence on read performance. Furthermore, note that MIM does not change the commit phase, during which data are written to the NVMe SSD. Thus, the lifetime of NVMe SSD is not influenced by MIM as long as the data amount written to the file system is not changed.

6.3 Comparison with Ceph storage engines

We also conduct experiments to compare MIM with other Ceph storage engines, KStore and BlueStore [7], which do not employ journaling, under the mentioned three types of workloads. In these experiments, the microwrite size is set to 8 kB and the number of threads is set to 8. To analyze the performance and stability, we record the instantaneous performance for the middle 100 s during runtime in terms of IOPS and write latency of MIM, Ceph FileStore, KStore, and BlueStore.

The results under FIO workload, Varmail workload, and FileServer workload are depicted in Figures 12–14, respectively. These figures indicate the following.

• Under all three types of workloads, Ceph FileStore suffers from severe performance fluctuation in terms of both IOPS and latency. After implementing MIM, the performance becomes very stable.

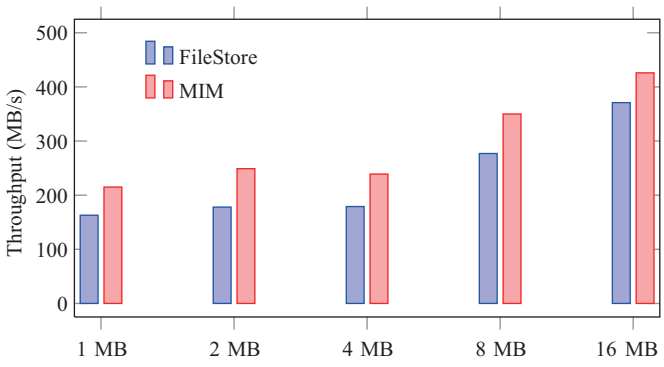


Figure 15 (Color online) Throughput of MIM and FileStore for large writes.

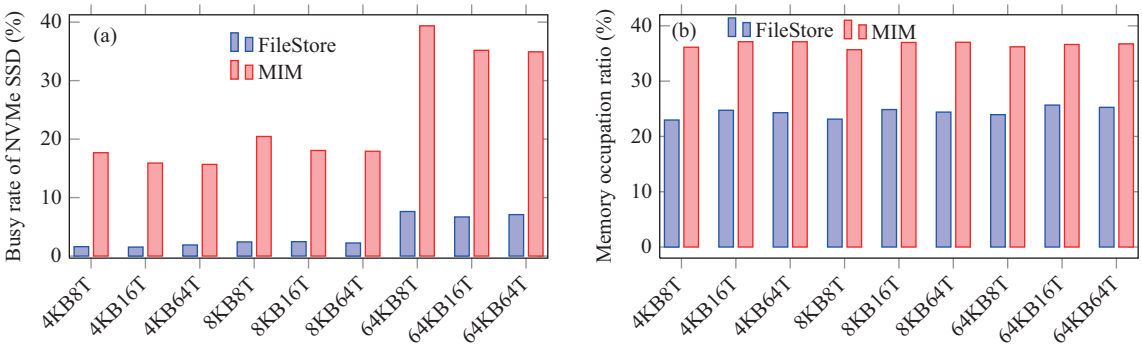


Figure 16 (Color online) (a) Average busy rate of NVMe SSD in MIM and Ceph FileStore; (b) The memory consumption of MIM and Ceph FileStore.

• Under all three types of workloads, the performances of both KStore and BlueStore are much more stable than that of FileStore in terms of IOPS and latency, because they do not use a journaling mechanism. KStore exhibits more fluctuated performance than BlueStore, because the compaction operations within the LSM tree in KStore have different overheads.

• MIM performs much better than KStore and BlueStore in terms of both IOPS and latency.

• Under the Varmail workload and the FileServer workload, the peak IOPS of FileStore is sometimes higher than that of MIM. This can be explained as follows. Both workloads comprise complex operations. When the proportion of write operations is very small, MIM has little gain, but incurs some overhead, which harms the peak performance.

6.4 Large writes

We measure the throughput of sequentially writing a 4 GB file, where the write size ranges from 1 MB to 16 MB. The results of MIM and FileStore are plotted in Figure 15, where MIM outperforms FileStore by 10%–20% for each evaluated value of the write size. Although MIM can hardly provide the benefit of write merging because of large write sizes, which are comparable to a block size of 4 MB, MIM can still act as a high-speed buffer, leading to a slightly higher write throughput.

6.5 Resource utilization/consumption

To see how the NVMe SSD can be utilized with the memory overhead caused by MIM, we measure the SSD busy rate and the memory consumption under the FIO workload (because FIO workload comprises pure microwrite operations). The average busy rates of NVMe SSD and the memory consumption in FileStore and MIM are depicted in Figures 16(a) and (b), respectively. Figure 16(a) shows that MIM improves the utilization of NVMe SSD significantly. However, according to Figure 16(b), MIM only incurs at most 13% additional memory, which is usually affordable in practice.

7 Conclusion

We observed and analyzed the performance fluctuation phenomenon in file systems with journaling on NVMe SSD. Motivated by this, we proposed MIM, a simple yet very effective memory acceleration architecture that can be easily incorporated into existing journaling file systems. MIM introduces a data-structure HTMLL in the memory to support write merging, and employs a flushing scheme to fully exploit the benefit of merging while preventing the journal from growing too long. It adopts a new checkpointing process to preserve data durability. We also implemented MIM in Ceph FileStore. The experimental results obtained on the testbed showed that MIM can provide stable and much higher performance in terms of both IOPS and write latency as compared to the original Ceph FileStore.

Acknowledgements This work was supported in part by National Key R&D Program of China (Grant No. 2018YFB1004704), National Natural Science Foundation of China (Grant Nos. 61832005, 61872171), Natural Science Foundation of Jiangsu Province (Grant No. BK20190058), Key R&D Program of Jiangsu Province (Grant No. BE2017152), Science and Technology Program of State Grid Corporation of China (Grant No. 52110418001M), and Collaborative Innovation Center of Novel Software Technology and Industrialization.

References

1 Tweedie S. Ext3, journaling filesystem. In: Proceedings of Ottawa Linux Symposium, Ottawa, 2000. 24–29
2 Mathur A, Cao M, Bhattacharya S, et al. The new EXT4 filesystem: current status and future plans. In: Proceedings of the Linux symposium, Ottawa, 2007. 21–33
3 Weil S A, Brandt S A, Miller E L, et al. Ceph: a scalable, high-performance distributed file system. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI), Seattle, 2006. 307–320
4 Wei Q, Chen J, Chen C. Accelerating file system metadata access with byte-addressable nonvolatile memory. ACM Trans Storage, 2015, 11: 1–28
5 Sehgal P, Basu S, Srinivasan K, et al. An empirical study of file systems on NVM. In: Proceedings of the 31st International Conference on Mass Storage Systems and Technologies (MSST), Santa Clara, 2015. 1–14
6 Chen C, Yang J, Wei Q, et al. Optimizing file systems with fine-grained metadata journaling on byte-addressable NVM. ACM Trans Storage, 2017, 13: 1–25
7 Lee D-Y, Jeong K, Han S-H, et al. Understanding write behaviors of storage backends in ceph object store. In: Proceedings of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST), Santa Clara, 2017
8 Roselli D S, Lorch J R, Anderson T E, et al. A comparison of file system workloads. In: Proceedings of 2000 USENIX Annual Technical Conference (ATC), San Diego, 2000. 41–54
9 Leung A W, Pasupathy S, Goodson G R, et al. Measurement and analysis of large-scale network file system workloads. In: Proceedings of 2008 USENIX Annual Technical Conference (ATC), Boston, 2008. 2–5
10 Dong M, Ota K, Yang L T, et al. LSCD: a low-storage clone detection protocol for cyber-physical systems. IEEE Trans Comput-Aided Des Integr Circ Syst, 2016, 35: 712–723
11 Li D, Dong M, Tang Y, et al. A novel disk I/O scheduling framework of virtualized storage system. Cluster Comput, 2019, 22: 2395–2405
12 Joo Y, Park S, Bahn H. Exploiting I/O reordering and I/O interleaving to improve application launch performance. ACM Trans Storage, 2017, 13: 1–17
13 Chahal D, Nambiar M. Cloning io intensive workloads using synthetic benchmark. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, L'AQUILA, 2017. 317–320
14 Madireddy S, Balaprakash P, Carns P, et al. Analysis and correlation of application I/O performance and system-wide I/O activity. In: Proceedings of the 12th IEEE International Conference on Networking, Architecture, and Storage (NAS), Shenzhen, 2017. 1–10
15 Li D, Dong M, Tang Y, et al. Triple-L: improving CPS disk I/O performance in a virtualized NAS environment. IEEE Syst J, 2015, 11: 152–162
16 Jannen W, Yuan J, Zhan Y, et al. BetrFS: a right-optimized write-optimized file system. In: Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, 2015. 301–315
17 Best S. Journaling file systems. Linux Magaz, 2002, 4: 24–31
18 Chen J, Tan Z, Wu F, et al. sJournal: a new design of journaling for file systems to provide crash consistency. In: Proceedings of the 9th IEEE International Conference on Networking, Architecture, and Storage (NAS), Tianjin, 2014. 53–62
19 Lee W, Lee K, Son H, et al. WALDIO: eliminating the filesystem journaling in resolving the journaling of journal anomaly. In: Proceedings of 2015 USENIX Annual Technical Conference (ATC), Santa Clara, 2015. 235–247
20 Dua R, Kohli V, Patil S, et al. Performance analysis of union and cow file systems with docker. In: Proceedings of 2016 International Conference on Computing, Analytics and Security Trends (CAST), India, 2016. 550–555
21 Son M, Ahn J, Yoo S. Nonvolatile write buffer-based journaling bypass for storage write reduction in mobile devices. IEEE Trans Comput-Aided Design Integr Circ Syst, 2017, 37: 1747–1759
22 Huang K, Zhou J, Huang L, et al. NVHT: an efficient key-value storage library for non-volatile memory. J Parall Distrib Comput, 2018, 12: 339–354
23 Nightingale E B, Veeraraghavan K, Chen P M, et al. Rethink the sync. ACM Trans Comput Syst, 2018, 26: 6
24 Aghayev A, Ts'o T, Gibson G, et al. Evolving EXT4 for shingled disks. In: Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, 2017. 105–120
25 Rodeh O, Teperman A. ZFS-a scalable distributed file system using object disks. In: Proceedings of 2003 International Conference on Mass Storage Systems and Technologies (MSST), San Diego, 2003. 207–218
26 Rodeh O, Bacik J, Mason C. Btrfs: the linux B-tree filesystem. ACM Trans Storage, 2013, 9: 1–32
27 Chen J, Wang J, Tan Z H, et al. Effects of recursive update in copy-on-write file systems: a BTRFS case study. Can J Electr Comput Eng, 2014, 37: 113–122
28 Choi H J, Lim S-H, Park K H. JFTL: a flash translation layer based on a journal remapping for flash memory. ACM Trans Storage, 2009, 4: 1–22

29 Lee E, Yoo S, Jang J-E, et al. Shortcut-JFS: a write efficient journaling file system for phase change memory. In: Proceedings of 2012 IEEE Conference on Mass Storage Systems and Technologies (MSST), Pacific Grove, 2012. 1–6
30 Chen T-Y, Chang Y-H, Chen S-H. Enabling write-reduction strategy for journaling file systems over byte-addressable NVRAM. In: Proceedings of the 54th International Conference on Design Automation Conference (DAC), Austin, 2017. 1–6
31 O'Neil P, Cheng E, Gawlick D, et al. The log-structured merge-tree (LSM-tree). Acta Inform, 1996, 33: 351–385
32 Shetty P J, Spillane R P, Malpani R R, et al. Building workload-independent storage with VT-trees. In: Proceedings of 11th USENIX Conference on File and Storage Technologies (FAST), San Jose, 2013. 17–30
33 Wu X, Xu Y, Shao Z, et al. LSM-trie: an LSM-tree-based ultra-large key-value store for small data items. In: Proceedings of 2015 USENIX Annual Technical Conference (ATC), Santa Clara, 2015. 71–82
34 Lu L, Pillai T S, Gopalakrishnan H, et al. Wisckey: separating keys from values in SSD-conscious storage. ACM Trans Storage, 2017, 13: 5
35 Griffiths N. nmon performance: a free tool to analyze AIX and linux performance. 2003. https://sourceforge.net/projects/nmon/
36 Son Y, Kim S, Yeom H Y, et al. High-performance transaction processing in journaling file systems. In: Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST), Oakland, 2018. 227–240
37 Rajimwale A, Prabhakaran V, Davis J D. Block management in solid-state devices. In: Proceedings of 2009 USENIX Annual Technical Conference (ATC), San Diego, 2009
38 Tarasov V, Zadok E, Shepler S. Filebench: a flexible framework for file system benchmarking. The USENIX Magaz, 2016, 41: 6–12