PAPER

Copy-on-Write with Adaptive Differential Logging for Persistent Memory

Taeho HWANG[†a)], ***Member and*** **Youjip WON**[††b)], ***Nonmember***

SUMMARY File systems based on persistent memory deploy Copy-on-Write (COW) or logging to guarantee data consistency. However, COW has a write amplification problem and logging has a double write problem. Both COW and logging increase write traffic on persistent memory. In this work, we present *adaptive differential logging* and *zero-copy logging* for persistent memory. Adaptive differential logging applies COW or logging selectively to each block. If the updated size of a block is smaller than or equal to half of the block size, we apply logging to the block. If the updated size of a block is larger than half of the block size, we apply COW to the block. Zero-copy logging treats an user buffer on persistent memory as a redo log. Zero-copy logging does not incur any additional data copy. We implement adaptive differential logging and zero-copy logging on both NOVA and PMFS file systems. Our measurement on real workloads shows that adaptive differential logging and zero-copy logging get 150.6% and 149.2% performance improvement over COW, respectively.

key words: *file system, persistent memory, consistency, COW, logging*

1. Introduction

Traditional file systems like `Ext4`, `XFS`, `Btrfs` and `F2FS` for hard disk drive (HDD) or solid-state drive (SSD) deploy Copy-on-Write (COW) or journaling (logging) to guarantee consistency of data and metadata. Since block devices like HDD and SSD are accessible at block-granularity, traditional file systems update metadata at block-granularity. For example, although COW needs to update 8byte metadata at an internal node to point a new leaf node, block-based COW updates an internal node at block-granularity. In addition, block-based journaling exploits journal log entry at block-granularity to update 128byte or 256byte inode.

As the persistent memory (PM) technology such as STT-MRAM [1], 3D XPoint [2] and NVDIMM [3] emerges, several groups have designed the file systems for hybrid DRAM/PM [4]–[12]. These file systems support several techniques optimized for PM to guarantee consistency of metadata like short-circuit shadow paging [6] and fine-grained logging [5]. These schemes eliminates additional I/O and data copy by updating the metadata at byte granularity rather than at block granularity. Meanwhile, the PM-based file systems pay little attention to consistency of data and deploy the existing techniques to guarantee consistency of data like COW [4], [9] or logging [9]. However, COW has a write amplification problem, and logging has double write and read-tracking problems.

When a `write` system call is issued, COW allocates a new block and performs the write to the allocated block. For partial writes where the write is applied to part of a block rather than an entire block, the remaining area in the block where the write is not performed is filled with the original data. If the write size is smaller than the block size, COW has to copy the original data to a new block. Dulloor et al. [5] pointed out that COW causes a very huge write amplification if the block size is 1GB and the write size is a few hundred megabytes. Also, if a partial write is applied to two blocks instead of one block, COW has to copy the original data to each new block. When analyzing a write trace of some users, a write that is smaller than the block size or an unaligned partial write workload occupies about 63.12% of the total write [13]. In this workload, COW causes a performance degradation due to the write amplification. On the other hand, when a `write` system call is issued, logging copies the original data or new data to the log area. In the case of undo logging, the write is performed twice. For example, data journaling mode of `Ext3` guarantees consistency of file data, but it causes a performance degradation due to the double write [14].

The problems of write amplification and double write occur because the original data is copied to the newly allocated block or log area. In this paper, we propose two novel schemes optimized for PM to guarantee consistency of user data. The proposed schemes alleviate write amplification and double write problems. First, we propose *adaptive differential logging (ADL)* to minimize copying of the original data. We compared the size of data copy caused by COW or logging according to write request size (Table 1). COW generates smaller data copy than logging for a write of multiple blocks. Logging generates smaller data copy than COW for a small write like 1KB. In order to minimize data copy, ADL deploys both COW and logging selectively. When a write request is applied to multiple blocks, ADL selectively deploys COW or logging depending on the write size for each block. Second, we propose *zero-copy logging (ZCL)* to skip copying of original data. Persistent heaps for PM enable user to allocate buuffer from PM. Data on the buffer persists. In order to eliminate copying original data to log area, ZCL treats the user buffer as a redo log when the user buffer is allocated from PM and performs logging without any additional copy.

We evaluated adaptive differential logging and zero-

Manuscript received January 3, 2019.
Manuscript revised July 26, 2019.
Manuscript publicized September 25, 2019.
[†]The author is with Hanyang University, Seoul, Korea.
[††]The author is with KAIST, Daejeon, Korea.
a) E-mail: htaeh@hanyang.ac.kr
b) E-mail: ywon@kaist.ac.kr (Corresponding author)
DOI: 10.1587/transinf.2019EDP7002

copy logging with microbenchmark and macrobenchmark compared to COW. With sequential write workload and 5KB write size, adaptive differential logging and ZCL show performance improvement by 88.4% and 173.8%, respectively. When we run fillseq and overwrite workloads that are generated by key-value library, adaptive differential logging shows performance improvement by 170.5% and 87.2%, respectively. With the trace-based real workload, adaptive differential logging and ZCL show performance improvement by 150.6% and 149.2%, respectively.

2. Background

2.1 Copy-on-Write and Logging

Logging The file systems such as `Ext4` [15], `XFS` [16] deploy logging (or journaling) when updating metadata. Logging is divided into redo and undo logging depending on how it works. Redo logging stores new data to log area first. If a certain time elapses or the log area is insufficient, a check-pointing process is performed to reflect the new data to the file system area. On the other hand, undo logging stores the original data to the log area, and then reflects the new data to the file system area. Undo logging has a double write overhead because it performs additional writes to the log area. In the case of redo logging, since the latest data is in the log area, it has a read tracking overhead. In addition, block-based file systems have to perform I/O and logging (or journaling) at block granularity although the updated data size is a few bytes.

Copy-on-Write The file systems such as `ZFS` [17] and `Btrfs` [18] deploy Copy-on-Write (COW) when updating tree-based metadata. When a `write` system call is issued, COW performs updates out-of-place. In block-based file systems, updating a single leaf node incurs updating other nodes from a leaf node to a root node. In COW, updating the tree has a cascading overhead. When a few data in the leaf node is updated, COW also has a huge write amplification.

2.2 Persistent Memory

Since block devices such as hard disks are accessible at block granularity, legacy file systems manage metadata and data at block granularity. Persistent memory (PM) such as STT-MRAM [1], 3D XPoint [2] and NVDIMM [3] is accessible at byte granularity and guarantees persistence of data. Several researchers designed file systems on PM that manage metadata at byte granularity [4]–[12]. PM is connected to the memory bus like DRAM, and processors access PM directly. Modern processors support 8byte atomic writes. The order of store instructions applied to the PM may change due to a cache in the processor. PM-based file systems flush a particular cacheline through some instructions such as `clflush` or `clwb` to ensure ordering between store instructions.

2.3 File Systems for Persistent Memory

NOVA is a log-structured file system [4]. Legacy log-structured file systems keep log area linear and suffer log cleaning overhead. In order to resolve the overhead of keeping the log area linear, NOVA manages the log area as singly linked list per `inode`. NOVA adds a log entry to the log area. After adding the log entry, NOVA updates a tail pointer pointing to the end of the log area and uses the last log entry as the latest version. By atomically updating the 8byte tail pointer, NOVA guarantees the consistency of file system operations, including `write`. In addition, NOVA guarantees data consistency through COW.

BPFS is a file system that provides shadow paging to ensure consistency of metadata and data [6]. In order to solve the cascading update in legacy shadow paging, BPFS proposed short-circuit shadow paging. When a `write` system call is issued to a file, BPFS updates a pointer in the lowest node where only one pointer is updated. Updating the pointer is performed atomically by deploying 8byte atomic write.

PMFS is a file system that provides logging to ensure consistency of metadata [5]. In order to solve the overhead caused by legacy block-level logging, PMFS provides fine-grained logging by utilizing the accessibility of PM at byte granularity. In the fine-grained logging, a transaction consists of `START`, `DATA` and `CHECKPOINT` log entries. The size of the log entries is a cacheline. The transaction with the `CHECKPOINT` log entry is considered to be completed without system failure. The layout of both PMFS and BPFS is b-tree.

2.4 The Size of Data Copy

We compared the size of data copy that occurs in the PM-based file systems according to a write size requested by an user and consistency schemes. We assumed that COW generates minimal write to the internal node with the short-circuit shadow paging [6]. If the fine-grained journaling [5] is deployed for COW, the writes to the internal node can be reduced. In this comparison, we did not consider a write to the internal node. Table 1 shows the comparison result. If the write size is smaller than half of the block size, such as 1KB, logging generates a smaller data copy than COW. In the case of an aligned write, COW generates a smaller data copy than logging when the write size is larger than half of the block size, such as 3KB. In addition, if a write size is much larger than the block size, such as 64KB, COW

Table 1 The size of data copy according to consistency scheme and write size (block size = 4KB)

	Write size (KB)				
	1	2	3	4	64
COW (Aligned)	4	4	4	4	64
COW (Unaligned)	8	8	8	8	68
Logging	2	4	6	8	128

generates a very small data copy than logging. However, in the case of an unaligned write that a 3KB partial write that is applied to the multiple blocks, COW generates a 8KB data copy. This is larger than logging, which generates a 6KB data copy.

Although the size of updated data is less than block size, block-based file systems execute write at block-granularity. However, since PM is accessible at byte granularity, PM-based file systems are able to execute write at the size of updated data at byte granularity. Byte accessibility of PM enables fine-grained logging. For example, in order to process 1KB write, block-based logging generates 8KB IO for original data and new data. PM-based logging generates 2KB data copy. In case of COW, both block-based COW and PM-based COW generate 4KB IO or data copy for 1KB write. In this case, we only focus on user data at a leaf node.

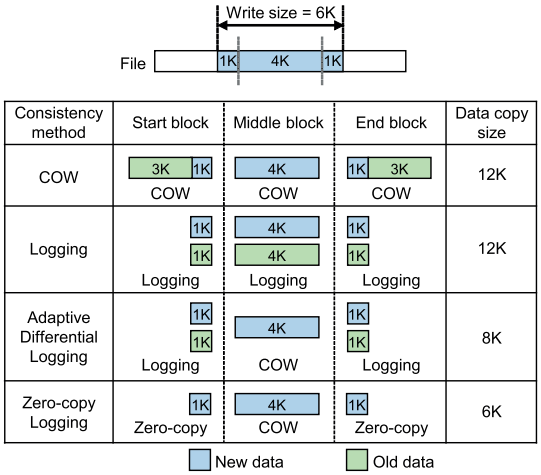


Fig. 1 Comparison of consistency scheme (unit = byte)

3. Proposal

We propose *Adaptive Differential Logging* (ADL) and *Zero-copy Logging* (ZCL). ADL applies fine-grained logging and COW selectively to each block. ZCL exploits an user buffer on persistent memory as log area. ADL reduces data copy and ZCL eliminates data copy. When write request is issued, we check the persistence of the user buffer. If the user buffer is on volatile memory, we apply ADL for the write. If the user buffer is on persistent memory, we apply ZCL for the write.

3.1 Adaptive Differential Logging

Usually, if the write size is larger than half of the block size, COW generates a small data copy. However, in the case of small partial writes that are not aligned and applied to multiple blocks, COW may generate more data copies than logging. Thus, we propose *adaptive differential logging (ADL)* that selectively applies different consistency schemes to each block according to the write size to the block, rather than the write request size. When the write size to the block is smaller than or equal to half of the block size, logging is applied to the block. And, when the write size to the block is larger than half of the block size, COW is applied to the block. If the write request size is larger than the block size and the write is applied to several blocks, COW or logging is selectively applied to each block. When write is applied to three or more blocks, the write of block size is applied to all the middle blocks except the start and end blocks. Therefore, we apply COW to all the middle blocks and check the write size of the start and end blocks. If the write size of the start and end blocks is smaller than or equal to half of block size, we perform logging on that block. In ADL, we adopted undo logging.

3.2 Zero-Copy Logging

In addition to file systems for hybrid DRAM/PM, persistent heaps [19]–[22] are being proposed. Schemes for persistent heap suggest a new programming model for PM, such as `pmalloc` [21] and `libvmmalloc` [23]. Since data in persistent heap is stored in PM, the data persists in the case of system failure. We propose *zero-copy logging (ZCL)* that exploits the user buffer as a redo log. If the user buffer is allocated from the persistent heap, we perform ZCL. Because the user buffer is used as a redo log, ZCL writes the user data to the file in-place without any additional logging. In the case of normal redo logging, the user data is first written to the log area. On the other hand, ZCL only records information about the persistent user buffer to the log area. A `write` system call of block-based file system is returned to user after storing user data to a buffer cache. User data in the buffer cache is flushed to block device periodically or by a `sync` system call. On the other hand, PM-based file systems store user data to PM directly. After a `write` system call is returned, file block contains the latest data. Thus, ZCL does not require read-tracking, unlike normal redo logging. Also, unlike undo logging, it does not have double write problems.

3.3 Amount of Transaction

Figure 1 illustrates the size of data copy according to each consistency method when 6KB write is applied to several blocks. In the case of COW, a data copy is generated as much as the block size for each block. In the case of logging, twice as many data copies are generated for each block. In the case of ADL, logging is applied to the start block and the end block, and COW is applied to the middle block. ZCL also performs logging to the start block and the end block. However, since the user buffer is located on PM and used as a redo log, we do not perform additional data copy for logging in the case of ZCL.

4. Design

In this section, we describe adaptive differential logging with an example of a log-structured file system like NOVA [4] and zero-copy logging with an example of a file

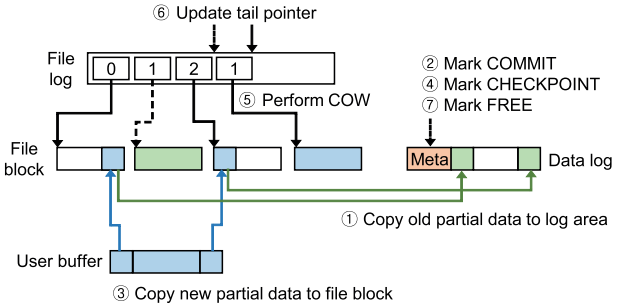


Fig. 2 Process of COW with adaptive differential logging in the case of a log-structured file system

system with b-tree layout like PMFS [5].

4.1 Adaptive Differential Logging

In a log-structured file system, such as NOVA [4], one transaction for `write` is completed by atomically updating the tail pointer that points the end of the log area. In a file system with b-tree layout, such as BPFS [6], one transaction for `write` is completed by atomically updating a pointer of the lowest node where only one pointer is updated. Unlike NOVA and BPFS, which guarantee data consistency through COW, PMFS [5] propose a method to guarantee data consistency through COW in design. In the case of PMFS, if multiple node pointers are updated, we can log node pointers in the internal node with fine-grained logging and complete a transaction for `write` by adding a `CHECKPOINT` log entry to the log area. If only one node pointer is updated, we can deploy a 8byte atomic write like BPFS.

In the case of PMFS, we can complete both fine-grained logging for COW and ADL simultaneously by adding one `COMMIT` log entry to the log area. In the case of NOVA and BPFS, we make a log entry COW-aware to operate ADL with COW that updates 8byte pointer value atomically. The log entry for ADL contains the pointer value for COW and its address before COW is performed. We can identify whether COW is completed or not through the pointer value. If the pointer value in the log entry is different with current pointer value that is updated by COW, we assume that COW is completed. We can acquire the current pointer value that is updated by COW through the address field in the log entry. On the other hand, we can identify whether ADL is completed or not through the `CHECKPOINT` log entry. If the `CHECKPOINT` log entry exists in the log area, we assume that ADL is completed. We perform recovery if either COW or ADL is not completed.

Figure 2 illustrates the process of ADL in the case of NOVA. The state field of a log entry has one of the following values: `FREE`, `COMMIT` and `CHECKPOINT`. The `FREE` mark indicates that the log entry is not in use. The `COMMIT` mark indicates that old file data is stored in log area before performing partial write. This mark also indicates that we are capable of performing undo recovery. The `CHECKPOINT` mark indicates that the partial write has been performed. When servicing `write` request, we first store old file data to log area (①) and then write the `COMMIT` mark to the log entry atomically (②). At this point, the log entry contains the old pointer value before COW is performed. After writing new file data in user buffer to the start or end blocks partially (③), we write the `CHECKPOINT` mark to the log entry atomically (④). We now perform COW (⑤), and then write new pointer value atomically (⑥).

The following shows whether recovery is performed or not according to the process in Fig. 2 for ADL. (i) If system failure occurs between ② and ④, the state field of the log entry is `COMMIT`. In this case, we assume that system failure occurs during partial overwriting. Blocks for ADL can have old data or can have new data. Blocks for COW have old data. Thus, we perform undo recovery. (ii) If system failure occurs between ④ and ⑥, the state field of the log entry is `CHECKPOINT`. However, the pointer value of the log entry is the same as the current pointer value for COW. In this case, we assume that system failure occurs after completing logging and before completing COW. Blocks for ADL have new data, but blocks for COW have old data. Thus, we perform undo recovery. (iii) If system failure occurs between ⑥ and ⑦, the state field of the log entry is `CHECKPOINT`. And, the old pointer value of the log entry is not the same as the current pointer value. In this case, we assume that system failure occurs after completing logging and COW, but the log entry is not retrieved. Blocks for ADL and COW have new data. Thus, we do not perform undo recovery. (iv) If system failure occurs before ② or after ⑦, the state field of the log entry is `FREE`. In this case, we assume that system failure occurs before partial overwriting or after completing write. Blocks for ADL and COW have old data before write or new data after write. Thus, we do nothing.

4.2 Zero-Copy Logging

Data in the user buffer may remain in the CPU cache before reaching PM. Therefore, before using the user buffer as a redo log, the user buffer must be flushed to PM via `clflush` or `clwb`. ZCL generates a data copy as much as the write size, and flushes twice as much as the write size. In our experiments, we confirmed that ZCL had lower performance than COW when the write size is larger than or equal to the block size. This is because the larger the write size, the greater the overhead of flushing the user buffer. Therefore, when the write size to the block is larger than or equal to the block size, ZCL is applied to the block like ADL.

Figure 3 illustrates the process of ZCL in the case of BPFS and PMFS [5]. In the Fig. 3, we considered the situation that only one of node pointers in the lowest internal node is updated. If several node pointers are updated, node pointer in higher node or root node pointer are updated atomically and the pointer value is stored in the log entry. The state field of a log entry has one of the following values: `FREE` and `COMMIT`. The `FREE` mark indicates that the log entry is not in use. The `COMMIT` mark indicates that the information of a persistent user buffer is recorded to log area.

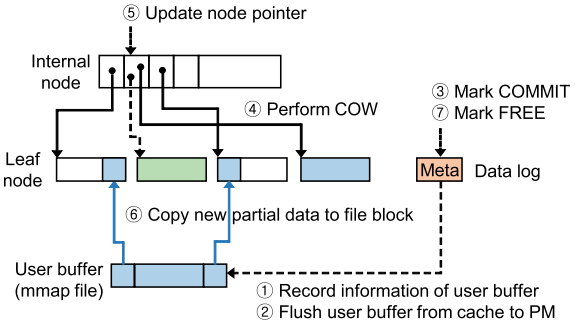


Fig. 3 Process of COW with zero-copy logging in the case of a file system with b-tree layout

This mark also indicates that we are capable of performing redo recovery. When servicing `write` request, we first store the information of user buffer that contains the latest user data to log area (①). As the data in the user buffer has not been applied to PM and can reside in CPU cache, we flush the user buffer (②). We write the `COMMIT` mark to the log entry atomically (③). At this point, the log entry contains the old pointer value before COW is performed. After performing COW (④), we write new point value atomically (⑤). Finally, after writing new file data in user buffer to the start or end blocks partially (⑥), we write the `FREE` mark to the log entry atomically (⑦).

The following shows whether recovery is performed or not according to the process in Fig. 3 for ZCL. (i) If system failure occurs between ③ and ⑤, the state field of the log entry is `COMMIT`. However, the pointer value of the log entry is the same as the current pointer value. In this case, we assume that system failure occurs before completing COW. Blocks for COW and ZCL have old data. Thus, we do not perform redo recovery although we have the redo log. (ii) If system failure occurs between ⑤ and ⑦, the state field of the log entry is `COMMIT`. And, the old pointer value of the log entry is not the same as the current pointer value. In this case, we assume that system failure occurs after completing COW. Blocks for COW have new data, but blocks for ZCL can have old data. Although blocks for ZCL have new data, we do not know. Thus, we perform redo recovery. (iii) If system failure occurs before ③ or after ⑦, the state field of the log entry is `FREE`. In this case, we assume that system failure occurs before logging or after completing write. Blocks for COW and ZCL have old data before write or new data after write. Thus, we do nothing.

5. Implementation

We implemented ADL and ZCL on a log-structured NOVA file system [4] and a PMFS file system [5] with b-tree layout. In the case of PMFS, we implemented COW by using a 8byte atomic write and fine-grained logging. We implemented logging that operates in conjunction with COW. In this paper, we assumed the 8byte write is the minimum unit of atomic write for PM.

Usually, persistent heaps [19], [20] deploy a `mmap` system call to allocate the user buffer from PM. When servicing the `mmap` system call in the file system, we managed which process requested the mapping and the mapping information in a list (*zcl_mmap_list*). When servicing a `write` system call, we checked if the user buffer is inside file-mapped address space via the *zcl_mmap_list*.

ZCL exploits the user buffer as log area. In order to apply ZCL to file system, the file system should be capable of identifying whether the user buffer is on PM or not. In current implementation, ZCL has a limitation. In some case, we cannot identify the persistence of the user buffer. Currently, the file system checks if the user buffer is mmapped-file or not. If the user buffer is mmaped-file, we can ensure the persistence of the user buffer. The user buffer can be allocated from PM through native persistent heap [22] that does not depend on file system. If the user buffer is on native persistent heap, we cannot ensure the persistence of the user buffer. Although ZCL can work for the user buffer, we apply ADL for the write.

5.1 Logging

We reserved part of file system for ADL and ZCL for logging. We assigned a separate log area for each core to minimize synchronization overhead. This is the same as NOVA's per-core journaling. The size of log entry containing the information of logged data is cacheline size. The number of log entries in the per-core log area is 64 by default. Therefore, one core can simultaneously perform ADL and ZCL for 64 writes. We allocate and deallocate log entries into circular queues. Updating head and tail pointers of the circular queue with allocation information at PM requires consistency method and generates frequent flushes. Thus, we keep head and tail pointers in DRAM, not PM. When file system attempts to recover the system after system failure, we check the status of all log entries in the log area and reconstruct the circular queue.

When the write is applied to three or more blocks, we perform logging on start and end block. Even if logging is applied to both blocks, the size of logging data is smaller than or equal to one block. Therefore, we use one log block for one write. Half of log block is used as log area for start block and the other half is used as log area for end block. In logging of some file systems such as PMFS [5], `START` log entry is added at the beginning of log area that is reserved for the transaction, and then log entries with old data are added, and finally `COMMIT` log entry is added. On the other hand, ADL uses only one log entry in order to reduce log tail contention that occurs when a lot of log entries are added simultaneously. We do not add the separate `START`/`COMMIT` log entry. Instead, we mark state field in log entry as `COMMIT` according to the state of transaction. As a result, we use one log entry and one log block for one write. In addition, we manage log entries in cacheline size and log blocks separately.

Usually, block-based file systems store data in a buffer before writing data to a block device. Due to I/O scheduling,

reordering may occur in the process of flushing buffered data to a block device. On the other hand, PM-based file systems usually copy user data directly to a file block without using a buffer when processing `write` system call. So, file block has the latest data after `write`. This process is a `CHECKPOINT` from a logging point of view. Before `write` system call is returned, we change the state of a log entry to `FREE`. We do not have a separate garbage collector to retrieve free log entries. We try to retrieve log entries when assigning log entries. We check the state of a log entry that a log head is pointing to and increase the log head pointer until finding a non-free log entry.

NOVA performs write on a file after acquiring a lock of the file. Since the file data is protected by the lock, one or more writes to the same file cannot be performed at the same time. When ADL is performed, the corresponding lock prevents COW or logging by different `write` system calls from being simultaneously applied to the same block. Therefore, we do not use a separate lock to prevent simultaneous COW or logging to the same block. But, in the case of PMFS, application should use file lock to protect file block that is being accessed by multiple process.

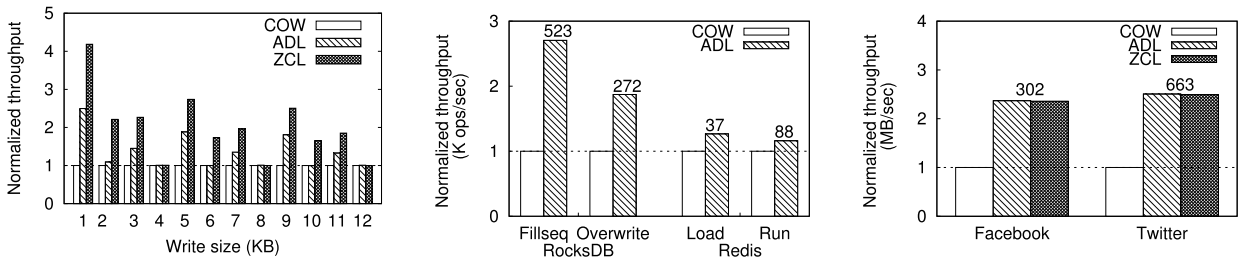
We reserved part of file system for ADL and ZCL for logging. If we find a log entry that is not in free state during recovery, we copy the old data from source kernel address in the log area to destination kernel address in the file. During ADL and ZCL, we record destination kernel address and the size of the log data. In the case of ADL, we managed the log block to the corresponding the log entry in a fixed location. Thus, we know destination kernel address in the fixed log block. On the other hand, in the case of ZCL, the file-mapped user buffer is used as the redo log and the file block is used as the log block. Thus, we record `inode` number of the mapped file and file offset from which we can extract destination kernel address because the log block to the corresponding log entry is not in the fixed location.

6. Evaluation

In this section we evaluated the performance of ADL and ZCL. We used iozone, db_bench and ycsb as microbenchmark and mobibench (with mobigen) as macrobenchmark. We compared ADL and ZCL against COW. We selected NOVA [4] and PMFS [5] file system that deploys COW for data consistency. In the case of PMFS, we also implemented COW. We configured 4KB as the block size. The experimental testbed consists of Intel (R) Core (TM) i7-6700 CPU @ 3.40GHz (8 processor, 8MB) CPU and 32GB RAM. We configure 24GB of 32GB RAM as the PMEM [24] to emulate PM.

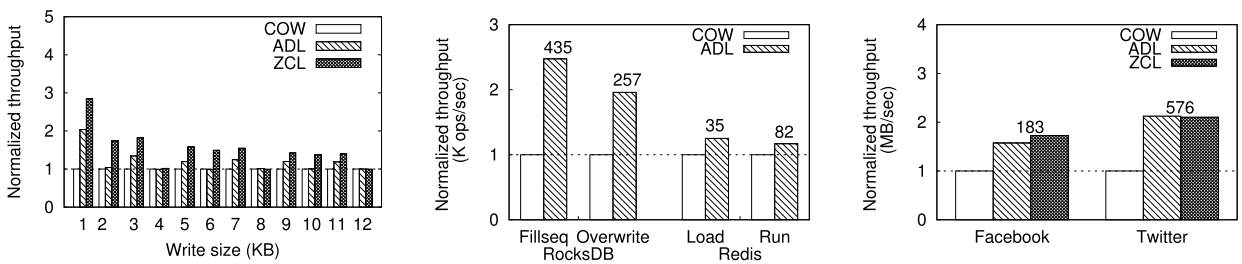
The benchmarks allocate the user buffer from volatile memory by issuing malloc(). For the evaluation of ADL, we did not modify the benchmark. For the evaluation of ZCL, we modified benchmark so that it uses the mmap-based user buffer like `libvmmalloc` [23] which is based on the `C` programming language, but we did not apply ZCL for some benchmark that uses the `Java` programming language.

In this section, we described the figures of NOVA. PMFS has similar trend and figures of NOVA. In Fig. 4 and Fig. 5, the figures above the bar mean the measured throughput.



(a) IOzone (b) Key-value library/store (c) Mobibench with mobigen

Fig. 4 Normalized throughput of NOVA



(a) IOzone (b) Key-value library/store (c) Mobibench with mobigen

Fig. 5 Normalized throughput of PMFS

6.1 Microbenchmark for Filesystem

We used sequential writes with a variety of write sizes from 1KB to 12KB for a pre-generated 1GB file and ran `IOzone` benchmark [25]. Figure 4 (a) shows the normalized throughput to COW. With the 1KB write size, ADL and ZCL show performance improvement by 149.7% and 318.5%, respectively, compared to COW. COW copies 3KB original data and 1KB new data. It generates a total data copy of 4KB. ADL copies 1KB original data and 1KB new data. It generates a total data copy of 2KB. ZCL copies 1KB new data. It generates a total data copy of 1KB. With the 2KB write size, COW, ADL and ZCL generate 4KB, 4KB and 2KB data copy, respectively. COW copies 2KB original data and 2KB new data. It generates a total data copy of 4KB. ADL copies 2KB original data and 2KB new data. It generates a total data copy of 4KB. ZCL copies 2KB new data. It generates a total data copy of 2KB. With the 3KB write size, COW generates between 4KB and 8KB data copy alternately. When 3KB write is applied to one block, ADL and ZCL apply COW to the block. When a 3KB write is applied to two blocks, ADL and ZCL apply logging to each block. Therefore, ADL and ZCL generate 6KB and 3KB data copy, respectively. If the write size is a multiple of 4KB, ADL and ZCL apply COW to every block. With the 5KB write size, the write is applied to two blocks. In this case, COW always generates a 8KB data copy. On the other hand, ADL and ZCL apply COW and logging to each block selectively. ADL generates between 6KB and 8KB data copy alternately. ZCL generates between 5KB and 6KB data copy alternately. With the 5KB write size, ADL and ZCL show performance improvement by 88.4% and 173.8%, respectively.

6.2 Microbenchmark for Key-Value Library/Store

RocksDB [26] is a key-value library for fast storage such as flash or RAM. RocksDB provides its own benchmark called `db_bench` for performance test. We configured the key size, the value size and the number of entries as the default values. The key size is 16B, the value size is 100B, and the number of entries is 1,000,000. We ran `fillseq` and `overwrite` workload. First, we ran the fillseq workload to insert the key into the database. The workload inserts the key into the database file in sequential order. Second, we ran the overwrite workload to update the value of the inserted key. The workload updates the value for the randomly selected key. In Fig. 4 (b), the graph on the left shows the normalized throughput to COW with RocksDB. For fillseq and overwrite workloads, ADL shows performance improvement by 170.5% and 87.2%, respectively. The percentage of writes whose write sizes are smaller than half of the block size during the entire write is 99.6% for both fillseq and overwrite workloads. Also, the corresponding write size is mostly 138B. Unlike fillseq workload, overwrite workload performs 2KB read 41K times, resulting in smaller performance improvement than fillseq workload.

Redis [27] is an in-memory key-value store. Redis provides snapshot and AOF (Append-Only-File) options for persistence. Snapshot saves the dataset to a file for each interval. On the other hand, AOF logs all write operations to a file. We set Redis to the AOF option to check the effect of the file system on Redis persistence for each write operation. We ran `YCSB` benchmark [28] to measure the performance of Redis. YCSB provides a core workload for a cloud system. We used `workload A` consisting of 50/50 reads and writes. Atikoglu et al. [29] collected traces of Facebook's Memcached and confirmed that the value of 90% in the trace is smaller than 500B. Therefore, we set the value size to 250B, because the small size value is a common value in the key-value store. The number of entries is 5,000,000. After we loaded the database, we ran the workload. In Fig. 4 (b), the graph on the right shows the normalized throughput to COW with Redis. For load and ran workloads, ADL shows performance improvement by 26.6% and 16.1%, respectively. The percentage of writes whose write size is smaller than half of the block size during the entire write is 95.2% and 99.9% for load and ran workloads, respectively. Also, the corresponding write size is mostly 315B.

6.3 Macrobenchmark

Campello et al. [13] collected the user's write traces and analyzed that 63.12% of write is smaller than 4KB write. We conducted experiments on two IO traces that can be acquired among those traces. The IO traces [30] were extracted while the user approached facebook and twitter. We replayed the trace through mobibench [31]. Figure 4 (c) shows the normalized throughput to COW. For facebook trace, ADL and ZCL show performance improvement by 136.9% and 136%, respectively. For twitter trace, ADL and ZCL show performance improvement by 150.6% and 149.2%, respectively. The percentage of writes whose write sizes are smaller than half of the block size in the IO trace is 81.4% and 77% for facebook and twitter traces, respectively. In addition, the ratio of write with size 1B or 4B in the IO trace are 66.5% and 99.1% for facebook and twitter traces, respectively. With 1B or 4B write size, ADL and ZCL perform almost the same size memory copy and flush. Therefore, the performance improvements of the two methods are similar for the trace.

6.4 Amount of Transaction

ADL and ZCL enhance write performance by reducing and eliminating the data copy of user data for write transaction. Table 2 illustrates the total amount of write transaction according to each workload.

We measured the amount of transaction in three workload of microbenchmark; IOzone, RocksDB and Redis. In the case of IOzone, it performs write requests to 1GB file with 1KB write size. In IOzone, the size of total write is 1GB. As COW performs 4KB data copy for each write request, the size of total transaction is 4GB. In ADL, the size

Table 2 The amount of transaction for user data (unit: byte)

	Total Write	COW	ADL	ZCL
IOzone (1KB)	1G	4G	2G	1G
RocksDB (Fillseq)	131M	3.9G	263M	-
Redis (Run)	750M	9.8G	1.5G	-
Mobibench (Facebook)	11.9M	57.5M	12.7M	11.9M
Mobibench (Twitter)	4.8M	21.7M	4.8M	4.8M

of total transaction is 2GB because the amount of data copy is twice the write request. As ZCL does not perform additional data copy, the size of total transaction is 1GB. In the case of RocksDB, it performs write requests 1,003,476 times and the size of total write is 131MB. The size of most write requests is 138Byte. As COW performs 4Kb data copy for 138Byte, the size of total transaction is 3.9GB. As ADL performs twice data copy, it generates 263MB data copy in total transactions. In the case of Redis, it performs write requests 2,501,170 times and the size of total write is 750MB. The size of most write requests is 315Byte. COW performs 4KB data copy for 315Byte and ADL performs 630Byte data copy for 315Byte.

We measured the amount of transaction in two workload of macrobenchmark; facebook and twitter. In the real workload of both facebook and twitter, some of write is smaller than half of page and some of write is as large as 4KB page. For write that is smaller than half of page, COW performs amplified data copy. For 4KB write, COW performs as much as write request. In the case of facebook workload, COW amplifies some of write from 825KB to 46.5MB in total. The total size of 4KB write is 11MB. So, COW generates 57.5MB (46.5+11) transaction in total. ADL amplifies some of write from 825KB to 1.7Mb in total and generates 12.7MB (1.7+11) transaction in total. As ZCL does not amplify write request, it generates about 11.9MB transaction that is the same size of write request in total.

6.5 Sensitivity to Write Latency of Persistent Memory

The read and write latency of persistent memory are expected to be asymmetric and the write latency is expected to be longer than the read latency. We simulated the write latency of persistent memory and measured the write performance according to each consistency scheme. We inserted delays after each `clflush` instruction for emulation of the write latency. For IOzone, we issued 1KB write requests to 1GB file. Figure 6 illustrates the write performance according to the write latency (delay). For 1KB write request, COW performs 4KB data copy and 4KB flush. ADL performs 2KB data copy and 2KB flush. Although ZCL does not generate additional data copy, it performs 1KB data copy and 2KB flush because it flushes the user buffer from CPU cache to persistent memory to ensure the persistence of the user data. As the write latency increases, the flush overhead of ZCL also increases and thus the performance of ZCL decreases. When there is no penalty of write latency, the performance of each consistency scheme depends on the amount of data copy. As the write latency increases, the performance of each consistency scheme depends on the amount of flush, not data copy.

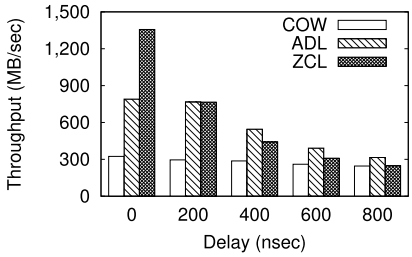


Fig. 6 Performance according to latency

6.6 Recovery Overhead

In order to recover the system from power failure quickly, the recovery overhead of consistency scheme should be low. We measured an additional recovery overhead of proposed consistency scheme. We selected NOVA as a target file system and fileserver workload of `filebench` [32]. In the fileserver workload, the user data and metadata for file system are generated and updated. We set the number of files to 5,000, the file size to 1MB and the IO size to 64KB. For simulation of power failure, we rebooted the system compulsorily while fileserver workload is being executed. After rebooting, we measured the mount time of the inconsistent file system. After recovery, legacy NOVA consumes 12,024usec for mounting and NOVA with ADL consumes 12,057usec for mounting. The recovery of ADL does not affect the recovery process of the file system. ADL creates log entry before it begins write request and then it frees log entry after it completes write request. So, there are as many valid log entries as the number of core. Compared to the number of write transactions, there are a few valid log entries for ADL and the recovery overhead of ADL is trivial.

7. Related Work

DiffTx [33] supports WAL and shadow paging differentially to reduce write traffic on flash-based SSD like ADL. DiffTx applies logging to pages to which partial update occurs and applies shadow paging to pages to which full update occurs. However, since DiffTx does not target PM, ZCL cannot be applied. In addition, DiffTx does not apply different consistency method to each block which is covered by a write. It applies to different consistency method according to a write.

OSP [34] proposes shadow paging with fine-grained persistence to resolve both double copy overhead of logging and write amplification of shadow paging. Although decreasing the size of a page can reduce the overhead of write amplification, it incurs increase in virtual-to-physical mapping information and lookup time of page table. So, OSP supports shadow paging with fine-grained persistence by exploiting cache line-level mapping. But, since OSP requires an additional 2bit per cacheline, it needs to modify

hardware.

Strata [11] is a cross-media file system that utilizes the advantages of PM, SSD and HDD. Strata logs data in PM for fast write. Strata directly accesses if data to be read is on PM, and caches file data in file data cache on DRAM for fast read if it is in SSD. Even if a small write smaller than block size is issued, strata stores only the latest data in the log and does not generate 4KB block having only the latest data in order to avoid write amplification overload caused by COW. However, if read is issued in the same block and there is no data to be read in the log, strata creates a block with the latest data in file data cache and returns it to an user. Therefore, strata sacrifices read performance for small write performance when read is issued on the same block where write was issued. On the other hand, ADL keeps blocks with the latest data after write. Therefore, in ADL, small write does not affect read performance. In addition, strata converts the data stored in the log into a read-optimized tree for fast read and stores it in PM. At this time, when data smaller than block size is stored in log due to a small write, strata performs digest by COW method.

NOVA-fortis [35] is file system based on NOVA file system. NOVA-fortis provides snapshot and reliability functions. NOVA-fortis maintains log entry with valid snapshot id and provides snapshot by using the corresponding log entry rather than the latest log entry. In addition, NOVA-fortis guarantees the reliability of the file system through checksum and parity. Ext4-DAX [36] provides the ability for users to directly access file data without going through the page cache. Ext4-DAX ensures consistency of metadata through journaling. However, since the user directly accesses the file data, it does not ensure data consistency. Aerie [7] is file system that allows PM access at the user level. Since PM is accessed without going through the kernel, Aerie ensures fast I/O performance. SCMFS [8] is file system that contains the file address space in the process virtual address space. SCMFS removes mapping information in file address space and file system address space because files are managed in virtual address space continuously. SCMFS also manages file system blocks by reusing memory management modules. SoupFS [37] is file system that is based on soft updates. PM-based file systems can suffer performance degradation due to cacheline flush. SoupFS maintains pointer-based dual views to improve performance by delaying synchronous flushes.

8. Conclusion

PM-based file systems provides PM-optimized consistency scheme, but they focus on metadata consistency. We focus on data consistency to resolve the write amplification overhead of COW and the double write overhead of logging. According to the write size to each block, the performance of COW and logging is different. In this work, we present adaptive differential logging (ADL) to minimize a data copy and zero-copy logging (ZCL) to eliminate a data copy. Our measurements show that COW with ADL and ZCL resolves the write amplification overhead of COW. Instead of applying one consistency method, adopting two different consistency methods adaptively shows better performance.

Acknowledgments

This work is funded by NRF (No.2017R1A4A1015498, Scalable I/O Stack for Future High Performance Storage), IITP (No.2018-0-00549, Extremely Scalable Order-preserving Operating System for Manycore and Non-volatile Memory) and IITP (No.2019-0-00118, Research and Development on Memory-Centric OS Technologies of Unified Data Model for Next-Generation Shared/Hybrid Memory).

References

[1] A.V. Khvalkovskiy, D. Apalkov, S. Watts, R. Chepulskii, R.S. Beach, A. Ong, X. Tang, A. Driskill-Smith, W.H. Butler, P.B. Visscher, D. Lottis, E. Chen, V. Nikitin, and M. Krounbi, "Basic principles of stt-mram cell operation in memory arrays," Journal of Physics D: Applied Physics, vol.46, no.7, p.074001, 2013.
[2] Intel, "Intel and Micron produce breakthrough memory technology," 2015. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/
[3] Micron, "Nvdimm," 2017. https://www.micron.com/products/dram-modules/nvdimm#
[4] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," Proc. USENIX FAST, 2016.
[5] S.R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," Proc. ACM EuroSys, 2014.
[6] J. Condit, E.B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," Proc. ACM SOSP, 2009.
[7] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M.M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," Proc. ACM EuroSys, 2014.
[8] X. Wu and A.L.N. Reddy, "Scmfs: a file system for storage class memory," Proc. ACM/IEEE SC, 2011.
[9] J. Ou and J. Shu, "Fast and failure-consistent updates of application data in non-volatile main memory file system," Proc. MSST, 2016.
[10] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," Proc. ACM EuroSys, 2016.
[11] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," Proc. ACM SOSP, pp.460–477, 2017.
[12] S. Zheng, L. Huang, H. Liu, L. Wu, and J. Zha, "Hmvfs: A hybrid memory versioning file system," Proc. MSST, 2016.
[13] D. Campello, H. Lopez, R. Koller, R. Rangaswami, and L. Useche, "Non-blocking writes to files," Proc. USENIX FAST, 2015.
[14] V. Prabhakaran, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," Proc. USENIX ATC, 2005.
[15] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," Proc. Linux Symposium, 2007.
[16] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system," Proc. USENIX ATC, 1996.
[17] J. Bonwick, M. Ahrens, V. Henson, M. Maybee, and M. Shellenbaum, "The zettabyte file system," Proc. FAST, work in progress, 2003.

[18] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," ACM Transactions on Storage (TOS), vol.9, no.3, p.9, 2013.
[19] Intel, "Persistent memory programming." http://pmem.io/pmdk/
[20] J. Coburn, A.M. Caulfield, A. Akel, L.M. Grupp, R.K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," Proc. ACM ASPLOS, pp.105–118, 2011.
[21] H. Volos, A.J. Tack, and M. Swift, "Mnemosyne: Lightweight persistent memory," Proc. ACM ASPLOS, pp.91–104 2011.
[22] T. Hwang, J. Jung, and Y. Won, "Heapo: Heap-based persistent object store," ACM Transactions on Storage (TOS), vol.11, no.1, p.3, 2015.
[23] Intel, "Libvmmalloc." http://pmem.io/pmdk/manpages/linux/master/libvmmalloc/libvmmalloc.7.html
[24] Intel, "Persistent memory block device," 2016. https://pmem.io/2016/02/22/pm-emulation.html
[25] Iozone, http://www.iozone.org/
[26] Facebook, "RocksDB." http://rocksdb.org
[27] J.L. Carlson, Redis in Action, Manning Publications, 2013.
[28] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," Proc. ACM SoCC, pp.143–154, 2010.
[29] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," ACM SIGMETRICS Performance Evaluation Review, pp.53–64, 2012.
[30] ESOS-Lab, "Mobigen traces," 2013. https://github.com/ESOS-Lab/Mobibench/tree/master/MobiGen
[31] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won, "Framework for analyzing android i/o stack behavior: from generating the workload to analyzing the trace," Future Internet, vol.5, no.4, pp.591–610, 2013.
[32] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," login: The USENIX Magazine, vol.41, no.1, pp.6–12, 2016.
[33] Y. Lu, J. Shu, J. Guo, and P. Zhu, "Supporting system consistency with differential transactions in flash-based ssds," IEEE Transactions on Computers (TOC), vol.65, no.2, pp.627–639, 2016.
[34] Y. Ni, J. Zhao, D. Bittman, and E. Miller, "Reducing nvm writes with optimized shadow paging," Proc. USENIX HotStorage, 2018.
[35] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T.B. Da Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," Proc. ACM SOSP, pp.478–496, 2017.
[36] "Supporting filesystems in persistent memory," 2014. https://lwn.net/Articles/610174/
[37] M. Dong and H. Chen, "Soft updates made simple and fast on non-volatile memory," Proc. USENIX ATC, 2017.

Taeho Hwang is a Ph.D. student in the Dept. of Computer Software at Hanyang University. He is working in Operating Systems Laboratory. His research interests are in file systems and storage systems to fully exploit persistent memory.

Youjip Won is ICT Endowed Chair Professor at School of Electrical Engineering, KAIST. He did his BS and MS in Dept. of Computer Science, Seoul National University, Seoul, Korea in 1990 and 1992, respectively. He received his Ph.D. in Computer Science from University of Minnesota in 1997. He worked for Intel Corp. as Server Performance Analyst till 1999. From 1999 till 2019, he was with Dept. of Computer Science, Hanyang University, Seoul, Korea. He is known for his work on the Android IO stack optimization, filesystem and block layer design for SSD and NVRAM. His research interests include Operating System, Distributed System, Storage System and Software support for byte-addressable NVRAM.