High-Performance Transaction Processing in Journaling File Systems

Yongseok Son, Sunggon Kim, and Heon Young Yeom, *Seoul National University*; Hyuck Han, *Dongduk Women's University*

https://www.usenix.org/conference/fast18/presentation/son

This paper is included in the Proceedings of the 16th USENIX Conference on File and Storage Technologies.
February 12–15, 2018 • Oakland, CA, USA
ISBN 978-1-931971-42-3

High-Performance Transaction Processing in Journaling File Systems

Yongseok Son, Sunggon Kim, Heon Young Yeom, and Hyuck Han†
Seoul National University, †Dongduk Women's University

Abstract

Journaling file systems provide crash-consistency to applications by keeping track of uncommitted changes in the journal area (journaling) and writing committed changes to their original area at a certain point (checkpointing). They generally use coarse-grained locking to access shared data structures and perform I/O operations by a single thread. For these reasons, journaling file systems often face the problem of lock contention and underutilization of I/O bandwidth on multi-cores with high-performance storage. To address these issues, we design journaling and checkpointing schemes that enable concurrent updates on data structures and parallelize I/O operations. We implement our schemes in EXT4/JBD2 and evaluate them on a 72-core machine with a high-performance NVMe SSD. The experimental results show that our optimized file system improves the performance by up to about 2.2x and 1.5x compared to the existing EXT4 file system and a recent scalable file system, respectively.

1 Introduction

A transaction in file systems is a group of file system modifications that must be atomic and durable [6, 12, 23]. To support transaction processing, many file systems have adopted a journaling technique to guarantee the atomicity and durability. Journaling logs modified metadata and data to the journal area in a transaction before updating the original area. After the transaction is committed, the committed transaction is written into the original area by checkpointing (write-ahead logging). With journaling, the file systems can provide crash-consistency to applications by recovering the committed transactions in case of crashes [5, 23, 24].

Although crash-consistency is supported using journaling file systems [20, 26, 29, 31], journaling can face a performance bottleneck on multi-cores and high-performance storage [15, 16, 21]. The performance bottleneck mainly arises from (1) data structures for transaction processing protected by non-scalable locks and (2) serialized I/O operations by a single thread. For example, in EXT4/JBD2, multiple application threads insert their own buffer into the running transaction by using coarse-grained locking which can negatively affect scalability on multi-cores. In addition, a single application thread performs checkpoint I/O operations which can underutilize high-performance storage.

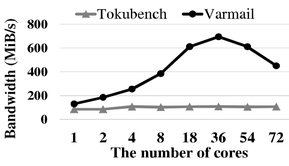
To handle these issues, previous studies [16, 21] investigated the locking and I/O operations of file systems. SpanFS [16] consists of a collection of micro file system services called domains. It distributes files and directories among the domains and delegates an I/O operation to the corresponding domain to reduce the lock contention and exploit the device parallelism. Min et al. [21] observed that file systems are hidden scalability bottlenecks in many I/O-intensive applications. They designed and implemented a benchmark to evaluate the scalability of file systems and found unexpected scalability behaviors. Our study is in line with these approaches [16, 21] in terms of investigating the locking and I/O operations of file systems. In contrast, we focus on internal operations of shared data structures and I/O processing in transaction processing.

In this paper, we propose a transaction processing with two main schemes to achieve high-performance I/O 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 techniques to transaction processing (i.e., running, committing, checkpointing, and recovery) on EXT4/JBD2 in Linux kernel 4.9.1.

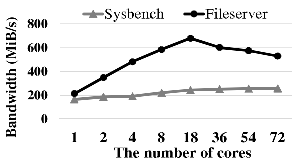
We evaluate the existing and our optimized file systems on a 72-core machine with an Intel P3700 NVMe SSD [14] using metadata and data-intensive workloads in the ordered and data journaling modes. The experimental results show that the optimized file system improves the performance by up to about 2.2x and 2.1x in the ordered mode and the data journaling mode, respectively, compared to the existing file system. Also, the optimized file system improves the performance by up to about 1.5x compared to SpanFS, a recently developed file system for multi-core scalability.

The contributions of our work are as follows:

- We analyze the locking and I/O operations in transaction processing of EXT4/JBD2 in terms of its procedures.
- We design and implement a transaction processing with concurrent lock-free updates on data structures



(a) Ordered mode



(b) Data journaling mode

Figure 1: Motivational evaluation (the number of threads is the same as that of the cores, and the detailed experimental environment is described in Section 4.)

and parallel I/O processing in a cooperative manner.

- We demonstrate that our optimized file system improves the performance compared to the existing file system and a recent scalable file system.

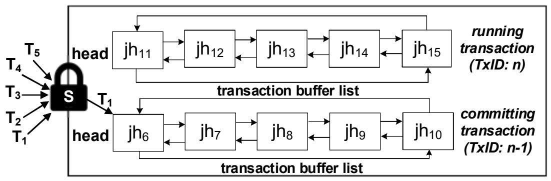
The rest of this paper is organized as follows: Section 2 describes the background and motivation. Section 3 presents the design and implementation of the proposed schemes. Section 4 shows the experimental results. Section 5 discusses the related works. Section 6 concludes this paper.

2 Background and Motivation

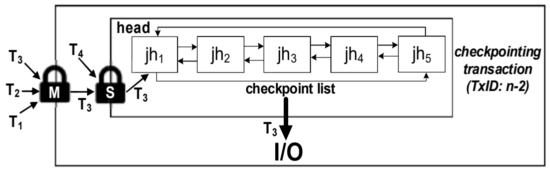
This paper focuses on the EXT4 journaling mechanism since EXT4 is the most widely used file system in Linux and more general than other file systems [16, 17]. EXT4 uses a fork of the journaling block device (JBD) called JBD2 [32] which performs transaction processing by using a variant of write-ahead logging (WAL) [11]. JBD is a file system-independent interface that can also be attached to other file systems, such as EXT3 and OCFS2 [10]. EXT4/JBD2 provides three journaling modes: write-back, ordered, and data journaling. The detailed description of each journaling mode can be found in previous studies [2, 16, 23, 25].

JBD2 adopts a single compound transaction. There is only one running transaction that absorbs all updates and one committing transaction at any time. An application thread starts a running transaction for each update and associates the update with the transaction. The running transaction has a linked list, which has the pointers to the modified blocks. When a periodic commit operation is invoked or an `fsync()` is called, a journal thread changes the state of the transaction to *committing*, and writes the blocks associated with the transaction into the journal area. After the transaction commits, the transaction is marked as checkpoint. The committed blocks are written back to the original area by an application thread during the checkpoint operation. Then, the journal area is reclaimed via the checkpoint operation.

We focus on the locking and I/O operations in transaction processing. Figure 1 shows a motivational evaluation using metadata and data-intensive workloads in the ordered and data journaling modes, respectively. As



(a) running and committing transactions



(b) checkpointing transaction

Figure 2: 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`))

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, 16], it is due to the lock contention on shared data structures and the serialization of I/O operations. For example, as shown in Figure 2(a) and 2(b), a spin lock (`j_list_lock`) is used to ensure the correct list operations for journal heads (`jhs`)[1] in the journaling lists (transaction buffer and checkpoint lists) [16], which are circular doubly linked lists. However, in multi-cores, this locking can incur 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(b), 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.

Running transaction. When application threads perform some file operations (e.g., `create()`), they start a transaction to handle the modifications. 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, or the threads are blocked if the transaction cannot be newly created. When a running transaction needs to be committed while a previous transaction is committing, the threads which try

[1]Journal head (`jh`) is a structure that associates the buffer (buffer_head (`bh`)) with the respective transaction [13]. The operations on the `bh` are protected by a spin lock (`jbd_lock_bh_state`) per `bh`.

to get a running transaction are blocked 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 [16, 23].

After getting the running transaction, 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. A thread, which acquires the list lock, associates the `jh` with the running transaction and inserts the `jh` into the tail of the list. Then, the thread releases the list lock and finishes the insert operation. Finally, the thread completes its own transaction processing by decreasing the number of updates.

When application threads perform some file operations, such as `truncate()`, the threads can invalidate buffers that are already associated with a transaction. In this case, by acquiring the state lock and the list lock, a thread removes the `jh` from the transaction buffer or checkpoint lists and disassociates the `jh` from the running or checkpoint transactions 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. The journal thread changes the running transaction to a committing transaction and its state to *committing* by acquiring the state lock. Then, the journal thread waits for other application threads to complete their transaction processing by checking the `t_updates` variable. 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 for the completion of I/Os. Then, the journal 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. The journal 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 under the list lock. The shadow list (`t_shadow`) includes the frozen buffers.

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. If the number of inserted buffers (`bufs`) is higher than the pre-defined threshold, the journal thread issues I/Os to the journal area by calling `submit_bh()` and prepares for the next I/Os. After issuing all the I/O requests for journaling, the journal thread waits for the completion of I/Os. And then removes the `jh` from the shadow list and inserts it into the forget list under the list lock. The forget list (`t_forget`) includes both the frozen buffers from the shadow list and buffers to be freed. After all the I/Os are completed, the journal thread writes the commit block for the transaction atomicity; 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 journal thread makes a checkpoint list with the buffers that are not freed and still dirty in the forget list under the list lock. Finally, the committed transaction is inserted into the tail of a checkpoint transaction list for checkpointing by acquiring the state and list locks.

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. A winner thread, which acquires the mutex lock, performs the checkpoint I/O operations while other threads are blocked 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. 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.

Under the mutex and list locks, the winner thread aggregates the buffers by fetching the `jh`s from the checkpoint list and inserting the fetched buffers into a checkpoint buffer (`j_chkpt_bhs`) to issue the I/Os in a batched manner. 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 one by one by fetching the `jh` from the checkpoint io list. When fetching the `jh`, the thread uses the list lock. After then, the thread sets the next transaction to be checkpointed in the checkpoint transaction list under the list lock. 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.

3 Design and Implementation

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

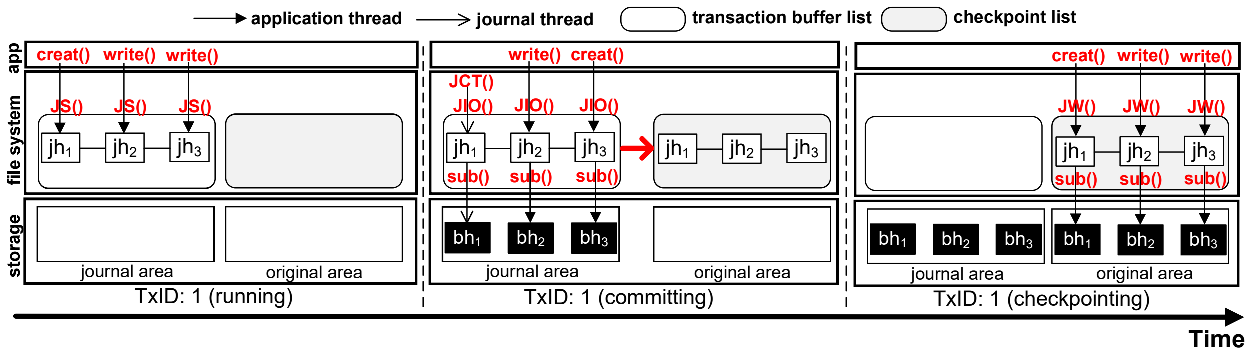


Figure 3: Overall architecture (app: application, jh: journal_head, bh: buffer_head, TxID: transaction ID, JS(): jbd2_journal_start(), JCT(): jbd2_journal_commit_transaction(), JIO(): journal_io_start(), JW(): jbd2_log_wait_for_space(), sub(): submit_bh())

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 [23]. 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 schemes do not sacrifice 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 (e.g., state lock) do not incur a significant overhead according to our evaluation, as well as other works [16]. However, such locks can be a performance bottleneck in a massive number of cores, which is beyond this paper; therefore, we leave the latent performance issue as a future work.

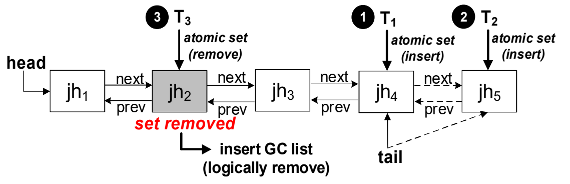
3.1 Design

Figure 3 shows an overall architecture of proposed schemes. When application threads update the metadata and data by calling system calls, such as `creat()` and `write()`, they start a transaction (i.e., TxID: 1) by calling `jbd2_journal_start()`. They insert their own modified buffer into the transaction buffer list concurrently. When the transaction needs to commit, the journal thread begins the commit process by calling `jbd2_journal_commit_transaction()` and starts journal I/Os. Application threads, which cannot create nor join a running transaction, join and perform the journal I/Os with the journal thread by calling `journal_io_start()`. They fetch the buffers in the transaction buffer list concurrently and write them to the journal area in parallel by calling `submit_bh()`. Then, the threads concurrently insert the committed buffers into the checkpoint list. When the space for journaling is not enough, application threads start to perform the checkpoint I/Os by calling `jbd2_log_wait_for_space()`. They fetch the committed buffers in the checkpoint list concurrently and write them to the original area in parallel by calling `submit_bh()`.

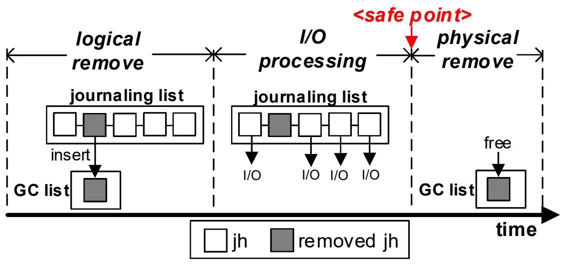
3.1.1 Concurrent updates on data structures

We manage the linked lists for transaction processing in a lock-free manner as shown in Figure 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. 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 [1], which allows us to identify the last element of the list and insert the item into the tail atomically.

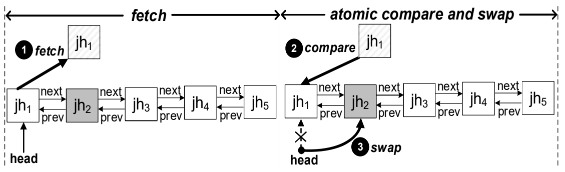
INSERT. We provide a concurrent insert operation to add an item to a list. In the existing transaction processing, the items are inserted into the tail of the list in the incoming order. Similar to the existing scheme but without locking, we concurrently update the tail by the incoming items using an atomic set instruction. In an example shown in Figure 4(a), 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



(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 4: Concurrent updates on data structures (jh: journal_head)

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 [22] 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 thus, threads can perform appropriate operations for the items. For the safe garbage collection (GC) of the logically removed items, we additionally maintain a GC list per transaction.

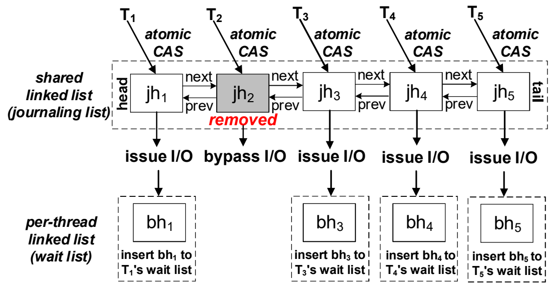
For example, as shown in Figure 4(a), when a thread (T_3) tries to remove the `jh` (jh_2), the thread marks the `jh` as *removed* atomically by executing the atomic set instruction. Then, the thread inserts the `jh` into the GC list using our concurrent insert operation as shown in Figure 4(b). And then, the 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 4(c), 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.

3.1.2 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 blocked 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 blocking 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 blocking 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 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 for the I/O completion in parallel.

As shown in Figure 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



T1
T2
T3
T4
T5
atomic CAS
shared linked list (journaling list)
head
tail
jh1
jh2
jh3
jh4
jh5
next
prev
removed
issue I/O
bypass I/O
per-thread linked list (wait list)
bh1
bh3
bh4
bh5
insert bh1 to T1's wait list
insert bh3 to T3's wait list
insert bh4 to T4's wait list
insert bh5 to T5's wait list

Figure 5: Parallel I/O in a cooperative manner (T: thread, jh: journal_head, bh: buffer_head)

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 Implementation

3.2.1 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 1, 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)[2]. First, the threads associate their `jh` with 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[3]. This instruction updates the tail and returns the old tail atomically. Then, the threads

PROCEDURE 1 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: }

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: }
```

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

[3]__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 [28].

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.

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 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.

3.2.2 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 2, 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 1, line 33, Procedure 2, line 11, and Procedure 3, line 2). Thus, the threads can join the I/O processing if it is initiated by the journal thread (Procedure 3, lines 5-6).

To handle the joined threads, we record the number of threads by executing atomic_add/sub instructions[4]

PROCEDURE 2 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: }
```

PROCEDURE 3 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: }
```

(Procedure 3, 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[5] (lines 9-17). If the fetched jh was logically removed, the thread bypasses and retries to fetch the next jh. Otherwise, each thread

[4]__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 [28].

[5]__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 [28].

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 1, line 36) and decreases* `t_update` *after inserting its* `jh` *(Procedure 1, line 42). Before the journal thread starts the parallel I/O processing by setting* `journal_io` *(Procedure 2, line 10), the thread waits until* `t_update` *becomes 0 (Procedure 2, 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 3, lines 22-32), the threads insert the `jh`s into a checkpoint list if the `jh`s 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 `jh`s 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 2, lines 12-13). This procedure satisfies the following property.

Property 2. *A transaction is committed or uncommitted (atomicity) according to the commit block.*

Every application thread that joins the I/O processing increases `num_io_threads` *before issuing I/O (Procedure 3, line 7) and decreases* `num_io_threads` *after completing I/O (Procedure 3, line 20). The journal thread waits until* `num_io_threads` *becomes 0 before the journal thread writes the commit block (Procedure 2, 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).

3.2.3 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 4, 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 from 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.*

PROCEDURE 4 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: }
```

A committed transaction is inserted into the tail of the checkpoint transaction list in the committed order (Procedure 2, lines 15-16). The last thread sets the next transaction to be checkpointed in the checkpoint transaction list in the committed order (Procedure 4, lines 19-20). This means that if transaction N-1 is committed prior to transaction N, 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 Property 1, 2, and 3. □

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

3.2.4 Recovery

In the existing recovery procedure, a single-threaded process (i.e., mount process) performs the recovery operation. To optimize the recovery procedure, we create multiple threads to perform scan and replay I/O operations in parallel and do not use any additional lock. When multiple threads start the scan operation, each thread makes a local wait list to complete the I/Os in parallel. Then, it atomically gets its own offset which is the logical position in the journal area by executing the `atomic_add` instruction. Each thread gets its own buffer based on the offset, issues the read request for the buffer in parallel, and inserts the buffer into own wait list. This process is repeated for all the buffers which need to be scanned. After the threads complete the I/Os for the scan operation in parallel, they concurrently insert the buffers included in their own wait list into a global list for the replay operation.

In the case of the replay operation, each thread makes a local wait list similar to the case of the scan operation and concurrently fetches the buffers to be replayed from the list. Then, it issues the write request in parallel and inserts the buffer into its own wait list. After issuing all the I/Os for the replay operation, the threads complete the I/Os in parallel. This recovery scheme makes the recovery procedure faster and more efficient.

4 Evaluation

4.1 Experimental setup

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 [14], 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 scheme (P-EXT4). In P-EXT4, we allow the application threads to perform the I/O operations by not blocking but joining them to the journal and checkpoint I/Os in a parallel and cooperative manner. However, we still update the data structures using `j_list_lock`. Through this evaluation, we compare the performance of our two schemes. We run metadata and data-intensive workloads, such as tokubench [9], sysbench [18], and filebench [34] with the parameters shown in Table 1. 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.

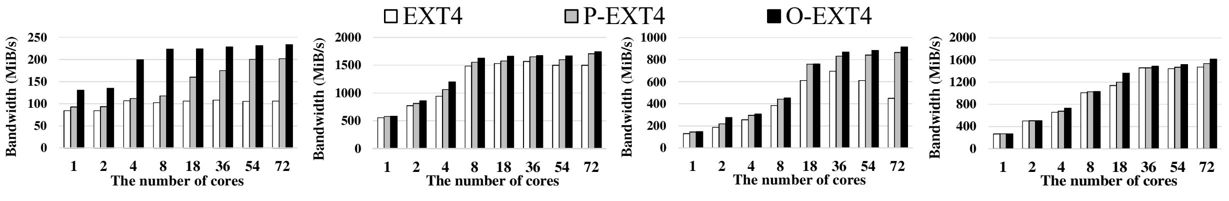
4.2 Performance results

4.2.1 Ordered mode

We present the performance results in the ordered mode as shown in Figure 6. In the case of tokubench as

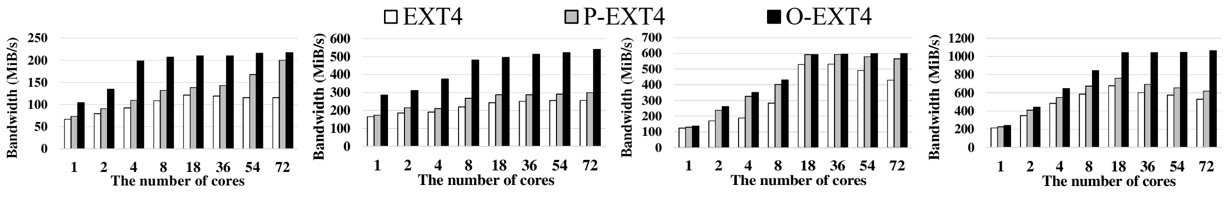
Benchmarks	Descriptions	Parameters
Tokubench (micro benchmark)	Metadata-intensive workload (file creation)	Files: 30,000,000, I/O sizes: 4KiB
Sysbench (micro benchmark)	Data-intensive workload (random write)	Files: 72, Each file size: 1GiB, I/O sizes: 4KiB
Filebench Varmail (macro benchmark)	Metadata-intensive workload (read/write ratio = 1:1)	Files: 300,000, Directory width: 10,000
Filebench Fileserver (macro benchmark)	Data-intensive workload (read/write ratio = 1:2)	Files: 1,000,000, Directory width: 10,000

Table 1: Workload descriptions and parameters



(a) Tokubench (b) Sysbench (c) Varmail (d) Fileserver

Figure 6: Ordered mode

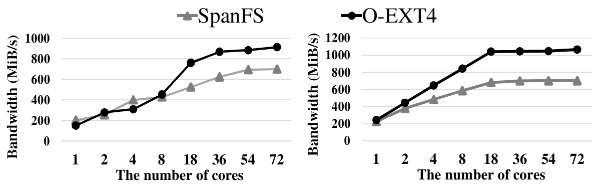


(a) Tokubench (b) Sysbench (c) Varmail (d) Fileserver

Figure 7: Data journaling mode

shown in Figure 6(a), 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 6(b), 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 6(c), 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 6(d), 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 number of metadata I/Os. Consequently, our optimized file system improves the performance in the ordered mode by reducing the lock contention and parallelizing the I/O operations, especially for metadata-intensive workloads.



(a) Varmail (b) Fileserver

Figure 8: Comparison with SpanFS

4.2.2 Data journaling mode

We present the performance results in the data journaling mode as shown in Figure 7. In the case of tokubench as shown in Figure 7(a), 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 7(b), 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.

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

Table 2: Device-level bandwidth and total execution time of main locks and write operations

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 3: Recovery performance

Under the varmail workload as shown in Figure 7(c), 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 7(d), P-EXT4 and O-EXT4 outperform EXT4 by 1.17x and 2.01x at 72 cores, respectively. O-EXT4 achieves up to 1064.6 MiB/s. The performance of P-EXT4 and 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.

4.2.3 Comparison with a scalable file system

We compare our optimized file system with SpanFS [16], 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 8, both file systems scale well until the performance is saturated in both workloads. Meanwhile, O-EXT4 generally shows better performance and improves the performance by up to 1.45x and 1.51x in the varmail and fileserver workloads, respectively, compared to SpanFS. Especially, in the case of the varmail workload, the performance of O-EXT4 is similar or slower than that of SpanFS at a 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.

4.3 Experimental analysis

Table 2 shows the total execution time for the main locks and the 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 achieves high-performance transaction processing by enabling both concurrent updates and parallel I/O.

4.4 Recovery performance

To evaluate the recovery performance, 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 3 shows the recovery performance of the ordered and data journaling modes in the file systems. The scan and replay operations occupy the main part 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.

5 Related Work

Lock-free data structures. Valois [33] provides lock-free data structures and algorithms for implementing a shared singly-linked list, allowing concurrent traversal, insertion, and deletion. Zhang et al. [35] introduce new lock-free and wait-free unordered linked list algorithms. They provide the first practical implementation of the unordered linked list that supports wait-free insert, remove, and lookup operations. Our study is inspired by these works [33, 35], and we use a variant of these implementations and apply it to transaction processing in a journaling file system.

Scalable database systems. Silo [30] is an in-memory database system designed for multi-core machines. Silo implements a variant of optimistic concurrency control in which transactions write their updates to shared memory only at commit time and uses a decentralized timestamp based technique to validate transactions at commit time. SiloR [37] adds additional features, such as logging, checkpointing, and recovery to Silo. It uses concurrency in all parts of the system. For example, the log is written concurrently to several disks, and a checkpoint is taken by several concurrent threads that also write to multiple disks. Our study is in line with these works [30, 37] in terms of investigating the multi-core scalability but we focus on the transaction processing in the file system.

Scalable kernels. Cerberus [27] mitigates contention on many shared data structures within OS kernels by clustering multiple commodity operating systems atop a virtual machine monitor. Boyd-Wickizer et al. [4] analyze the scalability of seven system applications running on Linux. They find that all applications trigger scalability bottlenecks inside a Linux kernel. RadixVM [7] 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 [27, 4, 7] and in line with them in terms of investigating the scalability of OS kernels on multi-cores. In contrast, we focus on transaction processing in file systems on high-performance storage.

Scalable storage stacks. Zheng et al. [36] 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 [15] 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. Our study is in line with these works [36, 15] 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.

Scalable file systems. IceFS [19] 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, the data and I/O within each cube are disentangled from the data and I/O outside of it. SpanFS [16] 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. [8] 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. [21] 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 [23] 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. ScaleFS [3] decouples the in-memory file system from the on-disk file system using per-core operation logs to improve many-core scalability. ScaleFS delays propagating updates to the disk until an `fsync` call, which merges the per-core logs and applies the operations to disk. In contrast with our scheme, ScaleFS uses the per-core log to avoid the lock contention and timestamp to sort the operations in the log. Our study is in line with these approaches [19, 16, 8, 21, 23, 3] 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.

6 Conclusion and Future Work

In this paper, we investigate the locking and I/O operations in transaction processing of the journaling file system. We find that the lock contention on shared data structures and I/O operations by a single thread can be performance bottlenecks on a multi-core platform incorporating high-performance storage. To handle this issue, we present a transaction processing with concurrent updates on data structures and parallel I/O operations. Experiments show that our optimized file system achieves higher performance and scales better than the existing file system. In future work, we will design and implement lock-free mechanisms to handle the locks for other shared resources, such as file, page cache, etc., and evaluate them in different storage environments.

7 Acknowledgments

We thank our shepherd, Dushyanth Narayanan, and anonymous reviewers for valuable comments that greatly improved our paper. This research was supported by National Research Foundation of Korea (NRF) (2015M3C4A7065645, 2015M3C4A7065646, 2016R1D1A1B03934393).

References

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

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

[3] Bhat, S. S., Eqbal, R., Clements, A. T., Kaashoek, M. F., and Zeldovich, N. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), ACM, pp. 69–86.

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

[5] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 228–243.

[6] Chutani, S., Anderson, O. T., Kazar, M. L., Leverett, B. W., Mason, W. A., Sidebotham, R. N., et al. The episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference* (1992), San Fransisco, CA, USA, pp. 43–60.

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

[8] Curtis-Maury, M., Devadas, V., Fang, V., and Kulkarni, A. 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, 2016), USENIX Association, pp. 419–434.

[9] Esmet, J., Bender, M. A., Farach-Colton, M., and Kuszmaul, B. C. The tokufs streaming file system. In *HotStorage* (2012).

[10] Fasheh, M. Ocfs2: The oracle clustered file system, version 2. In *Proceedings of the 2006 Linux Symposium* (2006), Citeseer, pp. 289–302.

[11] Gray, J., and Reuter, A. *Transaction processing: concepts and techniques*. Elsevier, 1992.

[12] Hagmann, R. *Reimplementing the Cedar file system using logging and group commit*, vol. 21. ACM, 1987.

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

[14] 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.

[15] Kang, J., Hu, C., Wo, T., Zhai, Y., Zhang, B., and Huai, J. Multilanes: Providing virtualized storage for os-level virtualization on manycores. *Trans. Storage 12*, 3 (June 2016), 12:1–12:31.

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

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

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

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

[20] Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., Vivier, L., and S, B. S. A. 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).

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

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

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

[24] Pillai, T. S., Alagappan, R., Lu, L., Chidambaram, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Application crash consistency and performance with ccfs. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association.

[25] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track* (2005), vol. 194, pp. 196–215.

[26] Reiser, H. Reiserfs, 2004.

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

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

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

[30] Tu, S., Zheng, W., Kohler, E., Liskov, B., and Madden, S. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 18–32.

[31] Tweedie, S. Ext3, journaling filesystem. In *Ottawa Linux Symposium* (2000), pp. 24–29.

[32] Tweedie, S. C. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo* (1998).

[33] Valois, J. D. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1995), PODC '95, ACM, pp. 214–222.

[34] Wilson, A. The new and improved filebench. In *Proceedings of 6th USENIX Conference on File and Storage Technologies* (2008).

[35] ZHANG, K., ZHAO, Y., YANG, Y., LIU, Y., AND SPEAR, M. Practical non-blocking unordered lists. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205* (New York, NY, USA, 2013), DISC 2013, Springer-Verlag New York, Inc., pp. 239–253.

[36] ZHENG, D., BURNS, R., AND SZALAY, A. S. 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* (New York, NY, USA, 2013), SC '13, ACM, pp. 69:1–69:12.

[37] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 465–477.