IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 9, 2021, accepted October 17, 2021, date of publication October 19, 2021, date of current version October 26, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3121423

SCJ: Segment Cleaning Journaling for Log-Structured File Systems

HYUNHO GWAK AND DONGKUN SHIN, (Member, IEEE)
Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Dongkun Shin (dongkun@skku.edu)

This work was supported by the Institute of Information and Communications Technology Planning and Evaluation (IITP) Grant by the Korean Government through MSIT (Software Star Laboratory) under Grant IITP-2017-0-00914.

ABSTRACT The append-only write scheme of the log-structured file system (LFS), which does not permit in-place updates, is suitable for flash memory. Therefore, LFSs have been adopted to manage various flash-memory-based storage systems. However, LFSs amplify user write requests owing to segment cleaning and frequent metadata write operations. Hence, the IO performance is degraded and the life span of the flash memory is reduced. In particular, when each segment cleaning invokes checkpointing to store dirty metadata modified via block relocation, irrelevant dirty metadata are written by the checkpointing operation, and the metadata write overhead occupies a significant proportion of the segment cleaning latency. Hence, we propose segment cleaning journaling (SCJ), which logs the block relocation information of segment cleaning at a specific journal area without invoking checkpointing. We implemented and evaluated the proposed SCJ by modifying F2FS. Experimental results show that SCJ can improve file system performance by 1.13–1.63 times compared with normal segment cleaning.

INDEX TERMS File systems, flash memory, operating systems, log-structured file systems, segment cleaning.

I. INTRODUCTION

NAND flash memory has been widely used in various systems, including mobile devices and server systems. Flash memory offers several advantages, such as high performance, low power consumption, and shock resistance, compared with HDDs. However, flash memory presents several limitations, such as the erase-before-write constraint [1], sequential programming on erase blocks, and limited program-and-erase cycles. Therefore, the write request on a logical address space sent from the host must be redirected to a writable flash memory location, and the logical-to-physical address translation must be managed. Modern flash storage devices such as solid-state drives (SSDs), embedded multimedia cards (eMMC), and universal flash storage (UFS) are operated based a specific internal firmware, known as the flash translation layer (FTL), which manages logical-to-physical address mapping and performs garbage collection (GC) to recycle dirty flash blocks. The GC operation moves all valid data in the GC victim blocks to clean blocks and erases the victim blocks to reuse them. Therefore, the GC process generates many flash read and write operations and delays the management of host IO requests. In particular, random write requests from the host render invalid pages more distributed, thereby increasing the GC overhead [2]–[4].

The log-structured file system (LFS) [5] can reduce the detrimental effects of random writes because its out-of-place update scheme generates sequential writes on storage. Therefore, LFSs are adopted in modern flash storage-based devices such as smartphones and ZNS SSDs [6], [7].

However, segment cleaning (also known as GC, segment recycling, or segment compaction) must be performed in LFSs to reclaim dirty segments. When free (empty) segments run out, the segment cleaning copies all valid blocks in the dirty segments to other segments such that they become free segments [5]. Segment cleaning invokes numerous block copy operations and degrades performance, particularly when the file system utilization is high. Therefore, many researchers have attempted to reduce the segment cleaning overhead of LFSs [3], [8]–[16]. Note that the file system-level segment cleaning is performed at a different level with the GC of the FTL firmware. Therefore, the segment cleaning optimization is irrelevant to FTL.

The associate editor coordinating the review of this manuscript and approving it for publication was Cristian Zambelli.

Another disadvantage of the LFS is its frequent metadata updates. When a data block is updated, its indirect index block that stores the location of the data block should be updated because the LFS reallocates the location of the data block. In addition, indirect block updates may incur another modification of higher-level metadata recursively. This is known as the wandering tree problem [17], [18]. The significant number of metadata writes causes frequent segment cleaning operations.

To mitigate the metadata overhead of LFSs, several optimization technologies have been adopted in modern LFSs. For example, the lazy indirect block update (LIBU) scheme [18] accumulates indirect block updates in memory instead of flushing them immediately to storage. The accumulated indirect blocks are periodically flushed during checkpointing. Hence, some indirect blocks can be lost when a system crash occurs, and the LFS must recover them via either roll-forward recovery or roll-back recovery. Whereas the roll-back recovery restores the file system state to the state saved by the last checkpoint, the roll-back recovery rebuilds lost indirect blocks by examining the segment summary, which is recorded each time the segment is completely used and contains information regarding the corresponding file number and the offset for each block in the segment [18].

Flash-friendly file system (F2FS) [19], which is designed considering the characteristics of flash memory, uses the lazy metadata update (LMU) scheme to delay the flush of not only node block updates, but also that of all file system metadata updates. LMU can significantly reduce the number of metadata write requests compared with LIBU. However, LMU presents the following disadvantages: First, roll-forward recovery becomes impossible because the segment summary blocks are not flushed immediately. Consequently, all file system updates after the last checkpoint may be lost. A simple solution to this problem is to perform checkpointing frequently to reduce the number of file system updates between successive checkpoints. However, frequent checkpointing decreases the benefit of LMU because accumulated metadata in the memory must be flushed during checkpointing. Therefore, F2FS supports roll-forward recovery only for *fsynced* files. For a fsync file, F2FS flushes its node block without flushing other metadata blocks. A specific flag within the node block is marked to signify that the corresponding file is fsynced. During recovery, the fsync-marked nodes are used to rebuild the lost metadata updates.

The second problem of LMU is associated with segment cleaning. Although many metadata updates are performed by block relocations during segment cleaning, the LMU scheme does not change the in-storage file system metadata immediately. Therefore, the in-storage metadata points to the old block locations and the old block cannot be utilized for new data until the modified metadata blocks are flushed to storage. If new write requests overwrite the old block locations, then the roll-back recovery cannot restore the file system state to the last checkpoint state. Hence, F2FS performs checkpointing after each segment cleaning to flush the modified metadata. After checkpointing, the reclaimed segment can be reused for the new data. However, checkpointing flushes all the modified data and metadata blocks, including those irrelevant to the segment cleaning operation. Therefore, a significant amount of write traffic occurs, and the segment cleaning latency is increased. Because segment cleaning is constantly invoked once the file system utilization becomes high, the checkpointing overhead of segment cleaning degrades the file system performance significantly.

In this study, we analyzed the checkpoint overhead during segment cleaning for various workloads. We discovered that more than half of the segment cleaning latency was caused by the checkpointing operation. In addition, we discovered that most of the metadata recorded at the checkpoints were not associated with segment cleaning. To remove unnecessary writes, we propose segment cleaning journaling (SCJ), which enables segment cleaning to be performed without checkpointing, i.e., metadata changes that are relevant to segment cleaning are logged at specific journal blocks. Hence, unnecessary writes by checkpointing can be avoided, and the segment cleaning latency can be reduced. In SCJ, checkpointing is delayed until excessive metadata updates are pending and the checkpointing interval increases significantly. This causes many pre-invalid blocks that are invalid but still referenced by in-storage metadata and are non-reclaimable. The pre-invalid blocks can be transformed to invalid blocks after their corresponding metadata are flushed by checkpointing. To solve this problem, we propose an adaptive checkpointing technique that performs checkpointing by considering accumulated pre-invalid blocks. When the performance degradation caused by the pre-invalid blocks is more significant than the metadata flush overhead of a checkpoint, checkpointing is invoked. We implemented and evaluated the proposed SCJ and adaptive checkpoint techniques by modifying F2FS. Our evaluation results showed that the file system performance using the proposed techniques was 1.13–1.63 times higher than that of the normal LFS.

The SCJ is based on our prior study, i.e., GC journaling [20]. However, in our prior study, a basic concept was suggested, whereas specific implementation issues and design considerations were disregarded. Herein, we propose an adaptive checkpoint technique to resolve problems in SCJ, and we present our implementations in detail, such as the recovery algorithm for SCJ.

The remainder of this paper is organized as follows: In Section II, the basic knowledge of the F2FS and the segment cleaning behavior of LFS are introduced; in Section III, the motivation of the proposed scheme is described; in Section IV and Section V, the proposed SCJ and the adaptive checkpoint schemes are described in detail, respectively; the experimental results are presented in Section VI; previous studies are presented in Section VII; the conclusion of this study is described in Section VIII.

II. BACKGROUND

A. F2FS

In this study, we used F2FS [19] to implement the proposed SCJ and adaptive checkpoint techniques. F2FS is an actively maintained LFS that is designed considering the characteristics of flash memory. F2FS uses several optimization technologies, as follows:

F2FS writes frequently updated file system metadata to the dedicated metadata area on the disk to prevent recursive metadata updates. The metadata area of F2FS stores the segment information table (SIT), node address table (NAT), and segment summary area (SSA). However, the node (indirect) blocks of F2FS are not recorded in the metadata area. This is because the number of node blocks is not fixed, unlike other file system metadata. In fact, F2FS uses the NAT structure to prevent recursive node block updates. If a data block is relocated, then F2FS updates only one node block and its NAT entry, thereby effectively addressing the wandering tree problem.

Another optimization technique for F2FS is multi-head logging. F2FS maintains six major log segments, i.e., hot, warm, and cold segments for each data and node. By separating hot and cold blocks into different segments, F2FS can reduce segment cleaning overhead. F2FS supports both append logging and threaded logging [18] to reduce IO delays due to segment cleaning. If the number of free segments is insufficient, then the F2FS changes the logging mode to threaded logging. Threaded logging enables the writing of new data to holes in the dirty segment. Therefore, threaded logging neither consumes free segments nor invokes segment cleaning. However, F2FS must invoke segment cleaning to secure free segments when the number of free segments becomes excessively small.

F2FS performs segment cleaning in two distinct manners: foreground and background. Foreground segment cleaning is triggered only when the number of free segments is insufficient to process incoming write requests. Background segment cleaning is triggered only when the number of free segments is below a threshold, and the file system is idle. Therefore, the IO delay due to background cleaning is insignificant. In this study, we focused on the foreground segment cleaning overhead.

B. SEGMENT CLEANING

Segment cleaning is the process of reclaiming scattered and invalidated blocks for further logging. Segment cleaning comprises the following three phases.

Victim selection. The segment cleaning process first identifies a victim segment among the dirty segments. Most LFSs adopt the greedy policy, which selects a segment with the smallest number of valid blocks to minimize the time spent on segment cleaning. F2FS adopts the greedy policy for foreground segment cleaning. To identify the victim segment promptly, F2FS maintains the number of valid blocks of each segment in the SIT.

Valid block migration. After the victim selection phase, the LFS copies all valid blocks of the victim segment to the free segment. During the copy process, node blocks that store the block addresses of the valid blocks are updated because the valid blocks are relocated. To update the node blocks, the LFS retrieves the segment summary information of the victim segment. The segment summary contains the file (inode) number and the offset for each block in the segment. Therefore, the LFS can identify and update the parent node blocks, which point to the valid blocks.

Metadata write. The LFS must flush the metadata updated during the valid block migration phase before new write requests overwrite the victim segment. Otherwise, when a sudden system crash occurs, the LFS loses overwritten data because the in-storage metadata points to the old block location. After flushing the metadata, the victim segment becomes a new free segment. F2FS performs checkpointing in the metadata write phase. However, this results in unnecessary writes because checkpointing flushes all dirty metadata blocks that are unrelated to segment cleaning.

C. CHECKPOINT

The LFS can maintain the consistency of the file system by rolling data back to the last checkpoint state when a sudden crash occurs. During the checkpoint process, the LFS flushes modified data and metadata to the non-volatile storage persistently. The flush operations of the checkpoint writeback dirty blocks in the cache to the storage. Therefore, frequent checkpointing significantly degrade performance. The LFS triggers a checkpoint procedure as follows:

Data and node block flush. First, the checkpoint process begins flushing all dirty data and node blocks in the memory to storage. Because LFS sequentially allocates new blocks for write operations in the free segment, the file system will issue one large write request to reduce the request handling overhead. Therefore, instead of immediately issuing a write request for each block, the LFS accumulates write requests in the buffer, and some data may remain in the buffer when the LFS triggers the checkpoint. In this phase, the LFS flushes all dirty blocks, including the blocks remaining in the buffer.

File system metadata flush. Flushing data and node blocks invokes modifications to the file system metadata, which include node block addresses and segment summaries. The LFS flushes the modified file system metadata to retrieve valid node blocks and data blocks during the recovery process. For example, F2FS flushes the NAT, which stores node block addresses. In the recovery process, F2FS can read valid node and data blocks using the NAT and node blocks, respectively.

Checkpoint metadata flush. After flushing the modified file system metadata, the LFS creates and flushes the checkpoint metadata (also known as the checkpoint region [5] or checkpoint pack [19]). The checkpoint metadata stores the current checkpoint version, log tail pointer, and addresses of the valid file system metadata. The LFS can retrieve the file system metadata using checkpoint metadata as well as

retrieve valid nodes and data blocks using the file system metadata. Therefore, after flushing the checkpoint metadata, the LFS can be rolled back to the current file system state.

The LFS performs checkpointing periodically to reduce the time required for roll-forward recovery [5]. For example, SpriteLFS uses a checkpoint interval of 30 seconds. However, frequent checkpointing results in performance degradation owing to the metadata flush overhead of the checkpoint. Furthermore, under the LMU scheme, the overhead of the checkpoint increases because all accumulated metadata in the memory must be flushed. Therefore, F2FS uses a checkpoint interval of 60 seconds and performs checkpointing only when idle. Another condition of F2FS to trigger checkpointing is when the memory contains a high percentage of dirty metadata.

III. MOTIVATION

We measured the time consumption of each segment cleaning phase to identify the reason contributing to performance degradation during segment cleaning. We executed F2FS in the LFS mode. In the LFS mode, no random writes exist except in the metadata area. Prior to the measurement, the file system utilization was set to 80% to invoke segment cleaning. In this experiments, the fileserver workload was used for the measurement. The configuration of the benchmark is shown in Table 1. The fileserver workload causes a large sequential write pattern and invokes little fsync calls. All measurements were performed on a computer system equipped with a 3GHz quad-core processor, 8GB of memory, and a SAMSUNG 970 SSD.

Fig. 1(a) presents the change in the segment cleaning latency for the fileserver workload. The latency was divided into three phases (init, copy, and checkpoint), where each phase represents the victim selection, valid block migration, and metadata write phase of the segment cleaning, respectively. Unlike the description of the background section, F2FS does not wait for the completion of write requests generated in the valid block migration phase. In fact, F2FS accumulates the write requests in the buffer, which are then flushed at the metadata write phase. Therefore, the time spent copying valid blocks is included in the metadata write phase. To exactly measure the time consumption of each phase, we modified F2FS such that the metadata write phase begins after the completion of the write requests generated in the valid block migration phase. Experimental results show that checkpointing constituted a significant proportion of the segment cleaning latency. The victim selection phase was short because the file system reads only several metadata blocks in the victim select phase, and those blocks were cached in the memory. The time spent on the valid block migration phase was similar to that on the metadata write phase. The time spent on the metadata write phase was 52% (on average) of the total segment cleaning latency. Similarly, in the other workloads shown in Table 1, checkpointing constituted a significant proportion of the segment cleaning latency. The time spent on the metadata write phase for the varmail, tpcc, and YCSB workloads were 44%, 57%, and 56% (on average) of the total segment cleaning latency, respectively.

Fig. 1(b) presents the change in the checkpoint latency and the amount of dirty metadata for the fileserver workload. The dirty meta shown in Fig. 1(b) represents the number of dirty metadata blocks accumulated in the memory. Experimental results show that the checkpoint latency increased with the amount of dirty metadata. The checkpoint latency and the dirty meta changed periodically during runtime. When segment cleaning was performed, checkpointing was invoked and the amount of dirty metadata decreased to zero. After segment cleaning, the amount of dirty metadata increased because file operations occurred between each segment cleaning operation. However, most of the dirty metadata blocks in the memory were irrelevant to the segment cleaning. Only few metadata blocks were modified owing to segment cleaning, i.e., less than 1% of the amount of dirty metadata.

In summary, checkpointing significantly increased the segment cleaning latency. However, most of the latency was caused by time spent recording metadata irrelevant to segment cleaning. Therefore, if the victim segment can be reclaimed without recording metadata that are irrelevant to segment cleaning, then the segment cleaning latency will be reduced significantly.

IV. SCJ

As mentioned earlier, checkpointing performed during segment cleaning can increase the segment cleaning latency by more than 2x. The long latency of segment cleaning adversely affects the response time of user requests. Hence, we propose the use of SCJ in the metadata write phase of the segment cleaning process instead of using checkpointing.

SCJ records only modified block addresses owing to segment cleaning instead of checkpointing, resulting in minimal metadata writes. Metadata modifications owing to other file operations are not recorded, and SCJ does not restore them. This recovery policy is in fact reflected in the traditional LFS using LMU, which does not retain the data recorded after the last checkpoint. For example, F2FS does not support the recovery of data blocks recorded after the last checkpoint. One exception is fsynced files that are restored even when they are recorded after the last checkpoint. To restore the fsynced file, F2FS flushes the metadata of the fsynced file, and this recovery scheme can be used in SCJ.

Unlike file system changes owing to other file operations, file system changes owing to segment cleaning must be flushed and restored during the recovery process to reclaim the victim segment. In fact, this is why F2FS performs checkpointing after each segment cleaning. If the modified metadata caused by segment cleaning are not recorded before the victim segment is overwritten with new data, then the old data blocks in the victim segment are lost. Therefore, recording the modified metadata is a prerequisite for reclaiming the victim segment. Furthermore, segment cleaning moves the data blocks to different locations. Therefore, restoring

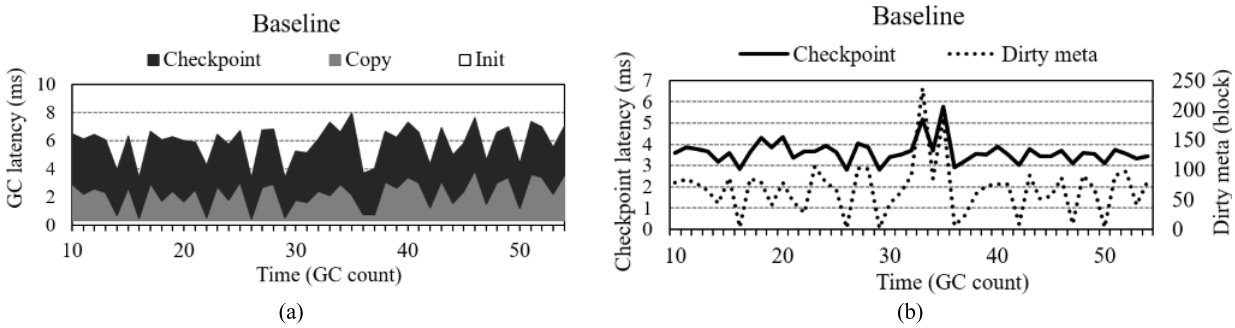

FIGURE 1. Effect of dirty metadata on segment cleaning latency: (a) garbage collection latency; (b) amount of dirty metadata and checkpoint latency.

file system changes owing to segment cleaning does not render the file state different from the last checkpoint state, unlike restoring other file operations such as file updates. SCJ flushes metadata modifications as journal blocks, which include only modifications owing to segment cleaning.

Segment cleaning based on SCJ comprises three phases, similar to the original segment cleaning. The victim selection phase is the same as the original operation. In SCJ, the modified block address is accumulated in a journal block during the valid block migration phase. The modified block address of each valid block is stored as an entry into the journal block. Subsequently, the entries of the journal block are used as replay logs during the recovery process, and the journal block stores the current checkpoint version for the recovery. Meanwhile, in the metadata write phase, the journal block is flushed instead of performing checkpointing.

If a block is recorded after the last checkpointing in the victim segment, then SCJ does not accumulate its block address in the journal block because it is not subject to recovery. Hence, SCJ should distinguish blocks recorded after the last checkpoint from valid blocks in the victim segment. F2FS maintains two types of valid block bitmaps, and one of them is a checkpoint valid bitmap that indicates valid blocks at the last checkpoint. We used these bitmaps to distinguish between valid blocks.

We implemented SCJ by modifying F2FS. The segment size of F2FS was 2 MB, and the block address was stored in 4 B. Hence, the maximum number of valid blocks in the victim block was 511, and the journal block can store a maximum of 511 x 8B = 4088 B entries. Because the checkpoint version was stored in 8 B, the number of journal block writes caused by one-segment cleaning did not exceed 4 KB. Therefore, the recording of journal blocks yielded a small amount of writing.

Fig. 2 shows an example of segment cleaning via SCJ. The valid blocks, B_1, B_2, and B_3, in the victim segment were copied to the free segment such that the victim segment becomes a free segment. Each entry of the journal block includes the old and new block addresses of a valid block that is copied during the segment cleaning. For example, in Fig. 2, because B_1 in 511 block address is copied into

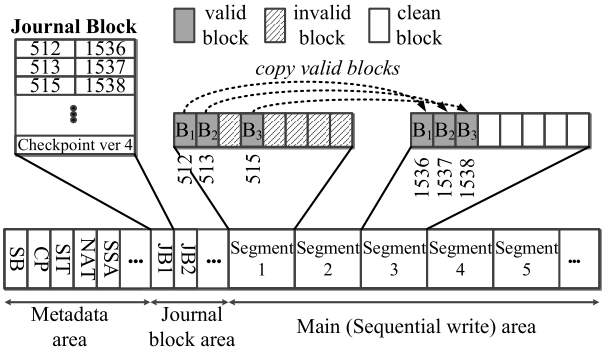

FIGURE 2. Example of segment cleaning via SCJ.

1536 block address, the journal entry contains the block address modification of B1. The last entry of the journal blocks contains the current checkpoint version number, which is 4. The journal block is flushed into the journal block area before the victim segment is changed into a free segment.

A. JOURNAL BLOCK AREA

The journal blocks were used as replay logs in the recovery process. Therefore, each journal block must be read in the order that it is created. Hence, in SCJ, each journal block is recorded sequentially in the journal block area. The journal block area is a region reserved in a fixed location of the storage space, such as the metadata area of F2FS.

If no free space is available to record the journal block in the journal block area, then checkpointing must be performed to create free space. Because checkpointing flushes all accumulated metadata in the memory, modifications of block addresses, which are included in the recorded journal blocks, are flushed. Therefore, all journal blocks recorded before checkpointing are invalidated. After checkpointing is performed, the journal block area can be reused, and the invalid journal blocks can be overwritten. Therefore, both valid and invalid journal blocks can co-exist in the journal block area. In the recovery process, the LFS must distinguish between valid and invalid journal blocks by examining the

checkpoint version stored in each journal block. The valid journal blocks store the same checkpoint version as that of the last checkpoint.

If the journal block area is extremely small, then checkpointing will be performed frequently to create free space. By contrast, if the journal block area is extremely large, then the available storage space for incoming write requests is reduced. Therefore, the journal block area must be set to an appropriate size. In our implementation, only one journal block was recorded per 2 MB of data write. Furthermore, because SCJ and adaptive checkpoint schemes are used simultaneously, all journal blocks in the journal block area were invalidated periodically. Hence, a large journal block area was unnecessary. A detailed description of the adaptive checkpoint is provided in the following section.

A fixed amount of journal block area is reserved in the metadata area of F2FS. The size of the journal block has to be set considering the trade-off between the journal area size and the performance. In the evaluation, we measured the checkpoint overhead and the performance while varying the size of the journal block area. As a result, we found that 256 KB is enough to prevent performance degradation since the proposed adaptive checkpoint scheme triggered the checkpoint periodically. The size of journal space, 256KB, occupied only 0.002% of the workload dataset size.

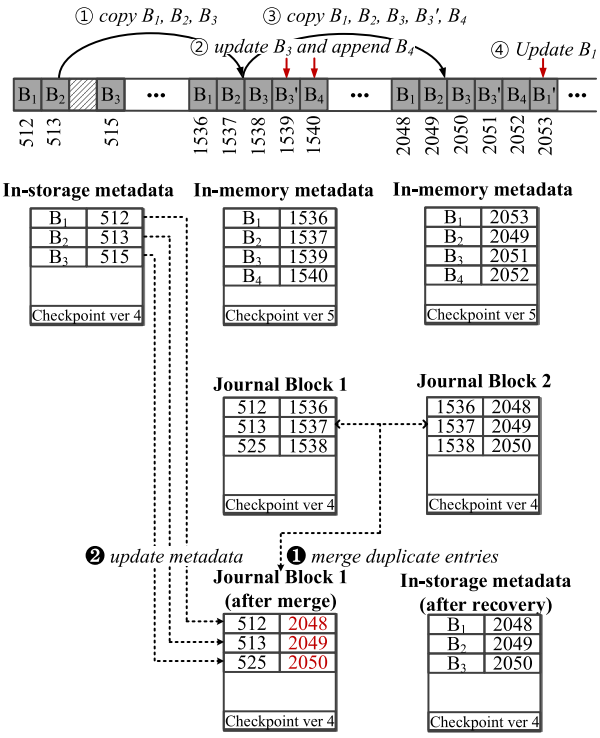

FIGURE 3. Example of journal block recovery.

B. RECOVERY

The recovery of SCJ is a process of applying block address modifications from the valid journal blocks. Because in-storage metadata point to old blocks that are overwritten after segment cleaning, the recovery operation must modify the metadata to point to the relocated block. The recovery of SCJ begins after the roll-back recovery of the LFS because the block address changes included in the invalid journal blocks are already flushed prior to the last checkpoint. The recovery of SCJ is performed in the following order:

Identify valid journal blocks. The recovery process begins with the identification of valid journal blocks. Because journal blocks are recorded in a fixed location in the storage space, the file system can read journal blocks sequentially. After reading the journal blocks, the file system examines the checkpoint version stored in each journal block to identify valid journal blocks.

Merge duplicate entries. The second phase of the recovery merges duplicate entries into valid journal blocks. The duplicate entries are generated only in an exceptional case when blocks copied by segment cleaning are re-copied by another segment cleaning. In this case, the block address modifications of the same block are stored in multiple journal blocks. Therefore, these modifications must be merged into one entry for recovery.

Update metadata. After merging duplicate entries, metadata that refer to the old block address are modified to a new block address. The metadata can be retrieved using a segment summary, and this is analogous to the LFS retrieving metadata for valid blocks during segment cleaning. Finally, the LFS performs checkpointing and flushes the modified metadata.

Fig. 3 shows a recovery example after a sudden crash occurs. Prior to the system crash, valid blocks were recorded in the 512, 513, and 515 block addresses in the first segment. Checkpointing was performed after these valid blocks were recorded in the first segment, and the checkpoint version was 4. After the checkpointing, two segment cleanings and three block writes occurred. Each time segment cleaning occurred (① and ③), a journal block was created and flushed to storage. When new writes occurred (② and ④), only in-memory metadata was modified and they were not flushed to storage. In this example, a system crash occurred after the third write (④) without additional checkpointing. The metadata listed in the first row of Fig. 3 are either in-storage metadata or in-memory metadata referenced by each operation. The first in-storage metadata was flushed through checkpointing before the first segment cleaning (①). Metadata modifications generated after the checkpointing were accumulated in the in-memory metadata. The metadata in the second row are journal block 1 and journal block 2 and they were flushed during the first and second segment cleaning operations, respectively. Finally, the third row indicates the recovery process using the in-storage metadata and journal blocks.

As shown in Fig. 3, the recovery was performed as follows: The first process was identifying valid journal blocks. Journal block 1 and journal block 2 were valid journal blocks because their checkpoint versions were the same as that of the last

checkpointing (i.e., version 4). Next, duplicate entries in the valid journal blocks were merged (❶). In the example shown in Fig. 3, the block address modifications of blocks B_1, B_2, and B_3 were stored in both journal block 1 and journal block 2. For example, one modification of the block address of B_1, i.e., from 512 to 1536, was stored in journal block 1. In addition, the other modification of the block address of the same block, i.e., from 1536 to 2048, was stored in journal block 2. These modifications can be merged into one entry, i.e., from 512 to 2048. After merging, the node block that points to the data blocks is retrieved using the segment summary and then updated (❷). Finally, the updated node block is flushed via checkpointing.

One exception in the recovery process is the fsync of F2FS. F2FS flushes the in-memory metadata of fsynced file, which can be restored using the metadata. For example, if fsync is called after the third write (④) in Fig. 3, in-memory metadata including the block address of B_1' is flushed to storage. Therefore, the LFS can restore the B_1' block using the in-storage metadata.

V. ADAPTIVE CHECKPOINT

Although SCJ can remove unnecessary metadata writes caused by frequent checkpointing, its block copy cost can be higher than that of normal segment cleaning. This is due to pre-invalid blocks that are invalid but still referenced by the in-storage metadata and are non-reclaimable. The pre-invalid blocks are created under the LMU scheme because the in-storage metadata do not change immediately after the relocation of the data block. They cannot be overwritten because crash recovery will need to restore them. Therefore, under the SCJ scheme, the LFS must copy the pre-invalid blocks in the victim segment during segment cleaning. In the normal segment cleaning process, the pre-invalid blocks need not be copied because all pre-invalid blocks in the victim segment become invalid blocks after checkpointing. B_3 in Fig. 3 is an example of a pre-invalid block. B_3, which is recorded in the 1538 block address, is updated and relocated to the 1539 block address. After the update, the old block located in the 1538 block address becomes a pre-invalid block because the in-storage metadata still points to the 1538 block address. The pre-invalid block was copied during second segment cleaning (③).

If checkpointing is delayed until excessive pre-invalid blocks are created, then the pre-invalid block copy overhead significantly increases the segment cleaning latency and degrades the file system performance. To solve this problem, we used an adaptive checkpoint, which invokes checkpointing when the performance degradation caused by the pre-invalid blocks is more significant than the metadata flush overhead of the checkpoint. We implemented the adaptive checkpoint scheme to trigger checkpoints whenever the number of accumulated pre-invalid blocks exceeded a threshold. For adaptive checkpoints, the LFS must monitor the number of pre-invalid blocks. If checkpointing is invoked frequently, then the write traffic on the metadata blocks increases. By contrast, if checkpointing is delayed for an extremely long duration, then the pre-invalid block copy overhead increases, and the segment cleaning latency increases. Therefore, an appropriate threshold must be determined by considering the abovementioned tradeoff. Based on experiments involving several workloads, we discovered that the overall performance was maximized when the threshold was approximately 128 MB. Therefore, in the evaluation, the threshold was set to 128 MB.

TABLE 1. Benchmark configurations.

fileserver	112,500 files, file size: 128KB, 14GB fileset, 50 threads
varmail	475,000 files, file size: 32KB, 14.5GB fileset, 16 threads
tpcc	DB size: 12GB, 1GB buffer pool, 16 connections
YCSB	DB size: 12GB, 1GB buffer pool, 32 connections

VI. EVALUATION

A. EXPERIMENTAL SETUP

We evaluated the performance improvements of the SCJ and adaptive checkpoint schemes. All experiments were performed on a PC using a 3GHz quad-core processor, 8GB of memory, and a SAMSUNG 970 SSD. In our implementation, we modified Linux kernel 4.15 to exploit the SCJ and adaptive checkpoint schemes.

Filebench [21] (fileserver and varmail) and OLTP benchmarks (tpcc [22] and YCSB [23] on MySQL) were used for the evaluation. Unlike fileserver workload, varmail workload causes a random write pattern with frequent fsync calls. Tpcc also causes a random write pattern. For the YCSB, we used workload A of YCSB, which has a mix of 50/50 reads and writes. We set the file system size to 16 GB and determined the dataset size of each benchmark such that the file system utilization was 80% by default. The configuration of each benchmark is presented in Table 1.

In the experiments, two different versions of the LFS were used: baseline (BL) and SCJ. The SCJ uses both the proposed SCJ and adaptive checkpoint schemes, whereas BL is the original F2FS in the LFS mode.

B. ADAPTIVE CHECKPOINT

We measured the effect of SCJ on the fileserver workload while varying the checkpoint interval, as shown in Fig. 4. The checkpoint interval is a threshold, and if the number of pre-invalid blocks exceeds the threshold, then the LFS triggers the checkpoint. We increased the checkpoint interval from 32 to 8192 MB. The BL does not use the SCJ and adaptive checkpoint schemes and performs checkpointing after each segment cleaning.

Fig. 4(a) presents the performances of the BL and SCJ. The performance of the SCJ improved as the checkpoint interval increased. However, when the checkpoint interval exceeded 128 MB, the performance of the SCJ began to deteriorate because of the following reason. As the checkpoint interval increased and the frequency of the checkpoints decreased, the checkpoint overhead decreased.

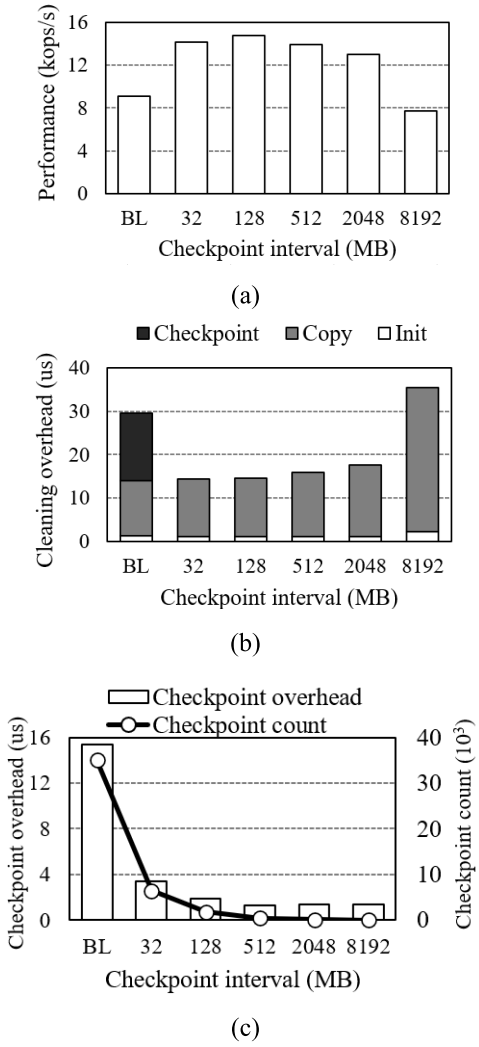

FIGURE 4. Performance of SCJ for various checkpoint intervals: (a) Performance; (b) segment cleaning overhead; (c) checkpoint overhead.

However, as the checkpoint interval increased, the segment cleaning overhead increased because of the pre-invalid blocks. The BL performed checkpointing the most frequently in the experiments and demonstrated worse performance than SCJ, except when the checkpoint interval of the SCJ was set to 8192 MB.

Fig. 4(b) shows the segment cleaning overhead. The segment cleaning overhead is the time spent on segment cleaning divided by the number of data block writes. The segment cleaning overhead comprised three components (init, copy, and checkpoint), as shown in Fig. 1. As the checkpoint interval increased, the copy overhead of the SCJ increased. Unlike SCJ, the BL does not copy pre-invalid blocks during segment cleaning. Hence, the BL showed the lowest copy overhead in the experiments. However, the BL significantly increased the segment cleaning overhead because it performed checkpointing after each segment cleaning. Fig. 4(c) shows the checkpoint overhead and checkpoint count. The checkpoint overhead is the time spent on checkpointing divided by the number of data block writes. Unlike the segment cleaning overhead, the checkpoint overhead decreased as the checkpoint interval increased. The BL showed the highest checkpoint count and checkpoint overhead in the experiments. Therefore, even when the checkpoint interval was set to 32 MB, the checkpoint overhead of SCJ was significantly lower than that of the BL.

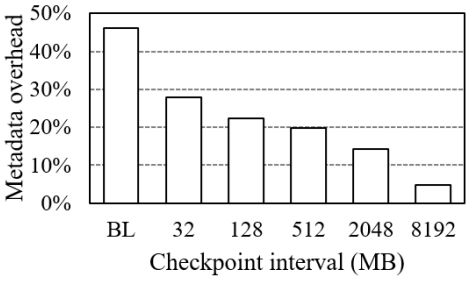

FIGURE 5. Metadata overhead.

The performance was inversely proportional to the total overhead, which is the sum of the segment cleaning overhead and the checkpoint overhead. For the fileserver workload, SCJ showed the smallest total overhead when the checkpoint interval was set to 128MB. Therefore, the performance of SCJ was maximized when the checkpoint interval was set to 128MB. Similarly, other workloads demonstrated the best performance when the checkpoint interval was set to 128 MB.

Fig. 5 presents the file system metadata overhead, which indicates the write traffic on the node blocks and the file system metadata blocks. The overhead values were normalized to the user data traffic. As the checkpoint interval increased, the number of metadata writes decreased because the LMU accumulated dirty metadata in the memory until checkpointing. However, when the checkpoint interval was set to 32 MB, the checkpoint overhead was reduced by 78%, whereas the metadata overhead was reduced by only 13% as compared with that of the BL. This discrepancy was caused by the checkpoint process. Because each checkpoint phase must wait for the completion of the previous phases, the checkpoint latency includes the waiting time for the completion of the write operations. Because this waiting time accounts for most of the checkpoint overheads, the reduction rate of the metadata write is smaller than the reduction rate of the checkpoint overhead.

C. JOURNAL BLOCK AREA SIZE

We measured the effect of SCJ on the fileserver workload while varying the size of the journal block area, as shown in Fig. 6. We increased the size of the journal block area from 4 KB to 1 MB. When the size of the journal block is 4 KB, the SCJ performs checkpointing at every segment cleaning, so it is identical with the BL.

Fig. 6(a) presents the performance changes of the SCJ as the size of the journal block area increased. When the size of the journal block exceeded 256 KB, there were no

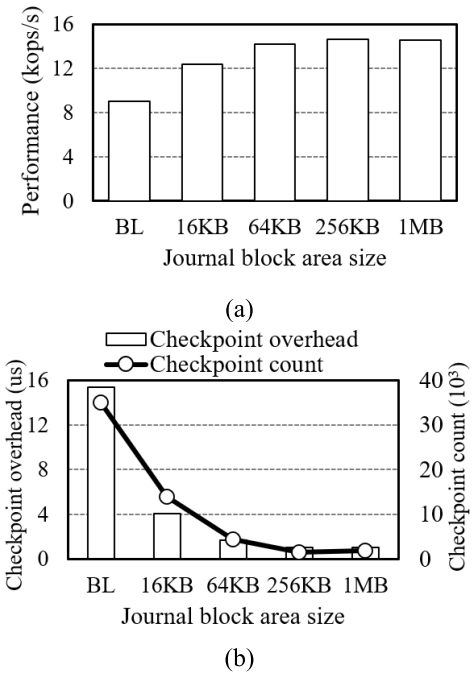

FIGURE 6. Performance of SCJ for various size of the journal block area: (a) Performance; (b) checkpoint overhead.

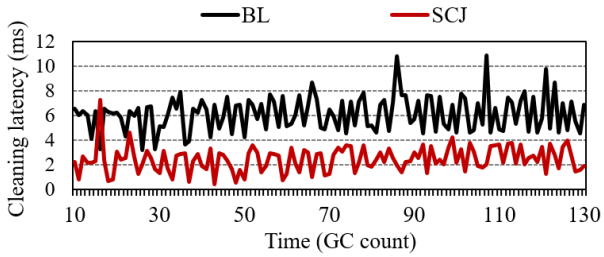

FIGURE 7. Segment cleaning latency of SCJ.

further improvement. Fig. 6(b) shows the checkpoint overhead and checkpoint count. The checkpoint overhead is the time spent on checkpointing divided by the number of data block writes. The checkpoint overhead decreased as the size of the journal block area increased, because the checkpointing must be performed if no free space is available to record the journal block. However, when the size of the journal block exceeded 256 KB, additional checkpointing was not performed to make free journal space because all journal blocks in the journal block area were invalidated periodically by the adaptive checkpoint. When the size of the journal block exceeded 256 KB, the checkpointing was triggered only at the adaptive checkpoint. The size of the journal block that is required for avoiding additional checkpointing is 256 KB, and this is only 0.002% of the workload dataset size. Therefore, the additional space overhead of the SCJ is negligible.

D. CHANGES IN SEGMENT CLEANING LATENCY

Fig. 7 presents the change in the segment cleaning latency of both the BL and SCJ for the fileserver workload.

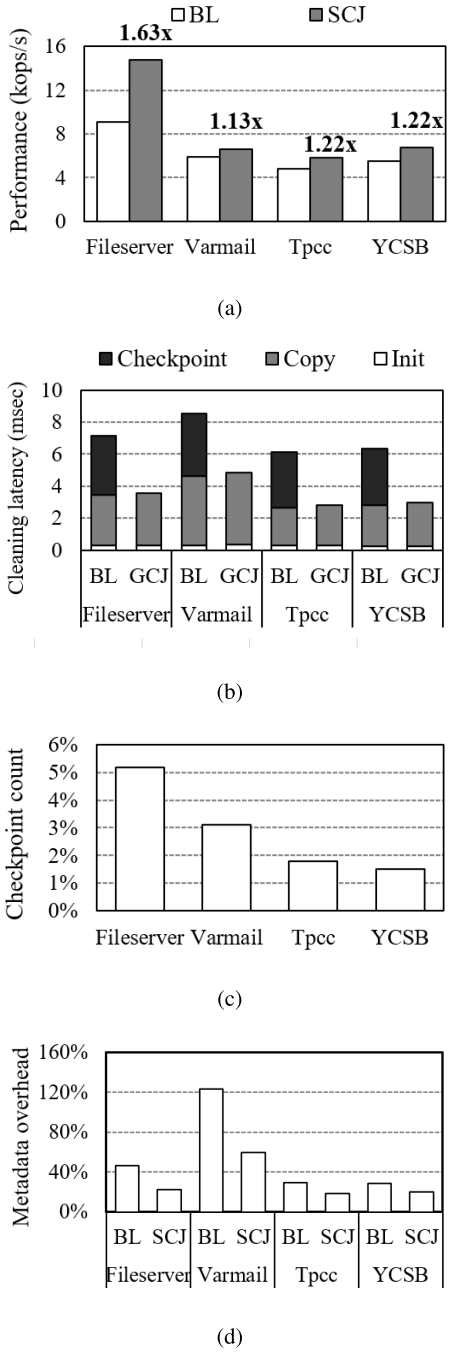

FIGURE 8. Evaluation for various workloads: (a) Performance; (b) segment cleaning latency; (c) checkpoint count; (d) metadata overhead.

As mentioned earlier, the segment cleaning latency was significantly affected by the time spent on checkpointing. Therefore, the segment cleaning latency of SCJ was lower than that of the BL, since segment cleaning was performed without checkpointing in SCJ.

E. OVERALL PERFORMANCE

We compared the overall performance of the benchmarks under BL and SCJ to evaluate the effects of the proposed

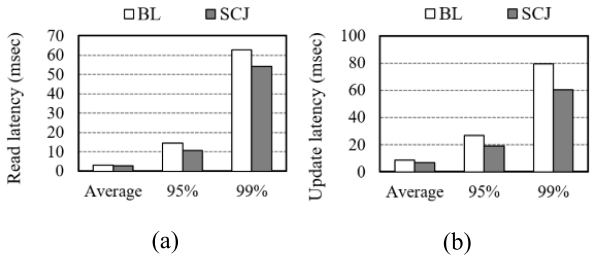

FIGURE 9. Read and update latency of YCSB: (a) Read latency; (b) update latency.

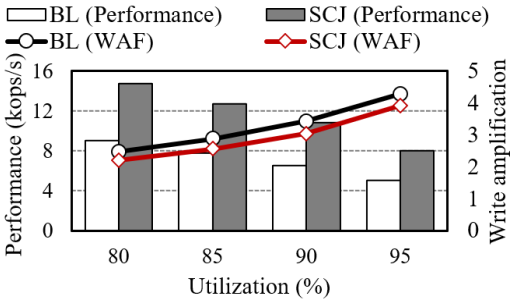

FIGURE 10. Performances for various file system utilizations (fileserver workload).

SCJ and adaptive checkpoint schemes, as shown in Fig. 8(a). Fig. 8(b) shows the segment cleaning latency; as shown, the performance of SCJ was 1.13–1.63 times higher than that of BL because SCJ can significantly reduce the segment cleaning latency. In the varmail workload, the segment cleaning overhead constituted a smaller proportion of the total IO latency compared with the other workloads. Therefore, the performance improvements of the varmail workloads were less significant than those of the other workloads.

Fig. 8(c) shows the checkpoint count of the SCJ. The count values were normalized to the checkpoint count of the BL. To remove unnecessary metadata writes, SCJ only performs checkpointing when the pre-invalid block overhead is larger than the metadata flush overhead of the checkpoint. Hence, the checkpoint count of SCJ was reduced to less than 6% of the checkpoint count of the BL for all the workloads. This implies that the BL performed checkpointing frequently and executed numerous unnecessary metadata writes. Fig. 8(d) presents the file system metadata overhead. As shown, SCJ reduced the metadata write traffic for all workloads.

Furthermore, we compared the IO latency of the different schemes. Fig. 9 presents the average, 95 percentile, and 99 percentile latency for the YCSB workload. Because the segment cleaning overhead and checkpoint overhead delay not only write requests, but also read requests, SCJ can decrease the latency of both the read and update operations of the YCSB. In the YCSB workload, the SCJ decreased the average and 99 percentile tail latency of the read operations by 11% and 10%, respectively, compared with the BL. Furthermore, the SCJ decreased the average and 99 percentile tail

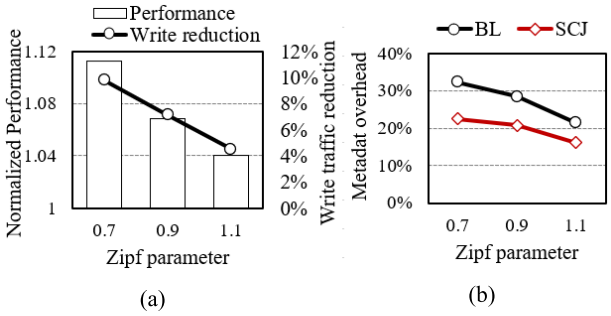

FIGURE 11. Performance of SCJ for various update localities: (a) Normalized performance and write traffic reduction; (b) metadata overhead.

latency of the update operation by 20% and 19%, respectively, compared with the BL.

F. FILE SYSTEM UTILIZATION

We compared the performances of different schemes while varying the file system utilization. By changing the file size of the target workload, we controlled the file system utilization. Fig. 10 presents the workload bandwidth and write amplification factor (WAF) of each technique for the fileserver workload. The WAF is the total write traffic invoked by the file system (including data block writes, metadata block writes, and segment cleaning) divided by the write traffic generated by the user workload. As the file system utilization increased, the WAF increased because more valid blocks must be copied during segment cleaning, and segment cleaning was invoked more frequently. However, as the number of copied valid blocks increased, the proportion of metadata block writes in the total write traffic decreased. Because SCJ only reduced the amount of metadata writes invoked by checkpointing, its performance improvements deteriorated as the file system utilization increased.

G. METADATA WRITE RATIO

We demonstrated the relationship between the proportion of metadata writes and the performance improvement of SCJ based on the fio benchmark. We executed a random write workload using fio, and the write size was set to 32 KB. Before executing the benchmark, we created a 12 GB dataset to invoke segment cleaning. We controlled the proportion of metadata writes by changing the zipf distribution parameter of the fio benchmark. As the zipf distribution parameter increased, the update locality of the data blocks increased, and the number of modified metadata blocks decreased. Fig. 11 presents the normalized performance, write traffic reduction, and metadata overhead. The performance values are the random write bandwidths of the SCJ normalized to those of the BL. Fig. 11(b) shows that the metadata overhead of both the BL and SCJ decreased as the update locality increased. Therefore, the discrepancy in the amount of write traffic and performance increased as the update locality decreased.

VII. RELATED WORK

The LFS has been widely used in various systems [3], [16], [19], [24]–[27] because of its append-only write scheme. The advantages of append-only write make LFS used for not only flash memory but also non-volatile memory such as the INTEL optane DC persistent memory [28]–[30]. However, the LFS has disadvantages that degrade the IO performance during segment cleaning, and many techniques have been proposed to overcome this performance degradation.

Separating hot data from cold data is effective in reducing segment cleaning costs [3], [4], [8]–[10], [31]–[35]. DAC [32] is a representative scheme for separating hot and cold data that clusters data blocks of similar write frequencies into the same logical group by monitoring accesses during runtime. Similarly, the SFS [3] captures the hotness semantics at the file block level and places data blocks with similar update likelihood into the same segment. F2FS [19], which we used to implement SCJ, runs multiple active log segments concurrently, and appends data and metadata to separate log segments.

Over-provisioning [36], [37] is a technique used to reduce the number of blocks copied by segment cleaning. It reserves a certain portion of the storage capacity to store valid blocks used for segment cleaning. Background segment cleaning [11]–[14] can reduce performance degradation during segment cleaning because background operations do not delay IO latency. Another approach to reducing segment cleaning overhead is to apply specific commands such as vector chunk copy command [15] and zone_compaction command [16], which are interfaces for copying data inside the SSD. These specific commands can reduce the data transfer traffic between the host and storage as well as reduce the segment cleaning latency.

Previous studies have focused on reducing the valid block copy cost of segment cleaning or performing segment cleaning at idle to reduce the IO delay. However, they cannot resolve the unnecessary metadata write problem caused by segment cleaning operations. Unlike previous studies, the proposed SCJ scheme reduces the metadata write overhead by performing segment cleaning without flushing irrelevant metadata.

VIII. CONCLUSION

The current segment cleaning operation of the LFS imposes unnecessary metadata write overheads. Increased write traffic shortens the life span of flash memory and increases segment cleaning latency. To optimize the performance of the file system, the segment cleaning latency must be reduced. To remove unnecessary metadata writes and reduce the segment cleaning overhead, we proposed SCJ, which performs segment cleaning without checkpointing. Furthermore, we proposed an adaptive checkpoint scheme to solve the pre-invalid block problem. Experimental results showed that the SCJ and adaptive checkpoint schemes can improve the file system performance by 1.13–1.63 times compared with normal segment cleaning.

REFERENCES

[1] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Trans. Embedded Comput. Syst.*, vol. 6, no. 3, p. 18, Jul. 2007.

[2] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 1, pp. 181–192, 2009.

[3] C. Min, K. Kim, H. Cho, S. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proc. 10th USENIX Conf. File Storage Technol. (FAST)*, 2012, pp. 1–16.

[4] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proc. 6th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2014, pp. 1–5.

[5] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, 1992.

[6] *Galaxy Note 10 Uses F2FS*. Accessed: Sep. 1, 2021. [Online]. Available: https://www.sammobile.com/news/galaxy-note-10-uses-f2fs-not-ext4-file-system-whats-the-difference

[7] *F2FS ZNS Support*. Accessed: Sep. 1, 2021. [Online]. Available: https://zonedstorage.io/linux/fs

[8] J. Wang and Y. Hu, "WOLF—A novel reordering write buffer to boost the performance of log-structured file systems," in *Proc. 1st USENIX Conf. File Storage Technol. (FAST)*, 2002, pp. 47–60.

[9] W. Wang, Y. Zhao, and R. Bunt, "Hylog: A high performance approach to managing disk layout," in *Proc. 3th USENIX Conf. File Storage Technol. (FAST)*, 2004, pp. 145–158.

[10] L. Yang, Z. Tan, F. Wang, S. Tu, and J. Shao, "M2H: Optimizing F2FS via multi-log delayed writing and modified segment cleaning based on dynamically identified hotness," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 808–811.

[11] T. Blackwell, J. Harris, and M. I. Seltzer, "Heuristic cleaning algorithms in log-structured file systems," in *Proc. USENIX Tech. Conf.*, 1995, pp. 277–288.

[12] D. Park, S. Cheon, and Y. Won, "Suspend-aware segment cleaning in log-structured file system," in *Proc. 7th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2015, pp. 1–5.

[13] Q. Li, A. Deng, C. Gao, Y. Liang, L. Shi, and E. H.-M. Sha, "Optimizing fragmentation and segment cleaning for CPS based storage devices," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, Apr. 2019, pp. 242–249.

[14] C. Wu, C. Ji, and C. J. Xue, "Reinforcement learning based background segment cleaning for log-structured file system on mobile devices," in *Proc. IEEE Int. Conf. Embedded Softw. Syst. (ICESS)*, Jun. 2019, pp. 1–8.

[15] *Open-Channel Solid State Drives Specification Revision 2.0*. Accessed: Sep. 1, 2021. [Online]. Available: http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf

[16] K. Han, H. Gwak, D. Shin, and J. Hwang, "ZNS+: Advanced zoned namespace interface for supporting in-storage zone compaction," in *Proc. 15th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2021, pp. 147–162.

[17] (2005). *JFFS3 Design Issues*. [Online]. Available: https://www.linux-mtd.infradead.org

[18] Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Optimizations of LFS with slack space recycling and lazy indirect block update," in *Proc. 3rd Annu. Haifa Experim. Syst. Conf. (SYSTOR)*, 2010, pp. 1–9.

[19] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *Proc. 13th USENIX Conf. File Storage Technol. (FAST)*, 2015, pp. 273–286.

[20] H. Gwak, Y. Kang, and D. Shin, "Reducing garbage collection overhead of log-structured file systems with GC journaling," in *Proc. Int. Symp. Consum. Electron. (ISCE)*, Jun. 2015, pp. 1–2.

[21] *Filebench*. Accessed: Sep. 1, 2021. [Online]. Available: https://github.com/lebench/lebench/wiki

[22] *Percona-Lab/TPCC-MySQL*. Accessed: Sep. 1, 2021. [Online]. Available: https://github.com/Percona-Lab/tpcc-mysql

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput. (SoCC)*, 2010, pp. 143–154.

[24] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, "Don't stack your log on my log," in *Proc. 2nd Workshop Interact. NVM/Flash with Operating Syst. Workloads (INFLOW)*, 2014, pp. 1–10.

[25] S. Lee, M. Liu, S. Jun, S. Xu, and J. Kim, "Application-managed flash," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 339–353.

[26] J. Zhang, J. Shu, and Y. Lu, "ParaFS: A log-structured file system to exploit the internal parallelism of flash devices," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2016, pp. 87–100.
[27] C. Ji, L. Chang, R. Pan, C. Wu, C. Gao, L. Shi, T. Kuo, and C. J. Xue, "Pattern-guided file compression with user-experience enhancement for log-structured file system on mobile devices," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST)*, pp. 127–140, 2021.
[28] *Intel Optane DC Persistent Memory*. Accessed: Sep. 1, 2021. [Online]. Available: https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/optane-dc-persistent-memory.html
[29] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 323–338.
[30] S. Zheng, M. Hoseinzadeh, and S. Swanson, "Ziggurat: A tiered file system for non-volatile main memories and disks," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST)*, 2019, pp. 207–219.
[31] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang, "Managing flash memory in personal communication devices," in *Proc. IEEE Int. Symp. Consum. Electron. (ISCE)*, Dec. 1997, pp. 177–182.
[32] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Softw.-Pract. Exp.*, vol. 29, no. 3, pp. 267–290, Mar. 1999.
[33] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang, "Efficient identification of hot data for flash memory storage systems," *ACM Trans. Storage*, vol. 2, no. 1, pp. 22–40, Feb. 2006.
[34] J. Lee and J.-S. Kim, "An empirical study of hot/cold data separation policies in solid state drives (SSDs)," in *Proc. 6th Int. Syst. Storage Conf. (SYSTOR)*, 2013, pp. 1–6.
[35] Y.-Y. Lu, C.-H. Wu, and Y.-S. Chen, "A machine-learning-based data classifier to reduce the write amplification in SSDs," in *Proc. Int. Conf. Res. Adapt. Convergent Syst.*, Oct. 2020, pp. 213–218.
[36] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2008, pp. 1–14.
[37] Y. Oh, J. Choi, D. Lee, and S. H. Noh, "Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems," in *Proc. 10th USENIX Conf. File Storage Technol. (FAST)*, 2012, pp. 1–14.

HYUNHO GWAK received the B.E. degree in computer engineering from Sungkyunkwan University, Suwon, South Korea, in 2014, where he is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering. His research interests include embedded software, flash memory, file systems, and operating systems.

DONGKUN SHIN (Member, IEEE) received the Ph.D. degree in computer science and engineering from Seoul National University, Seoul, South Korea, in 2004. From 2004 to 2007, he was a Senior Engineer with Samsung Electronics, South Korea. He is currently a Professor with the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, South Korea. His research interests include embedded software, low-power systems, computer architecture, and real-time systems.