Ph.D. DISSERTATION

Optimizing File Systems for High-Performance Storage Devices

고성능 저장장치를 위한 파일시스템 최적화

FEBRUARY 2018

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yongseok Son

Ph.D. DISSERTATION

Optimizing File Systems for High-Performance Storage Devices

고성능 저장장치를 위한 파일시스템 최적화

FEBRUARY 2018

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yongseok Son

Optimizing File Systems for High-Performance Storage Devices

고성능 저장장치를 위한 파일시스템 최적화

지도교수 염헌영

이 논문을 공학박사 학위논문으로 제출함

2017 년 12 월

서울대학교 대학원

전기·컴퓨터 공학부

손용석

손용석의 공학박사 학위논문을 인준함

2017 년 12 월

위 원 장	엄현상	(인)
부위원장	염헌영	(인)
위 원	유승주	(인)
위 원	이재욱	(인)
위 원	한 혁	(인)

Abstract

High-performance storage technologies such as solid-state drives (SSDs) provide low-latency, high throughput, and high I/O parallelism to legacy storage systems. SSDs access data without mechanical overhead, and they often leads to order-of-magnitude improvements in performance over legacy storage devices such as hard disk drives (HDDs). However, replacing HDDs with SSDs while keeping the software I/O stack or not exploiting SSD features does not lead to maximum performance.

In this dissertation, we optimize file systems to fully exploit the SSD features (e.g., low-latency and high I/O parallelism). First, we analyze and explore I/O strategies in the existing file systems on low-latency SSDs. The file systems issue and complete several I/O requests when blocks are not contiguous, which does not take advantage of the low-latency of SSDs. To address this problem, we propose efficient I/O strategies, which transfer requests from discontiguous host memory buffers in the file systems to discontiguous storage segments in a single I/O request. Thus, they enable file systems to fully exploit the performance of low-latency SSDs.

Second, we investigate the locking and I/O parallelism in the existing file systems on highly parallel SSDs. In the file systems, the coarse-grained locking to access shared data structures is used and I/O operations are serialized by a single thread. For these reasons, the file systems often face the problem of lock contention and underutilization of I/O bandwidth on multi-cores with highly parallel SSDs. To address these issues, we enable concurrent updates on data structures and parallelize I/O operations.

We implement our techniques in EXT4/JBD2 and evaluate them on low-latency and highly parallel SSDs. The experimental results show that our optimized file system improves the performance compared to the existing EXT4 file system.

Keywords: File system, Operating System, High-Performance Storage Device, Solid-State Drive

Student Number: 2013-30241

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Over the last few decades, enhancing the performance of storage devices has been an important challenge for computer systems in research and industry. Many data-intensive applications have demanded high throughput and low-latency. For many years, hard disk drives (HDDs) [1–4] have been used as the most common primary storage device. However, the performance of HDDs lags far behind that of the processor and the main memory due to mechanical overhead (i.e., rotational and seek time), and this HDD performance bottleneck has worsened in modern computer systems.

Semiconductor technology has introduced non-volatile memory (NVM) such as MRAM [5], PCM [6], and NAND flash [7,8] to computer system communities, and it opened up research challenges. NVM accesses data in low-latency and highly parallel way, and this often leads to orders-of-magnitude improvements in performance over HDDs. Such recent developments in NVM technologies have

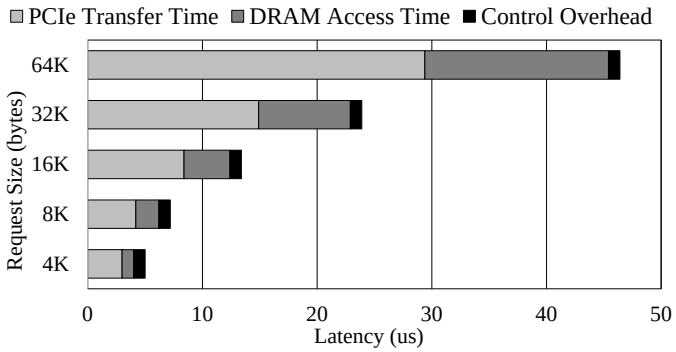
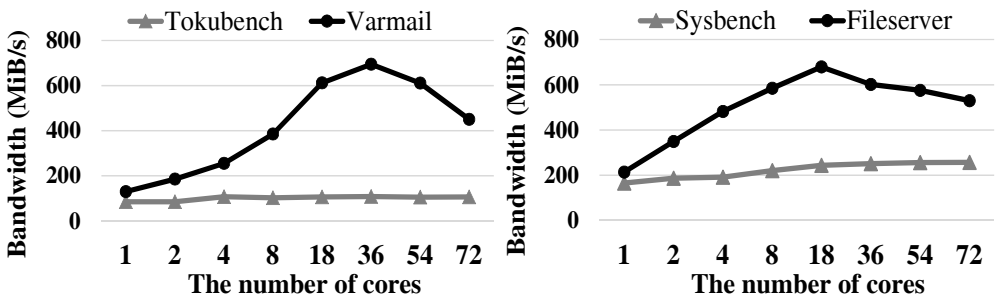


Figure 1.1: Latency Breakdown on low-latency storage (the detailed experimental environment is described in Section 4.1)

closed the performance gap between main memory and storage. Accordingly, NVM has improved I/O performance in various environments such as cloud platforms, social network services, large websites, etc. However, replacing HDD with NVM while keeping the software I/O stack does not lead to maximum performance as it is optimized for HDDs. To exploit the performance of NVM, researchers [9–15] have reconstructed the traditional software I/O stack and performed several optimizations.

Figure 1.1 shows a latency breakdown in our low-latency storage [16] with varying request sizes. As shown in the figure, the PCIe transfer time accounts for the major portion of the total PCIe communication time. The total time for one 64 KiB request is 42% less than that for sixteen 4 KiB requests owing to the benefit of the PCIe communication. This shows that a single request in larger granularity is more efficient than multiple small requests. Therefore, processing a large request is a better method for PCIe-based fast storage device. As our observations, existing I/O strategies prevent file systems from taking advantage of fast storage's full performance even if the block I/O subsystem is optimized. They process I/O requests by issuing and completing the request one by one when the storage segments of the requests are discontinuous.



(a) Ordered mode (b) Data journaling mode

Figure 1.2: Scalability evaluation on highly parallel storage (the number of threads is the same as that of the cores and the detailed experimental environment is described in Section 4.2)

Figure 1.2 shows a scalability and I/O performance using metadata and data-intensive workloads in the ordered and data journaling modes, respectively, in our highly parallel storage [17]. As shown in the figure, the performance does not scale well or decreases as the number of cores grows. Based on our analysis and other studies [15, 18], it is due to the contention on shared data structures and serialization of I/O operations.

1.2 Approach and Contributions

To achieve lower latency and higher parallelism, we propose two main optimizations for the file system. First, we propose I/O strategies of file systems in terms of latency. The key idea is to transfer data from discontiguous host memory buffers of file systems to discontiguous storage segments in a single I/O request, which existing block-based file systems cannot provide. We note that current storage protocols such as SATA and NVMe [19] support data transfer only from discontiguous host memory segments to contiguous storage segments in a single I/O request.

Second, we propose schemes to achieve high I/O parallelism as follows: (1) We use lock-free data structures and operations to reduce the lock contention. This scheme allows multiple threads to access the data structures (e.g., linked lists) concurrently. (2) We propose a parallel I/O scheme that performs I/O operations by multiple threads in a parallel and cooperative manner. This scheme allows multiple threads to cooperate in I/O processing and issue/complete the I/Os in parallel while not sacrificing the consistency of the file system.

We apply and implement the two optimizations on EXT4/JBD2. Our techniques provide higher performance while preserving all features and the same consistency level of the existing file system. We evaluate our optimized file systems for low-latency and parallelism using a DRAM-based SSD and Intel P3700 NVMe SSD, respectively. The experimental results show that the optimized file systems improve the performance compared to the existing file system.

The contributions of this dissertation can be summarized as follows:

- We analyze the main obstacles that increase the latency and reduce the parallelism of high-performance storage.
- We propose several optimization techniques for journaling file systems and implement them on EXT4/JBD2.
- Experimental results show that the optimized file system could achieve significant performance improvements, compared to the existing file system, while providing the same level of consistency.

1.3 Dissertation Structure

This dissertation is organized as follows:

Chapter 2 analyzes I/O path and strategy in the file system in terms of I/O latency and parallelism.

Chapter 3 designs and implements our schemes.

Chapter 4 evaluates our optimized file systems in terms of I/O low-latency and parallelism using varying workloads.

Chapter 5 summarizes related works and compares them with our works.

Chapter 6 summarizes our optimizations and contributions.

Chapter 2

Background

2.1 High-performance Storage Devices

High-performance storage provides low-latency and highly parallel accessing to data. Non-volatile memory (NVM) technologies, including PCM [20], spin-transfer torque memory [21], MRAM [5], and NAND flash [7,8] are anticipated to be faster than existing storage technologies (e.g., hard disk drives (HDDs)). The most significant features of NVM [5,20,21] are low latency, high throughput, and high parallelism without mechanical overheads. Previous studies [21, 22] suggest that NVM will have bandwidth and latency similar to DRAM and mention that the devices will be 50,000x faster than HDDs.

Modern PCIe-attached NVM-based SSDs [6, 22, 23] have emerged in many studies, and the arrival of the NVMe interface [19, 24] implies that PCIe-attached SSDs will be one of the target designs for fast NVM. Also, they employ significant amount of parallelism by having multiple channels, where each channel has multiple memory chips. Such a highly parallelized structure

provides rich opportunities for parallelism.

2.2 Crash Consistency in File Systems

Modern file systems provide crash consistency to applications. They employ journaling or copy-on-write (COW) mechanisms for transaction processing. Journaling file systems such as EXT4 [25], XFS [26], JFS [27], and ReiserFS [28] use a variant of write-ahead logging (WAL) [29], which first writes the metadata and/or data to journal area before in-place updates to metadata or/and data in storage for atomicity and durability. COW file systems [30], such as BTRFS [31], ZFS [32], and log-structured file system [33], use out-of-place updates to support crash consistency. They copy and modify the data for atomic update and then free the previous data through garbage collection.

This dissertation focuses on the EXT4 journaling mechanism since EXT4 is the most widely used file system in Linux and general to other file systems [15,34]. The EXT4 uses a fork of the journaling block device (JBD) called JBD2. The JBD is a file system-independent interface that can also be attached to other file systems such as EXT3 and OCFS2. It performs journal updates, commits, and checkpoint operations. EXT4 offers three journaling modes, such as write-back, ordered, and data journaling [15,35–37]. Write-back is the weakest crash consistency mode among the three journaling modes. This mode writes the metadata into the journal area, but the user data may be written into the original area in the file system after its metadata has been committed to the journal. In this mode, the ordering between the data and metadata is not preserved. The ordered mode provides stronger crash consistency than the write-back mode by keeping order between the metadata and data. Similar to the write-back mode, this mode writes the metadata into the journal while the

data is directly to the original area in the file system before the metadata is written into the journal.

The data journaling mode supports the highest crash consistency with data integrity. Both metadata and data are written into the journal area prior to being written into the original area in the file system to ensure they are updated atomically to persistent storage; they are either committed or aborted together in a transaction. However, the overhead of the data journaling mode is the largest among the journaling modes since the data is written to storage twice.

When an application updates blocks, a new transaction starts or the modification is compounded to the already running transaction activated by another application, which is a compound transaction scheme; EXT4 has only one running transaction and one committing transaction at any time [15, 38]. When the commit occurs at an interval of journal commit (5 seconds) or `fsync` call, the updated blocks are written into the journal area.

The transaction finishes the commit work after writing the commit block[1] into the end of the written blocks in the journal area. This commit block decides whether the transaction is committed or uncommitted. In a system fail or sudden power outage, the file system is remounted, and the file system scans the blocks in the journal area. Then, the file system replays the metadata/data blocks with a commit block and discards the blocks without a commit block. Checkpointing is triggered periodically and activated when the amount of the free space in the journal area drops below a certain threshold. The checkpoint operation writes the metadata/data in the committed transactions into the original area. The journal area is reclaimed via checkpoint so that the transaction can be continuously processed by writing the metadata/data into the free

[1] A commit block generates a flush command to preserve the ordering between journal metadata/data and the commit block

space in the journal area.

2.3 Read and Write Operations in the Existing File Systems

In this section, we describe current I/O strategies such as read-ahead and write-back. These strategies are applied to most Linux file systems in the same manner, and they are used by default when applications open files. When a buffered read is used, the file system performs the read-ahead technique to take advantage of spatial locality. To do this, the file system selects the user requested page(s) as well as additional adjacent pages. This technique is especially useful for sequential read patterns, as the next accessed pages are more likely to already be in the page cache, resulting in a higher page cache hit rate.

Figure 2.1a shows an example of a read-ahead operation in the existing file system. There are five pages (Page 0-4). Page 0 is the page requested by the user, and the other pages (Page1-4) are contiguous pages that the file system wants to read ahead. Each page is mapped to LBA 30, 31, 32, 33, and 34, respectively.

The file system performs read-ahead operations only for non up-to-date pages. It checks whether the LBAs for pages are contiguous with each other to ensure that each request has only contiguous pages. For example, in the case of Figure 2.1a, since Page 2 is already up-to-date, it incurs a hole in LBA. Thus, the file system first merges Page 0 (LBA: 30) and Page 1 (LBA: 31) into a request and issues the request (Request #1) since the LBAs of Page1 and Page3 are not contiguous. After I/O completion (polling) of the request, the file system rechecks the contiguity between the LBAs of Page 3 and Page 4. The file system merges these pages into a single request and issues the request (Request #2). In the read-ahead operation, the contiguity of the LBA is dependent on the

state of the page.

When a buffered write is used, the file system performs write-back operations for dirty pages if the dirty rate of the pages in the page cache is higher than the threshold. The file system chooses the dirty pages from the page cache and obtains the LBAs for the pages. The file system then checks the contiguity among pages. We note that the file system selects dirty pages without considering whether the pages are contiguous or not, unlike the prefetched pages.

Figure 2.1b describes an example of a write-back operation. There are five dirty pages (Page 0-4). The pages are mapped to each LBA: 1, 2, 20, 89, and 45, respectively. The file system merges the two contiguous pages (Page 0 (LBA: 1) and Page 1 (LBA: 2)) into a single request and issues the request (Request #1). After I/O completion of the request, the file system rechecks the LBA of the next page (Page 2) against the following page (Page 3). Since they are discontinuous with each other, the file system first issues the request (Request #2) for Page 2 and completes the I/O. Likewise, in order, the remaining pages (Page 3, Page 4) are issued and completed as separate requests (Request #3 and Request #4). In this example, since the LBAs (20, 89, and 45) are all discontinuous, the file system performs four operations.

In write-back operations, the LBA's contiguity is dependent on the sequence of the dirty pages. Consequently, current read-ahead and write-back operations can reduce the bandwidth by incurring several requests instead of one large request among discontinuous pages.

2.4 Journal I/O in the Journaling File Systems

We describe journal I/O operations, such as journal metadata/data, commit, and checkpoint, based on the data journaling mode. Figure 2.2a shows the journal metadata/data and commit operations in a transaction for the existing

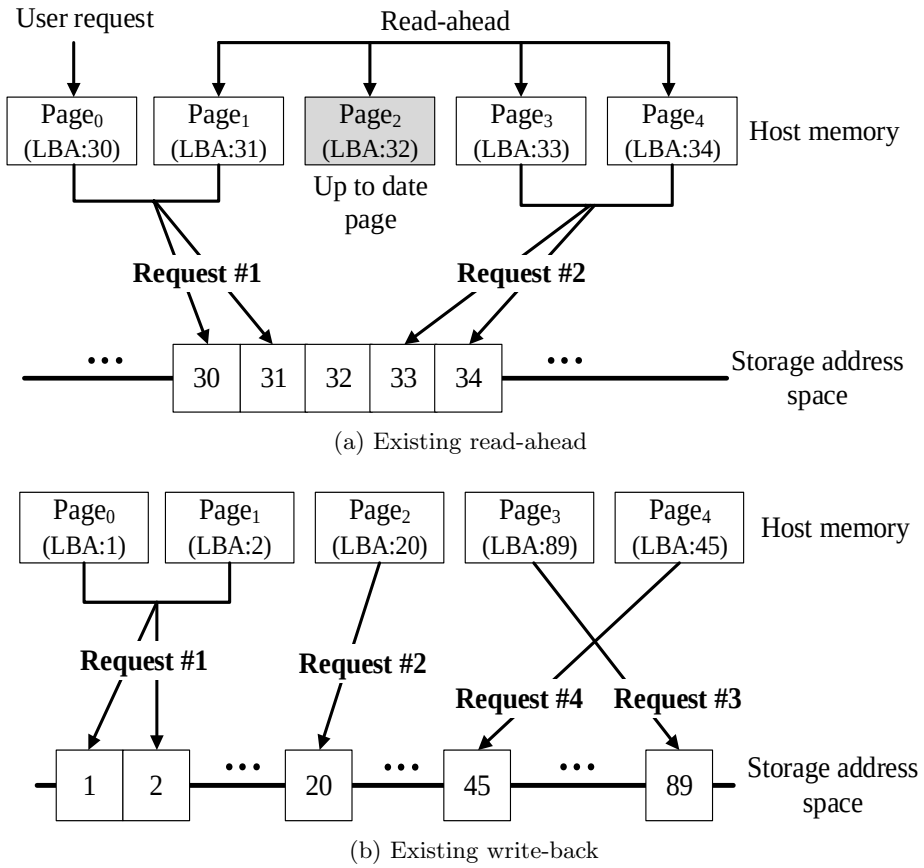
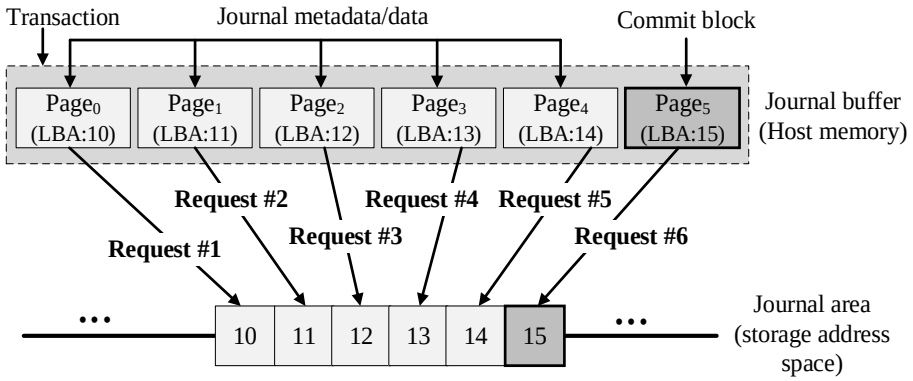


Figure 2.1: Read-ahead and write-back of existing file system

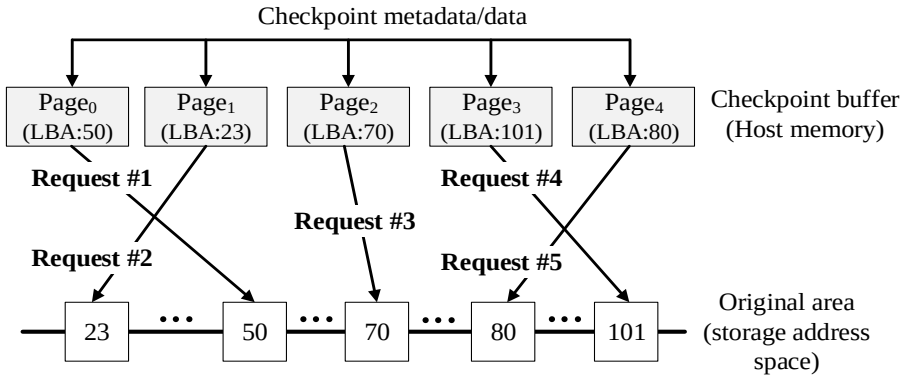
file system. There are five pages (Page 0-4) for journal blocks and one page (Page 5) for a commit block to be written into the journal area. They are mapped to each LBA: 10, 11, 12, 13, 14, and 15 respectively. The file system gets a journal block from the journal buffer, issues the I/O and completes the I/O repeatedly until the I/O for the journal blocks in the transaction are finished. After the five pages (Page 0-4) for the journal blocks completely are written into the journal area, the page (Page 5) for a commit block is written into the journal area resulting in the transaction being committed. As a result, the journal I/Os for each LBA make several requests (Request #1-#6).

We note that the I/O pattern for the journal metadata/data written to the journal area is sequential. In a conventional block I/O subsystem, the sequential writes in the journal I/O can be merged by using functions supported by an I/O scheduler. For example, the JBD2 module uses two functions such as `blk_start_plug()` and `blk_finish_plug()`. The I/O scheduler merges sequential write requests between `blk_start_plug()` and `blk_finish_plug()` call into a single large request. This mechanism is a disk-friendly feature. However, in the case of high-performance storage devices, an I/O request bypasses the I/O scheduler due to the well-known performance issue [14]. Thus, the JBD2 module cannot explicitly use the features mentioned above due to the absence of the I/O scheduler, and this leads to individual write requests for the journal I/O. In our work, we use an optimized block I/O subsystem without the I/O scheduler since it shows the best performance among all configurations. The sequential writes in the journal I/O are performed as individual requests one by one in the optimized block I/O subsystem.

Figure 2.2b shows the checkpoint operations in the existing file system. There are five pages (Page 0-4) for checkpoint updates in the checkpoint buffer, and they are mapped to each LBA: 23, 50, 70, 101, and 80, respectively. Since the checkpoint operation writes the metadata/data blocks in the committed transaction to the original area, their LBAs can be discontiguous to each other. The file system gets a block (page) from the checkpoint buffer and then issues and completes the block iteratively. As shown in this figure, there are five separate requests (Request #1-#5). In conclusion, journaling and checkpoint operations are issued and completed by each request per page. This current I/O operation can reduce the bandwidth by incurring several requests instead of one large request among pages.



(a) Existing journal metadata/data



(b) Existing checkpoint

Figure 2.2: Journal metadata/data and checkpoint of existing file system

2.5 Recovery in the Journaling File Systems

In this section, we describe the recovery procedure in the existing journaling file system. After a system crash or power outage, the mount process reads the journal blocks from the journal area and replays the changes until the file system is consistent again. The changes are atomic in that they are either replayed completely during recovery or are not replayed at all if they had not yet been completely written to the journal area before the crash occurred.

We analyze the recovery I/O path in the JBD2 module. The module per-

forms read operations for getting all the journal blocks in the journal area. The module then performs the checksum operation for the scanned blocks and then selects blocks in the committed transactions. After all blocks to be replayed are selected, the blocks are written to their original area by a sync operation, which writes the blocks one by one. The journal area is initialized after the blocks in the journal area are completely written to the original area.

Figure 2.3 shows an example of a recovery procedure in terms of I/O operations. The mount process reads the blocks (Page 0-4) mapped to each LAB: 1, 2, 3, and 4, in the journal area through four requests (Request #1-#4). The JBD2 module goes through the block device layer directly for the read operation, and therefore, no read-ahead is performed. Similar to the case of journal metadata/data I/O, in the Linux I/O scheduler-based system, adjacent blocks can be merged. However, the I/O stack without the I/O scheduler performs the I/Os as individual requests.

During the recovery procedure, the mount process identifies the journal blocks and a commit block. If the commit block exists, as shown in the figure, the journal blocks can be recovered and are written into the original area with LBA: 33, 56, 78, through three requests (Request #4-#6). Consequently, as our observation, the existing mount process performs inefficient I/O operations by issuing several requests to storage.

2.6 Existing Locking and I/O Parallelism in Journaling File Systems

In this section, we investigate the locking and I/O parallelism in EXT4/JBD2. As shown in Figure 2.4a and 2.4b, a spin lock (`j_list_lock`) is used to ensure the correct list operations for journal heads (`jhs`)[2] in the journaling lists

[2]Journal head (`jh`) is a structure that associates the buffer (buffer_head (`bh`)) with the respective transaction [39]. The operations on the `bh` are protected by a spin lock

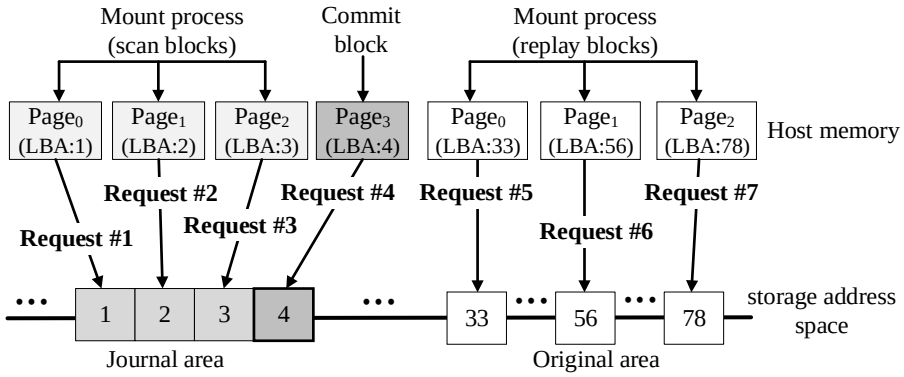
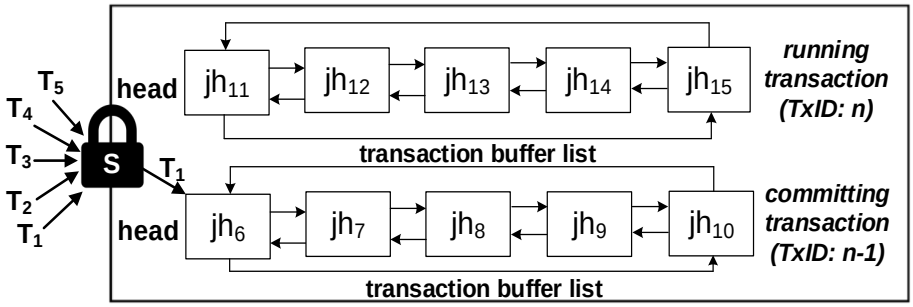


Figure 2.3: Existing recovery I/O operations

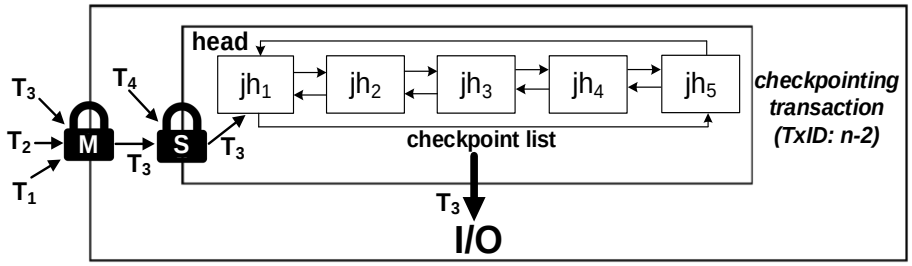
(transaction buffer and checkpoint lists) [15], which are circular doubly linked lists. However, in multi-cores, this locking can incur a contention on the shared data structures and limit the scalability. In addition, only a single thread performs the journal and checkpoint I/Os. For example, as shown in Figure 2.4b, T_3 performs I/O operations for checkpointing by acquiring a mutex lock (`j_checkpoint_mutex`). Such serialized I/O operations can limit the I/O parallelism on high-performance storage. We will explain the transaction processing in terms of locking and I/O operations with the following simplified procedures.

Running transaction. When application threads perform some file operations (e.g., `create()`), they start a transaction to handle the modifications (Procedure 1, lines 3 and 31-39). To process the transaction, the threads first check if a running transaction is available or not. If a running transaction is available, the threads join the running transaction by increasing the number of updates (`t_updates`) in the transaction under the state lock (`j_state_lock`) which is a read-write lock; the `t_updates` variable indicates the number of current threads that join the transaction. Otherwise, a new transaction is created,

(`jbd_lock_bh_state`) per `bh`.



(a) running and committing transaction



(b) checkpointing transaction

Figure 2.4: Examples of existing locking and I/O operations (T: thread, TxID: transaction ID, jh: journal_head, S: spin lock (`j_list_lock`)), M: mutex lock (`j_checkpoint_mutex`)

or the threads are scheduled[3] if the transaction cannot be newly created.

After getting the running transaction (line 4), the threads modify their own buffer and then try to insert it into a transaction buffer list by using the `jh` of the buffer (`bh`). To insert the `jh`, the threads try to acquire a list lock (`j_list_lock`) which is a spin lock (lines 5-6). A thread, which acquires the list lock, associates the `jh` to the running transaction (line 45) and inserts the `jh` into the tail of the list (line 46). Then, the thread releases the list lock and finishes the insert operation (line 7). Finally, the thread completes its own transaction processing

[3]When a running transaction needs to be committed while a previous transaction is committing, the threads which try to get a running transaction are scheduled until the running transaction is available. It is because there are only one running transaction and one committing transaction at any time in the compound transaction scheme [15, 36].

PROCEDURE 1 C-like pseudo-code of running transaction in EXT4/JBD2

```
1: create(dir, ...){
2:      /* create a new file */
3:     handle = jbd2_journal_start(journal, ...);
4:     transaction = handle->transaction;
5:     spin_lock(journal->j_list_lock);
6:     add_buffer(bh->jh, transaction, transaction->t_buffers);
7:     spin_unlock(journal->j_list_lock);
8:     jbd2_journal_stop(handle);
9: }

10: truncate(dentry, ...){
11:      /* truncate a file */
12:     journal_unmap_buffer(journal, bh);
13: }

14: journal_unmap_buffer(journal, bh){
15:     /* invalidate a buffer */
16:     write_lock(journal->j_state_lock);
17:     spin_lock(journal->j_list_lock);
18:     transaction = bh->jh->transaction;
19:     if(!bh->jh->cp_transaction){
20:         head = jh->cp_transaction->t_checkpoint_list;
21:         del_buffer(bh->jh, bh->jh->cp_transaction, head);
22:     }else if(transaction == journal->j_committing_transaction){
23:         set_buffer_free(bh);
24:     }else if(transaction == journal->j_running_transaction){
25:         head = journal->j_running_transaction->t_buffers;
26:         del_buffer(bh->jh, transaction, head);
27:     }
28:     spin_unlock(journal->j_list_lock);
29:     write_unlock(journal->j_state_lock);
30: }
```

by decreasing the number of updates (lines 8 and 40-43).

When application threads perform some file operations, such as `truncate()`, the threads can invalidate buffers that are already associated with a transaction

```
31: jbd2_journal_start(journal, ...){
32:     if(j_running_transaction is not available)
33:         <create a new transaction or call schedule()>
34:     read_lock(journal->j_state_lock);
35:     handle->transaction = journal->j_running_transaction;
36:     atomic_add(transaction->t_updates, 1);
37:     read_unlock(journal->j_state_lock);
38:     return handle;
39: }

40: jbd2_journal_stop(handle){
41:     /* complete a transaction */
42:     atomic_sub(handle->transaction->t_updates, 1);
43: }

44: add_buffer(jh, transaction, head) [a] {
45:     jh->transaction = transaction;
46:     tail = head->prev;
47:     if(!head){
48:         jh->next = jh->prev = head = jh;
49:     } else{
50:         jh->prev = tail; jh->next = head; tail->next = head->prev = jh;
51:     }
52: }

53: del_buffer(jh, transaction, head) [a]{
54:     if(head == jh){
55:         head = jh->next;
56:         if(head == jh)
57:             head = NULL;
58:     }
59:     jh->prev->next = jh->next; jh->next->prev = jh->prev;
60:     jh->transaction = NULL;
61: }
```

[a] The `jh` is inserted into/removed from a transaction buffer list or checkpoint list by using the `prev/next/transaction` or `cpprev/cpnext/cp_transaction` fields of the `jh`, respectively.

(lines 10-13, 14-30, and 48-51). In this case, by acquiring the state lock and the list lock (lines 16-17), a thread removes the `jh` from the transaction buffer or checkpoint lists (line 49) and disassociates the `jh` with the running or checkpoint transactions (line 50) if it is associated with the running or checkpoint transactions, respectively. If the `jh` is associated with a committing transaction, the thread sets the `jh` as *freed*; both the `jh` and its buffer will be freed later during the commit procedure. As discussed above, EXT4/JBD2 ensures correct updates on the transaction state and the transaction buffer list by the state lock and the list lock, respectively.

Committing transaction. To commit a transaction, a journal thread wakes up and processes a commit procedure (Procedure 2). First, the thread changes the running transaction to a committing transaction and its state to *committing*. Then, the thread initializes the running transaction by acquiring the state lock (lines 2-5). And then, the journal thread waits for other threads to complete their transaction processing by checking the `t_updates` variable (line 6). Therefore, if the `jh` is already associated with a running transaction, the `jh` must be moved to a committing transaction. Meanwhile, the committing transaction does not accept any new modifications, and the next modification triggers the creation of a new running transaction. With the committing transaction, the journal thread prepares for journal I/Os by creating a wait list, which is used to wait the I/Os (line 8). Then, the thread fetches the `jh` from the head (`t_buffers`) of the transaction buffer list and creates a copy of its buffer called frozen buffer (`frozen_bh`) to preserve the contents of the buffer (lines 9-11). And then, the thread removes the `jh` from the list by updating the head of the list to the next of the head, and inserts the `jh` into the shadow list[4]

[4]The shadow list (`t_shadow`) includes the frozen buffers which preserves the contents of the buffers.

under the list lock (lines 12-15).

To perform a batched journal I/O, the journal thread aggregates the frozen buffer by inserting it into a write buffer (`wbuf`) and the wait list (lines 16-17). If the number of inserted buffers (`bufs`) is higher than the pre-defined threshold, the thread issues I/Os to the journal area by calling `submit_bh()` and prepares for the next I/Os (lines 18-22). After issuing all the I/O requests for journaling, the thread waits for the I/Os and then removes the `jh` from the shadow list and inserts it into the forget list[5] under the list lock (lines 24-32). After all the I/Os are completed, the journal thread writes the commit block for the transaction atomicity (line 33); if a crash occurs, the file system can replay or discard the transaction according to the existence of the commit block of the transaction. Then, the thread makes a checkpoint list with the buffers that are not freed and still dirty in the forget list under the list lock (lines 34-42). Finally, the committed transaction is inserted into the tail of a checkpoint transaction list for checkpointing by acquiring the state and list locks (lines 43-48).

Checkpointing transaction. When a transaction needs to be checkpointed, application threads try to acquire a checkpoint mutex lock (`j_checkpoint_mutex`) and perform a batched I/O operation (Procedure 3, line 2). A thread, which acquires the mutex lock, performs the checkpoint I/O operations while others are blocked by the lock until the I/O operations are completed. Then, the thread tries to acquire the list lock to get the transaction and access its checkpoint list (lines 3-9). The list lock is used since other threads can access the checkpoint list to remove the `jh`s when they free the buffers of the `jh`s, which do not need to be checkpointed.

[5]The forget list (`t_forget`) includes both the frozen buffers from the shadow list and buffers to be freed. In some cases, when an application thread frees a buffer which is associated with a transaction but not needed to be checkpointed, the thread inserts the `jh` of the buffer into the forget list. By doing so, the `jh` is not inserted into the checkpoint list at the commit procedure.

PROCEDURE 2 C-like pseudo-code of committing transaction in EXT4/JBD2

```
1: jbd2_journal_commit_transaction(journal){
2:     transaction = journal->j_running_transaction;
3:     write_lock(journal->j_state_lock);
4:     journal->j_committing_transaction = transaction;
5:     journal->j_running_transaction = NULL;
6:     while(atomic_read(transaction->t_updates)){...}
7:     write_unlock(journal->j_state_lock);
8:     create_wait_list(local_wait_list); // create a local wait list
9:     while(transaction->t_buffers){
10:        jh = transaction->t_buffers;
11:        <making a frozen buffer (frozen_bh)>
12:        spin_lock(journal->j_list_lock);
13:        del_buffer(jh, transaction, transaction->t_buffers);
14:        add_buffer(jh, transaction, transaction->t_shadow);
15:        spin_unlock(journal->j_list_lock);
16:        wbuf[bufs++] = jh->frozen_bh;
17:        add_wait_list(local_wait_list, jh->frozen_bh);
18:        if(bufs == journal->j_wbufsize){ /*j_wbufsize: 341*/
19:            for(i=0 ; i<bufs ; i++)
20:               submit_bh(WRITE, wbuf[i]);
21:            bufs=0;
22:        }
23:    }
24:    while(!list_empty(local_wait_list)){
25:        frozen_bh = list_entry(local_wait_list.prev, ...);
26:        wait_on_buffer(frozen_bh);
27:        jh = transaction->t_shadow->prev;
28:        spin_lock(journal->j_list_lock);
29:        del_buffer(jh, transaction, transaction->t_shadow);
30:        add_buffer(jh, transaction, transaction->t_forget);
31:        spin_unlock(journal->j_list_lock);
32:    }
33:    <issue and complete a commit block>
```

Under the mutex and list locks, the thread aggregates the buffers by fetching the `jh`s from the checkpoint list and inserting the fetched buffers into a

```
34:     spin_lock(journal->j_list_lock);
35:     while(transaction->t_forget){
36:         jh = transaction->t_forget;
37:         jh->transaction = NULL;
38:         if(!buffer_freed(jh->bh) && jbddirty(jh->bh))
39:             add_buffer(jh, transaction, transaction->t_checkpoint_list);
40:         del_buffer(jh, transaction, trasnaction->t_forget);
41:     }
42:     spin_unlock(journal->j_list_lock);
43:     write_lock(journal->j_state_lock);
44:     spin_lock(journal->j_list_lock);
45:     <insert the committed transaction into a checkpoint transaction list
46:         (journal->j_checkpoint_transactions)>
47:     spin_unlock(journal->j_list_lock);
48:     write_unlock(journal->j_state_lock);
49: }
```

checkpoint buffer (`j_chkpt_bhs`) to issue the I/Os in a batched manner (lines 9-21). Similar to the commit procedure, the `jh` is removed and re-inserted into a checkpoint io list, which is used for I/O completion. If the number of aggregated buffers (`batch_count`) is higher than the pre-defined threshold, the thread releases the list lock and issues the I/Os. Then, the thread prepares for the next I/Os by acquiring the list lock. After issuing all the I/Os, the thread completes them in a batched manner by using the checkpoint io list under the list lock (lines 22-34). After then, the thread sets the next transaction to be checkpointed in the checkpoint transaction list. Finally, the checkpointed transaction is freed, which denotes the end of a life cycle of the transaction, and the list lock and the mutex lock are released (lines 32 and 35-36).

PROCEDURE 3 C-like pseudo-code of checkpointing transaction in EXT4/JBD2

```
1: jbd2_log_wait_for_space(journal){
2:     mutex_lock(journal->j_checkpoint_mutex);
3:     spin_lock(journal->j_list_lock);
4:     if((transaction = journal->j_checkpoint_transactions) == NULL){
5:         spin_unlock(journal->j_list_lock);
6:         mutex_unlock(journal->j_checkpoint_mutex);
7:         return;
8:     }
9:     while(transaction->t_checkpoint_list){
10:        jh = transaction->t_checkpoint_list;
11:        journal->j_chkpt_bhs[batch_count++] = jh->bh;
12:        del_buffer(jh, transaction, transaction->t_checkpoint_list);
13:        add_buffer(jh, transaction, transaction->t_checkpoint_io_list);
14:        if((batch_count == JBD3_NR_BATCH)){/*JBD3_NR_BATCH:64*/
15:            spin_unlock(journal->j_list_lock);
16:            for(i=0;i<batch_count;i++)
17:                submit_bh(WRITE, journal->j_chkpt_bhs[i]);
18:            batch_count = 0;
19:            spin_lock(journal->j_list_lock);
20:        }
21:    }
22:    while(transaction->t_checkpoint_io_list){
23:        jh = transaction->t_checkpoint_io_list;
24:        spin_unlock(journal->j_list_lock);
25:        wait_on_buffer(jh->bh);
26:        spin_lock(journal->j_list_lock);
27:        del_buffer(jh, transaction, transaction->t_checkpoint_io_list);
28:        if(transaction->t_checkpoint_list == NULL &&
29:            transaction->t_checkpoint_io_list == NULL){
30:            <set the next transaction to be checkpointed
31:                    in the checkpoint transaction list>
32:            free(transaction);
33:        }
34:    }
35:    spin_unlock(journal->j_list_lock);
36:    mutex_unlock(journal->j_checkpoint_mutex);
37: }
```

Chapter 3

Design and Implementation

3.1 Optimizing File Systems for Low-latency Storage Devices

In this section, we describe the design of our optimization techniques to increase the bandwidth per thread for read and write operations. The key idea is to combine multiple and individual pages into a single large request and issue/complete it irrespective of the LBAs for each page.

3.1.1 Design

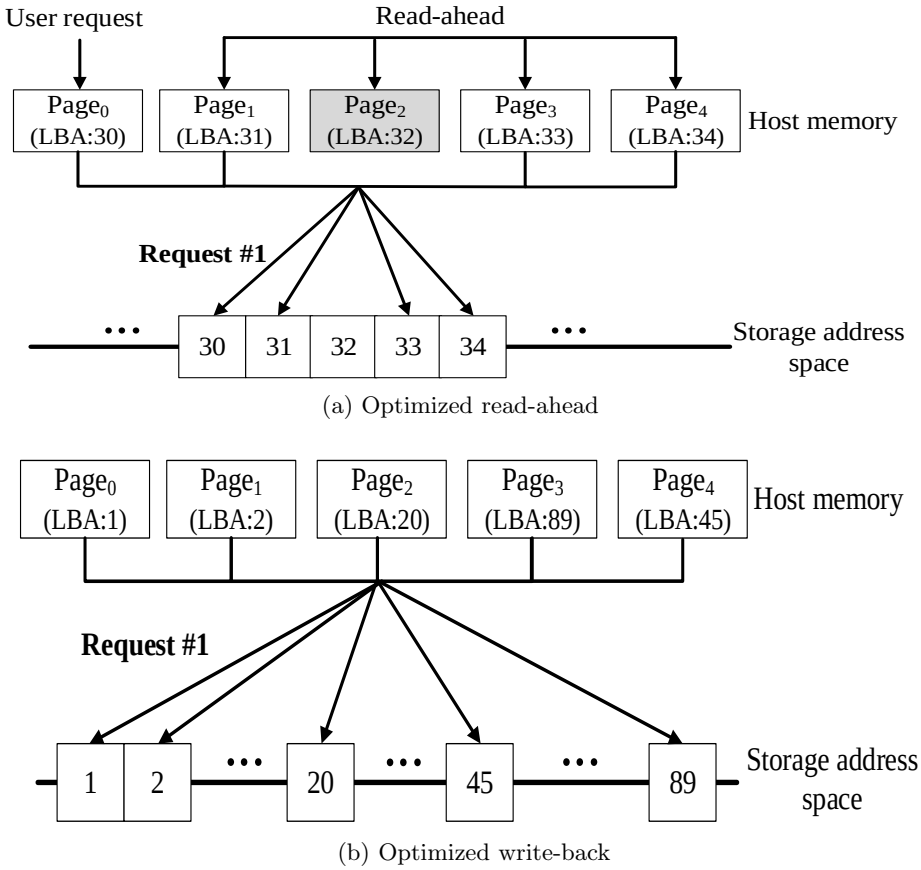
Read and write operations in the optimized file system

In the case of the read operation, we observe the read-ahead dilemma of whether to use prefetching or not. Enabling the read-ahead technique is not beneficial to the random access workload since the prefetched data is not normally expected under the workload. Thus, disabling the technique is advantageous to the random access workload while it degrades the sequential read throughput significantly as a side effect; performance is decreased by about 50% without

prefetching. To resolve this dilemma, the baseline system disables the context lookup feature [14]. It reduces the number of prefetch pages under a random read workload while still providing a sufficient number of prefetched pages for the sequential read workload. As a result, the random read performance is improved without degrading the sequential read performance.

Figure 3.1a outlines the read-ahead of the optimized file system. There are five pages (Page 0-4) where Page 0 is the demanded page and the other pages (Page 1-4) are contiguous pages which the file system wants to read ahead. The pages are mapped to each LBA, and the state of Page 2 is already up-to-date so that the pages to be read are Page 0, 1, 3, and 4. In this situation, the existing file system issues two requests such as a request for Page 0-1 and another request for Page 3-4. Unlike the existing file system, our optimized file system gathers the pages (Page 0, 1, 3, and 4) and issues/completes a single large request (Request #1) with gathered pages. This scheme demonstrates that the file system increases the number of I/Os per request and reduces the number of operations for issue and completion irrespective of the LBA's contiguity according to the state of the pages.

Figure 3.1b describes write-back of the optimized file system. There are five dirty pages (Page 0-4) from the page cache, and they are mapped to each LBA. The optimized file system merges the dirty pages from the page cache into a single request (Request #1). This scheme shows that the sequence of the dirty pages negatively affects the bandwidth per thread. It also demonstrates that the performance of the read operations can be improved by reducing the flushing time whenever the write-back operation occurs. Finally, our scheme does not sacrifice the consistency of the current file systems by preserving the metadata and journaling mechanism.



(a) Optimized read-ahead

(b) Optimized write-back

Figure 3.1: Read-ahead and write-back of the optimized file system

Journal I/O in the optimized file system

In this section, we describe the optimization techniques in the journal I/O operations based on the data journaling mode. This optimization provides efficient journaling/checkpoint operations and reduces the time for I/O operations. Our journal I/O scheme still guarantees the same consistency as that of the existing file system.

To increase the bandwidth per journal I/O operation, similar to optimization for the read and write operations, the optimized file system combines

multiple and individual journal pages (blocks) into a large request and issues/completes the request. When a transaction starts to commit, the existing file system makes a temporary I/O buffer in which the journal blocks are included, which was updated by the transaction. Then, the existing file system performs I/O for the block from an I/O buffer one by one. In contrast, the optimized file system makes blocks in the I/O buffer into a large request and issues the request to the device driver.

Figure 3.2a shows the journal metadata/data and commit operations in a transaction for the optimized file system. There are five pages (Page 0-4) for the journal blocks and one page (Page 5) for a commit block. In this example, the pages (Page 0-5) are mapped to LBA 10, 11, 12, 13, 14, and 15. In contrast to the existing file system, the optimized file system merges the journal blocks into a single request (Request #1) and issues the request. After the transfer of the journal blocks is finished, the I/Os are completed at once. To provide the crash consistency at the same level as that of the existing file system, after the I/Os for journal blocks are completely finished, we issue and complete the I/O for the commit block in a request (Request #2). Consequently, our scheme still supports the same consistency with that of the existing file system since our scheme preserves the write ordering between the journal block and the commit block.

When a new transaction is started, the file system checks whether there is enough space left in the journal area to write all potential buffers requested. If there is enough space, the transaction is continuously progressed. Otherwise, the upcoming I/O needs to stall pending a checkpoint to free up some more journal space. Therefore, fast checkpoint operation is required to increase the I/O performance by reducing the stall time. Figure 3.2b describes the optimized checkpoint operations. The checkpoint buffer includes the metadata and

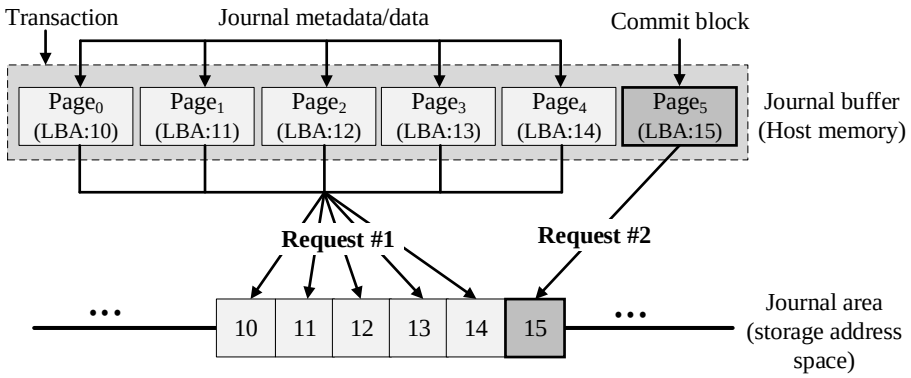
data to be rewritten into the original area. There are five pages (Page 0-5). In this example, the pages are mapped to LBA 50, 23, 70, 101, and 80. When the checkpoint is activated, the optimized file system makes the blocks in the checkpoint buffer into a single request (Request #1) and issues/completes the request at once, irrespective of the LBA's contiguity.

Our scheme allows the file system to increase the number of I/Os per request and reduces the number of operations for issue and completion in journal/checkpoint operations. Consequently, our scheme reduces the transfer time for journal I/O and supports shorter journal work by providing efficient I/O operations.

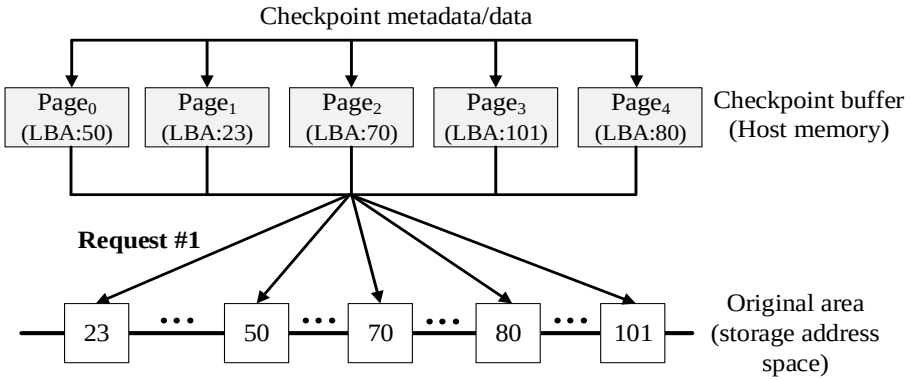
Recovery in the optimized file system

In this section, we present an efficient recovery mechanism in the journaling file system. We provide the optimizations for both scan and replay operations. Our scheme is to make several pages to be scanned and replayed into large requests. In the ordered mode, only the metadata in the committed transaction is replayed to the original metadata area while both the metadata and data in the committed transaction are replayed to their original area in the data journaling mode. Similar to the existing file system, the optimized file system initializes the journal area after the recovery procedure is completely finished.

Figure 3.3 shows an example of the optimized recovery procedure in the optimized file system. There are four pages (Page 0-3) mapped to LBA 1, 2, 3, and 4 in the journal area. When the system is restarted after a system crash or power outage, the optimized file system reads the pages in the journal area. Unlike the existing recovery procedure, the optimized file system reads the pages in a request (Request #1). This optimization allows the mount process to the scanning and selecting the pages to be replayed faster. After scanning, the selected



(a) Optimized journal metadata/data



(b) Optimized checkpoint

Figure 3.2: Journal metadata/data and checkpoint of the optimized file system

pages (Page 0-2) mapped to LBA 33, 56, and 78 are written into the original area. In contrast to the existing file system, the optimized file system makes the pages into a request (Request #2). In this example, our scheme issues and completes the two requests for each scan and replay operation. Consequently, the optimized file system decreases the recovery time and allows the mount process to perform more efficient recovery I/O operations.

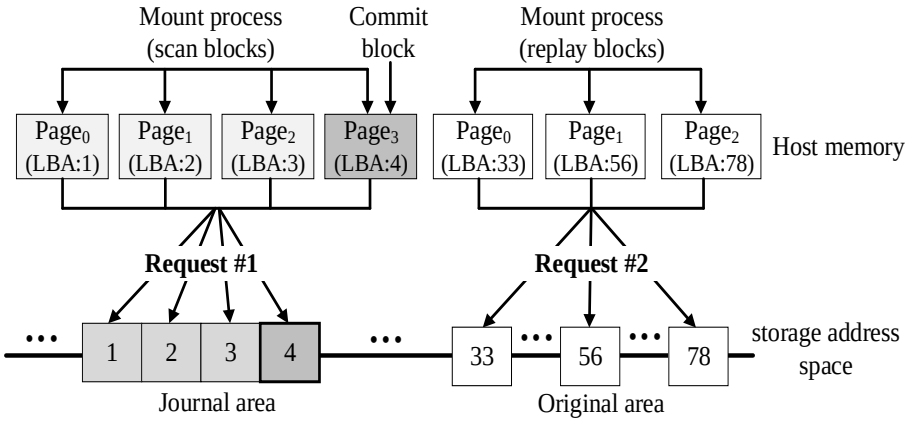


Figure 3.3: Optimized recovery procedure

3.1.2 Implementation

Our optimization requires a DMA engine of the storage device to support the capability of transferring data between discontiguous host memory pages of the file system and discontiguous storage address spaces. To support the capability, the DMA engine of the DRAM-SSD is customized by using a set of descriptors for an I/O request. A descriptor is defined, which includes a mapping of host memory segment, storage segment, and data size. A data structure is defined called Block Control Table (BCT) to contain the descriptors. BCT can contain 1,024 descriptors maximally. Therefore, the 1,024 segments in a single request can be dispatched at once.

Our optimizations can be applied on other types of fast storage devices, which support an ultra-low latency (e.g., a few microseconds) and the capability of transferring data between discontiguous host memory segments and discontiguous storage address spaces. For example, the devices with fast storage medium (a type of memory such as PCM, STT-MRAM, and so on) has an ultra-low latency. RAMCloud also provides low latency access by storing all data in DRAM at all times. The file systems on the fast storage devices or fast

storage systems with fast remote memory access can be optimized if the devices or systems can support the transfer capability. In terms of standard interfaces, the SATA and NVMe do not support the data transfer between discontiguous host memory segments and discontiguous storage address spaces. Therefore, our optimizations are hard to be applied on current flash-based SSDs with the SATA and NVMe interfaces.

Our scheme requires modification of file system and device driver. In read and write operations, the existing file system builds the BIO structure to forward requests to the block layer; the structure is the basic container for block I/O. The existing file system upon the existing block layer identifies only pages with contiguous LBAs and adds the pages to the BIO structure; the BIO structure has the starting LBA in the `bi_sector` field of the structure. To implement our scheme, we define a new data structure called PIO (Proposed I/O) for the page transfers between the file system and the device driver. The PIO structure consists of page vectors, the total request size, and the number of pages. Each page vector contains the page, length, offset, and sector addresses (LBA) to represent the mapping between a single page of the file system and a specific LBA. This enables the file system to transfer multiple LBAs to the device driver.

To this end, we implemented new functions between the file system and the device driver. When a PIO instance for I/O operation arrives from the file system, the device driver allocates as many request descriptors as the total number of pages (`nr_pages` in the PIO structure). Then, the device driver prepares a DMA operation; the device driver calculates the appropriate DMA address for each page in the `dma_map_sg()` function and completes the allocated request descriptors with the DMA address (host memory address), storage address (LBA), and length. Finally, the device driver issues the request descriptors to the storage device.

The file system adds pages for read-ahead operations to a page pool list. The file system then finds the LBAs of the pages from the list one by one. The optimized file system moves pages from the page pool list to the page_vec structure of the PIO with information for each page by not checking the contiguity among the pages. The file system then transfers the PIO via a customized read interface exported by the device driver. We modified the `ext4_readpages()` that is almost identical to other file systems except the block retrieval mechanism.

When write-back occurs, the optimized file system finds dirty pages from the page cache via `pagevec_lookup_tag()` to get the LBAs of the dirty pages. The file system does not check the contiguity of the pages and merges them into page_vec in PIO. In the write-back operation, we modified `ext4_writepages()`. Similar to the optimized read operation, the file system issues the request with PIO via a customized write interface exposed by the device driver. We expect that applying these optimization techniques to other file systems would be relatively easy since the modifications are included in common functions of the Linux file system.

For journaling and recovery optimization, we changed the JBD2 module. We modified `jbd2_journal_commit_transaction()` for journal metadata/data optimization; this function is the primary function for the commit procedure. When the transaction commit occurs, the journal thread wakes up and performs the journal commit procedure. In this situation, the optimized file system gets the journal metadata/data buffer to be transferred and sends the buffer to the device driver using a customized function that is similar to those used in read/write operations. The journal thread issues and completes the I/O for the journal blocks at once and subsequently the I/O for the commit block.

For optimization of the checkpoint, we modified `flush_batch()`, which performs the checkpoint I/O operation in the JBD2 module. The optimized file

system gets the checkpoint buffer and transfers it to the device driver via the customized function. The file system issues and completes the I/O for the checkpoint at once. In short, we add two functions as the interfaces between JBD2 and the device driver. By using the interfaces, when our file system issues the journal metadata/data or checkpoint buffer, our device driver prepares the DMA operation for a single large request similar to read and write operations.

To provide faster recovery, we modified `jbd2_journal_recover()` that is the primary function for recovery when mounting a device. To support a large read request for journal blocks, the optimized file system aggregates the blocks to be read and then issues/completes the aggregated blocks by a request via the customized function. In the replay operation, the mount process writes the selected blocks to the original area via the customized function to write the dirty blocks to be replayed at once without their contiguity. Consequently, in the recovery procedure, we modify the read and re-write operations for journal blocks, which then makes the procedure more efficient and decreases the recovery and remount time.

3.2 Optimizing File Systems for Highly Parallel Storage Devices

To achieve higher I/O performance on multi-cores with high-performance storage, we aim to reduce the lock contention and maximize I/O parallelism in transaction processing. To do this, we propose a transaction processing with two main schemes that enable concurrent updates on shared data structures and cooperatively parallelize I/O operations. We apply these schemes to the transaction processing in EXT4/JBD2.

We maintain the compound transaction scheme of EXT4/JBD2 to exploit its advantages [36]. For example, it provides a better performance when the same

metadata or data is frequently updated within a short period of time. With this advantage, we implement our schemes in the compound transaction. We also preserve the existing ordering of write operations and transactions, such as the ordering of journal blocks and a commit block, committing and checkpointing, and checkpoints. Therefore, our implementation does not require modifications to the existing recovery procedure while not sacrificing the consistency of the file system.

Furthermore, we do not optimize all locking operations in transaction processing but focus on the list lock for management of journal heads and the checkpoint mutex lock for serialized I/O operations. Compared to the list lock and the mutex lock, other locks, such as state lock do not incur a significant overhead according to our evaluation as well as other works [15]. However, such locks can be a performance bottleneck in a massive number of cores, which is beyond this dissertation; therefore, we leave the latent performance issue as a future work.

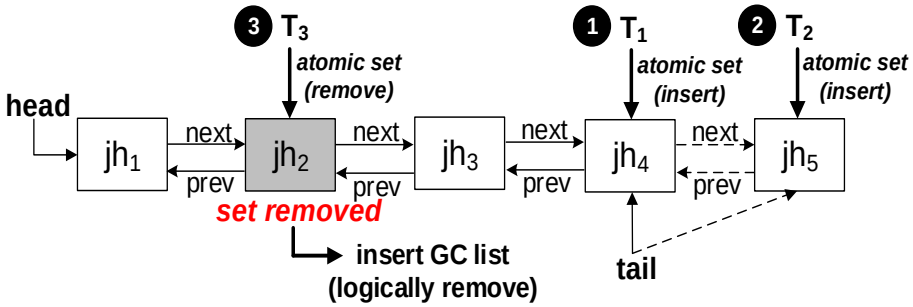
3.2.1 Design

Concurrent updates on data structures

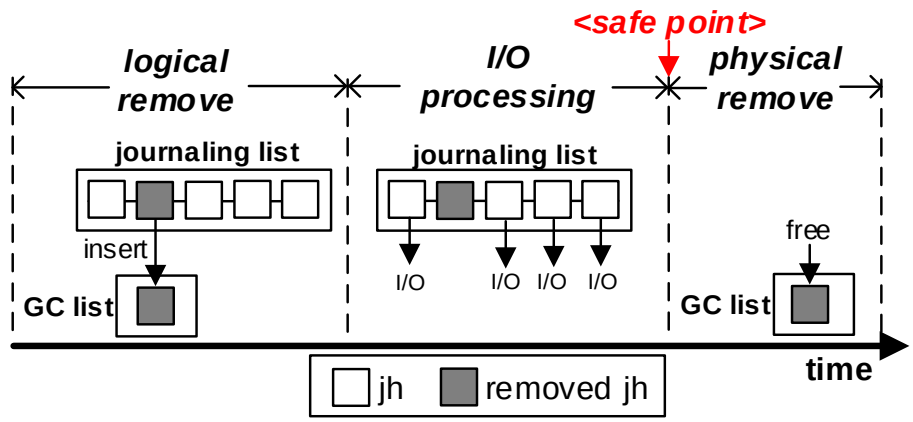
We manage the linked lists for transaction processing in a lock-free manner as shown in Figure 3.4. To this end, instead of the existing circular doubly linked lists, we use non-circular doubly linked lists and add the `tail` to the lists to enable lock-free operations[1].

INSERT. We provide a concurrent insert operation to add an item to a list. In the existing transaction processing, the items are inserted into the tail

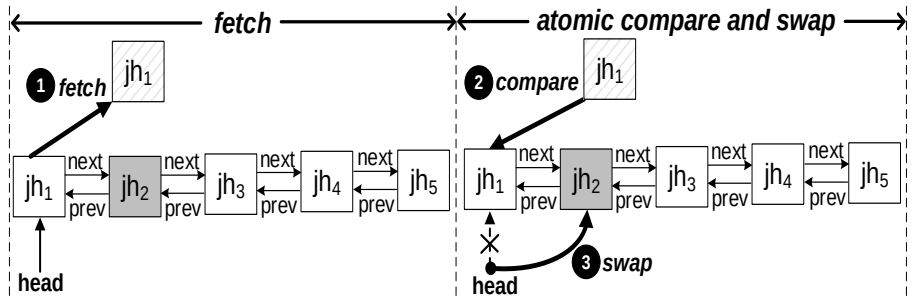
[1] In the circular doubly linked list, when an item is inserted into the list, the multiple pointers that link the item, head, and tail are updated, which makes the atomic insert operation difficult. Instead, we add the tail and set the tail's next item as a constant *NULL* variable [40], which allows us to identify the last element of the list and insert the item into the tail atomically.



(a) Insert and remove operations in a lock-free manner (T: thread)



(b) Two-phase removal (GC: garbage collection)



(c) Fetch operations in a lock-free manner

Figure 3.4: Concurrent updates on data structures

of the list in the incoming order. Similar to the existing scheme but without locking, we concurrently update the tail by the incoming items using an atomic

set instruction. In an example shown in Figure 3.4a, before jh_5 is inserted into a journaling list (e.g., transaction buffer list or checkpoint list), the journaling list consists of four `jh`s, and the tail points jh_4 which is inserted by T_1. When T_2 inserts jh_5, the thread atomically updates the tail and the jh_5's previous item by jh_5 and jh_4, respectively, by executing the atomic set operation. By updating the previous item (jh_4) of jh_5 atomically, the next item of jh_4 is decided as jh_5. This insert operation allows multiple threads to add their item concurrently by updating the tail and linking atomically.

REMOVE. We provide a concurrent remove operation to delete an item from a list. When items are removed from the list concurrently without locking, the *invalid reference* problem [41] can occur. To address this issue, we propose a two-phase remove operation that marks an item as "removed" (logical remove) and then frees the item (physical remove) at a *safe point* when no other threads hold any references to the transaction and logically removed items. This scheme ensures safe access to the items of the list and so threads can perform appropriate operations for the items. For safe garbage collection (GC) for the logically removed items, we additionally maintain a GC list per transaction.

For example, as shown in Figure 3.4a, when a thread (T_3) tries to remove the `jh` (jh_2), the thread marks the `jh` as *removed* atomically by executing the atomic set instruction. Then, the thread inserts the `jh` into the GC list using our concurrent insert operation as shown in Figure 3.4b. And then, threads perform I/O for the valid `jh` or bypass the I/O for the logically removed `jh` while traversing the list safely. When the transaction arrives at the *safe point*, all items in the GC list are reclaimed. The *safe point* is the point when a transaction is checkpointed. At this point, no other threads reference the logically removed `jh`s in the transaction nor insert any logically removed `jh`s into the GC list of the transaction since all the transaction processing is over. Therefore, we can

free all the logically "removed" `jh`s at the *safe point*.

FETCH. Finally, we provide a concurrent fetch operation to get an item while traversing a list. In the existing transaction processing, the list traversal occurs when no threads insert any items into the list (e.g., journal and checkpoint I/O processing). This ensures that all threads see a consistent view of the list, including valid next pointers of all items. Under this condition, we can simply enable the concurrent fetch operation by using an atomic compare and swap (CAS) instruction. In the example shown in Figure 3.4c, a thread first fetches the current head (jh_1). Then, the thread compares the fetched jh_1 with the current head, and changes the head to jh_1's next item by using the CAS operation. If the thread fails the CAS operation, it repeats the procedure above. This fetch operation allows multiple threads to extract individual items concurrently by updating the head atomically. Consequently, through our concurrent update scheme, multiple threads can insert/remove/fetch their items in the lists concurrently and safely without the existing list lock.

Parallel I/O in a cooperative manner

We provide a parallel I/O in a cooperative manner to maximize the I/O parallelism. In the existing transaction processing, application threads can be scheduled out while the serialized I/O operations (e.g., journal and checkpoint I/O) are performed. On the other hand, in our scheme, we allow the application threads to perform the I/O operations by not scheduling but joining them to the I/O operations. For example, in the case of journal I/O, we allow the threads that cannot get a running transaction to join the I/Os by not scheduling them. In the case of checkpoint I/O, we allow the threads to join the I/Os by eliminating the mutex lock. By joining the multiple threads to the I/O processing, they fetch buffers from the shared linked lists (e.g., journaling lists), issue the

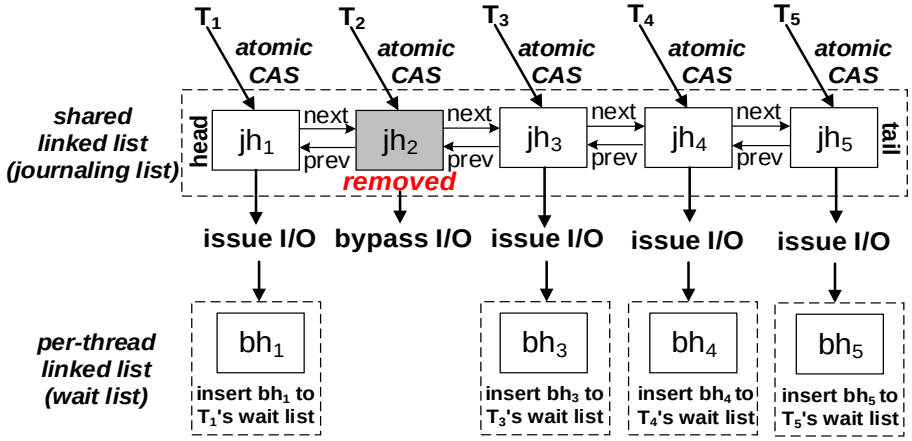


Figure 3.5: Parallel I/O in a cooperative manner (T: thread)

I/Os of the buffers, and complete them in parallel. For better parallelism, we use our concurrent fetch operation and per-thread wait list, which is a linked list used to wait the I/O operations in parallel.

As shown in Figure 3.5, each thread fetches the `jh` concurrently by executing the atomic CAS instruction. Then, each thread issues the I/O of the buffer (i.e., `bh`) associated with the `jh` and inserts the buffer into its own wait list. After all the I/Os are issued, each thread completes its own I/O using its own wait list. Meanwhile, if the fetched `jh` was removed logically, the thread (T_2) does not perform the I/O for the `jh` but fetches the next `jh`.

Using this scheme, multiple threads can cooperate in I/O processing by issuing/completing I/Os in parallel. This can make a commit and checkpoint procedure faster by increasing the I/O parallelism and minimizing the blocking time. We note that our parallel I/O operations can change the I/O ordering between buffers inside a transaction. However, such a change does not sacrifice the atomicity since we write the commit block after all journal blocks are written, which will be described in Section 3.2.2.

The optimized file system with our two schemes preserves the consistency

of the file system by satisfying the following properties: (1) Every block associated with a transaction is written to the journal area at a commit procedure. (2) A transaction is committed or uncommitted (atomicity) according to the commit block. (3) Committed transaction N-1 is checkpointed prior to committed transaction N. We will explain how to apply our schemes to transaction processing and how to satisfy the properties in detail.

3.2.2 Implementation

Running transaction

This section presents our running transaction. We enable multiple application threads to insert/remove the journal heads into/from the transaction buffer list concurrently. Similar to the existing procedure, when the threads start a transaction, they get a running transaction and increase the number of updates in the transaction (Procedure 4, lines 3-4 and 31-39). Meanwhile, in our running procedure, we allow the application threads to cooperate in I/O processing for journal I/Os by calling `journal_io_start()` (lines 32-33), which will be described in Section 3.2.2.

After getting the running transaction, we insert the `jh` into the transaction buffer list by using our concurrent insert operation (lines 5-6 and 44-51). First, the threads associate their `jh` to the running transaction. Then, they update the tail (`t_buffers_tail`) by their `jh` and the `jh`'s previous item by the old tail by executing the `atomic_set` instruction[2]. This instruction updates the tail and returns the old tail atomically. Then, the threads check whether the old tail exists or not. If it does not exist, the head (`t_buffers`) of the list is updated by the inserted `jh`, which becomes the first item in the list. Otherwise, the next item of the old tail is updated by the inserted `jh`.

[2] __sync_lock_test_and_set(type *ptr, type value): This built-in function performs an atomic exchange operation. It writes the value into *ptr, and returns the previous contents of *ptr [42].

PROCEDURE 4 C-like pseudo-code of our running transaction

```
 1: create(dir, ...){
 2:      /* create a new file */
 3:     handle = jbd2_journal_start(journal, ...);
 4:     transaction = handle->transaction;
 5:     add_buffer(bh->jh, transaction,
 6:         transaction->t_buffers, transaction->t_buffers_tail);
 7:     jbd2_journal_stop(handle);
 8: }

 9: truncate(dentry, ...){
10:      /* truncate a file */
11:     journal_unmap_buffer(journal, bh);
12: }

13: journal_unmap_buffer(journal, bh){
14:     /* invalidate a buffer */
15:     write_lock(journal->j_state_lock);
16:     transaction = bh->jh->transaction;
17:     if(!bh->jh->cp_transaction){
18:         head = jh->cp_transaction->gc_head;
19:         tail = jh->cp_transaction->gc_tail;
20:         del_buffer(jh, transaction, head, tail);
21:     }else if(transaction == journal->j_committing_transaction){
22:         set_buffer_free(bh);
23:         atomic_set(jh->removed, removed);
24:     }else if(transaction == journal->j_running_transaction){
25:         head = journal->j_running_transaction->gc_head;
26:         tail = journal->j_running_transaction->gc_tail;
27:         del_buffer(jh, transaction, head, tail);
28:     }
29:     write_unlock(journal->j_state_lock);
30: }
```

For remove operations, we use our two-phase remove operation. When the threads remove their `jh`, they get the GC list of the transaction if the `jh` is

```
31: jbd2_journal_start(journal, ...){
32:     if(j_running_transaction is not available)
33:         /*create a new transaction or call journal_io_start(journal)*/
34:     read_lock(journal->j_state_lock);
35:     handle->transaction = journal->j_running_transaction;
36:     atomic_add(transaction->t_updates, 1);
37:     read_unlock(journal->j_state_lock);
38:     return handle;
39: }

40: jbd2_journal_stop(handle){
41:     /* complete a transaction */
42:     atomic_sub(handle->transaction->t_updates, 1);
43: }

44: add_buffer(jh, transaction, head, tail) {
45:     jh->transaction = transaction;
46:     jh->prev = atomic_set(tail, jh);
47:     if(jh->prev == NULL)
48:         head = jh;
49:     else
50:         jh->prev->next = jh;
51: }

52: del_buffer(jh, transaction, head, tail) {
53:     atomic_set(jh->removed, removed);
54:     jh->gc_prev = atomic_set(tail, jh);
55:     if(jh->gc_prev == NULL)
56:         head = jh;
57:     else
58:         jh->gc_prev->gc_next = jh;
59:     bh->jh = jh->bh = NULL; /* unlink the bh from the jh */
60:     jh->transaction = NULL;
61: }
```

associated with running or checkpoint transactions (lines 17-20 and 24-27). For the logical remove operation (lines 52-61), the thread marks the `jh` as *removed*

by executing the `atomic_set` instruction and inserts the `jh` into the GC list atomically by using `gc_prev/next` fields of the `jh`. Then, the `bh` is unlinked from the removed `jh` (line 59), and the `jh`'s `transaction` or `cp_transaction` field is set to `NULL` in the case of running or checkpointing transaction, respectively (line 60). This means that the `jh` is not associated with the `bh` and the transaction any longer. Thus, the `jh` becomes an obsolete structure, and the `bh` gets freed at this point. This operation on the `bh` is performed safely since the operation is protected by a spin lock (`jbd_lock_bh_state`) per `bh` as same as the existing scheme. Meanwhile, in the case of committing transaction, the thread only marks the `jh` as *removed* (line 23), and both `bh` and `jh` will be freed during the commit procedure.

Committing transaction

In this section, we present our committing transaction. During the existing commit procedure, the journal thread updates the lists under the list lock and performs journal I/O operations by a single thread. On the other hand, in our commit procedure, we update the lists by using our concurrent update operations and parallelize the I/O operations in a cooperative manner.

To commit a transaction, the journal thread gets a committing transaction similar to the existing procedure (Procedure 5, lines 3-9). Then, the journal thread starts the parallel I/O by setting the `journal_io` variable (line 10). This informs application threads that the I/O processing is initiated. Note that in the existing procedure, application threads are blocked when a running transaction is not available and cannot be newly created. Instead of blocking the threads, we enable the threads to perform the I/O processing along with the journal thread by calling `journal_io_start()` (Procedure 4, line 33, Procedure 5, line 11, and Procedure 6, line 2). Thus, the threads can join the I/O processing if

it is initiated by the journal thread (Procedure 6, lines 5-6).

To handle the joined threads, we record the number of threads by executing `atomic_add/sub` instructions[3] (Procedure 6, lines 7 and 20) and create the per-thread wait list for the parallel I/O completion (line 8). Then, we allow each thread to fetch the `jh` from the transaction buffer list by using our concurrent fetch operation, which executes the `atomic_cas` instruction[4] (lines 9-17). If the fetched `jh` was logically removed, the thread bypasses and retries to fetch the next `jh`. Otherwise, each thread creates a frozen buffer, submits the I/O of the buffer to the journal area, and inserts the buffer into its own wait list in parallel.

After all the I/Os are issued, we stop new upcoming threads from joining the I/O processing by unsetting the `journal_io` variable (line 18). Then, the joined threads complete the I/O by using their own wait list (lines 19 and 22-32). Through the procedure above, the parallel I/O is completed by writing all the buffers to the journal area. This procedure satisfies the following property.

Property 1. *Every block associated with a transaction is written to the journal area at a commit procedure.*

Every application thread increases `t_update` *before inserting its* `jh` *(Procedure 4, line 36) and decreases* `t_update` *after inserting its* `jh` *(Procedure 4, line 42). Before the journal thread starts the parallel I/O processing by setting* `journal_io` *(Procedure 5, line 10), the thread waits until* `t_update` *becomes 0 (Procedure 5, line 7). This prevents application threads from starting and finishing the I/O processing before all the* `jh`*s are inserted into the transaction buffer list. Thus, it ensures that all the buffers associated with the transaction are written to the journal area even if the parallel I/O is enabled.* □

While completing the I/Os (Procedure 6, lines 22-32), the threads insert the

[3] __sync_add/sub_and_fetch(type *ptr, type val): These built-in functions atomically add/subtract the value of val to/from the variable that *ptr points to. The functions return the new value of the variable that *ptr points to [42].

[4] __sync_val_compare_and_swap(type *ptr, type oldval, type newval): This built-in function performs an atomic compare and swap operation. If the current value of *ptr is oldval, then write newval into *ptr. Otherwise, no operation is performed. The function returns the contents of *ptr before the operation [42].

PROCEDURE 5 C-like pseudo-code of our committing transaction (1)

```
 1: /*the journal thread commits a transaction*/
 2: jbd2_journal_commit_transaction(journal){
 3:     commit_transaction = journal->j_running_transaction;
 4:     write_lock(journal->j_state_lock);
 5:     journal->j_committing_transaction = commit_transaction;
 6:     journal->j_running_transaction = NULL;
 7:     while(atomic_read(transaction->t_updates)){...}
 8:     write_unlock(journal->j_state_lock);
 9:     transaction = journal->j_committing_transaction;
10:     atomic_set(transaction->journal_io, start);
11:     journal_io_start(journal);
12:     while(atomic_read(transaction->num_io_threads) != 0);
13:     <issue and complete a commit block>
14:     write_lock(journal->j_state_lock);
15:     <insert the committed transaction into a checkpoint transaction list
16:         (journal->j_checkpoint_transactions) using our concurrent insert>
17:     write_unlock(journal->j_state_lock);
18:     atomic_set(transaction->cp_io, start);
19: }
```

jhs into a checkpoint list if the jhs are not removed logically and their buffers are still dirty. In this processing, for simplicity and efficiency, we make the checkpoint list while completing the I/Os before the commit block is written. However, the list is not used for checkpointing until the commit procedure is finished to preserve the ordering of committing and checkpointing.

In addition, we use the wait lists instead of the shadow list and include all the frozen buffers in the wait lists. Instead of the forget list, we use the GC list and insert the jhs which are associated with buffers to be freed to the GC list. After completing all the I/Os, the journal thread waits until all the journal I/Os are finished by using the number of joined threads before writing the commit block (Procedure 5, lines 12-13). This procedure satisfies the following property.

Property 2. *A transaction is committed or uncommitted (atomicity) according*

PROCEDURE 6 C-like pseudo-code of our committing transaction (2)

```
 1: /*the journal thread performs journal I/Os with application threads*/
 2: journal_io_start(journal){
 3:     if((transaction = journal->j_committing_transaction) == NULL)
 4:         return;
 5:     if(atomic_read(transaction->journal_io) == stop)
 6:         return;
 7:     atomic_add(transaction->num_io_threads, 1);
 8:     create_wait_list(local_wait_list); // create a local wait list per thread
 9:     while((jh = transaction->t_buffers) != NULL){
10:         if(atomic_cas(transaction->t_buffers, jh, jh->next) != jh)
11:             continue;
12:         if(atomic_read(jh->removed) == removed)
13:             continue;
14:         <make a frozen buffer (frozen_bh)>
15:         submit_bh(WRITE, jh->frozen_bh);
16:         add_wait_list(local_wait_list, jh->frozen_bh);
17:     }
18:     atomic_set(transaction->journal_io, stop);
19:     wait_journal_io(wait_list);
20:     atomic_sub(transaction->num_io_threads, 1);
21: }

22: wait_journal_io(local_wait_list){
23:     while(!wait_list_empty(local_wait_list){
24:         frozen_bh = list_entry(local_wait_list.next, ...);
25:         wait_on_buffer(frozen_bh);
26:         jh = frozen_bh->bh->jh;
27:         jh->transaction = NULL;
28:         if(atomic_read(jh->removed) != removed && jbddirty(jh->bh))
29:             add_buffer(jh, transaction, transaction->t_checkpoint_list,
30:                 transaction->t_checkpoint_list_tail);
31:     }
32: }
```

to the commit block.

Every application thread that joins the I/O processing increases `num_io_threads` *before issuing I/O (Procedure 6, line 7) and decreases* `num_io_threads` *after*

completing I/O (Procedure 6, line 20). The journal thread waits until `num_io_threads` *becomes 0 before the journal thread writes the commit block (Procedure 5, line 12). This means that all the journal blocks are written before the commit block is written to the journal area. Thus, it ensures the atomicity of the transaction by preserving the ordering between the journal blocks and the commit block.* □

Finally, the journal thread inserts the committed transaction into the checkpoint transaction list by using the state lock (`j_state_lock`) and our concurrent insert operation, and sets the `cp_io` variable to start the checkpoint I/O (lines 14-18).

Checkpointing transaction

This section presents our checkpointing transaction. In the existing procedure, when a transaction needs to be checkpointed, an application thread performs checkpoint I/O operations by acquiring a checkpoint mutex lock (`j_checkpoint_mutex`). Meanwhile, other application threads, which fail to acquire the lock, are blocked until the checkpoint is finished, which can underutilize the I/O parallelism.

To enable a parallel checkpoint I/O, we allow the threads to join the I/O processing instead of using the mutex lock and the checkpoint buffer. However, even with the parallel I/O, the I/O issue/complete operations are still inefficient since the list lock is used to fetch/insert the `jh`s from/into the checkpoint/checkpoint io lists. Thus, we fetch the `jh`s by using our concurrent fetch operation, issue the I/Os, and complete the I/Os by using the per-thread wait list in parallel instead of the global checkpoint io list.

When a checkpoint is triggered, application threads get a transaction to be checkpointed if the transaction is available (Procedure 7, lines 2-3). Then, the threads check whether the transaction can be checkpointed or not by using the `cp_io` variable (lines 4-5). Similar to our commit procedure, we record the number of joined threads, and each thread creates its own wait list (lines 6-7).

For the concurrent and parallel I/O issue, each thread concurrently fetches the `jh` from the checkpoint list, submits the I/O of the buffer associated with the `jh` to the original area, and inserts the buffer into the wait list of each thread in parallel (lines 8-15). If the fetched `jh` was removed logically, the thread retries to fetch the next `jh`. After issuing all the I/Os, we stop new upcoming threads from joining the I/O processing by unsetting the `cp_io` variable (line 16). Then, the joined threads disassociate the `jh`s with the transaction while completing the I/Os (lines 17 and 28-34).

After completing all the I/Os, we find the last remaining thread by decreasing the number of joined threads (line 18). The last thread sets the next transaction to be checkpointed by updating the head of the checkpoint transaction list to the next of the head using the atomic CAS operation (lines 19-20). This procedure satisfies the following property.

Property 3. *Committed transaction N-1 is checkpointed prior to committed transaction N.*

A committed transaction is inserted into tail of the checkpoint transaction list in committed order (Procedure 5, 15-16). The last thread sets the next transaction to be checkpointed in the checkpoint transaction list in committed order (Procedure 7, 19-20). This means that if transaction N-1 is committed prior to transaction N, the transaction N is not checkpointed prior to transaction N-1. Thus, it ensures that all the buffers in the transaction are written to the original area in the committed order. Consequently, our optimized file system preserves the consistency of the file system by satisfying Properties 1, 2, and 3. □

And then, the last thread physically removes all the obsolete `jh`s in the GC list of the transaction (lines 21-24). At this point, we can reclaim the `jh`s safely. It is because all the transaction processing is ended: (1) No other threads reference the logically removed `jh`s in the transaction since all the I/O processing is ended. (2) No other threads insert any logically removed `jh`s into the GC list of the transaction since all the `jh`s in the transaction are disassociated with

PROCEDURE 7 C-like pseudo-code of our checkpointing transaction

```
1: jbd2_log_wait_for_space(journal){
2:     if((transaction = journal->j_checkpoint_transactions) == NULL)
3:         return;
4:     if(atomic_read(transaction->cp_io) == stop)
5:         return;
6:     atomic_add(transaction->cp_num_io_threads, 1);
7:     create_wait_list(local_wait_list); // create a local wait list per thread
8:     while((jh = transaction->t_checkpoint_list) != NULL){
9:         if(atomic_cas(transaction->t_checkpoint_list, jh, jh->next) != jh))
10:            continue;
11:        if(atomic_read(jh->removed) == removed)
12:            continue;
13:        submit_bh(WRITE, jh->bh);
14:        add_wait_list(local_wait_list, jh->bh);
15:    }
16:    atomic_set(transaction->cp_io, stop);
17:    wait_cp_io(local_wait_list);
18:    if(atomic_sub(transaction->cp_num_io_threads, 1) == 0){
19:        <set the next transaction to be checkpointed
20:                in the checkpoint transaction list using atomic_cas>
21:        while((jh = transaction->gc_head) != NULL){
22:            transaction->gc_head = jh->gc_next;
23:            free(jh);
24:        }
25:        free(transaction);
26:    }
27: }

28: wait_cp_io(local_wait_list){
29:    while(!wait_list_empty(local_wait_list){
30:        bh = list_entry(local_wait_list.next, ...);
31:        wait_on_buffer(bh);
32:        bh->jh->cp_transaction = NULL;
33:    }
34: }
```

the transaction. Finally, the last thread frees the checkpointed transaction (line 25).

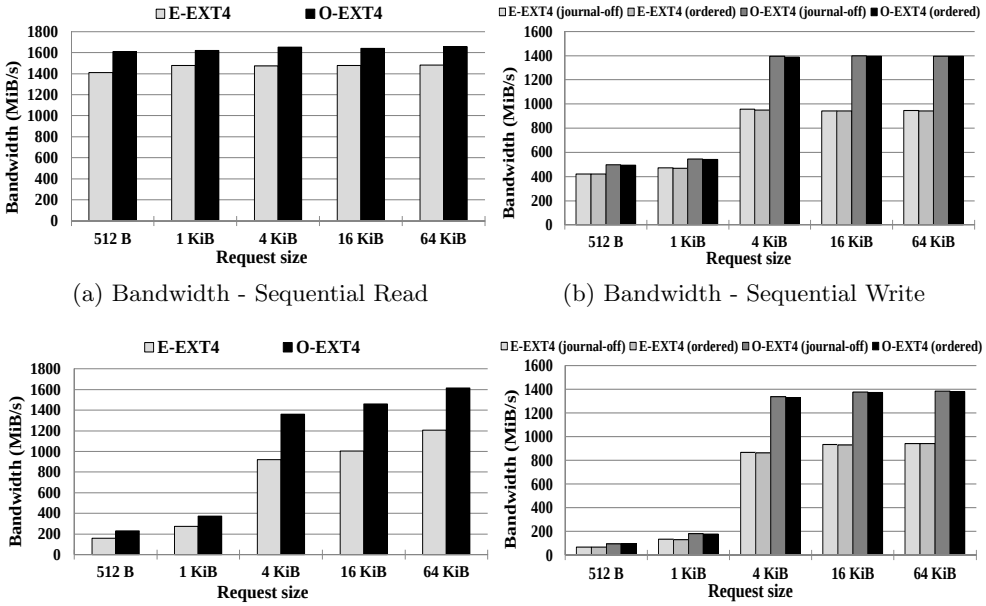
Chapter 4

Evaluation

4.1 Evaluating the Optimized File System for Low-latency Storage

Our machine has an Intel Xeon E5630 2.53GHz quad core processor (total 8 cores with hyper-threading), 8 GiB memory, and runs Linux 3.14.3. As shown in Figure 4.1, we used a battery-backed DRAM-SSD [16] as a fast storage device in the system [14, 43–45]. It has 512 GiB capacity in total (i.e., 64 GiB capacity per module * 8 DDR3 modules) and a PCIe interface. To increase capacity, a PCIe expansion card can be used to increase the number of PCIe slots, increasing the number of SSDs in a machine. The peak throughput is about 1.6 GiB/s for read and 1.4 GiB/s for write. The latency is 5 us and 7 us for reading and writing 4 KiB, respectively. To show the performance benefit from each optimization technique under different journaling modes, we evaluated the ordered (default) and data journaling modes of the EXT4 file system. We used the FIO benchmark [46] to measure the performance in terms of bandwidth for the two modes. To evaluate the optimized file system in realistic workloads, we used

Figure 4.1: The DRAM-based SSD used in this study



(a) Bandwidth - Sequential Read

(b) Bandwidth - Sequential Write

(c) Bandwidth - Random Read

(d) Bandwidth - Random Write

Figure 4.2: FIO benchmark results (ordered mode)

TPC-C benchmark for the ordered mode and filebench for the data journaling mode. We run each test five times and report the average and standard deviation. The standard deviations are always under 1% of the mean; graphs omit error bars.

4.1.1 Run-time Performance

Ordered mode

FIO benchmark results. We ran the FIO benchmark with diverse patterns, multiple request sizes, and buffered I/O under 8 threads (each thread creates a 3 GiB file) in terms of bandwidth, as shown in Figure 4.2. Overall, the performance is improved 35% on average compared to the existing I/O file system.

In the case of sequential read, Figure 4.2a shows that the optimized file system improves performance by 14% on average compared to the existing file systems. The performance gap is lower than those of other workloads, since the page cache hit rate is higher due to the read-ahead technique. In sizes that are less than 4 KiB, the performance of sequential read is highest among the I/O patterns since the prefetched pages increase the hit rate for sequential small block requests. However, existing file systems cannot fully utilize the I/O bandwidth, whereas the optimized file system reaches peak throughput (1.6 GiB/s).

In the case of write performance on the ordered mode, we note that the optimizations for journal I/O has little improvement on performance. Our optimization of journal I/O improves only about 1% on average compared to existing journal I/O since data-intensive workload such as the FIO benchmark generates small journal I/O for metadata. As shown in Figure 4.2b and 4.2d, the performance of the journal-off mode is almost the same as that of the ordered mode.

For sequential write as shown in Figure 4.2b, the optimized write-back achieves 38% better performance on average. In the case of small request sizes (i.e., 512 B and 1 KiB), the file systems perform read-modify-write operations since the request size does not match the page (block) size (i.e., 4 KiB). As the

read-modify-write generates unrequested I/Os, it wastes the bandwidth and largely decreases the I/O performance. The overall performance gains are increased compared to the case of sequential read. In this case, in addition to the hit rate of sequential write is lower than that of sequential read, the file system selects more scattered pages by choosing dirty pages all over the page cache; it increases the number of separate requests to storage. Eventually, since the optimized file system merges the requested pages irrespective of contiguity, it provides 1.4 GiB/s while the bandwidth of the existing file systems is saturated to 1 GiB/s when the request size is larger than 4 KiB.

For random read and write, Figure 4.2c and 4.2d show that the optimized file system improves the performance by average 39% and 40%, and up to 48% and 54%, respectively, compared to the existing file system. When the request size is less than 4 KiB during random read and write, the hit rate of small blocks is rapidly decreased compared to sequential read and write, which decreases the I/O performance. In the case of random write, more frequent read-modify-write operations are performed compared to sequential write and the performance of random write at small size is the lowest among the I/O patterns.

The performance gains of random workloads are higher than those of sequential workloads because the random workloads generate more multiple and separate requests. Although the performance is low in the cases of small random patterns (less than 4 KiB) compared to sequential patterns, the optimized file system shows full performance when the request size is larger than 4 KiB.

TPC-C results. To evaluate performance of the optimized file system in realistic workloads, we conducted TPC-C benchmark [47] with InnoDB [48]. We configured the experimental parameters as shown in Table 4.1 with other parameter sets as the default. In the default configuration, InnoDB provides atomicity of database page with redundant writes called double write buffer [49].

Parameters	Values
Page size (KiB)	4
DB buffer size (GiB)	6
Warehouse	500
Number of clients	8
Ramp-up time (seconds)	180
Measured time (seconds)	600

Table 4.1: Experimental parameters for InnoDB

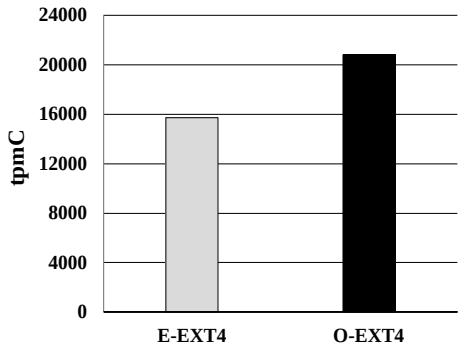
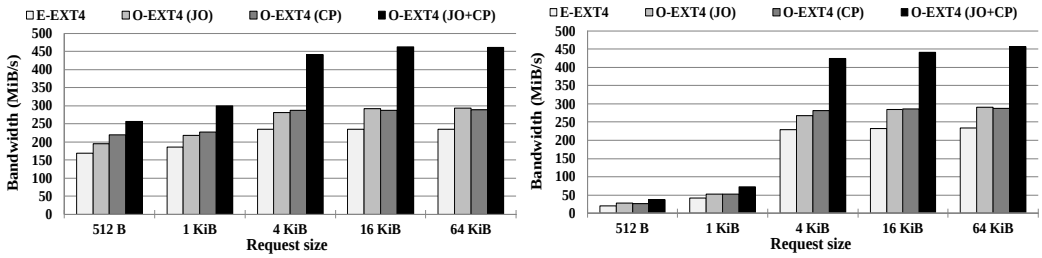


Figure 4.3: TPC-C results (ordered mode)

Therefore, the ordered mode is sufficient (i.e., metadata journaling) to provide crash consistency in the case of InnoDB. In [49], the authors showed that a smaller page size (4 KiB) leads to better transaction throughput instead of the default page size (16 KiB). Thus, we configured the page size as 4 KiB. In the TPC-C workload [50], the read:write ratio is kept at 1.9:1, and the pattern is random access.

Figure 4.3 shows the Transaction Per Minute type C (tpmC) for the existing and optimized file systems. As shown in figure, the optimized file system improves the performance by 32.3% compared to the existing file system. The optimized file system achieves up to 20807 tpmC. This result demonstrates that the existing file systems lag behind the optimized file system in the database



(a) Bandwidth - Sequential Write

(b) Bandwidth - Random Write

Figure 4.4: FIO benchmark results (data journaling)

workload. Similar to the FIO test in the ordered mode, our optimization of journal I/O little improves the performance (about 1%) since the TPC-C generates small metadata journal I/O compared to data I/O.

Data journaling mode

FIO benchmark results. To evaluate the performance for data journaling in the optimized file system, we conducted the FIO benchmark as shown in Figure 4.4 in the case of sequential and random write. We denote that JO and CP are journaling and checkpoint operations, respectively. The sequential and random read performance of the FIO benchmark is almost the same as those of the ordered mode. Since the read-only workload does not start the transaction, it does not generate any journal I/O. The performance of the existing data journaling mode is decreased by about 4x compared to that of the ordered mode in the case of 4 KiB due to the journaling operations with redundant data writes.

For the sequential write, in the existing file system, the performance is improved by 10%/26.2% from 512B/1KiB to 1KiB/4KiB respectively. However, the performance is saturated at about 235 MiB/s when the request size is larger than 4 KiB. The journal operation (JO), checkpoint (CP), and JO+CP in the

optimized file system I/O improve the performance by 15%/17%, 29.9%/48.9%, and 52.2%/61.1% in the case of 512B/1KiB respectively. In more than 4 KiB, the JO/CP/JO+CP in the optimized file system improves the performance by 19.7%/22.1%/88%, 23.9%/22.4%/96.9%, and 24.5%/22.9%/95.8% in the case of 4 KiB, 16 KiB, and 64 KiB respectively. The small size affects the bandwidth in the existing and optimized file systems since a request size of less than 4 KiB incurs read-modify-write operations. Accordingly, the performance improvement in the request sizes less than 4 KiB is less than that in the request sizes larger than 4 KiB. In terms of the performance improvement when larger than 4 KiB, JO and CP each show a relatively small improvement, but the optimized JO+CP noticeably improves the performance of the existing file system. As a result, it demonstrates that the optimization is necessary for both JO and CP.

The performance results for random write are almost the same as those of the sequential write workloads. However, in the case of less than 4 KiB, the performance improvement gap is much smaller. It is because the small random blocks are more read-modify-write operations than small sequential blocks. In the random workloads, optimized JO+CP operations improve the performance by 95% in the case of 64 KiB compared to the existing operations.

Filebench results. We evaluated the performance of the data journaling mode in existing and optimized file systems by using filebench [51]. In filebench, we used the fileserver workload that is write-intensive. In this workload, to provide crash consistency with data integrity, the data journaling mode is required because the file server does not provide atomic updates for data.

As shown in Table 4.2, we configured the I/O size as 16 KiB, the number of files as 5,000, the mean file size as 6 MiB, the number of clients as 64, and measured time as 600 s with other parameters sets as default. As shown in

Parameters	Values
I/O size (KiB)	16
The number of files	5,000
Meanfile size (MiB)	6
The number of clients	64
Measured time (seconds)	600

Table 4.2: Experimental parameters for fileserver

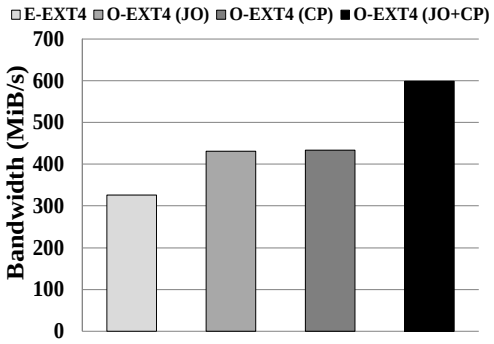
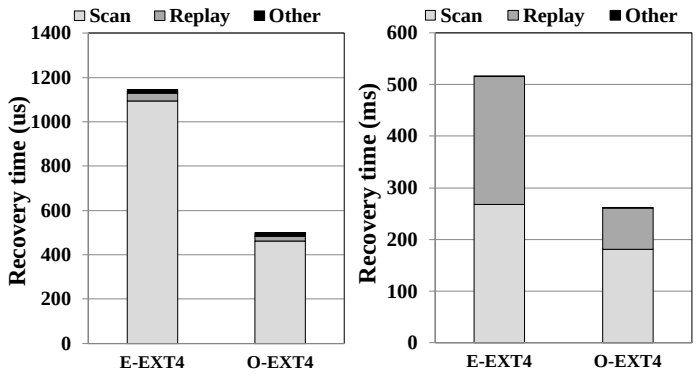


Figure 4.5: Fileserver results (data journaling)

Figure 4.5, the optimized JO, CP, and JO+CP operations in the optimized file system improve the performance by 32.1%, 33%, and 83.6% compared to those in the existing file system, respectively. We achieve high performance by up to about 600 MiB/s. The optimized file system provides higher performance than the existing file system while providing strong consistency.

4.1.2 Recovery Performance

To measure the recovery time in the existing and optimized file systems, we cut the power randomly while running the random write workload of the FIO benchmark in the ordered and data journaling modes. We conducted this evaluation more than 20 times, and the existing and optimized file systems were remounted correctly by scanning and replaying the blocks in the journal area.



(a) Ordered mode (b) Data journaling mode

Figure 4.6: Recovery performance

We compared the recovery performance when the number of replayed blocks was almost the same in the existing and optimized file systems. The recovery time occupied more than 92% of the total remount time while the scan and replay operations occupied a great part of the total recovery time in the existing and optimized file systems.

As shown in Figure 4.6, the optimized file system improves the recovery performance by about 2.28x/1.97x compared to the existing file system in the case of the ordered and data journaling modes, respectively. In this evaluation, the total replayed blocks in the existing/optimized file system are 3/3 and 23615/24164 in the ordered and data journaling mode, respectively.

In the case of the ordered mode as shown in Figure 4.6a, the scan and replay operations of existing/optimized file systems take about 1093us/461us and 37us/22us, respectively. In the case of data journaling mode as shown in Figure 4.6b, the scan and replay operations of the existing/optimized file systems take about 268ms/181ms and 248ms/80ms, respectively. According to this result, the optimized file system improves the scan and replay operations by making the blocks into a single request. Consequently, our scheme can also

Benchmark	FIO					
I/O types	SR	SW	RR	RW	JO	CP
Existing file system	15 (0.6)	14 (0.6)	2.5 (0.2)	1.5 (0.2)	1 (0)	0 (0)
Optimized file system	63 (0.6)	128 (0.04)	8 (0.14)	128 (0.04)	3.1 (0.1)	0 (0)

(a) FIO

Benchmark	TPC-C			
I/O types	Read	Write	JO	CP
Existing file system	1 (0.04)	2 (0.06)	1 (0)	1 (0)
Optimized file system	5 (0.2)	43 (0.6)	2 (0.1)	2.7 (0.4)

(b) TPC-C

Table 4.3: The average page counts in a single request in the ordered mode (SR: Sequential Read, SW: Sequential Write, RR: Random Read, RW: Random Write, JO: Journal operation, CP: Checkpoint, the numbers in parentheses are standard deviations)

Benchmarks	FIO				Fileserver	
I/O types	SW		RW		Random I/O	
	JO	CP	JO	CP	JO	CP
Existing file system	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
Optimized file system	338 (0.7)	31.6 (0.6)	337.9 (0.1)	48.3 (0.5)	332.6 (0.9)	61.2 (0.4)

Table 4.4: The average page counts in a single request in the data journaling mode (JO: Journal operation, CP: Checkpoint, SW: Sequential Write, RW: Random Write, the numbers in parentheses are standard deviations)

be applied to the recovery procedure to provide faster remount time.

4.1.3 Experimental Analysis

We analyzed the main factor of the performance improvement in the optimized file system. We measured the page counts per request with I/O operations. Table 4.3 shows the average page counts in a single request of the existing and optimized file systems. The unit of request size was 4 KiB for the EXT4 file system, and the counts were measured at the device driver level. Our technique increases the average page counts in a single request by 4.2x, 9.14x, 3.2x, and 85.3x in the case of SR, SW, RR, and RW under FIO, respectively. The results show that the number of write operations is higher than that of the read op-

Modes	Ordered		Data journaling	
Operations	Scan	Replay	Scan	Replay
Existing file system	1 (0)	1 (0)	1 (0)	1 (0)
Optimized file system	32 (0.08)	3 (0.4)	32 (0.08)	1024 (0.04)

Table 4.5: The average page counts in a single request during recovery (the numbers in parentheses are standard deviations)

erations. The reason is that write-back operations occur when the dirty rate is higher than the threshold, which accumulates the dirty pages.

Although the average page counts of the sequential read are higher than those of the random read, the performance gap of sequential read is lower than that of the random read operations. The reason is that the sequential read patterns provide a very high hit rate. In the case of TPC-C, the existing file systems cannot merge the pages into a large request since the patterns are almost random. Meanwhile, the optimized file system increases the average page counts by 5x and 21.5x for read and write, respectively.

We note that the optimized journal I/O in the ordered mode does not affect the performance of our fast storage device and data-intensive workload. The performance of journal-off is almost the same as that of the ordered mode. As shown in Table 4.3, in the ordered mode, only small journal blocks are infrequently written to the journal area.

In contrast, our optimization in the data journaling mode improves the performance since both data and metadata generate journaling and checkpoint. As shown in Table 4.4, in the case of the FIO benchmark with the data journaling mode, JO/CP in the optimized file system merges the 338/337.9 and 31.6/48.3 requests at once for the sequential and random write workloads, respectively. Thus, the average page counts are larger by 338x/337.9x and 31.6x/48.3x than those of existing file system. In the case of the fileserver workload, the average

counts are larger by 332.6x and 61.2x compared to those of the existing file system for JO and CP, respectively. In the FIO and fileserver workloads, the optimized file system significantly increases the page counts in a single request since each page is processed by only one request in the existing file system.

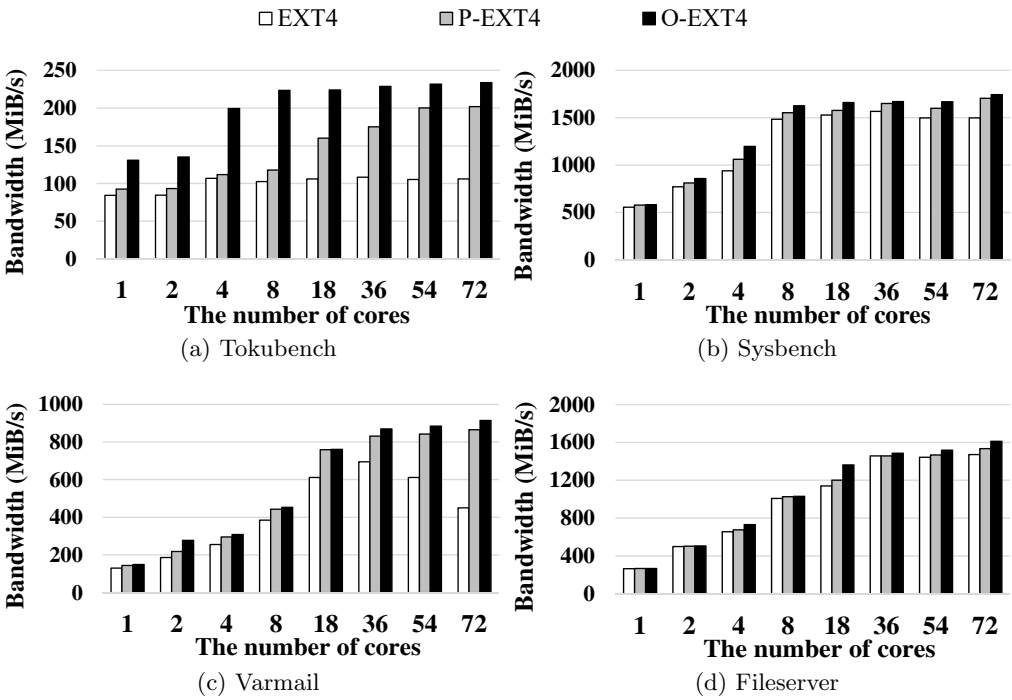
As shown in Table 4.5, in the analysis of the recovery performance, the existing file system processes the requests for scan and replay operations as several requests one by one. In contrast, the optimized file system makes 32/32 and 3/1024 blocks in the case of ordered/data journaling modes for scan and replay operations into a single request, respectively. This result shows that the optimized file system increases the average number of pages in a single request by 32x/32x and 3x/1024x in case of the ordered/data journaling modes for scan and replay operations compared to the existing file system, respectively. Consequently, the I/O performance is improved, and the recovery time is reduced by a large request.

4.2 Evaluating the Optimized File System for Highly Parallel Storage

We perform all of the experiments on a 72-core machine with four Intel Xeon E7-8870 processors (without hyperthreading), 16 GiB DRAM, and PCI 3.0 interface. For storage, the machine has an 800 GiB Intel P3700 NVMe SSD [17], which has 18 channels. The machine runs Ubuntu 16.04.1 LTS distribution with a Linux kernel 4.9.1. We evaluate the existing EXT4 and fully optimized EXT4 (O-EXT4) file systems in the ordered (default) and data journaling modes. To present a performance breakdown, we also evaluate an optimized EXT4 with our parallel I/O (P-EXT4), which performs our parallel I/O for journaling and checkpointing without `j_checkpoint_mutex` but updates the data structures using `j_list_lock`. We run metadata and data-intensive workloads, such as

Benchmarks	Descriptions	Parameters
Tokubench	Metadata-intensive (file creation)	Files: 30,000,000, I/O sizes: 4KiB
Sysbench	Data-intensive (random write)	Files: 72, Each file size: 1GiB, I/O sizes: 4KiB
Varmail	Metadata-intensive	Files: 300,000, Directory width: 10,000
Fileserver	Data-intensive	Files: 1,000,000, Directory width: 10,000

Table 4.6: Experimental parameters



(a) Tokubench (b) Sysbench

(c) Varmail (d) Fileserver

Figure 4.7: Ordered mode

tokubench [52], sysbench [53] and filebench [51] with the parameters shown in Table 4.6. We vary the number of cores from 1 to 72, and the number of threads is equal to that of the cores. We run each test ten times and report the average.

4.2.1 Run-time Performance

Ordered mode

We present the performance results in the ordered mode as shown in Figure 4.7. In the case of tokubench as shown in Figure 4.7a, the performance growth of EXT4 is not noticeable as the number of cores increases. P-EXT4 improves the performance by 1.9x compared to EXT4. However, compared to full optimization, this result shows the limitation of our parallel I/O scheme, which does not handle the lock contention. Through full optimization, O-EXT4 improves the performance by 2.2x at 72 cores compared to EXT4. Meanwhile, the performance of O-EXT4 is almost the same beyond 18 cores since the bandwidth is saturated due to the limited write bandwidth and the channels of the SSD. In the case of sysbench as shown in Figure 4.7b, P-EXT4 and O-EXT4 improve the performance by 13.8% and 16.3%, respectively, compared to EXT4 at 72 cores. The performance improvement is lower than that of tokubench since sysbench as a data-intensive workload generates far fewer journal I/Os for metadata.

Under the varmail workload as shown in Figure 4.7c, P-EXT4 and O-EXT4 scale well compared to the case of tokubench and outperform EXT4 by 1.92x and 2.03x at 72 cores, respectively. O-EXT4 achieves up to 914.3 MiB/s. Since the workload generates a mixture of read/write operations unlike tokubench, the available bandwidth increases, and therefore, the performance gradually scales at all cores. Meanwhile, the performance of EXT4 decreases beyond 54 cores due to the lock contention. Under the fileserver workload as shown in Figure 4.7d, P-EXT4 and O-EXT4 outperform EXT4 by 4.3% and 9.6% at 72 cores, respectively. All the file systems scale in a similar trend at each core, and the performance gap is not noticeable. The reason is that, similar to the case of sysbench, the fileserver workload is data-intensive, which generates a low

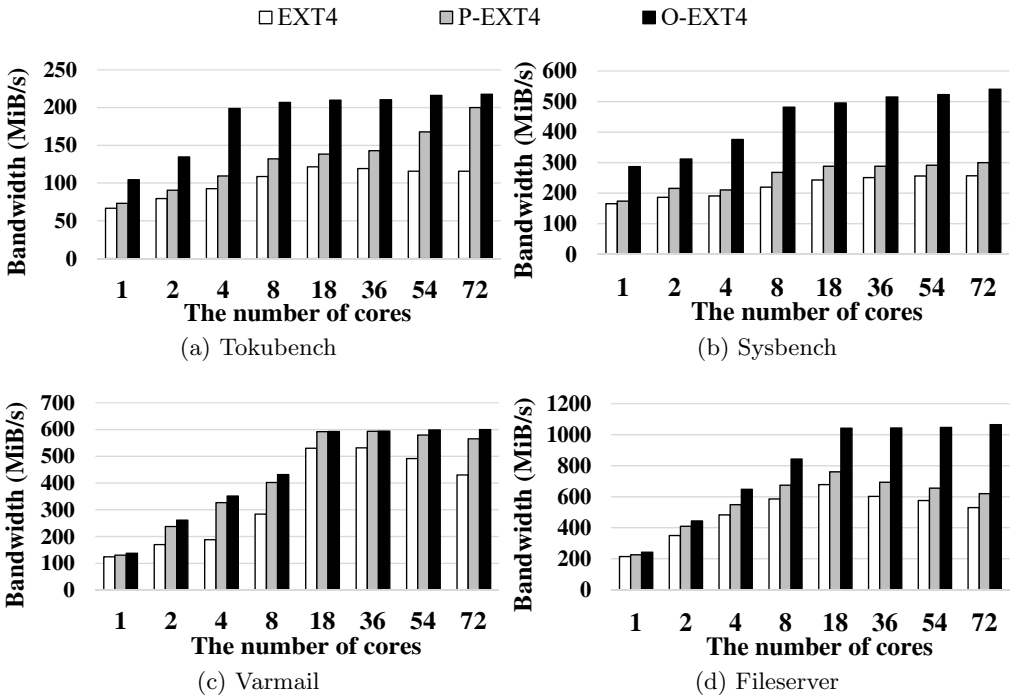


Figure 4.8: Data journaling mode

number of metadata I/Os. Consequently, our optimized file system improves the performance in the ordered mode by reducing the lock contention and parallelize the I/O operations especially for metadata-intensive workloads.

Data journaling mode

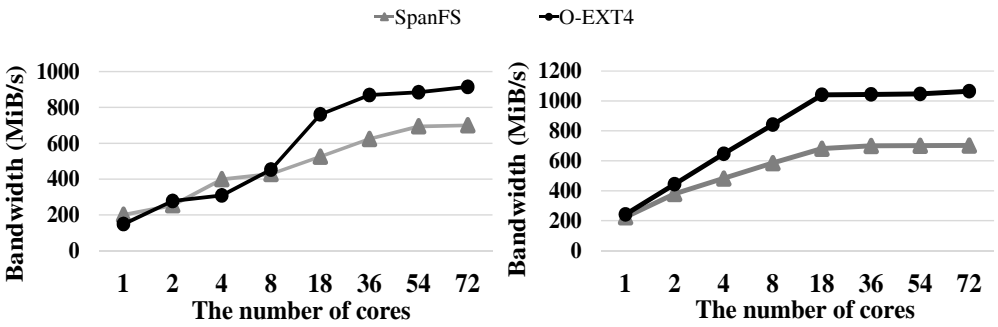
We present the performance results in the data journaling mode as shown in Figure 4.8. In the case of tokubench as shown in Figure 4.8a, P-EXT4 and O-EXT4 outperform EXT4 by 73% and 88.2% at 72 cores, respectively. The results show that the overall aspect of the performance is similar to that in the ordered mode. In the case of sysbench as shown in Figure 4.8b, P-EXT4 and O-EXT4 show 1.17x and 2.1x faster performance than EXT4 at 72 cores, respectively. The performance improvement is higher than that in the ordered

mode since the workload generates many journal I/Os for data. Also, the results show that the improvement by our parallel I/O scheme is low due to the list lock contention.

Under the varmail workload as shown in Figure 4.8c, P-EXT4 and O-EXT4 outperform EXT4 by 31.3% and 39.3% at 72 cores, respectively. Unlike the case of the ordered mode, the performance is saturated and sustained beyond 18 cores since writing both the metadata and the data makes the performance reach the full bandwidth faster. Meanwhile, the performance of EXT4 decreases due to the lock contention. In the case of fileserver as shown in Figure 4.8d, P-EXT4 and O-EXT4 outperform EXT4 by 1.45x and 2.01x at 72 cores, respectively. O-EXT4 achieves up to 1064.6 MiB/s. The performance of P-EXT4 and E-EXT4 decreases beyond 36 cores, which demonstrates the need for both concurrent updates on data structures and parallel I/O. Meanwhile, O-EXT4 scales well to 18 cores and increases the performance until 72 cores. Beyond 36 cores, the rate of bandwidth growth is reduced due to the bandwidth limit of the SSD. Consequently, our optimized file system achieves higher performance in the data journaling mode, and the benefit becomes larger in data-intensive workloads.

Comparison with a scalable file system

We compare our optimized file system with SpanFS [15], a scalable file system. We use the varmail and fileserver workloads in the ordered and data journaling modes, respectively. We set the number of domains in SpanFS as same as that of the cores. As shown in Figure 4.9, both file systems scale well until the performance is saturated in both workloads. Meanwhile, O-EXT4 generally shows better performance and improves performance by up to 1.45x and 1.51x in the varmail and fileserver workloads, respectively, compared to SpanFS. Es-



(a) Varmail (b) Fileserver

Figure 4.9: Comparison with SpanFS

pecially, in the case of the varmail workload, the performance of O-EXT4 is similar or slower than that of SpanFS at small number of cores while O-EXT4 shows better performance than SpanFS as the number of cores increases. The results show that our scheme can deliver better performance than the scheme that distributes file services.

4.2.2 Recovery Performance

In EXT4/JBD2, a single-threaded process (i.e., mount process) performs recovery operations which can underutilize both multi-cores and I/O parallelism. To increase the parallelism, similar to our journal and checkpoint I/O schemes, we perform scan and replay I/O operations in parallel by creating multiple threads without any additional locking. To evaluate the performance and test the correctness of recovery, we used tokubench and fileserver workloads in the ordered and data journaling modes, respectively. While running the benchmarks, we randomly cut the power of the machine, and both existing and optimized file systems are recovered to a consistent state after more than 30 crashes.

Table 4.7 shows the recovery performance of the ordered and data journaling modes in the file systems. The scan and replay operations occupy the main part

Modes	Ordered			Data journaling		
Operations	scan	replay	other	scan	replay	other
EXT4	331 ms	62 ms	7 ms	311 ms	81 ms	5 ms
O-EXT4	125 ms	34 ms	9 ms	117 ms	37 ms	4 ms

Table 4.7: Recovery performance

of the total recovery time in all cases. Through parallelizing scan and replay I/O operations, O-EXT4 improves the recovery performance by 2.38x and 2.51x compared to EXT4 in the ordered and data journaling modes, respectively. This result demonstrates that our schemes can also be applied to the recovery procedure to provide faster recovery time.

4.2.3 Experimental Analysis

Table 4.8 shows the total execution time for main locks and device-level bandwidth at 72 cores in the case of the sysbench workload in the data journaling mode. For this experiment, we measured the execution time by using a time function (`getrawmonotonic()`) for lower overhead and more correctness. As shown in the table, in EXT4, the execution time of the checkpoint mutex and list locks take a large portion of the total write time. In P-EXT4, the bandwidth increases by 16.3%, and the write time decreases by 15.7% compared to EXT4, respectively. As the total write time decreases, the time of the list and state locks decreases while the list lock still takes up 10.8% of the total write time. This demonstrates that the list lock contention can be a performance bottleneck in our parallel I/O scheme. In O-EXT4, the bandwidth increases by 2.06x, and the write time decreases by 2.08x compared to EXT4. This is achieved by removing the list lock contention via our concurrent update scheme. Meanwhile, the contention on the state lock increases due to the removal of the list lock but the portion is still small. Consequently, this result demonstrates that O-EXT4

File systems	EXT4	P-EXT4	O-EXT4
Device-level bandwidth	692 MiB/s	805 MiB/s	1426 MiB/s
Write time	52220 s (100%)	45124 s (100%)	25078 s (100%)
`j_checkpoint_mutex`	17946 s (34.4%)	0	0
`j_list_lock`	6132 s (11.7%)	4890 s (10.8%)	0
`j_state_lock`	102 s (0.2%)	87 s (0.2%)	182 s (0.7%)
Others	28040 s (53.7%)	40147 s (89%)	24896 s (99.3%)

Table 4.8: Device-level bandwidth and total execution time of main locks and write operations

achieves high-performance transaction processing by enabling both concurrent updates and parallel I/O.

Chapter 5

Related Work

5.1 Analysis and Evaluation of High-Performance storage

eNVy [54] presents a non-volatile main memory storage system built with flash memory. It uses a special controller equipped with a battery-backed SRAM buffer to hide the block-addressable nature of flash. The Rio file cache [55] uses a battery-backed main memory to make writes persistent. It can survive OS crashes and be as safe and permanent as disk. It achieves the performance of main memory with the reliability of disk by eliminating all reliability-induced writes to disk.

Kim et al. [6] explore the opportunities for PCM technology within enterprise storage systems. They present the results of a performance study of an all-PCM SSD prototype. They compare the PCM SSD prototype to an eMLC flash SSD to understand the performance characteristics of the PCM SSD as another storage tier. They state that the IOPS/$ of a tiered storage system can be improved by adding PCM. Vucinic et al. [23] explore the limits of communi-

cation latency with a PCM-based storage device over PCI express. They devise dubbed DC Express, which is a communication protocol. This protocol eliminates unnecessary packet exchanges and avoids context and mode switching. Our study is in line with such studies [54,55] in terms of using a battery-backed memory and such studies [6,23] in terms of exploring PCIe based SSDs. Meanwhile, we focus on the OS-level I/O stack including file systems.

Researchers have recently performed empirical studies of file systems and application performance on NVM [56–58] and concentrated efforts to improve the performance of fast storage devices. Sehgal et al. [56] evaluate the performance of various traditional Linux file systems under various real-world workloads on NVM simulated using ramdisk and compare it against an NVM optimized file system like PMFS [10]. They demonstrate that the traditional file systems can be tuned to perform better than their default settings on NVM with a performance comparable to that PMFS. Zhang et al. [57] provide an analysis of storage application performance with NVM. Their evaluation results show that NVM improves the storage application performance significantly compared to flash-based SSDs and HDDs. They present that even if NVM has somewhat higher latency and lower bandwidth than DRAM, this difference has a modest impact on application performance. Lee et al. [58] explore the use of NVM storage from the operating system (OS) perspective. They investigate the effectiveness of current I/O mechanisms and the efficient use of NVM storage.

5.2 Study of Journaling File Systems

There are some studies on the journaling file systems. Lu et al. [59] conduct a comprehensive study of Linux file system code evolution. They mention that open-source local file systems (e.g., EXT4) are a critical component of modern storage since many recent distributed file systems (e.g., Google GFS, Hadoop

DFS, etc) replicate data objects across local file systems. They analyze eight years of Linux file system changes and derive numerous new insights into the file system development process. Prabhakaran et al. [37] provide a detailed analysis of journaling file systems by using semantic block analysis (SBA) and semantic trace playback (STP). SBA enables users to understand the internal behavior and policies of the file system, and STP allows users to quantify how changing the file system will impact the performance of real workloads.

DualFS [60] is a new high-performance journaling file system, which provides the same consistency guarantees as existing file systems but with better performance. Different from our approach, DualFS places data and metadata in different devices and manages them in different ways. For example, DualFS organizes the metadata device as a log-structured file system and the data device as a group respectively. Kang et al. [15] propose a scalable file system on fast storage devices. In contrast to our goal, they focus on the lock contention as a bottleneck in the journaling file system. To reduce lock contention, they distribute files and directories among the domains, which consist of a collection of micro-file system services, and provide a global file system view on top of the domains, and maintain consistency in case of system crashes.

Lee et al. [61] present a novel buffer cache architecture that subsumes the functionality of caching and journaling. They reduce the I/O traffic of journaling using non-volatile memory. In contrast, we focus on PCIe-based storage as primary storage and provide more efficient I/O operations between the file system and the lower layer. OptFS [30] proposes decoupled ordering and durability primitives such as osync() and dsync() in HDD-based storage to reduce the frequent flush commands from the journaling file system. They trade freshness for performance while maintaining crash consistency. Unlike our scheme, their target is HDD-based storage that includes a volatile cache.

5.3 File and I/O System Optimizations for Low-latency Storage

There are a number of file system designs and suggested optimizations [9,10,12, 62] for SCM. SCMFS [12] is a new file system designed for storage class memory. This system utilizes the existing memory management module in the operating system to achieve high performance by keeping contiguous space for each file in the logical address space. BPFS [9] is optimized for small random writes by fine-grained updates instead of transferring data in bulk, which leads to unnecessary traffic over the memory bus for NVRAM. PMFS [10] is a persistent memory (PM) load/store accessible file system similar to SCMFS and BPFS. PMFS exploits byte-addressability of persistent memory in order to avoid the overhead of block-oriented storage and to enable direct PM access by applications. NOVA [62] is a log-structured file system (LFS) designed for hybrid memory systems. By extending LFS ideas to leverage NVM, NOVA proposes fast and efficient garbage collection, quick recovery, and strong consistency guarantees compared to conventional file operations and mmap-based load/store accesses. These schemes [9, 10, 12, 62] are optimized for persistent memory attached to the processor's memory bus. However, they are not appropriate for PCIe-based SSDs.

NV-Heaps [63] provides user-level transactional updates to persistent data and proposes a programming model to simplify PM programming. NV-Heaps forces the programmer to employ a specific object framework and requires modifications to the processor. Mnemosyne [64] also proposes a transaction mechanism to update data in NVM. It supports direct access and reduces latency by bypassing many software layers. These studies [63,64] provide fast mechanisms for object persistency, but they do not replace files or file systems.

In addition, various studies on optimizing I/O stack for PCIe-based SSDs

have been conducted. Seppanen et al. [11] state that an I/O scheduler in a traditional block I/O subsystem serializes and merges requests. It is efficient for HDDs since HDDs have seek overhead and lack parallelism. When storage devices such as flash-based SSDs are used, the system with the conventional block I/O subsystem cannot fully exploit the parallelism in the SSDs since the I/O scheduler processes the requests in a serialized and batched manner. Thus, the authors propose a new Linux block I/O subsystem without SCSI/ATA layers and an I/O scheduler to reduce latency and exploit parallelism of storage devices. However, the proposed block I/O subsystem still processes requests asynchronously; interrupts are used to communicate with storage devices.

Yang et al. [13] show that a synchronous I/O (polling) between the host and the storage device delivers better performance than an asynchronous I/O (interrupt) when the device has ultra-low latency. Since the system with interrupt-based I/O completes all I/Os asynchronously and it causes the interrupt handing overheads, the polling mechanism is more appropriated for storage devices with ultra-low latency. Yu et al. [14] improve the I/O bandwidth by combining multiple block requests among multiple threads into one I/O request and dispatching the request to storage. These studies [11,13,14] improve the performance by focusing on the optimization of the block I/O subsystem. Meanwhile, our study focuses on the I/O operations in the file system on top of the optimized block I/O subsystem including these techniques [11, 13, 14].

Moneta [65] is an architecture for a PCIe-attached storage array built from emulated PCM storage. This architecture uses a series of hardware/software (device driver) optimizations that improve its performance for next-generation NVM such as PCM. Moneta-D [66] is an extended version of Moneta that removes software overhead by using a user-level driver, which bypasses the kernel and file systems. Due to the user-level driver, Moneta-D requires additional

hardware functionalities for security and authority. In contrast to these studies [65, 66], we focus on optimization of the software stack and file systems. Our scheme is implemented at the kernel level and does not require additional support to avoid security concerns.

pVM [67] is a system software abstraction that provides applications with automatic OS-level memory capacity scaling, flexible memory placement policies across NVM, and fast object storage. It extends the OS virtual memory and abstracts NVM as a NUMA node with support for NVM-based memory placement mechanisms. This article is similar to pVM in terms of exploring NVM in the OS. Meanwhile, we focus on the existing I/O stack and file systems instead of the virtual memory system.

Several works have researched how to optimize software stacks for fast network access. IX [68] proposes a dataplane operating system by using hardware virtualization to separate the management and scheduling functions of the kernel from network processing. IX optimizes both bandwidth and latency by processing batches of packets to completion and eliminating synchronization on multi-core servers. Arrakis [69] presents an operating system that splits the traditional role of the kernel. Applications have direct access to virtualized I/O devices by allowing most I/O operations to skip the kernel, while the kernel is re-engineered to provide network and disk protection without kernel mediation of every operation. Similar to IX, Arrakis uses hardware virtualization to separate the I/O dataplane from the control plane. Both IX [68] and Arrakis [69] provide optimized networking stack by reducing the overhead of the operating systems. This article is in line with these schemes [68, 69] in terms of reducing the software overhead. Unlike these schemes, we focus on the storage stack for fast storage devices.

5.4 Study of Scalability in Operating Systems

Hive [70] is an operating system designed for large scale shared-memory multiprocessors. It is structured as an internal distributed system of independent kernels called cells to improve reliability and scalability. Cerberus [71] mitigates contention on many shared data structures within OS kernels by clustering multiple commodity operating systems atop a virtual machine monitor. Baumann et al. [72] investigate a new OS structure, the multikernel. To solve scalability problems for OSs, they structure the OS as a distributed system of cores that communicate using messages and share no memory. Corey [73] is an exokernel based operating system that follows a principle, which allows applications to control the sharing of kernel resources. Its abstractions ensure that each kernel data structure is used by only one core by default, while giving applications the ability to specify when sharing of kernel data is necessary.

Boyd-Wickizer et al. [74] analyze the scalability of seven system applications running on Linux. They find that all applications trigger scalability bottlenecks inside a Linux kernel. RadixVM [75] presents a scalable virtual memory address space for non-overlapping operations. It avoids cache line contention using three techniques, which are radix trees, Refcache, and targeted TLB shootdowns. Our study is inspired by these works [70–75] and in line with them in terms of investigating the scalability of OS kernels on multi-cores. In contrast, we focus on the transaction processing in file systems on high-performance storage.

5.5 File and I/O System Optimizations for Highly Parallel Storage

Zheng et al. [76] present a storage system for arrays of commodity SSDs. They create dedicated I/O threads for each SSD and deploy a set-associative parallel page cache, which divides the global page cache into small and independent sets

to reduce lock contention. MultiLanes [18] is a virtualized storage system for OS-level virtualization on many cores. It builds an isolated I/O stack on top of a virtualized storage device to eliminate contention on shared kernel data structures and locks. Bjørling et al. [77] propose a new design for I/O management in the block layer. They address the scalability of the Linux block layer and propose a new Linux block layer, which maintains a per-core request queue. They design multiple I/O submission/completion queues to minimize cache coherence across CPU cores. Jericho [78] is a new I/O stack that improves affinity between threads, and buffers in the storage I/O path for NUMA multicore systems. Jericho consists of a NUMA aware file system and a DRAM cache organized in slices mapped to NUMA nodes. Our study is in line with these works [18, 76–78] in terms of mitigating the contention on shared resources. In contrast, we focus on updating the data structures concurrently in a lock-free manner in journaling file systems.

ScaleFS [79] extends a scalable in-memory file system to support consistency on an on-disk file system by using per core operation logs. IceFS [80] partitions the on-disk resources among a new container abstraction called cubes to provide isolated I/O stacks for localized reaction to faults, fast recovery, and concurrent file system updates. Thus, data and I/O within each cube are disentangled from the data and I/O outside of it. SpanFS [15] is a scalable file system that consists of a collection of micro file system services called domains. It distributes the files and directories among the domains and provides a global file system view on top of the domains to maintain consistency. Each domain performs its file system service, such as data allocation and journaling, independently. Curtis-Maury et al. [81] present a data partitioning mode to parallelize the majority of file system operations. They also provide a fine-grained lock-based multiprocessor model for incremental advances in parallelism.

Min et al. [82] analyze the many-core scalability of five file systems by using their open source benchmark suite (i.e., FxMark). They observe that file systems are hidden scalability bottlenecks in many I/O-intensive applications. iJournaling [36] improves the performance of an `fsync()` call. It journals only the corresponding file-level transaction to the ijournal area for an fsync call while exploiting the advantage of the compound transaction scheme. iJournaling also handles multiple fsync calls simultaneously by allowing each core to have its own ijournal area to improve the scalability. Our study is in line with these approaches [15,36,79–82] in terms of investigating the scalability and parallelism of the file systems. In contrast, we enable concurrent updates on data structures in a lock-free manner and parallelize I/O operations cooperatively in transaction processing by focusing its internal operations.

Chapter 6

Conculsion

6.1 Summary

High-performance storage devices such as solid-state drives (SSDs) are becoming one of attractive storage solutions for various computer systems. According to development of the storage devices, optimizing the file systems is essential in order to fully exploit their features. As our observations, the existing I/O operations and locking in file systems can be performance bottlenecks on high-performance SSDs.

This dissertation proposes two key optimizations, 1) efficient I/O strategies for low-latency SSDs, which transfers requests from discontiguous host memory buffers to discontiguous storage segments in a single I/O request, and 2) concurrent updates on data structures and parallel I/O operations for highly parallel SSDs. Experiments show that our optimized file system achieves higher performance than the existing file system.

6.2 Future work

In the future work, we will extend our techniques and the scope of I/O optimizations for high-performance storage devices.

Extending I/O optimizations for low-latency storage devices. Our techniques for low-latency storage devices are limited to our customized DRAM-SSD. However, our techniques can be applied to other storage protocols or devices as well as other file systems. For example, current NVMe protocol transfers data only from discontiguous host memory buffers to contiguous storage segments in one I/O request. We can add the feature, which transfers data from discontiguous host memory buffers to discontiguous storage segments in one I/O request, to the NVMe protocol. By doing so, we standardize our technique by adding the new feature to the NVMe protocol and also our optimized file system can be used for low-latency storage devices with the NVMe protocol.

Also, we will perform a holistic end-to-end I/O stack or cross-layer optimizations for the low-latency storage devices. For example, we can broaden the scope of our optimizations to cover the whole local file system layers (e.g., VFS, block layer, and device driver), distributed file systems, user applications (e.g., database systems), and network layers. In the existing storage system, there are many layers, which can generate a performance bottleneck in the low-latency storage devices. Thus, we first find out the performance bottleneck by measuring the latency for each layer. And then, we will merge the redundant operations between layers and minimize the whole I/O path to maximize the performance.

Extending I/O optimizations for highly parallel storage devices. In this paper, our techniques for highly parallel storage devices are limited to the locking for transaction processing in EXT4/JBD2. We can extend the

techniques to the locks for other shared resources in the file systems such as file, page cache, etc. For example, EXT4 uses a coarse-grained lock (mutex) per file. This locking ensures correct updates on the file, and thus the file consistency is preserved. A previous study [82] shows the overhead from the file locking such as an inode mutex. When applications are accessing a shared file, such file locking become the bottleneck. Thus, we will extend our optimization techniques to the file locking mechanism to update the file updates concurrently.

For another example, the Linux kernel adopts a page cache organized as an address space radix tree to cache recently accessed blocks for better I/O performance. The OS uses a read-copy-update (RCU) lock to protect correct updates of the radix tree [15,76,83]. Previous studies [76,83] show the page cache locking overhead and reduce the overhead by using a set-associative parallel page cache which divides the global page cache into small and independent sets to reduce lock contention. SpanFS [15] leverages the Linux OS block device architecture to provide a dedicated buffer cache address space for each domain to avoid lock contention. For more efficiency, we will extend our technique using a lock-free data structure to the page cache.

For different storage configuration, we will consider the performance in multiple storage devices on RAID. In RAID, the I/O operations are performed for each device in parallel. However, we may rethink the RAID performance on many cores with a number of highly parallel storage devices considering the scalability and parallelism. Finally, after we solve the performance bottleneck in the local file and storage systems, we will extend the optimizations to other systems such as distributed file systems and database systems. Consequently, we will consider the whole layers in a small or large system to maximize the performance from high-performance hardware.

Bibliography

[1] E. Grochowski and R. F. Hoyt, "Future trends in hard disk drives," *IEEE Transactions on Magnetics*, vol. 32, no. 3, pp. 1850–1854, 1996.

[2] A. Al Mamun, G. Guo, and C. Bi, *Hard disk drive: mechatronics and control*, vol. 23. CRC press, 2006.

[3] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," *ACM SIGMETRICS Performance Evaluation Review*, vol. 22, 1994.

[4] Y. J. Yu, D. I. Shin, H. Eom, and H. Y. Yeom, "Ncq vs. i/o scheduler: Preventing unexpected misbehaviors," *ACM Transactions on Storage (TOS)*, vol. 6, no. 1, p. 2, 2010.

[5] R. R. Katti, H. L. Stadler, and J.-C. Wu, "Non-volatile magnetic random access memory," 1994. US Patent 5,289,410.

[6] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pp. 33–45, 2014.

[7] J.-D. Lee, S.-H. Hur, and J.-D. Choi, "Effects of floating-gate interference on nand flash memory cell operation," *IEEE Electron Device Letters*, vol. 23, no. 5, pp. 264–266, 2002.

[8] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of nand flash memory," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pp. 2–2, USENIX Association, 2012.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[10] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[11] E. Seppanen, M. O'Keefe, and D. Lilja, "High performance solid state storage under linux," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010.

[12] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[13] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, 2012.

[14] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom, "Optimizing the block i/o subsystem for fast storage devices," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 6:1–6:48, 2014.

[15] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai, "Spanfs: A scalable file system on fast storage devices," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, (Santa Clara, CA), pp. 249–261, USENIX Association, July 2015.

[16] TAILWINDSTORAGE, "Extreme s3804." `http://www.taejin.co.kr`, 2014.

[17] Intel Solid State Drive DC P3700 Series. `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-p3700-spec.pdf`, 2015.

[18] J. Kang, C. Hu, T. Wo, Y. Zhai, B. Zhang, and J. Huai, "Multilanes: Providing virtualized storage for os-level virtualization on manycores," *Trans. Storage*, vol. 12, pp. 12:1–12:31, June 2016.

[19] NVM express. `http://www.nvmexpress.org`, 2012.

[20] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L. Lung, and C. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, 2008.

[21] B. Dieny, R. Sousa, G. Prenat, and U. Ebels, "Spin-dependent phenomena and their implementation in spintronic devices," in *VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on*, 2008.

[22] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From aries to mars: Transaction support for next-generation, solid-state drives," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[23] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. Le Moal, T. Bunker, J. Xu, S. Swanson, *et al.*, "DC express: shortest latency protocol for reading phase change memory over PCI express," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pp. 309–315, 2014.

[24] Y. Son, H. Kang, H. Han, and H. Y. Yeom, "An empirical evaluation and analysis of the performance of nvm express solid state drive," *Cluster Computing*, pp. 1–13, 2016.

[25] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier, and B. S. A. S, "A and viver, l. the new ext4 filesystem: current status and future plans," in *In Ottawa Linux Symposium. http://ols.108.redhat.com/2007/ Reprints/mathur-Reprint.pdf*, 2007.

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

[27] JFS for Linux. `http://oss.software/ibm.com/jfs`, 2002.

[28] H. Reiser, "Reiserfs," 2004.

[29] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 1992.

[30] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 228–243, ACM, 2013.

[31] 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.

[32] J. Bonwick and B. Moore, "ZFS: The last word in file systems," 2007.

[33] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[34] D. Kim, J. Park, K.-g. Lee, and S. Lee, *Forensic Analysis of Android Phone Using Ext4 File System Journal Log*, pp. 435–446. Dordrecht: Springer Netherlands, 2012.

[35] A. C. Arpaci-Dusseau, "Model-based failure analysis of journaling file systems," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, (Washington, DC, USA), pp. 802–811, IEEE Computer Society, 2005.

[36] D. Park and D. Shin, "ijournaling: Fine-grained journaling for improving the latency of fsync system call," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 787–798, USENIX Association, 2017.

[37] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proceedings of the Annual*

Conference on USENIX Annual Technical Conference, ATEC '05, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2005.

[38] S. C. Tweedie, "Journaling the linux ext2fs filesystem," in *The Fourth Annual Linux Expo*, 1998.

[39] A. Hatzieleftheriou and S. V. Anastasiadis, "Improving bandwidth efficiency for consistent multistream storage," *Trans. Storage*, vol. 9, pp. 2:1–2:27, Mar. 2013.

[40] K. Apt, F. S. De Boer, and E.-R. Olderog, *Verification of sequential and concurrent programs*. Springer Science & Business Media, 2010.

[41] J. Östlund and T. Wrigstad, "Multiple aggregate entry points for ownership types," *ECOOP 2012–Object-Oriented Programming*, pp. 156–180, 2012.

[42] R. M. Stallman and G. DeveloperCommunity, *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009.

[43] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient memory-mapped i/o on fast storage device," *Trans. Storage*, vol. 12, pp. 19:1–19:27, May 2016.

[44] Y. Son, N. Song, H. Han, H. Eom, and H. Yeom, "Design and evaluation of a user-level file system for fast storage devices," *Cluster Computing*, vol. 18, no. 3, pp. 1075–1086, 2015.

[45] Y. Son, J. W. Choi, H. Eom, and H. Y. Yeom, "Optimizing the file system with variable-length i/o for fast storage devices," in *Proceedings of the 4th*

Asia-Pacific Workshop on Systems, APSys '13, (New York, NY, USA), pp. 14:1–14:6, ACM, 2013.

[46] J.Axboe, "Fiobenchmark." `http://freecode.com/projects/fio`, 1998.

[47] tpcc-mysql. `https://github.com/Percona-Lab/tpcc-mysql`.

[48] P. Fruhwirt, P. Kieseberg, S. Schrittwieser, M. Huber, and E. Weippl, "Innodb database forensics: Reconstructing data manipulation queries from redo logs," in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, 2012.

[49] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh, "Durable write cache in flash memory ssd for relational and nosql databases," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, 2014.

[50] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica, "Tpc-e vs. tpc-c: Characterizing the new tpc-e benchmark via an i/o comparison study," *SIGMOD Rec.*, vol. 39, no. 3, pp. 5–10, 2011.

[51] A. Wilson, "The new and improved filebench," in *Proceedings of 6th USENIX Conference on File and Storage Technologies*, 2008.

[52] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul, "The tokufs streaming file system," in *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, (Berkeley, CA, USA), pp. 14–14, USENIX Association, 2012.

[53] A. Kopytov, "Sysbench: a system performance benchmark," *URL: http://sysbench. sourceforge. net*, 2004.

[54] M. Wu and W. Zwaenepoel, "eNVy: a non-volatile, main memory storage system," *ACM SIGOPS Operating Systems Review*, vol. 28, no. 5, pp. 86–97, 1994.

[55] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio File Cache: Surviving Operating System Crashes," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (New York, NY, USA), pp. 74–83, ACM, 1996.

[56] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on nvm," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pp. 1–14, May 2015.

[57] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pp. 1–10, May 2015.

[58] E. Lee, H. Bahn, S. Yoo, and S. H. Noh, "Empirical study of nvm storage: An operating system's perspective and implications," in *Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '14, (Washington, DC, USA), pp. 405–410, IEEE Computer Society, 2014.

[59] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," *Trans. Storage*, vol. 10, pp. 3:1–3:32, Jan. 2014.

[60] J. Piernas, T. Cortes, and J. M. Garcia, "The design of new journaling file systems: The dualfs case," *IEEE Transactions on Computers*, vol. 56, pp. 267–281, Feb 2007.

[61] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pp. 73–80, 2013.

[62] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, (Santa Clara, CA), pp. 323–338, USENIX Association, Feb. 2016.

[63] 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," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[64] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[65] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[66] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[67] S. Kannan, A. Gavrilovska, and K. Schwan, "pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, (New York, NY, USA), pp. 13:1–13:16, ACM, 2016.

[68] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 49–65, 2014.

[69] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI 14)*, 2014.

[70] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault containment for shared-memory multiprocessors," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, (New York, NY, USA), pp. 12–25, ACM, 1995.

[71] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang, "A case for scaling applications to many-core with os clustering," in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys'11, (New York, NY, USA), pp. 61–76, ACM, 2011.

[72] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 29–44, ACM, 2009.

[73] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, *et al.*, "Corey: An operating system for many cores.," in *OSDI*, vol. 8, pp. 43–57, 2008.

[74] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich, *et al.*, "An analysis of linux scalability to many cores.," in *OSDI*, vol. 10, pp. 86–93, 2010.

[75] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Radixvm: Scalable address spaces for multithreaded applications," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 211–224, ACM, 2013.

[76] D. Zheng, R. Burns, and A. S. Szalay, "Toward millions of file system iops on low-cost, commodity hardware," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 69:1–69:12, ACM, 2013.

[77] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: Introducing multi-queue ssd access on multi-core systems," in *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, (New York, NY, USA), pp. 22:1–22:10, ACM, 2013.

[78] S. Mavridis, Y. Sfakianakis, A. Papagiannis, M. Marazakis, and A. Bilas, "Jericho: Achieving scalability through optimal data placement on multicore systems," in *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pp. 1–10, IEEE, 2014.

[79] R. Eqbal, *ScaleFS: A multicore-scalable file system.* PhD thesis, Massachusetts Institute of Technology, 2014.

[80] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Physical disentanglement in a container-based file system.," in *OSDI*, pp. 81–96, 2014.

[81] M. Curtis-Maury, V. Devadas, V. Fang, and A. Kulkarni, "To waffinity and beyond: A scalable architecture for incremental parallelization of file system code," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (GA), pp. 419–434, USENIX Association, 2016.

[82] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding manycore scalability of file systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, (Denver, CO), pp. 71–85, USENIX Association, 2016.

[83] "A parallel page cache: Iops and caching for multicore systems," in *Presented as part of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, (Boston, MA), USENIX, 2012.

요약

Solid-State Drive (SSD) 와 같은 고성능 저장장치 기술은 낮은 지연시간, 높은 대역폭, 그리고 높은 입출력 병렬성을 제공한다. SSD는 기계적인 오버헤드없이 데이터에 접근이 가능하도록 해주며, 하드 디스크와 같은 기존 저장장치에 비해 수십배의 성능향상을 가져온다. 그러나, 기존 소프트웨어 입출력 계층을 그대로 사용하거나 SSD의 특징을 활용하지 않으면 최대 성능에 도달하지 못할 수 있다.

본 논문에서는 SSD 특징들 (예: 낮은 지연시간, 높은 병렬성)을 최대한 활용할 수 있도록 파일 시스템을 최적화한다. 이를 위해 첫째, 지연시간이 낮은 SSD 기반에서 파일시스템의 기존 입출력 방식들을 분석한다. 해당 방식은 블락들이 비연속적일 경우, 여러 개의 입출력 요청으로 나누어서 처리하게 된다. 따라서, 이러한 방식은 해당 SSD의 특징을 최대한 활용하지 못한다. 이러한 문제를 해결하기 위해서, 본 논문은 효율적인 입출력 방식을 제안한다. 제안하는 방식에서는 하나의 입출력 요청으로 파일 시스템의 비연속 호스트 메모리 버퍼들을 비연속 저장소 세그먼트들로 전송한다. 따라서 이는 파일시스템이 지연시간이 낮은 SSD의 성능을 최대한 활용할 수 있게 해준다.

둘째, 높은 병렬성을 지닌 SSD 기반에서 파일시스템의 기존 락킹과 입출력 병렬성을 분석한다. 파일시스템에서는 공유 자료구조에 접근하기 위해 락킹이 사용되며, 입출력은 단일 스레드에 의해 직렬화된다. 이러한 이유로 파일시스템은 종종 높은 병렬성을 지닌 SSD와 멀티코어 환경에서 락 경쟁을 발생시키고 입출력 대역폭을 최대로 활용하지 못하는 문제에 직면한다. 이러한 문제를 해결하기 위해서 자료구조에 대한 동시적인 업데이트와 입출력 동작을 병렬화시킨다.

본 논문은 제안하는 방식들을 EXT4/JBD2에 구현하고 이들을 낮은 지연시간과 높은 병렬성을 가진 SSD기반에서 평가한다. 실험결과를 통해 최적화된 파일시스템이 기존 파일시스템에 비해 성능이 향상되었음을 확인할 수 있었다.

주요어: 파일시스템, 운영체제, 고성능 저장장치, Solid-State Drive
학번: 2013-30241

Ph.D. DISSERTATION

Optimizing File Systems for High-Performance Storage Devices

고성능 저장장치를 위한 파일시스템 최적화

FEBRUARY 2018

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yongseok Son

Ph.D. DISSERTATION

Optimizing File Systems for High-Performance Storage Devices

고성능 저장장치를 위한 파일시스템 최적화

FEBRUARY 2018

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yongseok Son

Optimizing File Systems for High-Performance Storage Devices

고성능 저장장치를 위한 파일시스템 최적화

지도교수 염헌영

이 논문을 공학박사 학위논문으로 제출함

2017 년 12 월

서울대학교 대학원

전기·컴퓨터 공학부

손용석

손용석의 공학박사 학위논문을 인준함

2017 년 12 월

위 원 장	엄현상	(인)
부위원장	염헌영	(인)
위 원	유승주	(인)
위 원	이재욱	(인)
위 원	한 혁	(인)

Abstract

High-performance storage technologies such as solid-state drives (SSDs) provide low-latency, high throughput, and high I/O parallelism to legacy storage systems. SSDs access data without mechanical overhead, and they often leads to order-of-magnitude improvements in performance over legacy storage devices such as hard disk drives (HDDs). However, replacing HDDs with SSDs while keeping the software I/O stack or not exploiting SSD features does not lead to maximum performance.

In this dissertation, we optimize file systems to fully exploit the SSD features (e.g., low-latency and high I/O parallelism). First, we analyze and explore I/O strategies in the existing file systems on low-latency SSDs. The file systems issue and complete several I/O requests when blocks are not contiguous, which does not take advantage of the low-latency of SSDs. To address this problem, we propose efficient I/O strategies, which transfer requests from discontiguous host memory buffers in the file systems to discontiguous storage segments in a single I/O request. Thus, they enable file systems to fully exploit the performance of low-latency SSDs.

Second, we investigate the locking and I/O parallelism in the existing file systems on highly parallel SSDs. In the file systems, the coarse-grained locking to access shared data structures is used and I/O operations are serialized by a single thread. For these reasons, the file systems often face the problem of lock contention and underutilization of I/O bandwidth on multi-cores with highly parallel SSDs. To address these issues, we enable concurrent updates on data structures and parallelize I/O operations.

We implement our techniques in EXT4/JBD2 and evaluate them on low-latency and highly parallel SSDs. The experimental results show that our optimized file system improves the performance compared to the existing EXT4 file system.

Keywords: File system, Operating System, High-Performance Storage Device, Solid-State Drive

Student Number: 2013-30241

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Over the last few decades, enhancing the performance of storage devices has been an important challenge for computer systems in research and industry. Many data-intensive applications have demanded high throughput and low-latency. For many years, hard disk drives (HDDs) [1–4] have been used as the most common primary storage device. However, the performance of HDDs lags far behind that of the processor and the main memory due to mechanical overhead (i.e., rotational and seek time), and this HDD performance bottleneck has worsened in modern computer systems.

Semiconductor technology has introduced non-volatile memory (NVM) such as MRAM [5], PCM [6], and NAND flash [7,8] to computer system communities, and it opened up research challenges. NVM accesses data in low-latency and highly parallel way, and this often leads to orders-of-magnitude improvements in performance over HDDs. Such recent developments in NVM technologies have

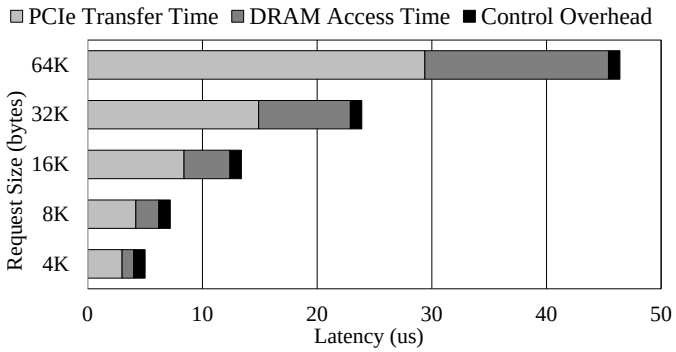
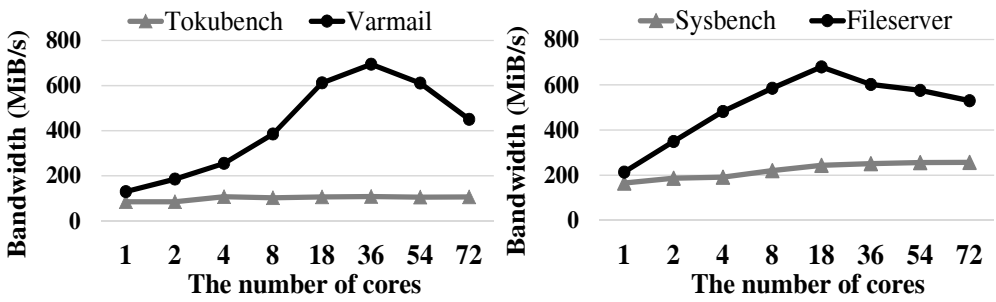


Figure 1.1: Latency Breakdown on low-latency storage (the detailed experimental environment is described in Section 4.1)

closed the performance gap between main memory and storage. Accordingly, NVM has improved I/O performance in various environments such as cloud platforms, social network services, large websites, etc. However, replacing HDD with NVM while keeping the software I/O stack does not lead to maximum performance as it is optimized for HDDs. To exploit the performance of NVM, researchers [9–15] have reconstructed the traditional software I/O stack and performed several optimizations.

Figure 1.1 shows a latency breakdown in our low-latency storage [16] with varying request sizes. As shown in the figure, the PCIe transfer time accounts for the major portion of the total PCIe communication time. The total time for one 64 KiB request is 42% less than that for sixteen 4 KiB requests owing to the benefit of the PCIe communication. This shows that a single request in larger granularity is more efficient than multiple small requests. Therefore, processing a large request is a better method for PCIe-based fast storage device. As our observations, existing I/O strategies prevent file systems from taking advantage of fast storage's full performance even if the block I/O subsystem is optimized. They process I/O requests by issuing and completing the request one by one when the storage segments of the requests are discontinuous.



(a) Ordered mode (b) Data journaling mode

Figure 1.2: Scalability evaluation on highly parallel storage (the number of threads is the same as that of the cores and the detailed experimental environment is described in Section 4.2)

Figure 1.2 shows a scalability and I/O performance using metadata and data-intensive workloads in the ordered and data journaling modes, respectively, in our highly parallel storage [17]. As shown in the figure, the performance does not scale well or decreases as the number of cores grows. Based on our analysis and other studies [15, 18], it is due to the contention on shared data structures and serialization of I/O operations.

1.2 Approach and Contributions

To achieve lower latency and higher parallelism, we propose two main optimizations for the file system. First, we propose I/O strategies of file systems in terms of latency. The key idea is to transfer data from discontiguous host memory buffers of file systems to discontiguous storage segments in a single I/O request, which existing block-based file systems cannot provide. We note that current storage protocols such as SATA and NVMe [19] support data transfer only from discontiguous host memory segments to contiguous storage segments in a single I/O request.

Second, we propose schemes to achieve high I/O parallelism as follows: (1) We use lock-free data structures and operations to reduce the lock contention. This scheme allows multiple threads to access the data structures (e.g., linked lists) concurrently. (2) We propose a parallel I/O scheme that performs I/O operations by multiple threads in a parallel and cooperative manner. This scheme allows multiple threads to cooperate in I/O processing and issue/complete the I/Os in parallel while not sacrificing the consistency of the file system.

We apply and implement the two optimizations on EXT4/JBD2. Our techniques provide higher performance while preserving all features and the same consistency level of the existing file system. We evaluate our optimized file systems for low-latency and parallelism using a DRAM-based SSD and Intel P3700 NVMe SSD, respectively. The experimental results show that the optimized file systems improve the performance compared to the existing file system.

The contributions of this dissertation can be summarized as follows:

- We analyze the main obstacles that increase the latency and reduce the parallelism of high-performance storage.
- We propose several optimization techniques for journaling file systems and implement them on EXT4/JBD2.
- Experimental results show that the optimized file system could achieve significant performance improvements, compared to the existing file system, while providing the same level of consistency.

1.3 Dissertation Structure

This dissertation is organized as follows:

Chapter 2 analyzes I/O path and strategy in the file system in terms of I/O latency and parallelism.

Chapter 3 designs and implements our schemes.

Chapter 4 evaluates our optimized file systems in terms of I/O low-latency and parallelism using varying workloads.

Chapter 5 summarizes related works and compares them with our works.

Chapter 6 summarizes our optimizations and contributions.

Chapter 2

Background

2.1 High-performance Storage Devices

High-performance storage provides low-latency and highly parallel accessing to data. Non-volatile memory (NVM) technologies, including PCM [20], spin-transfer torque memory [21], MRAM [5], and NAND flash [7,8] are anticipated to be faster than existing storage technologies (e.g., hard disk drives (HDDs)). The most significant features of NVM [5,20,21] are low latency, high throughput, and high parallelism without mechanical overheads. Previous studies [21, 22] suggest that NVM will have bandwidth and latency similar to DRAM and mention that the devices will be 50,000x faster than HDDs.

Modern PCIe-attached NVM-based SSDs [6, 22, 23] have emerged in many studies, and the arrival of the NVMe interface [19, 24] implies that PCIe-attached SSDs will be one of the target designs for fast NVM. Also, they employ significant amount of parallelism by having multiple channels, where each channel has multiple memory chips. Such a highly parallelized structure

provides rich opportunities for parallelism.

2.2 Crash Consistency in File Systems

Modern file systems provide crash consistency to applications. They employ journaling or copy-on-write (COW) mechanisms for transaction processing. Journaling file systems such as EXT4 [25], XFS [26], JFS [27], and ReiserFS [28] use a variant of write-ahead logging (WAL) [29], which first writes the metadata and/or data to journal area before in-place updates to metadata or/and data in storage for atomicity and durability. COW file systems [30], such as BTRFS [31], ZFS [32], and log-structured file system [33], use out-of-place updates to support crash consistency. They copy and modify the data for atomic update and then free the previous data through garbage collection.

This dissertation focuses on the EXT4 journaling mechanism since EXT4 is the most widely used file system in Linux and general to other file systems [15,34]. The EXT4 uses a fork of the journaling block device (JBD) called JBD2. The JBD is a file system-independent interface that can also be attached to other file systems such as EXT3 and OCFS2. It performs journal updates, commits, and checkpoint operations. EXT4 offers three journaling modes, such as write-back, ordered, and data journaling [15,35–37]. Write-back is the weakest crash consistency mode among the three journaling modes. This mode writes the metadata into the journal area, but the user data may be written into the original area in the file system after its metadata has been committed to the journal. In this mode, the ordering between the data and metadata is not preserved. The ordered mode provides stronger crash consistency than the write-back mode by keeping order between the metadata and data. Similar to the write-back mode, this mode writes the metadata into the journal while the

data is directly to the original area in the file system before the metadata is written into the journal.

The data journaling mode supports the highest crash consistency with data integrity. Both metadata and data are written into the journal area prior to being written into the original area in the file system to ensure they are updated atomically to persistent storage; they are either committed or aborted together in a transaction. However, the overhead of the data journaling mode is the largest among the journaling modes since the data is written to storage twice.

When an application updates blocks, a new transaction starts or the modification is compounded to the already running transaction activated by another application, which is a compound transaction scheme; EXT4 has only one running transaction and one committing transaction at any time [15, 38]. When the commit occurs at an interval of journal commit (5 seconds) or `fsync` call, the updated blocks are written into the journal area.

The transaction finishes the commit work after writing the commit block[1] into the end of the written blocks in the journal area. This commit block decides whether the transaction is committed or uncommitted. In a system fail or sudden power outage, the file system is remounted, and the file system scans the blocks in the journal area. Then, the file system replays the metadata/data blocks with a commit block and discards the blocks without a commit block. Checkpointing is triggered periodically and activated when the amount of the free space in the journal area drops below a certain threshold. The checkpoint operation writes the metadata/data in the committed transactions into the original area. The journal area is reclaimed via checkpoint so that the transaction can be continuously processed by writing the metadata/data into the free

[1] A commit block generates a flush command to preserve the ordering between journal metadata/data and the commit block

space in the journal area.

2.3 Read and Write Operations in the Existing File Systems

In this section, we describe current I/O strategies such as read-ahead and write-back. These strategies are applied to most Linux file systems in the same manner, and they are used by default when applications open files. When a buffered read is used, the file system performs the read-ahead technique to take advantage of spatial locality. To do this, the file system selects the user requested page(s) as well as additional adjacent pages. This technique is especially useful for sequential read patterns, as the next accessed pages are more likely to already be in the page cache, resulting in a higher page cache hit rate.

Figure 2.1a shows an example of a read-ahead operation in the existing file system. There are five pages (Page 0-4). Page 0 is the page requested by the user, and the other pages (Page1-4) are contiguous pages that the file system wants to read ahead. Each page is mapped to LBA 30, 31, 32, 33, and 34, respectively.

The file system performs read-ahead operations only for non up-to-date pages. It checks whether the LBAs for pages are contiguous with each other to ensure that each request has only contiguous pages. For example, in the case of Figure 2.1a, since Page 2 is already up-to-date, it incurs a hole in LBA. Thus, the file system first merges Page 0 (LBA: 30) and Page 1 (LBA: 31) into a request and issues the request (Request #1) since the LBAs of Page1 and Page3 are not contiguous. After I/O completion (polling) of the request, the file system rechecks the contiguity between the LBAs of Page 3 and Page 4. The file system merges these pages into a single request and issues the request (Request #2). In the read-ahead operation, the contiguity of the LBA is dependent on the

state of the page.

When a buffered write is used, the file system performs write-back operations for dirty pages if the dirty rate of the pages in the page cache is higher than the threshold. The file system chooses the dirty pages from the page cache and obtains the LBAs for the pages. The file system then checks the contiguity among pages. We note that the file system selects dirty pages without considering whether the pages are contiguous or not, unlike the prefetched pages.

Figure 2.1b describes an example of a write-back operation. There are five dirty pages (Page 0-4). The pages are mapped to each LBA: 1, 2, 20, 89, and 45, respectively. The file system merges the two contiguous pages (Page 0 (LBA: 1) and Page 1 (LBA: 2)) into a single request and issues the request (Request #1). After I/O completion of the request, the file system rechecks the LBA of the next page (Page 2) against the following page (Page 3). Since they are discontinuous with each other, the file system first issues the request (Request #2) for Page 2 and completes the I/O. Likewise, in order, the remaining pages (Page 3, Page 4) are issued and completed as separate requests (Request #3 and Request #4). In this example, since the LBAs (20, 89, and 45) are all discontinuous, the file system performs four operations.

In write-back operations, the LBA's contiguity is dependent on the sequence of the dirty pages. Consequently, current read-ahead and write-back operations can reduce the bandwidth by incurring several requests instead of one large request among discontinuous pages.

2.4 Journal I/O in the Journaling File Systems

We describe journal I/O operations, such as journal metadata/data, commit, and checkpoint, based on the data journaling mode. Figure 2.2a shows the journal metadata/data and commit operations in a transaction for the existing

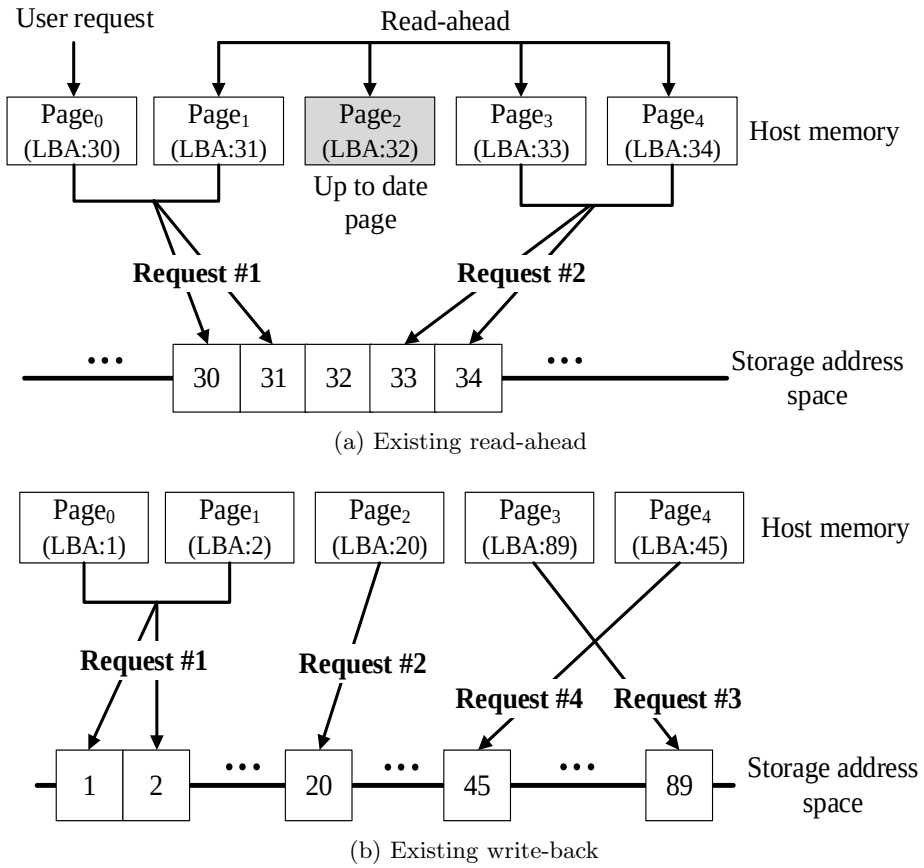
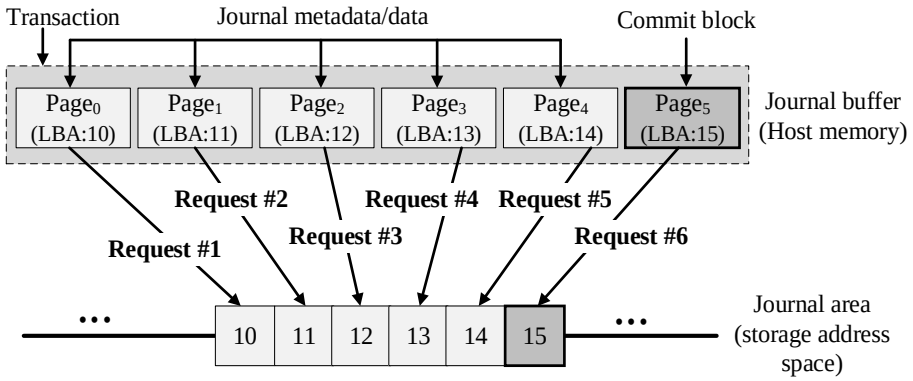


Figure 2.1: Read-ahead and write-back of existing file system

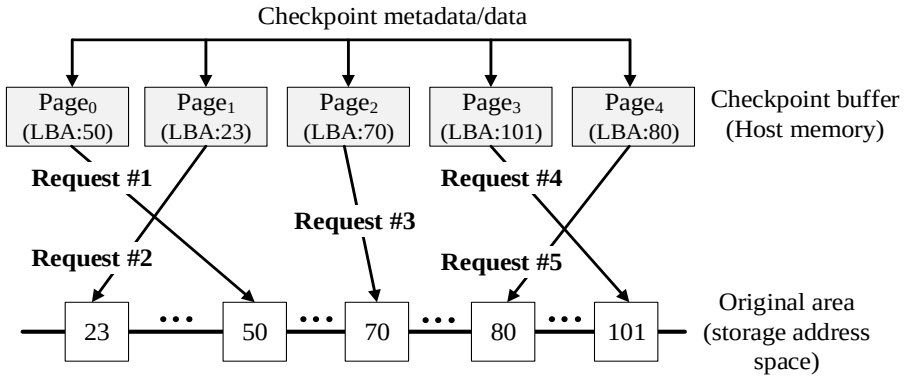
file system. There are five pages (Page 0-4) for journal blocks and one page (Page 5) for a commit block to be written into the journal area. They are mapped to each LBA: 10, 11, 12, 13, 14, and 15 respectively. The file system gets a journal block from the journal buffer, issues the I/O and completes the I/O repeatedly until the I/O for the journal blocks in the transaction are finished. After the five pages (Page 0-4) for the journal blocks completely are written into the journal area, the page (Page 5) for a commit block is written into the journal area resulting in the transaction being committed. As a result, the journal I/Os for each LBA make several requests (Request #1-#6).

We note that the I/O pattern for the journal metadata/data written to the journal area is sequential. In a conventional block I/O subsystem, the sequential writes in the journal I/O can be merged by using functions supported by an I/O scheduler. For example, the JBD2 module uses two functions such as `blk_start_plug()` and `blk_finish_plug()`. The I/O scheduler merges sequential write requests between `blk_start_plug()` and `blk_finish_plug()` call into a single large request. This mechanism is a disk-friendly feature. However, in the case of high-performance storage devices, an I/O request bypasses the I/O scheduler due to the well-known performance issue [14]. Thus, the JBD2 module cannot explicitly use the features mentioned above due to the absence of the I/O scheduler, and this leads to individual write requests for the journal I/O. In our work, we use an optimized block I/O subsystem without the I/O scheduler since it shows the best performance among all configurations. The sequential writes in the journal I/O are performed as individual requests one by one in the optimized block I/O subsystem.

Figure 2.2b shows the checkpoint operations in the existing file system. There are five pages (Page 0-4) for checkpoint updates in the checkpoint buffer, and they are mapped to each LBA: 23, 50, 70, 101, and 80, respectively. Since the checkpoint operation writes the metadata/data blocks in the committed transaction to the original area, their LBAs can be discontiguous to each other. The file system gets a block (page) from the checkpoint buffer and then issues and completes the block iteratively. As shown in this figure, there are five separate requests (Request #1-#5). In conclusion, journaling and checkpoint operations are issued and completed by each request per page. This current I/O operation can reduce the bandwidth by incurring several requests instead of one large request among pages.



(a) Existing journal metadata/data



(b) Existing checkpoint

Figure 2.2: Journal metadata/data and checkpoint of existing file system

2.5 Recovery in the Journaling File Systems

In this section, we describe the recovery procedure in the existing journaling file system. After a system crash or power outage, the mount process reads the journal blocks from the journal area and replays the changes until the file system is consistent again. The changes are atomic in that they are either replayed completely during recovery or are not replayed at all if they had not yet been completely written to the journal area before the crash occurred.

We analyze the recovery I/O path in the JBD2 module. The module per-

forms read operations for getting all the journal blocks in the journal area. The module then performs the checksum operation for the scanned blocks and then selects blocks in the committed transactions. After all blocks to be replayed are selected, the blocks are written to their original area by a sync operation, which writes the blocks one by one. The journal area is initialized after the blocks in the journal area are completely written to the original area.

Figure 2.3 shows an example of a recovery procedure in terms of I/O operations. The mount process reads the blocks (Page 0-4) mapped to each LAB: 1, 2, 3, and 4, in the journal area through four requests (Request #1-#4). The JBD2 module goes through the block device layer directly for the read operation, and therefore, no read-ahead is performed. Similar to the case of journal metadata/data I/O, in the Linux I/O scheduler-based system, adjacent blocks can be merged. However, the I/O stack without the I/O scheduler performs the I/Os as individual requests.

During the recovery procedure, the mount process identifies the journal blocks and a commit block. If the commit block exists, as shown in the figure, the journal blocks can be recovered and are written into the original area with LBA: 33, 56, 78, through three requests (Request #4-#6). Consequently, as our observation, the existing mount process performs inefficient I/O operations by issuing several requests to storage.

2.6 Existing Locking and I/O Parallelism in Journaling File Systems

In this section, we investigate the locking and I/O parallelism in EXT4/JBD2. As shown in Figure 2.4a and 2.4b, a spin lock (`j_list_lock`) is used to ensure the correct list operations for journal heads (`jhs`)[2] in the journaling lists

[2]Journal head (`jh`) is a structure that associates the buffer (buffer_head (`bh`)) with the respective transaction [39]. The operations on the `bh` are protected by a spin lock

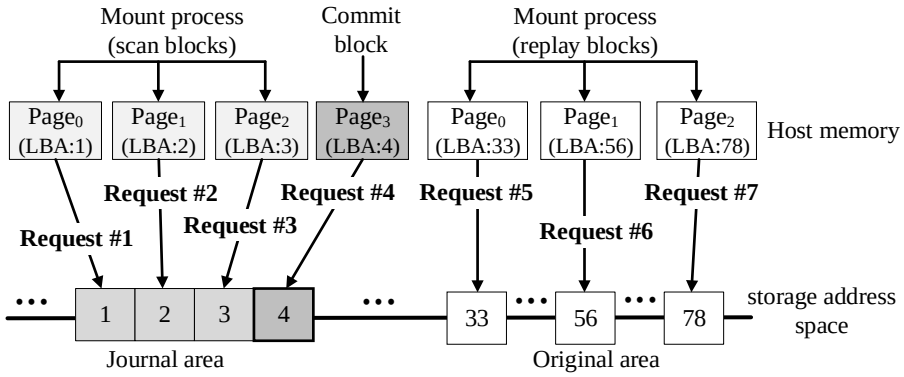
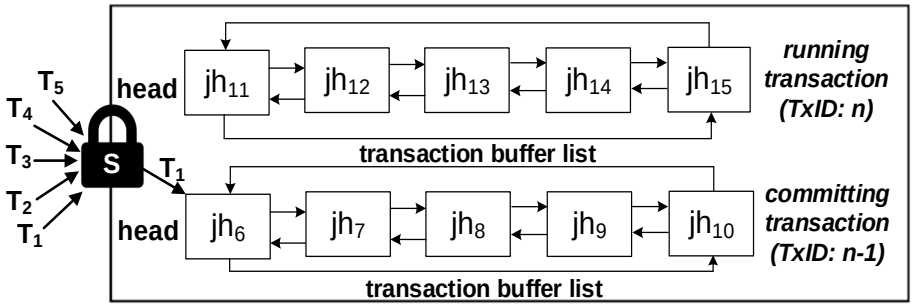


Figure 2.3: Existing recovery I/O operations

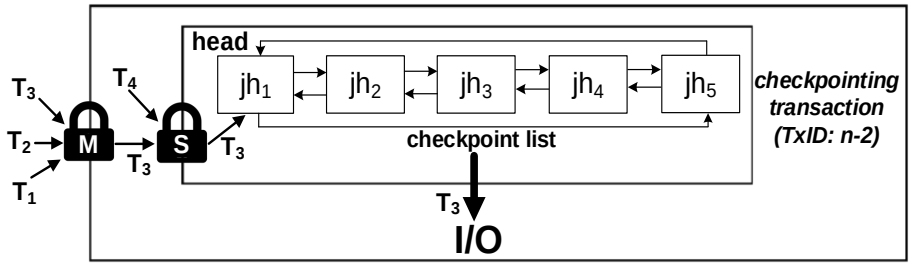
(transaction buffer and checkpoint lists) [15], which are circular doubly linked lists. However, in multi-cores, this locking can incur a contention on the shared data structures and limit the scalability. In addition, only a single thread performs the journal and checkpoint I/Os. For example, as shown in Figure 2.4b, T_3 performs I/O operations for checkpointing by acquiring a mutex lock (`j_checkpoint_mutex`). Such serialized I/O operations can limit the I/O parallelism on high-performance storage. We will explain the transaction processing in terms of locking and I/O operations with the following simplified procedures.

Running transaction. When application threads perform some file operations (e.g., `create()`), they start a transaction to handle the modifications (Procedure 1, lines 3 and 31-39). To process the transaction, the threads first check if a running transaction is available or not. If a running transaction is available, the threads join the running transaction by increasing the number of updates (`t_updates`) in the transaction under the state lock (`j_state_lock`) which is a read-write lock; the `t_updates` variable indicates the number of current threads that join the transaction. Otherwise, a new transaction is created,

(`jbd_lock_bh_state`) per `bh`.



(a) running and committing transaction



(b) checkpointing transaction

Figure 2.4: Examples of existing locking and I/O operations (T: thread, TxID: transaction ID, jh: journal_head, S: spin lock (`j_list_lock`)), M: mutex lock (`j_checkpoint_mutex`)

or the threads are scheduled[3] if the transaction cannot be newly created.

After getting the running transaction (line 4), the threads modify their own buffer and then try to insert it into a transaction buffer list by using the `jh` of the buffer (`bh`). To insert the `jh`, the threads try to acquire a list lock (`j_list_lock`) which is a spin lock (lines 5-6). A thread, which acquires the list lock, associates the `jh` to the running transaction (line 45) and inserts the `jh` into the tail of the list (line 46). Then, the thread releases the list lock and finishes the insert operation (line 7). Finally, the thread completes its own transaction processing

[3]When a running transaction needs to be committed while a previous transaction is committing, the threads which try to get a running transaction are scheduled until the running transaction is available. It is because there are only one running transaction and one committing transaction at any time in the compound transaction scheme [15, 36].

PROCEDURE 1 C-like pseudo-code of running transaction in EXT4/JBD2

```
1: create(dir, ...){
2:      /* create a new file */
3:     handle = jbd2_journal_start(journal, ...);
4:     transaction = handle->transaction;
5:     spin_lock(journal->j_list_lock);
6:     add_buffer(bh->jh, transaction, transaction->t_buffers);
7:     spin_unlock(journal->j_list_lock);
8:     jbd2_journal_stop(handle);
9: }

10: truncate(dentry, ...){
11:      /* truncate a file */
12:     journal_unmap_buffer(journal, bh);
13: }

14: journal_unmap_buffer(journal, bh){
15:     /* invalidate a buffer */
16:     write_lock(journal->j_state_lock);
17:     spin_lock(journal->j_list_lock);
18:     transaction = bh->jh->transaction;
19:     if(!bh->jh->cp_transaction){
20:         head = jh->cp_transaction->t_checkpoint_list;
21:         del_buffer(bh->jh, bh->jh->cp_transaction, head);
22:     }else if(transaction == journal->j_committing_transaction){
23:         set_buffer_free(bh);
24:     }else if(transaction == journal->j_running_transaction){
25:         head = journal->j_running_transaction->t_buffers;
26:         del_buffer(bh->jh, transaction, head);
27:     }
28:     spin_unlock(journal->j_list_lock);
29:     write_unlock(journal->j_state_lock);
30: }
```

by decreasing the number of updates (lines 8 and 40-43).

When application threads perform some file operations, such as `truncate()`, the threads can invalidate buffers that are already associated with a transaction

```
31: jbd2_journal_start(journal, ...){
32:     if(j_running_transaction is not available)
33:         <create a new transaction or call schedule()>
34:     read_lock(journal->j_state_lock);
35:     handle->transaction = journal->j_running_transaction;
36:     atomic_add(transaction->t_updates, 1);
37:     read_unlock(journal->j_state_lock);
38:     return handle;
39: }

40: jbd2_journal_stop(handle){
41:     /* complete a transaction */
42:     atomic_sub(handle->transaction->t_updates, 1);
43: }

44: add_buffer(jh, transaction, head) [a] {
45:     jh->transaction = transaction;
46:     tail = head->prev;
47:     if(!head){
48:         jh->next = jh->prev = head = jh;
49:     } else{
50:         jh->prev = tail; jh->next = head; tail->next = head->prev = jh;
51:     }
52: }

53: del_buffer(jh, transaction, head) [a]{
54:     if(head == jh){
55:         head = jh->next;
56:         if(head == jh)
57:             head = NULL;
58:     }
59:     jh->prev->next = jh->next; jh->next->prev = jh->prev;
60:     jh->transaction = NULL;
61: }
```

[a] The `jh` is inserted into/removed from a transaction buffer list or checkpoint list by using the `prev/next/transaction` or `cpprev/cpnext/cp_transaction` fields of the `jh`, respectively.

(lines 10-13, 14-30, and 48-51). In this case, by acquiring the state lock and the list lock (lines 16-17), a thread removes the `jh` from the transaction buffer or checkpoint lists (line 49) and disassociates the `jh` with the running or checkpoint transactions (line 50) if it is associated with the running or checkpoint transactions, respectively. If the `jh` is associated with a committing transaction, the thread sets the `jh` as *freed*; both the `jh` and its buffer will be freed later during the commit procedure. As discussed above, EXT4/JBD2 ensures correct updates on the transaction state and the transaction buffer list by the state lock and the list lock, respectively.

Committing transaction. To commit a transaction, a journal thread wakes up and processes a commit procedure (Procedure 2). First, the thread changes the running transaction to a committing transaction and its state to *committing*. Then, the thread initializes the running transaction by acquiring the state lock (lines 2-5). And then, the journal thread waits for other threads to complete their transaction processing by checking the `t_updates` variable (line 6). Therefore, if the `jh` is already associated with a running transaction, the `jh` must be moved to a committing transaction. Meanwhile, the committing transaction does not accept any new modifications, and the next modification triggers the creation of a new running transaction. With the committing transaction, the journal thread prepares for journal I/Os by creating a wait list, which is used to wait the I/Os (line 8). Then, the thread fetches the `jh` from the head (`t_buffers`) of the transaction buffer list and creates a copy of its buffer called frozen buffer (`frozen_bh`) to preserve the contents of the buffer (lines 9-11). And then, the thread removes the `jh` from the list by updating the head of the list to the next of the head, and inserts the `jh` into the shadow list[4]

[4]The shadow list (`t_shadow`) includes the frozen buffers which preserves the contents of the buffers.

under the list lock (lines 12-15).

To perform a batched journal I/O, the journal thread aggregates the frozen buffer by inserting it into a write buffer (`wbuf`) and the wait list (lines 16-17). If the number of inserted buffers (`bufs`) is higher than the pre-defined threshold, the thread issues I/Os to the journal area by calling `submit_bh()` and prepares for the next I/Os (lines 18-22). After issuing all the I/O requests for journaling, the thread waits for the I/Os and then removes the `jh` from the shadow list and inserts it into the forget list[5] under the list lock (lines 24-32). After all the I/Os are completed, the journal thread writes the commit block for the transaction atomicity (line 33); if a crash occurs, the file system can replay or discard the transaction according to the existence of the commit block of the transaction. Then, the thread makes a checkpoint list with the buffers that are not freed and still dirty in the forget list under the list lock (lines 34-42). Finally, the committed transaction is inserted into the tail of a checkpoint transaction list for checkpointing by acquiring the state and list locks (lines 43-48).

Checkpointing transaction. When a transaction needs to be checkpointed, application threads try to acquire a checkpoint mutex lock (`j_checkpoint_mutex`) and perform a batched I/O operation (Procedure 3, line 2). A thread, which acquires the mutex lock, performs the checkpoint I/O operations while others are blocked by the lock until the I/O operations are completed. Then, the thread tries to acquire the list lock to get the transaction and access its checkpoint list (lines 3-9). The list lock is used since other threads can access the checkpoint list to remove the `jh`s when they free the buffers of the `jh`s, which do not need to be checkpointed.

[5]The forget list (`t_forget`) includes both the frozen buffers from the shadow list and buffers to be freed. In some cases, when an application thread frees a buffer which is associated with a transaction but not needed to be checkpointed, the thread inserts the `jh` of the buffer into the forget list. By doing so, the `jh` is not inserted into the checkpoint list at the commit procedure.

PROCEDURE 2 C-like pseudo-code of committing transaction in EXT4/JBD2

```
1: jbd2_journal_commit_transaction(journal){
2:     transaction = journal->j_running_transaction;
3:     write_lock(journal->j_state_lock);
4:     journal->j_committing_transaction = transaction;
5:     journal->j_running_transaction = NULL;
6:     while(atomic_read(transaction->t_updates)){...}
7:     write_unlock(journal->j_state_lock);
8:     create_wait_list(local_wait_list); // create a local wait list
9:     while(transaction->t_buffers){
10:        jh = transaction->t_buffers;
11:        <making a frozen buffer (frozen_bh)>
12:        spin_lock(journal->j_list_lock);
13:        del_buffer(jh, transaction, transaction->t_buffers);
14:        add_buffer(jh, transaction, transaction->t_shadow);
15:        spin_unlock(journal->j_list_lock);
16:        wbuf[bufs++] = jh->frozen_bh;
17:        add_wait_list(local_wait_list, jh->frozen_bh);
18:        if(bufs == journal->j_wbufsize){ /*j_wbufsize: 341*/
19:            for(i=0 ; i<bufs ; i++)
20:               submit_bh(WRITE, wbuf[i]);
21:            bufs=0;
22:        }
23:    }
24:    while(!list_empty(local_wait_list)){
25:        frozen_bh = list_entry(local_wait_list.prev, ...);
26:        wait_on_buffer(frozen_bh);
27:        jh = transaction->t_shadow->prev;
28:        spin_lock(journal->j_list_lock);
29:        del_buffer(jh, transaction, transaction->t_shadow);
30:        add_buffer(jh, transaction, transaction->t_forget);
31:        spin_unlock(journal->j_list_lock);
32:    }
33:    <issue and complete a commit block>
```

Under the mutex and list locks, the thread aggregates the buffers by fetching the `jh`s from the checkpoint list and inserting the fetched buffers into a

```
34:     spin_lock(journal->j_list_lock);
35:     while(transaction->t_forget){
36:         jh = transaction->t_forget;
37:         jh->transaction = NULL;
38:         if(!buffer_freed(jh->bh) && jbddirty(jh->bh))
39:             add_buffer(jh, transaction, transaction->t_checkpoint_list);
40:         del_buffer(jh, transaction, trasnaction->t_forget);
41:     }
42:     spin_unlock(journal->j_list_lock);
43:     write_lock(journal->j_state_lock);
44:     spin_lock(journal->j_list_lock);
45:     <insert the committed transaction into a checkpoint transaction list
46:         (journal->j_checkpoint_transactions)>
47:     spin_unlock(journal->j_list_lock);
48:     write_unlock(journal->j_state_lock);
49: }
```

checkpoint buffer (`j_chkpt_bhs`) to issue the I/Os in a batched manner (lines 9-21). Similar to the commit procedure, the `jh` is removed and re-inserted into a checkpoint io list, which is used for I/O completion. If the number of aggregated buffers (`batch_count`) is higher than the pre-defined threshold, the thread releases the list lock and issues the I/Os. Then, the thread prepares for the next I/Os by acquiring the list lock. After issuing all the I/Os, the thread completes them in a batched manner by using the checkpoint io list under the list lock (lines 22-34). After then, the thread sets the next transaction to be checkpointed in the checkpoint transaction list. Finally, the checkpointed transaction is freed, which denotes the end of a life cycle of the transaction, and the list lock and the mutex lock are released (lines 32 and 35-36).

PROCEDURE 3 C-like pseudo-code of checkpointing transaction in EXT4/JBD2

```
1: jbd2_log_wait_for_space(journal){
2:     mutex_lock(journal->j_checkpoint_mutex);
3:     spin_lock(journal->j_list_lock);
4:     if((transaction = journal->j_checkpoint_transactions) == NULL){
5:         spin_unlock(journal->j_list_lock);
6:         mutex_unlock(journal->j_checkpoint_mutex);
7:         return;
8:     }
9:     while(transaction->t_checkpoint_list){
10:        jh = transaction->t_checkpoint_list;
11:        journal->j_chkpt_bhs[batch_count++] = jh->bh;
12:        del_buffer(jh, transaction, transaction->t_checkpoint_list);
13:        add_buffer(jh, transaction, transaction->t_checkpoint_io_list);
14:        if((batch_count == JBD3_NR_BATCH)){/*JBD3_NR_BATCH:64*/
15:            spin_unlock(journal->j_list_lock);
16:            for(i=0;i<batch_count;i++)
17:                submit_bh(WRITE, journal->j_chkpt_bhs[i]);
18:            batch_count = 0;
19:            spin_lock(journal->j_list_lock);
20:        }
21:    }
22:    while(transaction->t_checkpoint_io_list){
23:        jh = transaction->t_checkpoint_io_list;
24:        spin_unlock(journal->j_list_lock);
25:        wait_on_buffer(jh->bh);
26:        spin_lock(journal->j_list_lock);
27:        del_buffer(jh, transaction, transaction->t_checkpoint_io_list);
28:        if(transaction->t_checkpoint_list == NULL &&
29:            transaction->t_checkpoint_io_list == NULL){
30:            <set the next transaction to be checkpointed
31:                    in the checkpoint transaction list>
32:            free(transaction);
33:        }
34:    }
35:    spin_unlock(journal->j_list_lock);
36:    mutex_unlock(journal->j_checkpoint_mutex);
37: }
```

Chapter 3

Design and Implementation

3.1 Optimizing File Systems for Low-latency Storage Devices

In this section, we describe the design of our optimization techniques to increase the bandwidth per thread for read and write operations. The key idea is to combine multiple and individual pages into a single large request and issue/complete it irrespective of the LBAs for each page.

3.1.1 Design

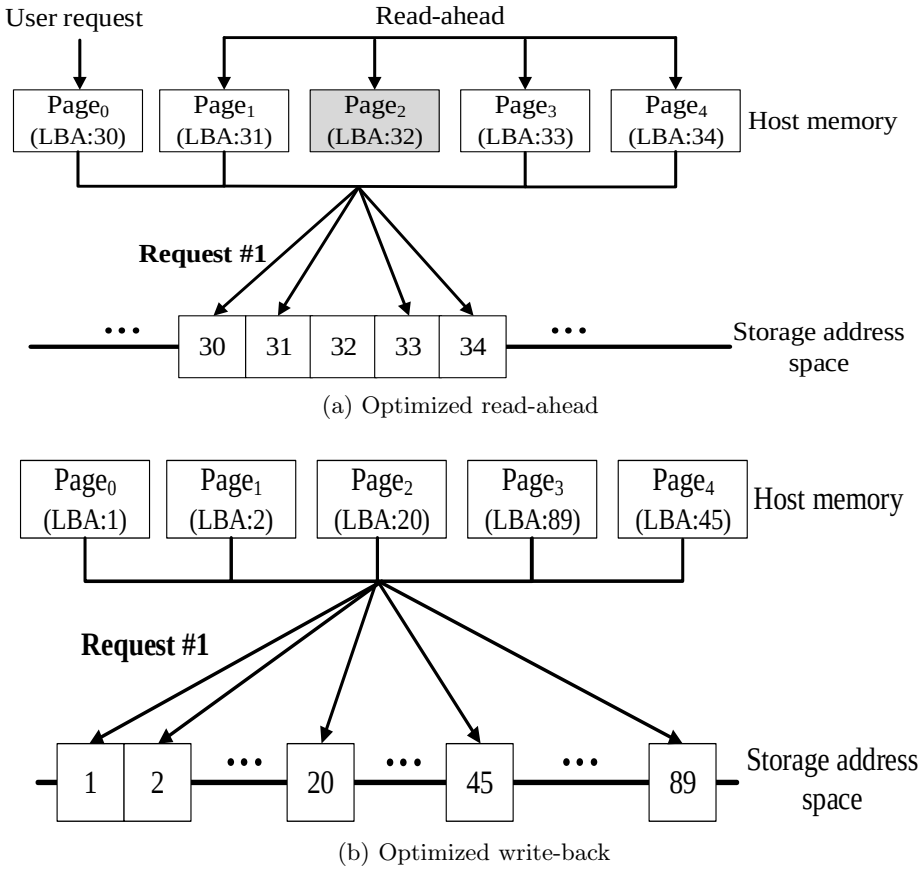
Read and write operations in the optimized file system

In the case of the read operation, we observe the read-ahead dilemma of whether to use prefetching or not. Enabling the read-ahead technique is not beneficial to the random access workload since the prefetched data is not normally expected under the workload. Thus, disabling the technique is advantageous to the random access workload while it degrades the sequential read throughput significantly as a side effect; performance is decreased by about 50% without

prefetching. To resolve this dilemma, the baseline system disables the context lookup feature [14]. It reduces the number of prefetch pages under a random read workload while still providing a sufficient number of prefetched pages for the sequential read workload. As a result, the random read performance is improved without degrading the sequential read performance.

Figure 3.1a outlines the read-ahead of the optimized file system. There are five pages (Page 0-4) where Page 0 is the demanded page and the other pages (Page 1-4) are contiguous pages which the file system wants to read ahead. The pages are mapped to each LBA, and the state of Page 2 is already up-to-date so that the pages to be read are Page 0, 1, 3, and 4. In this situation, the existing file system issues two requests such as a request for Page 0-1 and another request for Page 3-4. Unlike the existing file system, our optimized file system gathers the pages (Page 0, 1, 3, and 4) and issues/completes a single large request (Request #1) with gathered pages. This scheme demonstrates that the file system increases the number of I/Os per request and reduces the number of operations for issue and completion irrespective of the LBA's contiguity according to the state of the pages.

Figure 3.1b describes write-back of the optimized file system. There are five dirty pages (Page 0-4) from the page cache, and they are mapped to each LBA. The optimized file system merges the dirty pages from the page cache into a single request (Request #1). This scheme shows that the sequence of the dirty pages negatively affects the bandwidth per thread. It also demonstrates that the performance of the read operations can be improved by reducing the flushing time whenever the write-back operation occurs. Finally, our scheme does not sacrifice the consistency of the current file systems by preserving the metadata and journaling mechanism.



(a) Optimized read-ahead

(b) Optimized write-back

Figure 3.1: Read-ahead and write-back of the optimized file system

Journal I/O in the optimized file system

In this section, we describe the optimization techniques in the journal I/O operations based on the data journaling mode. This optimization provides efficient journaling/checkpoint operations and reduces the time for I/O operations. Our journal I/O scheme still guarantees the same consistency as that of the existing file system.

To increase the bandwidth per journal I/O operation, similar to optimization for the read and write operations, the optimized file system combines

multiple and individual journal pages (blocks) into a large request and issues/completes the request. When a transaction starts to commit, the existing file system makes a temporary I/O buffer in which the journal blocks are included, which was updated by the transaction. Then, the existing file system performs I/O for the block from an I/O buffer one by one. In contrast, the optimized file system makes blocks in the I/O buffer into a large request and issues the request to the device driver.

Figure 3.2a shows the journal metadata/data and commit operations in a transaction for the optimized file system. There are five pages (Page 0-4) for the journal blocks and one page (Page 5) for a commit block. In this example, the pages (Page 0-5) are mapped to LBA 10, 11, 12, 13, 14, and 15. In contrast to the existing file system, the optimized file system merges the journal blocks into a single request (Request #1) and issues the request. After the transfer of the journal blocks is finished, the I/Os are completed at once. To provide the crash consistency at the same level as that of the existing file system, after the I/Os for journal blocks are completely finished, we issue and complete the I/O for the commit block in a request (Request #2). Consequently, our scheme still supports the same consistency with that of the existing file system since our scheme preserves the write ordering between the journal block and the commit block.

When a new transaction is started, the file system checks whether there is enough space left in the journal area to write all potential buffers requested. If there is enough space, the transaction is continuously progressed. Otherwise, the upcoming I/O needs to stall pending a checkpoint to free up some more journal space. Therefore, fast checkpoint operation is required to increase the I/O performance by reducing the stall time. Figure 3.2b describes the optimized checkpoint operations. The checkpoint buffer includes the metadata and

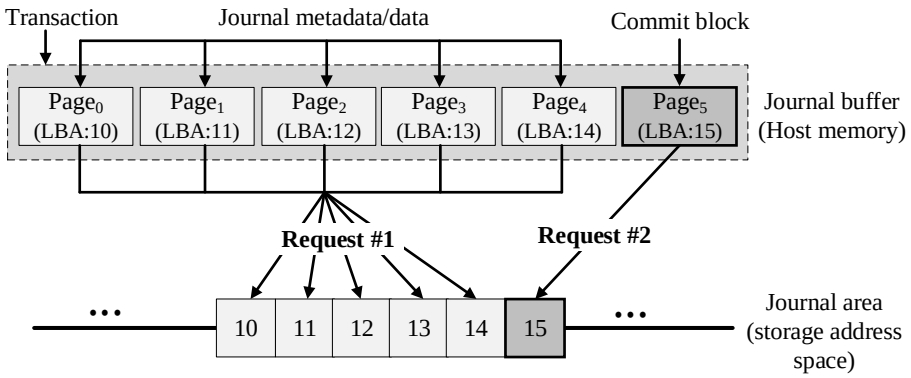
data to be rewritten into the original area. There are five pages (Page 0-5). In this example, the pages are mapped to LBA 50, 23, 70, 101, and 80. When the checkpoint is activated, the optimized file system makes the blocks in the checkpoint buffer into a single request (Request #1) and issues/completes the request at once, irrespective of the LBA's contiguity.

Our scheme allows the file system to increase the number of I/Os per request and reduces the number of operations for issue and completion in journal/checkpoint operations. Consequently, our scheme reduces the transfer time for journal I/O and supports shorter journal work by providing efficient I/O operations.

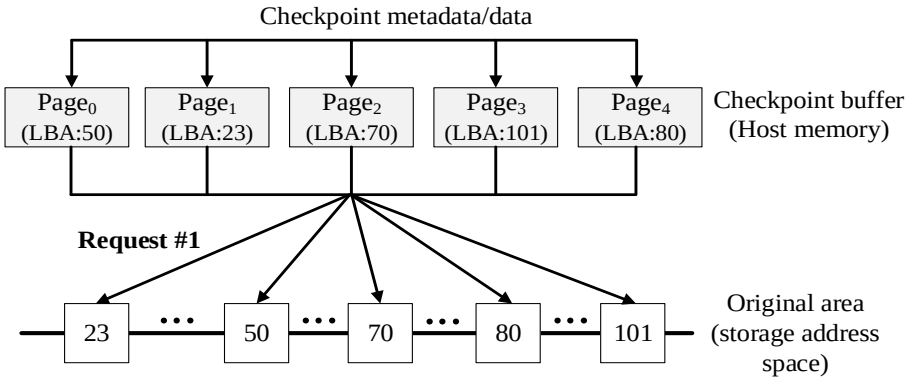
Recovery in the optimized file system

In this section, we present an efficient recovery mechanism in the journaling file system. We provide the optimizations for both scan and replay operations. Our scheme is to make several pages to be scanned and replayed into large requests. In the ordered mode, only the metadata in the committed transaction is replayed to the original metadata area while both the metadata and data in the committed transaction are replayed to their original area in the data journaling mode. Similar to the existing file system, the optimized file system initializes the journal area after the recovery procedure is completely finished.

Figure 3.3 shows an example of the optimized recovery procedure in the optimized file system. There are four pages (Page 0-3) mapped to LBA 1, 2, 3, and 4 in the journal area. When the system is restarted after a system crash or power outage, the optimized file system reads the pages in the journal area. Unlike the existing recovery procedure, the optimized file system reads the pages in a request (Request #1). This optimization allows the mount process to the scanning and selecting the pages to be replayed faster. After scanning, the selected



(a) Optimized journal metadata/data



(b) Optimized checkpoint

Figure 3.2: Journal metadata/data and checkpoint of the optimized file system

pages (Page 0-2) mapped to LBA 33, 56, and 78 are written into the original area. In contrast to the existing file system, the optimized file system makes the pages into a request (Request #2). In this example, our scheme issues and completes the two requests for each scan and replay operation. Consequently, the optimized file system decreases the recovery time and allows the mount process to perform more efficient recovery I/O operations.

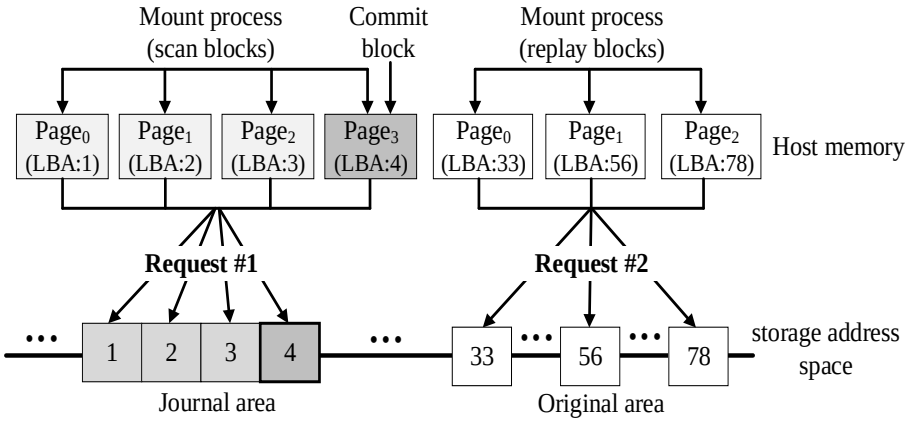


Figure 3.3: Optimized recovery procedure

3.1.2 Implementation

Our optimization requires a DMA engine of the storage device to support the capability of transferring data between discontiguous host memory pages of the file system and discontiguous storage address spaces. To support the capability, the DMA engine of the DRAM-SSD is customized by using a set of descriptors for an I/O request. A descriptor is defined, which includes a mapping of host memory segment, storage segment, and data size. A data structure is defined called Block Control Table (BCT) to contain the descriptors. BCT can contain 1,024 descriptors maximally. Therefore, the 1,024 segments in a single request can be dispatched at once.

Our optimizations can be applied on other types of fast storage devices, which support an ultra-low latency (e.g., a few microseconds) and the capability of transferring data between discontiguous host memory segments and discontiguous storage address spaces. For example, the devices with fast storage medium (a type of memory such as PCM, STT-MRAM, and so on) has an ultra-low latency. RAMCloud also provides low latency access by storing all data in DRAM at all times. The file systems on the fast storage devices or fast

storage systems with fast remote memory access can be optimized if the devices or systems can support the transfer capability. In terms of standard interfaces, the SATA and NVMe do not support the data transfer between discontiguous host memory segments and discontiguous storage address spaces. Therefore, our optimizations are hard to be applied on current flash-based SSDs with the SATA and NVMe interfaces.

Our scheme requires modification of file system and device driver. In read and write operations, the existing file system builds the BIO structure to forward requests to the block layer; the structure is the basic container for block I/O. The existing file system upon the existing block layer identifies only pages with contiguous LBAs and adds the pages to the BIO structure; the BIO structure has the starting LBA in the `bi_sector` field of the structure. To implement our scheme, we define a new data structure called PIO (Proposed I/O) for the page transfers between the file system and the device driver. The PIO structure consists of page vectors, the total request size, and the number of pages. Each page vector contains the page, length, offset, and sector addresses (LBA) to represent the mapping between a single page of the file system and a specific LBA. This enables the file system to transfer multiple LBAs to the device driver.

To this end, we implemented new functions between the file system and the device driver. When a PIO instance for I/O operation arrives from the file system, the device driver allocates as many request descriptors as the total number of pages (`nr_pages` in the PIO structure). Then, the device driver prepares a DMA operation; the device driver calculates the appropriate DMA address for each page in the `dma_map_sg()` function and completes the allocated request descriptors with the DMA address (host memory address), storage address (LBA), and length. Finally, the device driver issues the request descriptors to the storage device.

The file system adds pages for read-ahead operations to a page pool list. The file system then finds the LBAs of the pages from the list one by one. The optimized file system moves pages from the page pool list to the page_vec structure of the PIO with information for each page by not checking the contiguity among the pages. The file system then transfers the PIO via a customized read interface exported by the device driver. We modified the `ext4_readpages()` that is almost identical to other file systems except the block retrieval mechanism.

When write-back occurs, the optimized file system finds dirty pages from the page cache via `pagevec_lookup_tag()` to get the LBAs of the dirty pages. The file system does not check the contiguity of the pages and merges them into page_vec in PIO. In the write-back operation, we modified `ext4_writepages()`. Similar to the optimized read operation, the file system issues the request with PIO via a customized write interface exposed by the device driver. We expect that applying these optimization techniques to other file systems would be relatively easy since the modifications are included in common functions of the Linux file system.

For journaling and recovery optimization, we changed the JBD2 module. We modified `jbd2_journal_commit_transaction()` for journal metadata/data optimization; this function is the primary function for the commit procedure. When the transaction commit occurs, the journal thread wakes up and performs the journal commit procedure. In this situation, the optimized file system gets the journal metadata/data buffer to be transferred and sends the buffer to the device driver using a customized function that is similar to those used in read/write operations. The journal thread issues and completes the I/O for the journal blocks at once and subsequently the I/O for the commit block.

For optimization of the checkpoint, we modified `flush_batch()`, which performs the checkpoint I/O operation in the JBD2 module. The optimized file

system gets the checkpoint buffer and transfers it to the device driver via the customized function. The file system issues and completes the I/O for the checkpoint at once. In short, we add two functions as the interfaces between JBD2 and the device driver. By using the interfaces, when our file system issues the journal metadata/data or checkpoint buffer, our device driver prepares the DMA operation for a single large request similar to read and write operations.

To provide faster recovery, we modified `jbd2_journal_recover()` that is the primary function for recovery when mounting a device. To support a large read request for journal blocks, the optimized file system aggregates the blocks to be read and then issues/completes the aggregated blocks by a request via the customized function. In the replay operation, the mount process writes the selected blocks to the original area via the customized function to write the dirty blocks to be replayed at once without their contiguity. Consequently, in the recovery procedure, we modify the read and re-write operations for journal blocks, which then makes the procedure more efficient and decreases the recovery and remount time.

3.2 Optimizing File Systems for Highly Parallel Storage Devices

To achieve higher I/O performance on multi-cores with high-performance storage, we aim to reduce the lock contention and maximize I/O parallelism in transaction processing. To do this, we propose a transaction processing with two main schemes that enable concurrent updates on shared data structures and cooperatively parallelize I/O operations. We apply these schemes to the transaction processing in EXT4/JBD2.

We maintain the compound transaction scheme of EXT4/JBD2 to exploit its advantages [36]. For example, it provides a better performance when the same

metadata or data is frequently updated within a short period of time. With this advantage, we implement our schemes in the compound transaction. We also preserve the existing ordering of write operations and transactions, such as the ordering of journal blocks and a commit block, committing and checkpointing, and checkpoints. Therefore, our implementation does not require modifications to the existing recovery procedure while not sacrificing the consistency of the file system.

Furthermore, we do not optimize all locking operations in transaction processing but focus on the list lock for management of journal heads and the checkpoint mutex lock for serialized I/O operations. Compared to the list lock and the mutex lock, other locks, such as state lock do not incur a significant overhead according to our evaluation as well as other works [15]. However, such locks can be a performance bottleneck in a massive number of cores, which is beyond this dissertation; therefore, we leave the latent performance issue as a future work.

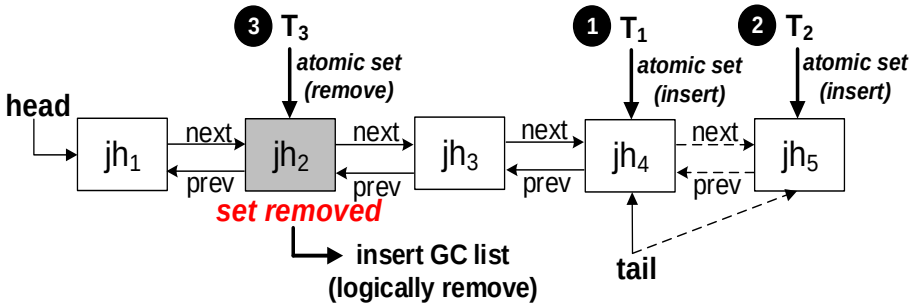
3.2.1 Design

Concurrent updates on data structures

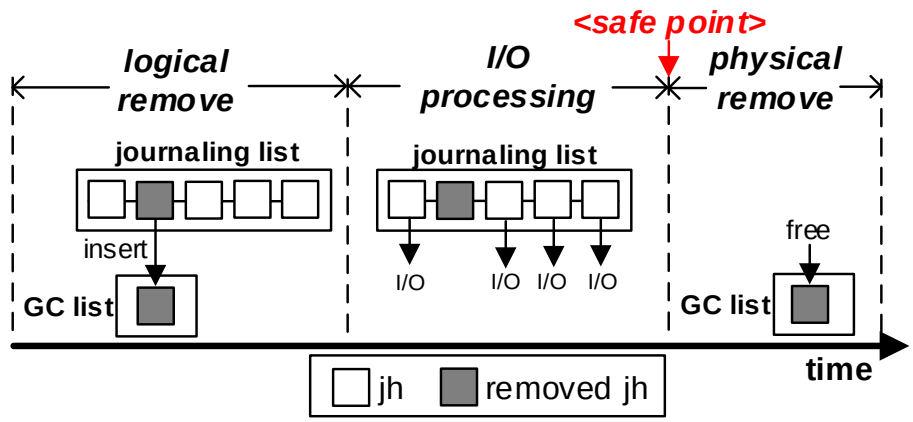
We manage the linked lists for transaction processing in a lock-free manner as shown in Figure 3.4. To this end, instead of the existing circular doubly linked lists, we use non-circular doubly linked lists and add the `tail` to the lists to enable lock-free operations[1].

INSERT. We provide a concurrent insert operation to add an item to a list. In the existing transaction processing, the items are inserted into the tail

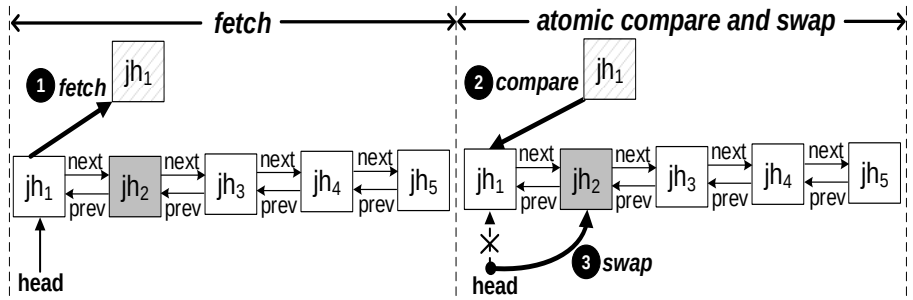
[1] In the circular doubly linked list, when an item is inserted into the list, the multiple pointers that link the item, head, and tail are updated, which makes the atomic insert operation difficult. Instead, we add the tail and set the tail's next item as a constant *NULL* variable [40], which allows us to identify the last element of the list and insert the item into the tail atomically.



(a) Insert and remove operations in a lock-free manner (T: thread)



(b) Two-phase removal (GC: garbage collection)



(c) Fetch operations in a lock-free manner

Figure 3.4: Concurrent updates on data structures

of the list in the incoming order. Similar to the existing scheme but without locking, we concurrently update the tail by the incoming items using an atomic

set instruction. In an example shown in Figure 3.4a, before jh_5 is inserted into a journaling list (e.g., transaction buffer list or checkpoint list), the journaling list consists of four `jh`s, and the tail points jh_4 which is inserted by T_1. When T_2 inserts jh_5, the thread atomically updates the tail and the jh_5's previous item by jh_5 and jh_4, respectively, by executing the atomic set operation. By updating the previous item (jh_4) of jh_5 atomically, the next item of jh_4 is decided as jh_5. This insert operation allows multiple threads to add their item concurrently by updating the tail and linking atomically.

REMOVE. We provide a concurrent remove operation to delete an item from a list. When items are removed from the list concurrently without locking, the *invalid reference* problem [41] can occur. To address this issue, we propose a two-phase remove operation that marks an item as "removed" (logical remove) and then frees the item (physical remove) at a *safe point* when no other threads hold any references to the transaction and logically removed items. This scheme ensures safe access to the items of the list and so threads can perform appropriate operations for the items. For safe garbage collection (GC) for the logically removed items, we additionally maintain a GC list per transaction.

For example, as shown in Figure 3.4a, when a thread (T_3) tries to remove the `jh` (jh_2), the thread marks the `jh` as *removed* atomically by executing the atomic set instruction. Then, the thread inserts the `jh` into the GC list using our concurrent insert operation as shown in Figure 3.4b. And then, threads perform I/O for the valid `jh` or bypass the I/O for the logically removed `jh` while traversing the list safely. When the transaction arrives at the *safe point*, all items in the GC list are reclaimed. The *safe point* is the point when a transaction is checkpointed. At this point, no other threads reference the logically removed `jh`s in the transaction nor insert any logically removed `jh`s into the GC list of the transaction since all the transaction processing is over. Therefore, we can

free all the logically "removed" `jh`s at the *safe point*.

FETCH. Finally, we provide a concurrent fetch operation to get an item while traversing a list. In the existing transaction processing, the list traversal occurs when no threads insert any items into the list (e.g., journal and checkpoint I/O processing). This ensures that all threads see a consistent view of the list, including valid next pointers of all items. Under this condition, we can simply enable the concurrent fetch operation by using an atomic compare and swap (CAS) instruction. In the example shown in Figure 3.4c, a thread first fetches the current head (jh_1). Then, the thread compares the fetched jh_1 with the current head, and changes the head to jh_1's next item by using the CAS operation. If the thread fails the CAS operation, it repeats the procedure above. This fetch operation allows multiple threads to extract individual items concurrently by updating the head atomically. Consequently, through our concurrent update scheme, multiple threads can insert/remove/fetch their items in the lists concurrently and safely without the existing list lock.

Parallel I/O in a cooperative manner

We provide a parallel I/O in a cooperative manner to maximize the I/O parallelism. In the existing transaction processing, application threads can be scheduled out while the serialized I/O operations (e.g., journal and checkpoint I/O) are performed. On the other hand, in our scheme, we allow the application threads to perform the I/O operations by not scheduling but joining them to the I/O operations. For example, in the case of journal I/O, we allow the threads that cannot get a running transaction to join the I/Os by not scheduling them. In the case of checkpoint I/O, we allow the threads to join the I/Os by eliminating the mutex lock. By joining the multiple threads to the I/O processing, they fetch buffers from the shared linked lists (e.g., journaling lists), issue the

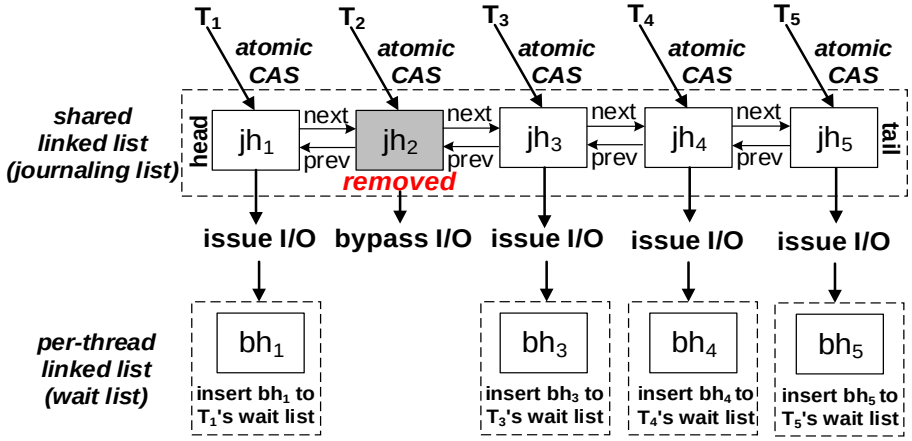


Figure 3.5: Parallel I/O in a cooperative manner (T: thread)

I/Os of the buffers, and complete them in parallel. For better parallelism, we use our concurrent fetch operation and per-thread wait list, which is a linked list used to wait the I/O operations in parallel.

As shown in Figure 3.5, each thread fetches the `jh` concurrently by executing the atomic CAS instruction. Then, each thread issues the I/O of the buffer (i.e., `bh`) associated with the `jh` and inserts the buffer into its own wait list. After all the I/Os are issued, each thread completes its own I/O using its own wait list. Meanwhile, if the fetched `jh` was removed logically, the thread (T_2) does not perform the I/O for the `jh` but fetches the next `jh`.

Using this scheme, multiple threads can cooperate in I/O processing by issuing/completing I/Os in parallel. This can make a commit and checkpoint procedure faster by increasing the I/O parallelism and minimizing the blocking time. We note that our parallel I/O operations can change the I/O ordering between buffers inside a transaction. However, such a change does not sacrifice the atomicity since we write the commit block after all journal blocks are written, which will be described in Section 3.2.2.

The optimized file system with our two schemes preserves the consistency

of the file system by satisfying the following properties: (1) Every block associated with a transaction is written to the journal area at a commit procedure. (2) A transaction is committed or uncommitted (atomicity) according to the commit block. (3) Committed transaction N-1 is checkpointed prior to committed transaction N. We will explain how to apply our schemes to transaction processing and how to satisfy the properties in detail.

3.2.2 Implementation

Running transaction

This section presents our running transaction. We enable multiple application threads to insert/remove the journal heads into/from the transaction buffer list concurrently. Similar to the existing procedure, when the threads start a transaction, they get a running transaction and increase the number of updates in the transaction (Procedure 4, lines 3-4 and 31-39). Meanwhile, in our running procedure, we allow the application threads to cooperate in I/O processing for journal I/Os by calling `journal_io_start()` (lines 32-33), which will be described in Section 3.2.2.

After getting the running transaction, we insert the `jh` into the transaction buffer list by using our concurrent insert operation (lines 5-6 and 44-51). First, the threads associate their `jh` to the running transaction. Then, they update the tail (`t_buffers_tail`) by their `jh` and the `jh`'s previous item by the old tail by executing the `atomic_set` instruction[2]. This instruction updates the tail and returns the old tail atomically. Then, the threads check whether the old tail exists or not. If it does not exist, the head (`t_buffers`) of the list is updated by the inserted `jh`, which becomes the first item in the list. Otherwise, the next item of the old tail is updated by the inserted `jh`.

[2] __sync_lock_test_and_set(type *ptr, type value): This built-in function performs an atomic exchange operation. It writes the value into *ptr, and returns the previous contents of *ptr [42].

PROCEDURE 4 C-like pseudo-code of our running transaction

```
 1: create(dir, ...){
 2:      /* create a new file */
 3:     handle = jbd2_journal_start(journal, ...);
 4:     transaction = handle->transaction;
 5:     add_buffer(bh->jh, transaction,
 6:         transaction->t_buffers, transaction->t_buffers_tail);
 7:     jbd2_journal_stop(handle);
 8: }

 9: truncate(dentry, ...){
10:      /* truncate a file */
11:     journal_unmap_buffer(journal, bh);
12: }

13: journal_unmap_buffer(journal, bh){
14:     /* invalidate a buffer */
15:     write_lock(journal->j_state_lock);
16:     transaction = bh->jh->transaction;
17:     if(!bh->jh->cp_transaction){
18:         head = jh->cp_transaction->gc_head;
19:         tail = jh->cp_transaction->gc_tail;
20:         del_buffer(jh, transaction, head, tail);
21:     }else if(transaction == journal->j_committing_transaction){
22:         set_buffer_free(bh);
23:         atomic_set(jh->removed, removed);
24:     }else if(transaction == journal->j_running_transaction){
25:         head = journal->j_running_transaction->gc_head;
26:         tail = journal->j_running_transaction->gc_tail;
27:         del_buffer(jh, transaction, head, tail);
28:     }
29:     write_unlock(journal->j_state_lock);
30: }
```

For remove operations, we use our two-phase remove operation. When the threads remove their `jh`, they get the GC list of the transaction if the `jh` is

```
31: jbd2_journal_start(journal, ...){
32:     if(j_running_transaction is not available)
33:         /*create a new transaction or call journal_io_start(journal)*/
34:     read_lock(journal->j_state_lock);
35:     handle->transaction = journal->j_running_transaction;
36:     atomic_add(transaction->t_updates, 1);
37:     read_unlock(journal->j_state_lock);
38:     return handle;
39: }

40: jbd2_journal_stop(handle){
41:     /* complete a transaction */
42:     atomic_sub(handle->transaction->t_updates, 1);
43: }

44: add_buffer(jh, transaction, head, tail) {
45:     jh->transaction = transaction;
46:     jh->prev = atomic_set(tail, jh);
47:     if(jh->prev == NULL)
48:         head = jh;
49:     else
50:         jh->prev->next = jh;
51: }

52: del_buffer(jh, transaction, head, tail) {
53:     atomic_set(jh->removed, removed);
54:     jh->gc_prev = atomic_set(tail, jh);
55:     if(jh->gc_prev == NULL)
56:         head = jh;
57:     else
58:         jh->gc_prev->gc_next = jh;
59:     bh->jh = jh->bh = NULL; /* unlink the bh from the jh */
60:     jh->transaction = NULL;
61: }
```

associated with running or checkpoint transactions (lines 17-20 and 24-27). For the logical remove operation (lines 52-61), the thread marks the `jh` as *removed*

by executing the `atomic_set` instruction and inserts the `jh` into the GC list atomically by using `gc_prev/next` fields of the `jh`. Then, the `bh` is unlinked from the removed `jh` (line 59), and the `jh`'s `transaction` or `cp_transaction` field is set to `NULL` in the case of running or checkpointing transaction, respectively (line 60). This means that the `jh` is not associated with the `bh` and the transaction any longer. Thus, the `jh` becomes an obsolete structure, and the `bh` gets freed at this point. This operation on the `bh` is performed safely since the operation is protected by a spin lock (`jbd_lock_bh_state`) per `bh` as same as the existing scheme. Meanwhile, in the case of committing transaction, the thread only marks the `jh` as *removed* (line 23), and both `bh` and `jh` will be freed during the commit procedure.

Committing transaction

In this section, we present our committing transaction. During the existing commit procedure, the journal thread updates the lists under the list lock and performs journal I/O operations by a single thread. On the other hand, in our commit procedure, we update the lists by using our concurrent update operations and parallelize the I/O operations in a cooperative manner.

To commit a transaction, the journal thread gets a committing transaction similar to the existing procedure (Procedure 5, lines 3-9). Then, the journal thread starts the parallel I/O by setting the `journal_io` variable (line 10). This informs application threads that the I/O processing is initiated. Note that in the existing procedure, application threads are blocked when a running transaction is not available and cannot be newly created. Instead of blocking the threads, we enable the threads to perform the I/O processing along with the journal thread by calling `journal_io_start()` (Procedure 4, line 33, Procedure 5, line 11, and Procedure 6, line 2). Thus, the threads can join the I/O processing if

it is initiated by the journal thread (Procedure 6, lines 5-6).

To handle the joined threads, we record the number of threads by executing `atomic_add/sub` instructions[3] (Procedure 6, lines 7 and 20) and create the per-thread wait list for the parallel I/O completion (line 8). Then, we allow each thread to fetch the `jh` from the transaction buffer list by using our concurrent fetch operation, which executes the `atomic_cas` instruction[4] (lines 9-17). If the fetched `jh` was logically removed, the thread bypasses and retries to fetch the next `jh`. Otherwise, each thread creates a frozen buffer, submits the I/O of the buffer to the journal area, and inserts the buffer into its own wait list in parallel.

After all the I/Os are issued, we stop new upcoming threads from joining the I/O processing by unsetting the `journal_io` variable (line 18). Then, the joined threads complete the I/O by using their own wait list (lines 19 and 22-32). Through the procedure above, the parallel I/O is completed by writing all the buffers to the journal area. This procedure satisfies the following property.

Property 1. *Every block associated with a transaction is written to the journal area at a commit procedure.*

Every application thread increases `t_update` *before inserting its* `jh` *(Procedure 4, line 36) and decreases* `t_update` *after inserting its* `jh` *(Procedure 4, line 42). Before the journal thread starts the parallel I/O processing by setting* `journal_io` *(Procedure 5, line 10), the thread waits until* `t_update` *becomes 0 (Procedure 5, line 7). This prevents application threads from starting and finishing the I/O processing before all the* `jh`*s are inserted into the transaction buffer list. Thus, it ensures that all the buffers associated with the transaction are written to the journal area even if the parallel I/O is enabled.* □

While completing the I/Os (Procedure 6, lines 22-32), the threads insert the

[3] __sync_add/sub_and_fetch(type *ptr, type val): These built-in functions atomically add/subtract the value of val to/from the variable that *ptr points to. The functions return the new value of the variable that *ptr points to [42].

[4] __sync_val_compare_and_swap(type *ptr, type oldval, type newval): This built-in function performs an atomic compare and swap operation. If the current value of *ptr is oldval, then write newval into *ptr. Otherwise, no operation is performed. The function returns the contents of *ptr before the operation [42].

PROCEDURE 5 C-like pseudo-code of our committing transaction (1)

```
 1: /*the journal thread commits a transaction*/
 2: jbd2_journal_commit_transaction(journal){
 3:     commit_transaction = journal->j_running_transaction;
 4:     write_lock(journal->j_state_lock);
 5:     journal->j_committing_transaction = commit_transaction;
 6:     journal->j_running_transaction = NULL;
 7:     while(atomic_read(transaction->t_updates)){...}
 8:     write_unlock(journal->j_state_lock);
 9:     transaction = journal->j_committing_transaction;
10:     atomic_set(transaction->journal_io, start);
11:     journal_io_start(journal);
12:     while(atomic_read(transaction->num_io_threads) != 0);
13:     <issue and complete a commit block>
14:     write_lock(journal->j_state_lock);
15:     <insert the committed transaction into a checkpoint transaction list
16:         (journal->j_checkpoint_transactions) using our concurrent insert>
17:     write_unlock(journal->j_state_lock);
18:     atomic_set(transaction->cp_io, start);
19: }
```

jhs into a checkpoint list if the jhs are not removed logically and their buffers are still dirty. In this processing, for simplicity and efficiency, we make the checkpoint list while completing the I/Os before the commit block is written. However, the list is not used for checkpointing until the commit procedure is finished to preserve the ordering of committing and checkpointing.

In addition, we use the wait lists instead of the shadow list and include all the frozen buffers in the wait lists. Instead of the forget list, we use the GC list and insert the jhs which are associated with buffers to be freed to the GC list. After completing all the I/Os, the journal thread waits until all the journal I/Os are finished by using the number of joined threads before writing the commit block (Procedure 5, lines 12-13). This procedure satisfies the following property.

Property 2. *A transaction is committed or uncommitted (atomicity) according*

PROCEDURE 6 C-like pseudo-code of our committing transaction (2)

```
 1: /*the journal thread performs journal I/Os with application threads*/
 2: journal_io_start(journal){
 3:     if((transaction = journal->j_committing_transaction) == NULL)
 4:         return;
 5:     if(atomic_read(transaction->journal_io) == stop)
 6:         return;
 7:     atomic_add(transaction->num_io_threads, 1);
 8:     create_wait_list(local_wait_list); // create a local wait list per thread
 9:     while((jh = transaction->t_buffers) != NULL){
10:         if(atomic_cas(transaction->t_buffers, jh, jh->next) != jh)
11:             continue;
12:         if(atomic_read(jh->removed) == removed)
13:             continue;
14:         <make a frozen buffer (frozen_bh)>
15:         submit_bh(WRITE, jh->frozen_bh);
16:         add_wait_list(local_wait_list, jh->frozen_bh);
17:     }
18:     atomic_set(transaction->journal_io, stop);
19:     wait_journal_io(wait_list);
20:     atomic_sub(transaction->num_io_threads, 1);
21: }

22: wait_journal_io(local_wait_list){
23:     while(!wait_list_empty(local_wait_list){
24:         frozen_bh = list_entry(local_wait_list.next, ...);
25:         wait_on_buffer(frozen_bh);
26:         jh = frozen_bh->bh->jh;
27:         jh->transaction = NULL;
28:         if(atomic_read(jh->removed) != removed && jbddirty(jh->bh))
29:             add_buffer(jh, transaction, transaction->t_checkpoint_list,
30:                 transaction->t_checkpoint_list_tail);
31:     }
32: }
```

to the commit block.

Every application thread that joins the I/O processing increases `num_io_threads` *before issuing I/O (Procedure 6, line 7) and decreases* `num_io_threads` *after*

completing I/O (Procedure 6, line 20). The journal thread waits until `num_io_threads` *becomes 0 before the journal thread writes the commit block (Procedure 5, line 12). This means that all the journal blocks are written before the commit block is written to the journal area. Thus, it ensures the atomicity of the transaction by preserving the ordering between the journal blocks and the commit block.* □

Finally, the journal thread inserts the committed transaction into the checkpoint transaction list by using the state lock (`j_state_lock`) and our concurrent insert operation, and sets the `cp_io` variable to start the checkpoint I/O (lines 14-18).

Checkpointing transaction

This section presents our checkpointing transaction. In the existing procedure, when a transaction needs to be checkpointed, an application thread performs checkpoint I/O operations by acquiring a checkpoint mutex lock (`j_checkpoint_mutex`). Meanwhile, other application threads, which fail to acquire the lock, are blocked until the checkpoint is finished, which can underutilize the I/O parallelism.

To enable a parallel checkpoint I/O, we allow the threads to join the I/O processing instead of using the mutex lock and the checkpoint buffer. However, even with the parallel I/O, the I/O issue/complete operations are still inefficient since the list lock is used to fetch/insert the `jh`s from/into the checkpoint/checkpoint io lists. Thus, we fetch the `jh`s by using our concurrent fetch operation, issue the I/Os, and complete the I/Os by using the per-thread wait list in parallel instead of the global checkpoint io list.

When a checkpoint is triggered, application threads get a transaction to be checkpointed if the transaction is available (Procedure 7, lines 2-3). Then, the threads check whether the transaction can be checkpointed or not by using the `cp_io` variable (lines 4-5). Similar to our commit procedure, we record the number of joined threads, and each thread creates its own wait list (lines 6-7).

For the concurrent and parallel I/O issue, each thread concurrently fetches the `jh` from the checkpoint list, submits the I/O of the buffer associated with the `jh` to the original area, and inserts the buffer into the wait list of each thread in parallel (lines 8-15). If the fetched `jh` was removed logically, the thread retries to fetch the next `jh`. After issuing all the I/Os, we stop new upcoming threads from joining the I/O processing by unsetting the `cp_io` variable (line 16). Then, the joined threads disassociate the `jh`s with the transaction while completing the I/Os (lines 17 and 28-34).

After completing all the I/Os, we find the last remaining thread by decreasing the number of joined threads (line 18). The last thread sets the next transaction to be checkpointed by updating the head of the checkpoint transaction list to the next of the head using the atomic CAS operation (lines 19-20). This procedure satisfies the following property.

Property 3. *Committed transaction N-1 is checkpointed prior to committed transaction N.*

A committed transaction is inserted into tail of the checkpoint transaction list in committed order (Procedure 5, 15-16). The last thread sets the next transaction to be checkpointed in the checkpoint transaction list in committed order (Procedure 7, 19-20). This means that if transaction N-1 is committed prior to transaction N, the transaction N is not checkpointed prior to transaction N-1. Thus, it ensures that all the buffers in the transaction are written to the original area in the committed order. Consequently, our optimized file system preserves the consistency of the file system by satisfying Properties 1, 2, and 3. □

And then, the last thread physically removes all the obsolete `jh`s in the GC list of the transaction (lines 21-24). At this point, we can reclaim the `jh`s safely. It is because all the transaction processing is ended: (1) No other threads reference the logically removed `jh`s in the transaction since all the I/O processing is ended. (2) No other threads insert any logically removed `jh`s into the GC list of the transaction since all the `jh`s in the transaction are disassociated with

PROCEDURE 7 C-like pseudo-code of our checkpointing transaction

```
1: jbd2_log_wait_for_space(journal){
2:     if((transaction = journal->j_checkpoint_transactions) == NULL)
3:         return;
4:     if(atomic_read(transaction->cp_io) == stop)
5:         return;
6:     atomic_add(transaction->cp_num_io_threads, 1);
7:     create_wait_list(local_wait_list); // create a local wait list per thread
8:     while((jh = transaction->t_checkpoint_list) != NULL){
9:         if(atomic_cas(transaction->t_checkpoint_list, jh, jh->next) != jh))
10:            continue;
11:        if(atomic_read(jh->removed) == removed)
12:            continue;
13:        submit_bh(WRITE, jh->bh);
14:        add_wait_list(local_wait_list, jh->bh);
15:    }
16:    atomic_set(transaction->cp_io, stop);
17:    wait_cp_io(local_wait_list);
18:    if(atomic_sub(transaction->cp_num_io_threads, 1) == 0){
19:        <set the next transaction to be checkpointed
20:                in the checkpoint transaction list using atomic_cas>
21:        while((jh = transaction->gc_head) != NULL){
22:            transaction->gc_head = jh->gc_next;
23:            free(jh);
24:        }
25:        free(transaction);
26:    }
27: }

28: wait_cp_io(local_wait_list){
29:    while(!wait_list_empty(local_wait_list){
30:        bh = list_entry(local_wait_list.next, ...);
31:        wait_on_buffer(bh);
32:        bh->jh->cp_transaction = NULL;
33:    }
34: }
```

the transaction. Finally, the last thread frees the checkpointed transaction (line 25).

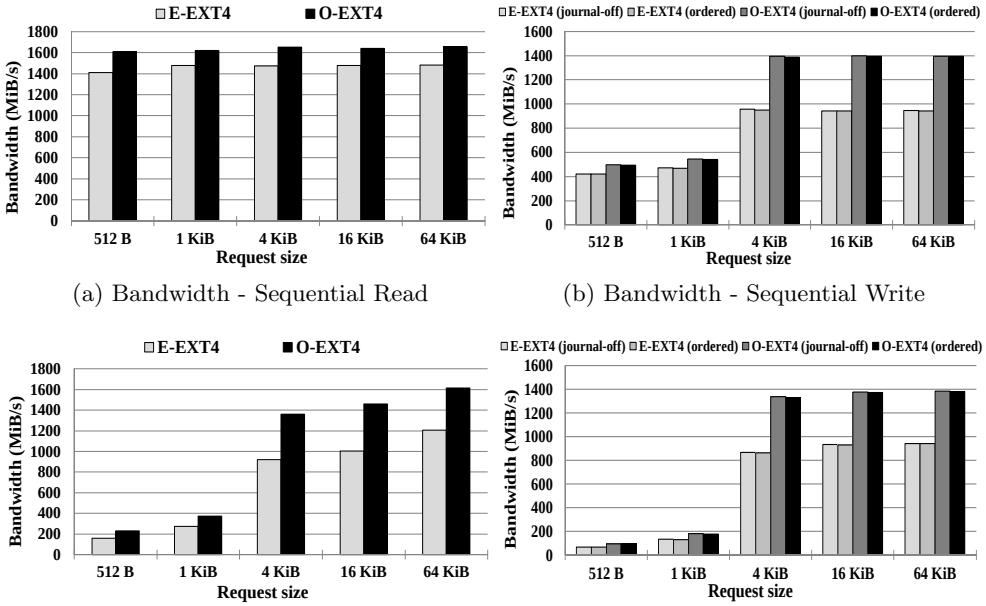
Chapter 4

Evaluation

4.1 Evaluating the Optimized File System for Low-latency Storage

Our machine has an Intel Xeon E5630 2.53GHz quad core processor (total 8 cores with hyper-threading), 8 GiB memory, and runs Linux 3.14.3. As shown in Figure 4.1, we used a battery-backed DRAM-SSD [16] as a fast storage device in the system [14, 43–45]. It has 512 GiB capacity in total (i.e., 64 GiB capacity per module * 8 DDR3 modules) and a PCIe interface. To increase capacity, a PCIe expansion card can be used to increase the number of PCIe slots, increasing the number of SSDs in a machine. The peak throughput is about 1.6 GiB/s for read and 1.4 GiB/s for write. The latency is 5 us and 7 us for reading and writing 4 KiB, respectively. To show the performance benefit from each optimization technique under different journaling modes, we evaluated the ordered (default) and data journaling modes of the EXT4 file system. We used the FIO benchmark [46] to measure the performance in terms of bandwidth for the two modes. To evaluate the optimized file system in realistic workloads, we used

Figure 4.1: The DRAM-based SSD used in this study



(a) Bandwidth - Sequential Read

(b) Bandwidth - Sequential Write

(c) Bandwidth - Random Read

(d) Bandwidth - Random Write

Figure 4.2: FIO benchmark results (ordered mode)

TPC-C benchmark for the ordered mode and filebench for the data journaling mode. We run each test five times and report the average and standard deviation. The standard deviations are always under 1% of the mean; graphs omit error bars.

4.1.1 Run-time Performance

Ordered mode

FIO benchmark results. We ran the FIO benchmark with diverse patterns, multiple request sizes, and buffered I/O under 8 threads (each thread creates a 3 GiB file) in terms of bandwidth, as shown in Figure 4.2. Overall, the performance is improved 35% on average compared to the existing I/O file system.

In the case of sequential read, Figure 4.2a shows that the optimized file system improves performance by 14% on average compared to the existing file systems. The performance gap is lower than those of other workloads, since the page cache hit rate is higher due to the read-ahead technique. In sizes that are less than 4 KiB, the performance of sequential read is highest among the I/O patterns since the prefetched pages increase the hit rate for sequential small block requests. However, existing file systems cannot fully utilize the I/O bandwidth, whereas the optimized file system reaches peak throughput (1.6 GiB/s).

In the case of write performance on the ordered mode, we note that the optimizations for journal I/O has little improvement on performance. Our optimization of journal I/O improves only about 1% on average compared to existing journal I/O since data-intensive workload such as the FIO benchmark generates small journal I/O for metadata. As shown in Figure 4.2b and 4.2d, the performance of the journal-off mode is almost the same as that of the ordered mode.

For sequential write as shown in Figure 4.2b, the optimized write-back achieves 38% better performance on average. In the case of small request sizes (i.e., 512 B and 1 KiB), the file systems perform read-modify-write operations since the request size does not match the page (block) size (i.e., 4 KiB). As the

read-modify-write generates unrequested I/Os, it wastes the bandwidth and largely decreases the I/O performance. The overall performance gains are increased compared to the case of sequential read. In this case, in addition to the hit rate of sequential write is lower than that of sequential read, the file system selects more scattered pages by choosing dirty pages all over the page cache; it increases the number of separate requests to storage. Eventually, since the optimized file system merges the requested pages irrespective of contiguity, it provides 1.4 GiB/s while the bandwidth of the existing file systems is saturated to 1 GiB/s when the request size is larger than 4 KiB.

For random read and write, Figure 4.2c and 4.2d show that the optimized file system improves the performance by average 39% and 40%, and up to 48% and 54%, respectively, compared to the existing file system. When the request size is less than 4 KiB during random read and write, the hit rate of small blocks is rapidly decreased compared to sequential read and write, which decreases the I/O performance. In the case of random write, more frequent read-modify-write operations are performed compared to sequential write and the performance of random write at small size is the lowest among the I/O patterns.

The performance gains of random workloads are higher than those of sequential workloads because the random workloads generate more multiple and separate requests. Although the performance is low in the cases of small random patterns (less than 4 KiB) compared to sequential patterns, the optimized file system shows full performance when the request size is larger than 4 KiB.

TPC-C results. To evaluate performance of the optimized file system in realistic workloads, we conducted TPC-C benchmark [47] with InnoDB [48]. We configured the experimental parameters as shown in Table 4.1 with other parameter sets as the default. In the default configuration, InnoDB provides atomicity of database page with redundant writes called double write buffer [49].

Parameters	Values
Page size (KiB)	4
DB buffer size (GiB)	6
Warehouse	500
Number of clients	8
Ramp-up time (seconds)	180
Measured time (seconds)	600

Table 4.1: Experimental parameters for InnoDB

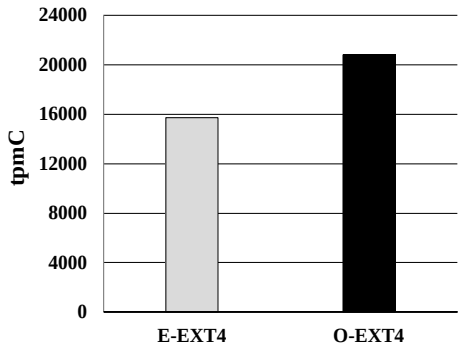
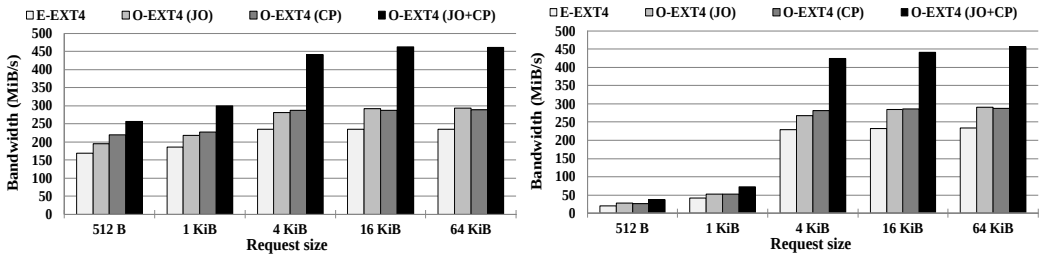


Figure 4.3: TPC-C results (ordered mode)

Therefore, the ordered mode is sufficient (i.e., metadata journaling) to provide crash consistency in the case of InnoDB. In [49], the authors showed that a smaller page size (4 KiB) leads to better transaction throughput instead of the default page size (16 KiB). Thus, we configured the page size as 4 KiB. In the TPC-C workload [50], the read:write ratio is kept at 1.9:1, and the pattern is random access.

Figure 4.3 shows the Transaction Per Minute type C (tpmC) for the existing and optimized file systems. As shown in figure, the optimized file system improves the performance by 32.3% compared to the existing file system. The optimized file system achieves up to 20807 tpmC. This result demonstrates that the existing file systems lag behind the optimized file system in the database



(a) Bandwidth - Sequential Write

(b) Bandwidth - Random Write

Figure 4.4: FIO benchmark results (data journaling)

workload. Similar to the FIO test in the ordered mode, our optimization of journal I/O little improves the performance (about 1%) since the TPC-C generates small metadata journal I/O compared to data I/O.

Data journaling mode

FIO benchmark results. To evaluate the performance for data journaling in the optimized file system, we conducted the FIO benchmark as shown in Figure 4.4 in the case of sequential and random write. We denote that JO and CP are journaling and checkpoint operations, respectively. The sequential and random read performance of the FIO benchmark is almost the same as those of the ordered mode. Since the read-only workload does not start the transaction, it does not generate any journal I/O. The performance of the existing data journaling mode is decreased by about 4x compared to that of the ordered mode in the case of 4 KiB due to the journaling operations with redundant data writes.

For the sequential write, in the existing file system, the performance is improved by 10%/26.2% from 512B/1KiB to 1KiB/4KiB respectively. However, the performance is saturated at about 235 MiB/s when the request size is larger than 4 KiB. The journal operation (JO), checkpoint (CP), and JO+CP in the

optimized file system I/O improve the performance by 15%/17%, 29.9%/48.9%, and 52.2%/61.1% in the case of 512B/1KiB respectively. In more than 4 KiB, the JO/CP/JO+CP in the optimized file system improves the performance by 19.7%/22.1%/88%, 23.9%/22.4%/96.9%, and 24.5%/22.9%/95.8% in the case of 4 KiB, 16 KiB, and 64 KiB respectively. The small size affects the bandwidth in the existing and optimized file systems since a request size of less than 4 KiB incurs read-modify-write operations. Accordingly, the performance improvement in the request sizes less than 4 KiB is less than that in the request sizes larger than 4 KiB. In terms of the performance improvement when larger than 4 KiB, JO and CP each show a relatively small improvement, but the optimized JO+CP noticeably improves the performance of the existing file system. As a result, it demonstrates that the optimization is necessary for both JO and CP.

The performance results for random write are almost the same as those of the sequential write workloads. However, in the case of less than 4 KiB, the performance improvement gap is much smaller. It is because the small random blocks are more read-modify-write operations than small sequential blocks. In the random workloads, optimized JO+CP operations improve the performance by 95% in the case of 64 KiB compared to the existing operations.

Filebench results. We evaluated the performance of the data journaling mode in existing and optimized file systems by using filebench [51]. In filebench, we used the fileserver workload that is write-intensive. In this workload, to provide crash consistency with data integrity, the data journaling mode is required because the file server does not provide atomic updates for data.

As shown in Table 4.2, we configured the I/O size as 16 KiB, the number of files as 5,000, the mean file size as 6 MiB, the number of clients as 64, and measured time as 600 s with other parameters sets as default. As shown in

Parameters	Values
I/O size (KiB)	16
The number of files	5,000
Meanfile size (MiB)	6
The number of clients	64
Measured time (seconds)	600

Table 4.2: Experimental parameters for fileserver

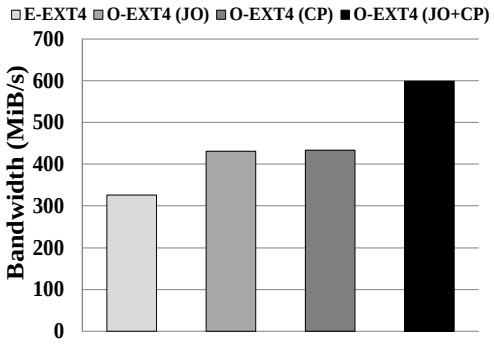
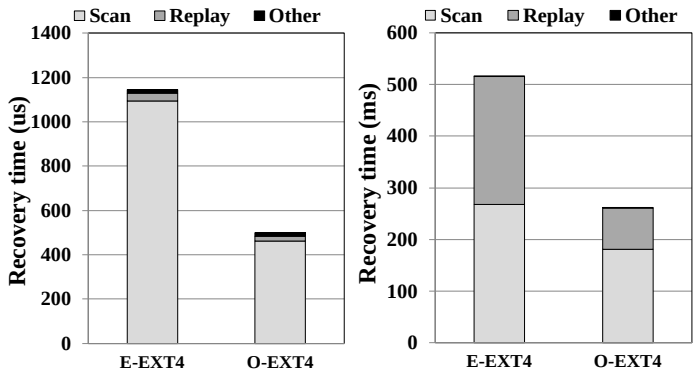


Figure 4.5: Fileserver results (data journaling)

Figure 4.5, the optimized JO, CP, and JO+CP operations in the optimized file system improve the performance by 32.1%, 33%, and 83.6% compared to those in the existing file system, respectively. We achieve high performance by up to about 600 MiB/s. The optimized file system provides higher performance than the existing file system while providing strong consistency.

4.1.2 Recovery Performance

To measure the recovery time in the existing and optimized file systems, we cut the power randomly while running the random write workload of the FIO benchmark in the ordered and data journaling modes. We conducted this evaluation more than 20 times, and the existing and optimized file systems were remounted correctly by scanning and replaying the blocks in the journal area.



(a) Ordered mode (b) Data journaling mode

Figure 4.6: Recovery performance

We compared the recovery performance when the number of replayed blocks was almost the same in the existing and optimized file systems. The recovery time occupied more than 92% of the total remount time while the scan and replay operations occupied a great part of the total recovery time in the existing and optimized file systems.

As shown in Figure 4.6, the optimized file system improves the recovery performance by about 2.28x/1.97x compared to the existing file system in the case of the ordered and data journaling modes, respectively. In this evaluation, the total replayed blocks in the existing/optimized file system are 3/3 and 23615/24164 in the ordered and data journaling mode, respectively.

In the case of the ordered mode as shown in Figure 4.6a, the scan and replay operations of existing/optimized file systems take about 1093us/461us and 37us/22us, respectively. In the case of data journaling mode as shown in Figure 4.6b, the scan and replay operations of the existing/optimized file systems take about 268ms/181ms and 248ms/80ms, respectively. According to this result, the optimized file system improves the scan and replay operations by making the blocks into a single request. Consequently, our scheme can also

Benchmark	FIO					
I/O types	SR	SW	RR	RW	JO	CP
Existing file system	15 (0.6)	14 (0.6)	2.5 (0.2)	1.5 (0.2)	1 (0)	0 (0)
Optimized file system	63 (0.6)	128 (0.04)	8 (0.14)	128 (0.04)	3.1 (0.1)	0 (0)

(a) FIO

Benchmark	TPC-C			
I/O types	Read	Write	JO	CP
Existing file system	1 (0.04)	2 (0.06)	1 (0)	1 (0)
Optimized file system	5 (0.2)	43 (0.6)	2 (0.1)	2.7 (0.4)

(b) TPC-C

Table 4.3: The average page counts in a single request in the ordered mode (SR: Sequential Read, SW: Sequential Write, RR: Random Read, RW: Random Write, JO: Journal operation, CP: Checkpoint, the numbers in parentheses are standard deviations)

Benchmarks	FIO				Fileserver	
I/O types	SW		RW		Random I/O	
	JO	CP	JO	CP	JO	CP
Existing file system	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
Optimized file system	338 (0.7)	31.6 (0.6)	337.9 (0.1)	48.3 (0.5)	332.6 (0.9)	61.2 (0.4)

Table 4.4: The average page counts in a single request in the data journaling mode (JO: Journal operation, CP: Checkpoint, SW: Sequential Write, RW: Random Write, the numbers in parentheses are standard deviations)

be applied to the recovery procedure to provide faster remount time.

4.1.3 Experimental Analysis

We analyzed the main factor of the performance improvement in the optimized file system. We measured the page counts per request with I/O operations. Table 4.3 shows the average page counts in a single request of the existing and optimized file systems. The unit of request size was 4 KiB for the EXT4 file system, and the counts were measured at the device driver level. Our technique increases the average page counts in a single request by 4.2x, 9.14x, 3.2x, and 85.3x in the case of SR, SW, RR, and RW under FIO, respectively. The results show that the number of write operations is higher than that of the read op-

Modes	Ordered		Data journaling	
Operations	Scan	Replay	Scan	Replay
Existing file system	1 (0)	1 (0)	1 (0)	1 (0)
Optimized file system	32 (0.08)	3 (0.4)	32 (0.08)	1024 (0.04)

Table 4.5: The average page counts in a single request during recovery (the numbers in parentheses are standard deviations)

erations. The reason is that write-back operations occur when the dirty rate is higher than the threshold, which accumulates the dirty pages.

Although the average page counts of the sequential read are higher than those of the random read, the performance gap of sequential read is lower than that of the random read operations. The reason is that the sequential read patterns provide a very high hit rate. In the case of TPC-C, the existing file systems cannot merge the pages into a large request since the patterns are almost random. Meanwhile, the optimized file system increases the average page counts by 5x and 21.5x for read and write, respectively.

We note that the optimized journal I/O in the ordered mode does not affect the performance of our fast storage device and data-intensive workload. The performance of journal-off is almost the same as that of the ordered mode. As shown in Table 4.3, in the ordered mode, only small journal blocks are infrequently written to the journal area.

In contrast, our optimization in the data journaling mode improves the performance since both data and metadata generate journaling and checkpoint. As shown in Table 4.4, in the case of the FIO benchmark with the data journaling mode, JO/CP in the optimized file system merges the 338/337.9 and 31.6/48.3 requests at once for the sequential and random write workloads, respectively. Thus, the average page counts are larger by 338x/337.9x and 31.6x/48.3x than those of existing file system. In the case of the fileserver workload, the average

counts are larger by 332.6x and 61.2x compared to those of the existing file system for JO and CP, respectively. In the FIO and fileserver workloads, the optimized file system significantly increases the page counts in a single request since each page is processed by only one request in the existing file system.

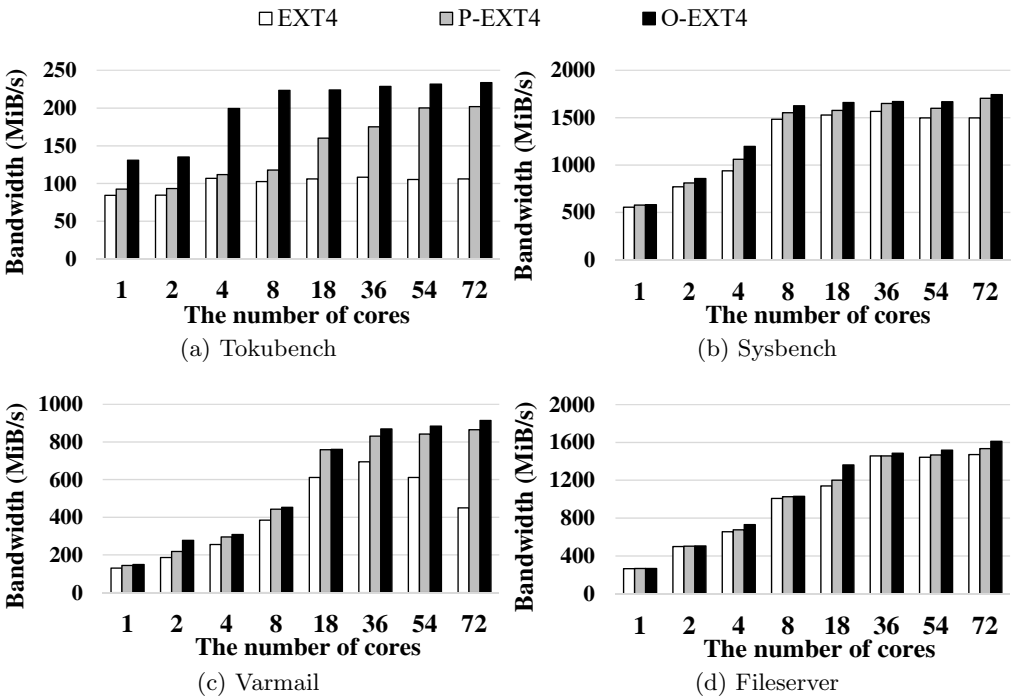
As shown in Table 4.5, in the analysis of the recovery performance, the existing file system processes the requests for scan and replay operations as several requests one by one. In contrast, the optimized file system makes 32/32 and 3/1024 blocks in the case of ordered/data journaling modes for scan and replay operations into a single request, respectively. This result shows that the optimized file system increases the average number of pages in a single request by 32x/32x and 3x/1024x in case of the ordered/data journaling modes for scan and replay operations compared to the existing file system, respectively. Consequently, the I/O performance is improved, and the recovery time is reduced by a large request.

4.2 Evaluating the Optimized File System for Highly Parallel Storage

We perform all of the experiments on a 72-core machine with four Intel Xeon E7-8870 processors (without hyperthreading), 16 GiB DRAM, and PCI 3.0 interface. For storage, the machine has an 800 GiB Intel P3700 NVMe SSD [17], which has 18 channels. The machine runs Ubuntu 16.04.1 LTS distribution with a Linux kernel 4.9.1. We evaluate the existing EXT4 and fully optimized EXT4 (O-EXT4) file systems in the ordered (default) and data journaling modes. To present a performance breakdown, we also evaluate an optimized EXT4 with our parallel I/O (P-EXT4), which performs our parallel I/O for journaling and checkpointing without `j_checkpoint_mutex` but updates the data structures using `j_list_lock`. We run metadata and data-intensive workloads, such as

Benchmarks	Descriptions	Parameters
Tokubench	Metadata-intensive (file creation)	Files: 30,000,000, I/O sizes: 4KiB
Sysbench	Data-intensive (random write)	Files: 72, Each file size: 1GiB, I/O sizes: 4KiB
Varmail	Metadata-intensive	Files: 300,000, Directory width: 10,000
Fileserver	Data-intensive	Files: 1,000,000, Directory width: 10,000

Table 4.6: Experimental parameters



(a) Tokubench (b) Sysbench

(c) Varmail (d) Fileserver

Figure 4.7: Ordered mode

tokubench [52], sysbench [53] and filebench [51] with the parameters shown in Table 4.6. We vary the number of cores from 1 to 72, and the number of threads is equal to that of the cores. We run each test ten times and report the average.

4.2.1 Run-time Performance

Ordered mode

We present the performance results in the ordered mode as shown in Figure 4.7. In the case of tokubench as shown in Figure 4.7a, the performance growth of EXT4 is not noticeable as the number of cores increases. P-EXT4 improves the performance by 1.9x compared to EXT4. However, compared to full optimization, this result shows the limitation of our parallel I/O scheme, which does not handle the lock contention. Through full optimization, O-EXT4 improves the performance by 2.2x at 72 cores compared to EXT4. Meanwhile, the performance of O-EXT4 is almost the same beyond 18 cores since the bandwidth is saturated due to the limited write bandwidth and the channels of the SSD. In the case of sysbench as shown in Figure 4.7b, P-EXT4 and O-EXT4 improve the performance by 13.8% and 16.3%, respectively, compared to EXT4 at 72 cores. The performance improvement is lower than that of tokubench since sysbench as a data-intensive workload generates far fewer journal I/Os for metadata.

Under the varmail workload as shown in Figure 4.7c, P-EXT4 and O-EXT4 scale well compared to the case of tokubench and outperform EXT4 by 1.92x and 2.03x at 72 cores, respectively. O-EXT4 achieves up to 914.3 MiB/s. Since the workload generates a mixture of read/write operations unlike tokubench, the available bandwidth increases, and therefore, the performance gradually scales at all cores. Meanwhile, the performance of EXT4 decreases beyond 54 cores due to the lock contention. Under the fileserver workload as shown in Figure 4.7d, P-EXT4 and O-EXT4 outperform EXT4 by 4.3% and 9.6% at 72 cores, respectively. All the file systems scale in a similar trend at each core, and the performance gap is not noticeable. The reason is that, similar to the case of sysbench, the fileserver workload is data-intensive, which generates a low

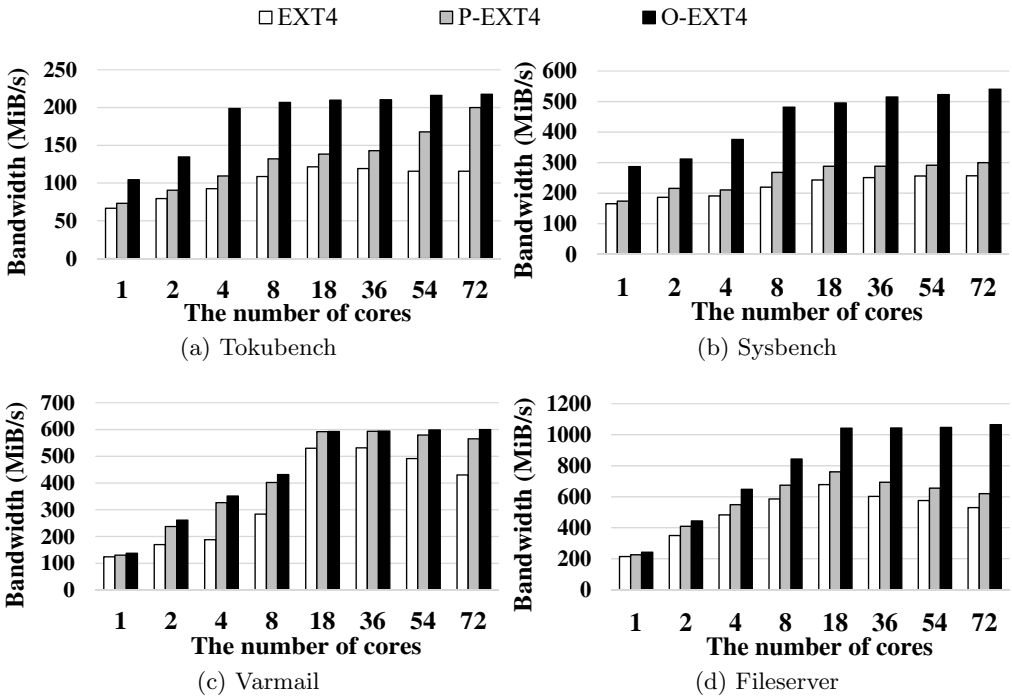


Figure 4.8: Data journaling mode

number of metadata I/Os. Consequently, our optimized file system improves the performance in the ordered mode by reducing the lock contention and parallelize the I/O operations especially for metadata-intensive workloads.

Data journaling mode

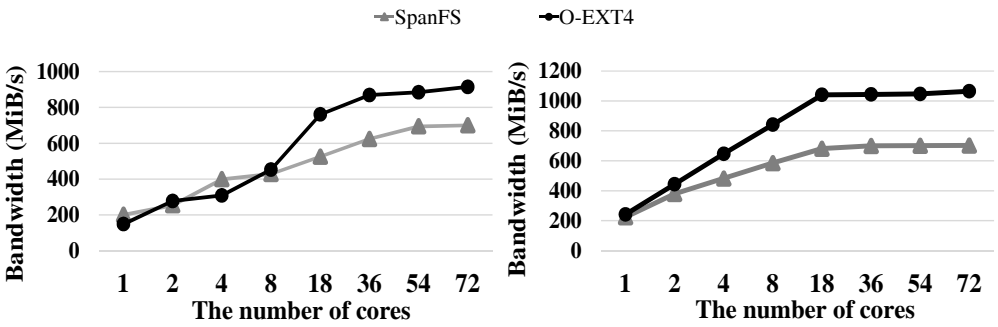
We present the performance results in the data journaling mode as shown in Figure 4.8. In the case of tokubench as shown in Figure 4.8a, P-EXT4 and O-EXT4 outperform EXT4 by 73% and 88.2% at 72 cores, respectively. The results show that the overall aspect of the performance is similar to that in the ordered mode. In the case of sysbench as shown in Figure 4.8b, P-EXT4 and O-EXT4 show 1.17x and 2.1x faster performance than EXT4 at 72 cores, respectively. The performance improvement is higher than that in the ordered

mode since the workload generates many journal I/Os for data. Also, the results show that the improvement by our parallel I/O scheme is low due to the list lock contention.

Under the varmail workload as shown in Figure 4.8c, P-EXT4 and O-EXT4 outperform EXT4 by 31.3% and 39.3% at 72 cores, respectively. Unlike the case of the ordered mode, the performance is saturated and sustained beyond 18 cores since writing both the metadata and the data makes the performance reach the full bandwidth faster. Meanwhile, the performance of EXT4 decreases due to the lock contention. In the case of fileserver as shown in Figure 4.8d, P-EXT4 and O-EXT4 outperform EXT4 by 1.45x and 2.01x at 72 cores, respectively. O-EXT4 achieves up to 1064.6 MiB/s. The performance of P-EXT4 and E-EXT4 decreases beyond 36 cores, which demonstrates the need for both concurrent updates on data structures and parallel I/O. Meanwhile, O-EXT4 scales well to 18 cores and increases the performance until 72 cores. Beyond 36 cores, the rate of bandwidth growth is reduced due to the bandwidth limit of the SSD. Consequently, our optimized file system achieves higher performance in the data journaling mode, and the benefit becomes larger in data-intensive workloads.

Comparison with a scalable file system

We compare our optimized file system with SpanFS [15], a scalable file system. We use the varmail and fileserver workloads in the ordered and data journaling modes, respectively. We set the number of domains in SpanFS as same as that of the cores. As shown in Figure 4.9, both file systems scale well until the performance is saturated in both workloads. Meanwhile, O-EXT4 generally shows better performance and improves performance by up to 1.45x and 1.51x in the varmail and fileserver workloads, respectively, compared to SpanFS. Es-



(a) Varmail (b) Fileserver

Figure 4.9: Comparison with SpanFS

pecially, in the case of the varmail workload, the performance of O-EXT4 is similar or slower than that of SpanFS at small number of cores while O-EXT4 shows better performance than SpanFS as the number of cores increases. The results show that our scheme can deliver better performance than the scheme that distributes file services.

4.2.2 Recovery Performance

In EXT4/JBD2, a single-threaded process (i.e., mount process) performs recovery operations which can underutilize both multi-cores and I/O parallelism. To increase the parallelism, similar to our journal and checkpoint I/O schemes, we perform scan and replay I/O operations in parallel by creating multiple threads without any additional locking. To evaluate the performance and test the correctness of recovery, we used tokubench and fileserver workloads in the ordered and data journaling modes, respectively. While running the benchmarks, we randomly cut the power of the machine, and both existing and optimized file systems are recovered to a consistent state after more than 30 crashes.

Table 4.7 shows the recovery performance of the ordered and data journaling modes in the file systems. The scan and replay operations occupy the main part

Modes	Ordered			Data journaling		
Operations	scan	replay	other	scan	replay	other
EXT4	331 ms	62 ms	7 ms	311 ms	81 ms	5 ms
O-EXT4	125 ms	34 ms	9 ms	117 ms	37 ms	4 ms

Table 4.7: Recovery performance

of the total recovery time in all cases. Through parallelizing scan and replay I/O operations, O-EXT4 improves the recovery performance by 2.38x and 2.51x compared to EXT4 in the ordered and data journaling modes, respectively. This result demonstrates that our schemes can also be applied to the recovery procedure to provide faster recovery time.

4.2.3 Experimental Analysis

Table 4.8 shows the total execution time for main locks and device-level bandwidth at 72 cores in the case of the sysbench workload in the data journaling mode. For this experiment, we measured the execution time by using a time function (`getrawmonotonic()`) for lower overhead and more correctness. As shown in the table, in EXT4, the execution time of the checkpoint mutex and list locks take a large portion of the total write time. In P-EXT4, the bandwidth increases by 16.3%, and the write time decreases by 15.7% compared to EXT4, respectively. As the total write time decreases, the time of the list and state locks decreases while the list lock still takes up 10.8% of the total write time. This demonstrates that the list lock contention can be a performance bottleneck in our parallel I/O scheme. In O-EXT4, the bandwidth increases by 2.06x, and the write time decreases by 2.08x compared to EXT4. This is achieved by removing the list lock contention via our concurrent update scheme. Meanwhile, the contention on the state lock increases due to the removal of the list lock but the portion is still small. Consequently, this result demonstrates that O-EXT4

File systems	EXT4	P-EXT4	O-EXT4
Device-level bandwidth	692 MiB/s	805 MiB/s	1426 MiB/s
Write time	52220 s (100%)	45124 s (100%)	25078 s (100%)
`j_checkpoint_mutex`	17946 s (34.4%)	0	0
`j_list_lock`	6132 s (11.7%)	4890 s (10.8%)	0
`j_state_lock`	102 s (0.2%)	87 s (0.2%)	182 s (0.7%)
Others	28040 s (53.7%)	40147 s (89%)	24896 s (99.3%)

Table 4.8: Device-level bandwidth and total execution time of main locks and write operations

achieves high-performance transaction processing by enabling both concurrent updates and parallel I/O.

Chapter 5

Related Work

5.1 Analysis and Evaluation of High-Performance storage

eNVy [54] presents a non-volatile main memory storage system built with flash memory. It uses a special controller equipped with a battery-backed SRAM buffer to hide the block-addressable nature of flash. The Rio file cache [55] uses a battery-backed main memory to make writes persistent. It can survive OS crashes and be as safe and permanent as disk. It achieves the performance of main memory with the reliability of disk by eliminating all reliability-induced writes to disk.

Kim et al. [6] explore the opportunities for PCM technology within enterprise storage systems. They present the results of a performance study of an all-PCM SSD prototype. They compare the PCM SSD prototype to an eMLC flash SSD to understand the performance characteristics of the PCM SSD as another storage tier. They state that the IOPS/$ of a tiered storage system can be improved by adding PCM. Vucinic et al. [23] explore the limits of communi-

cation latency with a PCM-based storage device over PCI express. They devise dubbed DC Express, which is a communication protocol. This protocol eliminates unnecessary packet exchanges and avoids context and mode switching. Our study is in line with such studies [54,55] in terms of using a battery-backed memory and such studies [6,23] in terms of exploring PCIe based SSDs. Meanwhile, we focus on the OS-level I/O stack including file systems.

Researchers have recently performed empirical studies of file systems and application performance on NVM [56–58] and concentrated efforts to improve the performance of fast storage devices. Sehgal et al. [56] evaluate the performance of various traditional Linux file systems under various real-world workloads on NVM simulated using ramdisk and compare it against an NVM optimized file system like PMFS [10]. They demonstrate that the traditional file systems can be tuned to perform better than their default settings on NVM with a performance comparable to that PMFS. Zhang et al. [57] provide an analysis of storage application performance with NVM. Their evaluation results show that NVM improves the storage application performance significantly compared to flash-based SSDs and HDDs. They present that even if NVM has somewhat higher latency and lower bandwidth than DRAM, this difference has a modest impact on application performance. Lee et al. [58] explore the use of NVM storage from the operating system (OS) perspective. They investigate the effectiveness of current I/O mechanisms and the efficient use of NVM storage.

5.2 Study of Journaling File Systems

There are some studies on the journaling file systems. Lu et al. [59] conduct a comprehensive study of Linux file system code evolution. They mention that open-source local file systems (e.g., EXT4) are a critical component of modern storage since many recent distributed file systems (e.g., Google GFS, Hadoop

DFS, etc) replicate data objects across local file systems. They analyze eight years of Linux file system changes and derive numerous new insights into the file system development process. Prabhakaran et al. [37] provide a detailed analysis of journaling file systems by using semantic block analysis (SBA) and semantic trace playback (STP). SBA enables users to understand the internal behavior and policies of the file system, and STP allows users to quantify how changing the file system will impact the performance of real workloads.

DualFS [60] is a new high-performance journaling file system, which provides the same consistency guarantees as existing file systems but with better performance. Different from our approach, DualFS places data and metadata in different devices and manages them in different ways. For example, DualFS organizes the metadata device as a log-structured file system and the data device as a group respectively. Kang et al. [15] propose a scalable file system on fast storage devices. In contrast to our goal, they focus on the lock contention as a bottleneck in the journaling file system. To reduce lock contention, they distribute files and directories among the domains, which consist of a collection of micro-file system services, and provide a global file system view on top of the domains, and maintain consistency in case of system crashes.

Lee et al. [61] present a novel buffer cache architecture that subsumes the functionality of caching and journaling. They reduce the I/O traffic of journaling using non-volatile memory. In contrast, we focus on PCIe-based storage as primary storage and provide more efficient I/O operations between the file system and the lower layer. OptFS [30] proposes decoupled ordering and durability primitives such as osync() and dsync() in HDD-based storage to reduce the frequent flush commands from the journaling file system. They trade freshness for performance while maintaining crash consistency. Unlike our scheme, their target is HDD-based storage that includes a volatile cache.

5.3 File and I/O System Optimizations for Low-latency Storage

There are a number of file system designs and suggested optimizations [9,10,12, 62] for SCM. SCMFS [12] is a new file system designed for storage class memory. This system utilizes the existing memory management module in the operating system to achieve high performance by keeping contiguous space for each file in the logical address space. BPFS [9] is optimized for small random writes by fine-grained updates instead of transferring data in bulk, which leads to unnecessary traffic over the memory bus for NVRAM. PMFS [10] is a persistent memory (PM) load/store accessible file system similar to SCMFS and BPFS. PMFS exploits byte-addressability of persistent memory in order to avoid the overhead of block-oriented storage and to enable direct PM access by applications. NOVA [62] is a log-structured file system (LFS) designed for hybrid memory systems. By extending LFS ideas to leverage NVM, NOVA proposes fast and efficient garbage collection, quick recovery, and strong consistency guarantees compared to conventional file operations and mmap-based load/store accesses. These schemes [9, 10, 12, 62] are optimized for persistent memory attached to the processor's memory bus. However, they are not appropriate for PCIe-based SSDs.

NV-Heaps [63] provides user-level transactional updates to persistent data and proposes a programming model to simplify PM programming. NV-Heaps forces the programmer to employ a specific object framework and requires modifications to the processor. Mnemosyne [64] also proposes a transaction mechanism to update data in NVM. It supports direct access and reduces latency by bypassing many software layers. These studies [63,64] provide fast mechanisms for object persistency, but they do not replace files or file systems.

In addition, various studies on optimizing I/O stack for PCIe-based SSDs

have been conducted. Seppanen et al. [11] state that an I/O scheduler in a traditional block I/O subsystem serializes and merges requests. It is efficient for HDDs since HDDs have seek overhead and lack parallelism. When storage devices such as flash-based SSDs are used, the system with the conventional block I/O subsystem cannot fully exploit the parallelism in the SSDs since the I/O scheduler processes the requests in a serialized and batched manner. Thus, the authors propose a new Linux block I/O subsystem without SCSI/ATA layers and an I/O scheduler to reduce latency and exploit parallelism of storage devices. However, the proposed block I/O subsystem still processes requests asynchronously; interrupts are used to communicate with storage devices.

Yang et al. [13] show that a synchronous I/O (polling) between the host and the storage device delivers better performance than an asynchronous I/O (interrupt) when the device has ultra-low latency. Since the system with interrupt-based I/O completes all I/Os asynchronously and it causes the interrupt handing overheads, the polling mechanism is more appropriated for storage devices with ultra-low latency. Yu et al. [14] improve the I/O bandwidth by combining multiple block requests among multiple threads into one I/O request and dispatching the request to storage. These studies [11,13,14] improve the performance by focusing on the optimization of the block I/O subsystem. Meanwhile, our study focuses on the I/O operations in the file system on top of the optimized block I/O subsystem including these techniques [11, 13, 14].

Moneta [65] is an architecture for a PCIe-attached storage array built from emulated PCM storage. This architecture uses a series of hardware/software (device driver) optimizations that improve its performance for next-generation NVM such as PCM. Moneta-D [66] is an extended version of Moneta that removes software overhead by using a user-level driver, which bypasses the kernel and file systems. Due to the user-level driver, Moneta-D requires additional

hardware functionalities for security and authority. In contrast to these studies [65, 66], we focus on optimization of the software stack and file systems. Our scheme is implemented at the kernel level and does not require additional support to avoid security concerns.

pVM [67] is a system software abstraction that provides applications with automatic OS-level memory capacity scaling, flexible memory placement policies across NVM, and fast object storage. It extends the OS virtual memory and abstracts NVM as a NUMA node with support for NVM-based memory placement mechanisms. This article is similar to pVM in terms of exploring NVM in the OS. Meanwhile, we focus on the existing I/O stack and file systems instead of the virtual memory system.

Several works have researched how to optimize software stacks for fast network access. IX [68] proposes a dataplane operating system by using hardware virtualization to separate the management and scheduling functions of the kernel from network processing. IX optimizes both bandwidth and latency by processing batches of packets to completion and eliminating synchronization on multi-core servers. Arrakis [69] presents an operating system that splits the traditional role of the kernel. Applications have direct access to virtualized I/O devices by allowing most I/O operations to skip the kernel, while the kernel is re-engineered to provide network and disk protection without kernel mediation of every operation. Similar to IX, Arrakis uses hardware virtualization to separate the I/O dataplane from the control plane. Both IX [68] and Arrakis [69] provide optimized networking stack by reducing the overhead of the operating systems. This article is in line with these schemes [68, 69] in terms of reducing the software overhead. Unlike these schemes, we focus on the storage stack for fast storage devices.

5.4 Study of Scalability in Operating Systems

Hive [70] is an operating system designed for large scale shared-memory multiprocessors. It is structured as an internal distributed system of independent kernels called cells to improve reliability and scalability. Cerberus [71] mitigates contention on many shared data structures within OS kernels by clustering multiple commodity operating systems atop a virtual machine monitor. Baumann et al. [72] investigate a new OS structure, the multikernel. To solve scalability problems for OSs, they structure the OS as a distributed system of cores that communicate using messages and share no memory. Corey [73] is an exokernel based operating system that follows a principle, which allows applications to control the sharing of kernel resources. Its abstractions ensure that each kernel data structure is used by only one core by default, while giving applications the ability to specify when sharing of kernel data is necessary.

Boyd-Wickizer et al. [74] analyze the scalability of seven system applications running on Linux. They find that all applications trigger scalability bottlenecks inside a Linux kernel. RadixVM [75] presents a scalable virtual memory address space for non-overlapping operations. It avoids cache line contention using three techniques, which are radix trees, Refcache, and targeted TLB shootdowns. Our study is inspired by these works [70–75] and in line with them in terms of investigating the scalability of OS kernels on multi-cores. In contrast, we focus on the transaction processing in file systems on high-performance storage.

5.5 File and I/O System Optimizations for Highly Parallel Storage

Zheng et al. [76] present a storage system for arrays of commodity SSDs. They create dedicated I/O threads for each SSD and deploy a set-associative parallel page cache, which divides the global page cache into small and independent sets

to reduce lock contention. MultiLanes [18] is a virtualized storage system for OS-level virtualization on many cores. It builds an isolated I/O stack on top of a virtualized storage device to eliminate contention on shared kernel data structures and locks. Bjørling et al. [77] propose a new design for I/O management in the block layer. They address the scalability of the Linux block layer and propose a new Linux block layer, which maintains a per-core request queue. They design multiple I/O submission/completion queues to minimize cache coherence across CPU cores. Jericho [78] is a new I/O stack that improves affinity between threads, and buffers in the storage I/O path for NUMA multicore systems. Jericho consists of a NUMA aware file system and a DRAM cache organized in slices mapped to NUMA nodes. Our study is in line with these works [18, 76–78] in terms of mitigating the contention on shared resources. In contrast, we focus on updating the data structures concurrently in a lock-free manner in journaling file systems.

ScaleFS [79] extends a scalable in-memory file system to support consistency on an on-disk file system by using per core operation logs. IceFS [80] partitions the on-disk resources among a new container abstraction called cubes to provide isolated I/O stacks for localized reaction to faults, fast recovery, and concurrent file system updates. Thus, data and I/O within each cube are disentangled from the data and I/O outside of it. SpanFS [15] is a scalable file system that consists of a collection of micro file system services called domains. It distributes the files and directories among the domains and provides a global file system view on top of the domains to maintain consistency. Each domain performs its file system service, such as data allocation and journaling, independently. Curtis-Maury et al. [81] present a data partitioning mode to parallelize the majority of file system operations. They also provide a fine-grained lock-based multiprocessor model for incremental advances in parallelism.

Min et al. [82] analyze the many-core scalability of five file systems by using their open source benchmark suite (i.e., FxMark). They observe that file systems are hidden scalability bottlenecks in many I/O-intensive applications. iJournaling [36] improves the performance of an `fsync()` call. It journals only the corresponding file-level transaction to the ijournal area for an fsync call while exploiting the advantage of the compound transaction scheme. iJournaling also handles multiple fsync calls simultaneously by allowing each core to have its own ijournal area to improve the scalability. Our study is in line with these approaches [15,36,79–82] in terms of investigating the scalability and parallelism of the file systems. In contrast, we enable concurrent updates on data structures in a lock-free manner and parallelize I/O operations cooperatively in transaction processing by focusing its internal operations.

Chapter 6

Conculsion

6.1 Summary

High-performance storage devices such as solid-state drives (SSDs) are becoming one of attractive storage solutions for various computer systems. According to development of the storage devices, optimizing the file systems is essential in order to fully exploit their features. As our observations, the existing I/O operations and locking in file systems can be performance bottlenecks on high-performance SSDs.

This dissertation proposes two key optimizations, 1) efficient I/O strategies for low-latency SSDs, which transfers requests from discontiguous host memory buffers to discontiguous storage segments in a single I/O request, and 2) concurrent updates on data structures and parallel I/O operations for highly parallel SSDs. Experiments show that our optimized file system achieves higher performance than the existing file system.

6.2 Future work

In the future work, we will extend our techniques and the scope of I/O optimizations for high-performance storage devices.

Extending I/O optimizations for low-latency storage devices. Our techniques for low-latency storage devices are limited to our customized DRAM-SSD. However, our techniques can be applied to other storage protocols or devices as well as other file systems. For example, current NVMe protocol transfers data only from discontiguous host memory buffers to contiguous storage segments in one I/O request. We can add the feature, which transfers data from discontiguous host memory buffers to discontiguous storage segments in one I/O request, to the NVMe protocol. By doing so, we standardize our technique by adding the new feature to the NVMe protocol and also our optimized file system can be used for low-latency storage devices with the NVMe protocol.

Also, we will perform a holistic end-to-end I/O stack or cross-layer optimizations for the low-latency storage devices. For example, we can broaden the scope of our optimizations to cover the whole local file system layers (e.g., VFS, block layer, and device driver), distributed file systems, user applications (e.g., database systems), and network layers. In the existing storage system, there are many layers, which can generate a performance bottleneck in the low-latency storage devices. Thus, we first find out the performance bottleneck by measuring the latency for each layer. And then, we will merge the redundant operations between layers and minimize the whole I/O path to maximize the performance.

Extending I/O optimizations for highly parallel storage devices. In this paper, our techniques for highly parallel storage devices are limited to the locking for transaction processing in EXT4/JBD2. We can extend the

techniques to the locks for other shared resources in the file systems such as file, page cache, etc. For example, EXT4 uses a coarse-grained lock (mutex) per file. This locking ensures correct updates on the file, and thus the file consistency is preserved. A previous study [82] shows the overhead from the file locking such as an inode mutex. When applications are accessing a shared file, such file locking become the bottleneck. Thus, we will extend our optimization techniques to the file locking mechanism to update the file updates concurrently.

For another example, the Linux kernel adopts a page cache organized as an address space radix tree to cache recently accessed blocks for better I/O performance. The OS uses a read-copy-update (RCU) lock to protect correct updates of the radix tree [15,76,83]. Previous studies [76,83] show the page cache locking overhead and reduce the overhead by using a set-associative parallel page cache which divides the global page cache into small and independent sets to reduce lock contention. SpanFS [15] leverages the Linux OS block device architecture to provide a dedicated buffer cache address space for each domain to avoid lock contention. For more efficiency, we will extend our technique using a lock-free data structure to the page cache.

For different storage configuration, we will consider the performance in multiple storage devices on RAID. In RAID, the I/O operations are performed for each device in parallel. However, we may rethink the RAID performance on many cores with a number of highly parallel storage devices considering the scalability and parallelism. Finally, after we solve the performance bottleneck in the local file and storage systems, we will extend the optimizations to other systems such as distributed file systems and database systems. Consequently, we will consider the whole layers in a small or large system to maximize the performance from high-performance hardware.

Bibliography

[1] E. Grochowski and R. F. Hoyt, "Future trends in hard disk drives," *IEEE Transactions on Magnetics*, vol. 32, no. 3, pp. 1850–1854, 1996.

[2] A. Al Mamun, G. Guo, and C. Bi, *Hard disk drive: mechatronics and control*, vol. 23. CRC press, 2006.

[3] B. L. Worthington, G. R. Ganger, and Y. N. Patt, "Scheduling algorithms for modern disk drives," *ACM SIGMETRICS Performance Evaluation Review*, vol. 22, 1994.

[4] Y. J. Yu, D. I. Shin, H. Eom, and H. Y. Yeom, "Ncq vs. i/o scheduler: Preventing unexpected misbehaviors," *ACM Transactions on Storage (TOS)*, vol. 6, no. 1, p. 2, 2010.

[5] R. R. Katti, H. L. Stadler, and J.-C. Wu, "Non-volatile magnetic random access memory," 1994. US Patent 5,289,410.

[6] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pp. 33–45, 2014.

[7] J.-D. Lee, S.-H. Hur, and J.-D. Choi, "Effects of floating-gate interference on nand flash memory cell operation," *IEEE Electron Device Letters*, vol. 23, no. 5, pp. 264–266, 2002.

[8] L. M. Grupp, J. D. Davis, and S. Swanson, "The bleak future of nand flash memory," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pp. 2–2, USENIX Association, 2012.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[10] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[11] E. Seppanen, M. O'Keefe, and D. Lilja, "High performance solid state storage under linux," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010.

[12] X. Wu and A. L. N. Reddy, "Scmfs: A file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, 2011.

[13] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, 2012.

[14] Y. J. Yu, D. I. Shin, W. Shin, N. Y. Song, J. W. Choi, H. S. Kim, H. Eom, and H. Y. Yeom, "Optimizing the block i/o subsystem for fast storage devices," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 6:1–6:48, 2014.

[15] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai, "Spanfs: A scalable file system on fast storage devices," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, (Santa Clara, CA), pp. 249–261, USENIX Association, July 2015.

[16] TAILWINDSTORAGE, "Extreme s3804." `http://www.taejin.co.kr`, 2014.

[17] Intel Solid State Drive DC P3700 Series. `http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-p3700-spec.pdf`, 2015.

[18] J. Kang, C. Hu, T. Wo, Y. Zhai, B. Zhang, and J. Huai, "Multilanes: Providing virtualized storage for os-level virtualization on manycores," *Trans. Storage*, vol. 12, pp. 12:1–12:31, June 2016.

[19] NVM express. `http://www.nvmexpress.org`, 2012.

[20] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L. Lung, and C. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, 2008.

[21] B. Dieny, R. Sousa, G. Prenat, and U. Ebels, "Spin-dependent phenomena and their implementation in spintronic devices," in *VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on*, 2008.

[22] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson, "From aries to mars: Transaction support for next-generation, solid-state drives," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[23] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. Le Moal, T. Bunker, J. Xu, S. Swanson, *et al.*, "DC express: shortest latency protocol for reading phase change memory over PCI express," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pp. 309–315, 2014.

[24] Y. Son, H. Kang, H. Han, and H. Y. Yeom, "An empirical evaluation and analysis of the performance of nvm express solid state drive," *Cluster Computing*, pp. 1–13, 2016.

[25] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier, and B. S. A. S, "A and viver, l. the new ext4 filesystem: current status and future plans," in *In Ottawa Linux Symposium. http://ols.108.redhat.com/2007/ Reprints/mathur-Reprint.pdf*, 2007.

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

[27] JFS for Linux. `http://oss.software/ibm.com/jfs`, 2002.

[28] H. Reiser, "Reiserfs," 2004.

[29] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 1992.

[30] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Optimistic crash consistency," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 228–243, ACM, 2013.

[31] 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.

[32] J. Bonwick and B. Moore, "ZFS: The last word in file systems," 2007.

[33] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[34] D. Kim, J. Park, K.-g. Lee, and S. Lee, *Forensic Analysis of Android Phone Using Ext4 File System Journal Log*, pp. 435–446. Dordrecht: Springer Netherlands, 2012.

[35] A. C. Arpaci-Dusseau, "Model-based failure analysis of journaling file systems," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, (Washington, DC, USA), pp. 802–811, IEEE Computer Society, 2005.

[36] D. Park and D. Shin, "ijournaling: Fine-grained journaling for improving the latency of fsync system call," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 787–798, USENIX Association, 2017.

[37] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *Proceedings of the Annual*

Conference on USENIX Annual Technical Conference, ATEC '05, (Berkeley, CA, USA), pp. 8–8, USENIX Association, 2005.

[38] S. C. Tweedie, "Journaling the linux ext2fs filesystem," in *The Fourth Annual Linux Expo*, 1998.

[39] A. Hatzieleftheriou and S. V. Anastasiadis, "Improving bandwidth efficiency for consistent multistream storage," *Trans. Storage*, vol. 9, pp. 2:1–2:27, Mar. 2013.

[40] K. Apt, F. S. De Boer, and E.-R. Olderog, *Verification of sequential and concurrent programs*. Springer Science & Business Media, 2010.

[41] J. Östlund and T. Wrigstad, "Multiple aggregate entry points for ownership types," *ECOOP 2012–Object-Oriented Programming*, pp. 156–180, 2012.

[42] R. M. Stallman and G. DeveloperCommunity, *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Paramount, CA: CreateSpace, 2009.

[43] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient memory-mapped i/o on fast storage device," *Trans. Storage*, vol. 12, pp. 19:1–19:27, May 2016.

[44] Y. Son, N. Song, H. Han, H. Eom, and H. Yeom, "Design and evaluation of a user-level file system for fast storage devices," *Cluster Computing*, vol. 18, no. 3, pp. 1075–1086, 2015.

[45] Y. Son, J. W. Choi, H. Eom, and H. Y. Yeom, "Optimizing the file system with variable-length i/o for fast storage devices," in *Proceedings of the 4th*

Asia-Pacific Workshop on Systems, APSys '13, (New York, NY, USA), pp. 14:1–14:6, ACM, 2013.

[46] J.Axboe, "Fiobenchmark." `http://freecode.com/projects/fio`, 1998.

[47] tpcc-mysql. `https://github.com/Percona-Lab/tpcc-mysql`.

[48] P. Fruhwirt, P. Kieseberg, S. Schrittwieser, M. Huber, and E. Weippl, "Innodb database forensics: Reconstructing data manipulation queries from redo logs," in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, 2012.

[49] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh, "Durable write cache in flash memory ssd for relational and nosql databases," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, 2014.

[50] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica, "Tpc-e vs. tpc-c: Characterizing the new tpc-e benchmark via an i/o comparison study," *SIGMOD Rec.*, vol. 39, no. 3, pp. 5–10, 2011.

[51] A. Wilson, "The new and improved filebench," in *Proceedings of 6th USENIX Conference on File and Storage Technologies*, 2008.

[52] J. Esmet, M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul, "The tokufs streaming file system," in *Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'12, (Berkeley, CA, USA), pp. 14–14, USENIX Association, 2012.

[53] A. Kopytov, "Sysbench: a system performance benchmark," *URL: http://sysbench. sourceforge. net*, 2004.

[54] M. Wu and W. Zwaenepoel, "eNVy: a non-volatile, main memory storage system," *ACM SIGOPS Operating Systems Review*, vol. 28, no. 5, pp. 86–97, 1994.

[55] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell, "The Rio File Cache: Surviving Operating System Crashes," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (New York, NY, USA), pp. 74–83, ACM, 1996.

[56] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on nvm," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pp. 1–14, May 2015.

[57] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, pp. 1–10, May 2015.

[58] E. Lee, H. Bahn, S. Yoo, and S. H. Noh, "Empirical study of nvm storage: An operating system's perspective and implications," in *Proceedings of the 2014 IEEE 22Nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, MASCOTS '14, (Washington, DC, USA), pp. 405–410, IEEE Computer Society, 2014.

[59] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," *Trans. Storage*, vol. 10, pp. 3:1–3:32, Jan. 2014.

[60] J. Piernas, T. Cortes, and J. M. Garcia, "The design of new journaling file systems: The dualfs case," *IEEE Transactions on Computers*, vol. 56, pp. 267–281, Feb 2007.

[61] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the buffer cache and journaling layers with non-volatile memory," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pp. 73–80, 2013.

[62] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, (Santa Clara, CA), pp. 323–338, USENIX Association, Feb. 2016.

[63] 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," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[64] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[65] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[66] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[67] S. Kannan, A. Gavrilovska, and K. Schwan, "pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, (New York, NY, USA), pp. 13:1–13:16, ACM, 2016.

[68] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pp. 49–65, 2014.

[69] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *Proceedings of the 11th Symposium on Operating System Design and Implementation (OSDI 14)*, 2014.

[70] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta, "Hive: Fault containment for shared-memory multiprocessors," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, (New York, NY, USA), pp. 12–25, ACM, 1995.

[71] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang, "A case for scaling applications to many-core with os clustering," in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys'11, (New York, NY, USA), pp. 61–76, ACM, 2011.

[72] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: A new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, (New York, NY, USA), pp. 29–44, ACM, 2009.

[73] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, *et al.*, "Corey: An operating system for many cores.," in *OSDI*, vol. 8, pp. 43–57, 2008.

[74] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich, *et al.*, "An analysis of linux scalability to many cores.," in *OSDI*, vol. 10, pp. 86–93, 2010.

[75] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Radixvm: Scalable address spaces for multithreaded applications," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 211–224, ACM, 2013.

[76] D. Zheng, R. Burns, and A. S. Szalay, "Toward millions of file system iops on low-cost, commodity hardware," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 69:1–69:12, ACM, 2013.

[77] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: Introducing multi-queue ssd access on multi-core systems," in *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, (New York, NY, USA), pp. 22:1–22:10, ACM, 2013.

[78] S. Mavridis, Y. Sfakianakis, A. Papagiannis, M. Marazakis, and A. Bilas, "Jericho: Achieving scalability through optimal data placement on multicore systems," in *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pp. 1–10, IEEE, 2014.

[79] R. Eqbal, *ScaleFS: A multicore-scalable file system.* PhD thesis, Massachusetts Institute of Technology, 2014.

[80] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Physical disentanglement in a container-based file system.," in *OSDI*, pp. 81–96, 2014.

[81] M. Curtis-Maury, V. Devadas, V. Fang, and A. Kulkarni, "To waffinity and beyond: A scalable architecture for incremental parallelization of file system code," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (GA), pp. 419–434, USENIX Association, 2016.

[82] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding manycore scalability of file systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, (Denver, CO), pp. 71–85, USENIX Association, 2016.

[83] "A parallel page cache: Iops and caching for multicore systems," in *Presented as part of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, (Boston, MA), USENIX, 2012.

요약

Solid-State Drive (SSD) 와 같은 고성능 저장장치 기술은 낮은 지연시간, 높은 대역폭, 그리고 높은 입출력 병렬성을 제공한다. SSD는 기계적인 오버헤드없이 데이터에 접근이 가능하도록 해주며, 하드 디스크와 같은 기존 저장장치에 비해 수십배의 성능향상을 가져온다. 그러나, 기존 소프트웨어 입출력 계층을 그대로 사용하거나 SSD의 특징을 활용하지 않으면 최대 성능에 도달하지 못할 수 있다.

본 논문에서는 SSD 특징들 (예: 낮은 지연시간, 높은 병렬성)을 최대한 활용할 수 있도록 파일 시스템을 최적화한다. 이를 위해 첫째, 지연시간이 낮은 SSD 기반에서 파일시스템의 기존 입출력 방식들을 분석한다. 해당 방식은 블락들이 비연속적일 경우, 여러 개의 입출력 요청으로 나누어서 처리하게 된다. 따라서, 이러한 방식은 해당 SSD의 특징을 최대한 활용하지 못한다. 이러한 문제를 해결하기 위해서, 본 논문은 효율적인 입출력 방식을 제안한다. 제안하는 방식에서는 하나의 입출력 요청으로 파일 시스템의 비연속 호스트 메모리 버퍼들을 비연속 저장소 세그먼트들로 전송한다. 따라서 이는 파일시스템이 지연시간이 낮은 SSD의 성능을 최대한 활용할 수 있게 해준다.

둘째, 높은 병렬성을 지닌 SSD 기반에서 파일시스템의 기존 락킹과 입출력 병렬성을 분석한다. 파일시스템에서는 공유 자료구조에 접근하기 위해 락킹이 사용되며, 입출력은 단일 스레드에 의해 직렬화된다. 이러한 이유로 파일시스템은 종종 높은 병렬성을 지닌 SSD와 멀티코어 환경에서 락 경쟁을 발생시키고 입출력 대역폭을 최대로 활용하지 못하는 문제에 직면한다. 이러한 문제를 해결하기 위해서 자료구조에 대한 동시적인 업데이트와 입출력 동작을 병렬화시킨다.

본 논문은 제안하는 방식들을 EXT4/JBD2에 구현하고 이들을 낮은 지연시간과 높은 병렬성을 가진 SSD기반에서 평가한다. 실험결과를 통해 최적화된 파일시스템이 기존 파일시스템에 비해 성능이 향상되었음을 확인할 수 있었다.

주요어: 파일시스템, 운영체제, 고성능 저장장치, Solid-State Drive
학번: 2013-30241