Z-Journal: Scalable Per-Core Journaling

Jongseok Kim and Cassiano Campes, *Sungkyunkwan University;* Joo-Young Hwang, *Samsung Electronics Co., Ltd.;* Jinkyu Jeong and Euiseong Seo, *Sungkyunkwan University*

https://www.usenix.org/conference/atc21/presentation/kim-jongseok

This paper is included in the Proceedings of the 2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the 2021 USENIX Annual Technical Conference is sponsored by USENIX.

Z-Journal: Scalable Per-Core Journaling

Jongseok Kim†, Cassiano Campes†, Joo-Young Hwang‡, Jinkyu Jeong† and Euiseong Seo†
†*Sungkyunkwan University, Republic of Korea.*
‡*Samsung Electronics Co., Ltd.*

Abstract

File system journaling critically limits the scalability of a file system because all simultaneous write operations coming from multiple cores must be serialized to be written to the journal area. Although a few scalable journaling approaches have been proposed, they required the radical redesign of file systems, or tackled only a part of the scalability bottlenecks. Per-core journaling, in which a core has its own journal stack, can clearly provide scalability. However, it requires a journal coherence mechanism because two or more cores can write to a shared file system block, so write order on the shared block must be preserved across multiple journals. In this paper, we propose a novel scalable per-core journal design. The proposed design allows a core to commit independently to other cores. The journal transactions involved in shared blocks are linked together through order-preserving transaction chaining to form a transaction order graph. The ordering constraints later will be imposed during the checkpoint process. Because the proposed design is self-contained in the journal layer and does not rely on the file system, its implementation, Z-Journal, can easily replace JBD2, the generic journal layer. Our evaluation with FxMark, SysBench and Filebench running on the ext4 file system in an 80-core server showed that it outperformed the current JBD2 by up to approx. 4000 %.

1 Introduction

The number of concurrent cores accessing a file system is ever increasing as the number of cores installed in a system increases. However, existing file systems show poor scalability for a few file system operations [16]. Especially, because write requests coming from all cores must be serialized to be written on the single journal, the journal layer acts as a representative scalability bottleneck [8, 16, 17, 19, 20].

The serialization occurs at two points in journaling; writes to the in-memory journal data structures, and to the on-disk journal area. The in-memory data structure accesses are parallelizable to some degree by applying lockless parallel data structures [19]. However, the parallelized memory operations, in the end, should be serialized for the on-disk journal writes. The serialization at the on-disk journal works as the more serious inhibitor for the file system scalability because the storage performance is still significantly slower than the main memory. In addition, the serialized journal writes hinder the performance gain earned from the ever-increasing internal-parallelism of modern SSDs [4].

A few research results have been proposed to achieve scalability in journaling. However, they require an explicit separation of the file system space [8], byte-addressible non-volatile memory (NVM) as the journaling device [20], or tight coupling of file system and journal layer [2]. Therefore, in order to apply them to existing systems, application modification, adoption of NVM, or radical changes to the file system design are necessary, respectively.

Not only performance but also stability and reliability are important criteria for choosing a file system. The file systems being used in production systems have obtained their reliability and performance through decades of improvement and refinement. Consequently, it is a difficult choice to migrate to a radically redesigned file system. This leads to the large demand for the scalable generic journal layer that can replace the existing ones, such as Journaling Block Device 2 (JBD2) [22].

The most intuitive approach to realize a scalable journal is having independent journal space and journal stack per-core. If the thread running on a core can write to the journal dedicated to the core independently to the other threads, the file system can achieve the complete performance scalability to the point of the maximum disk performance.

However, when two or more threads simultaneously perform write operations to the same file system block, inconsistency between write order to the in-memory buffers and that to the on-storage journals may occur. For example, as shown in Figure 1, let us suppose that core 0[1] modifies block 0 and block 1. In turn, core 1 updates block 0 and block 2.

[1]For brevity, we will use *core* to denote the thread running on the core unless otherwise stated.

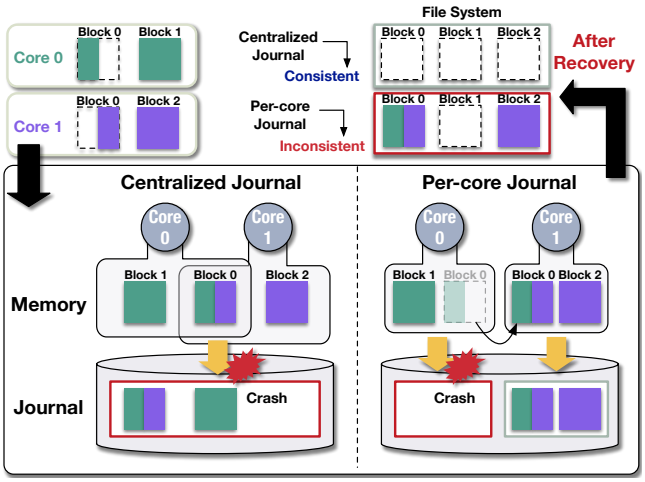


Figure 1: Inconsistency between write order to in-memory data and that to on-storage journal caused by per-core journaling without a coherence mechanism.

In this case, the journal commit issued by core 1 may finish earlier than that by core 0 when they independently operate. If a system crash occurs when core 1 committed, but core 0 did not, block 0 will be restored to a state that includes the modifications from both core 0 and core 1 during the recovery procedure. Because core 0 has not committed its modifications, the valid shape of block 0 after recovery must exclude the modifications by core 0, or have no modifications at all. This kind of inconsistency cannot happen in the conventional centralized journal.

Commonly, cores share metadata or files. Two cores have to share the metadata blocks even when they do not share any files if their files happen to be stored in the same block group [1], which is false sharing in this case. Regardless of whether being false or true, sharing blocks among cores is inevitable when the multiple cores access the same file system. Therefore, a scalable per-core journaling scheme must have a journal coherence mechanism that keeps the write order for the shared block modification.

This paper proposes *Z-Journal*, a scalable per-core journal scheme. Z-Journal includes a novel coherence mechanism. Z-Journal's coherence mechanism allows each core to commit transactions to its journal area independently to other cores. However, when shared-block writes exist in the journal transactions, Z-Journal forms write-order graphs among transactions sharing blocks through *order-preserving dependent transaction chaining* and commits them with the transactions. Imposing order-constraints over transactions will be performed when checkpointing the committed transactions. Through this journal coherence mechanism, Z-Journal enables scalable per-core journaling while keeping crash consistency.

Z-Journal is designed to provide an identical interface to JBD2. Therefore, it can be easily applied to the existing file systems that use JBD2, such as ext4 and OCFS2 [6], as their journal mechanism. However, to maximize the effectiveness of per-core journaling, it is desirable to eliminate false sharing among files coincidentally placed in the same block group. For this, we additionally propose a core-aware block-group allocation algorithm for the ext4 file system.

We implemented Z-Journal in the Linux kernel and applied it to the ext4 file system as its journal layer. For evaluation, we measured the performance and scalability of the Z-Journal and ext4 combination while executing FxMark [16], SysBench [12] and Filebench [21, 23] in an 80-core server.

The remainder of this paper is organized as follows. Section 2 introduces the background and motivation of this research. Section 3 proposes the design and implementation of Z-Journal, and Section 4 evaluates the proposed scheme using various benchmark workloads. After the related work is introduced in Section 5, Section 6 concludes our research.

2 Background and Motivation

2.1 Design of JBD2

A file system operation usually relates to the modification of multiple file system blocks. The conventional storage devices are unable to guarantee atomic writes of multiple blocks. When a sudden crash occurs during a file system operation, only a part of the modifications may be reflected on the storage, and the file system metadata and actual data may eventually mismatch each other. The partial update of the metadata or the mismatch between metadata and data can destroy the validity of the file system.

The journaling mechanism is a measure to ensure consistency by logging the file system changes caused by an operation in the predetermined location. After the file system commits the series of changes caused by a file system operation, the journal reflects the logged changes to the file system through the checkpoint operation. Once a file system operation is committed to the journal, the journal guarantees that the changes are reflected in the file system because it can replay the committed changes even after a system failure.

JBD2 is a generic journaling layer used in the Linux kernel. JBD2 groups a series of file system changes during an interval together into a unit called a transaction. A transaction is committed to the journal area, periodically, or on the conditions explained later. When a transaction is successfully committed to the journal, JBD2 leaves the commit block at the end of the journal record. When JBD2 performs recovery after crash or failure, it checkpoints the transactions having a commit block and discards the transactions without a commit block.

A transaction undergoes a few phases during its life cycle from its creation to checkpoint. Figure 2 shows the organization of transactions in different phases. A transaction is in one of the four states: ❶ *running*, ❷ *locked*, ❸ *committing*,

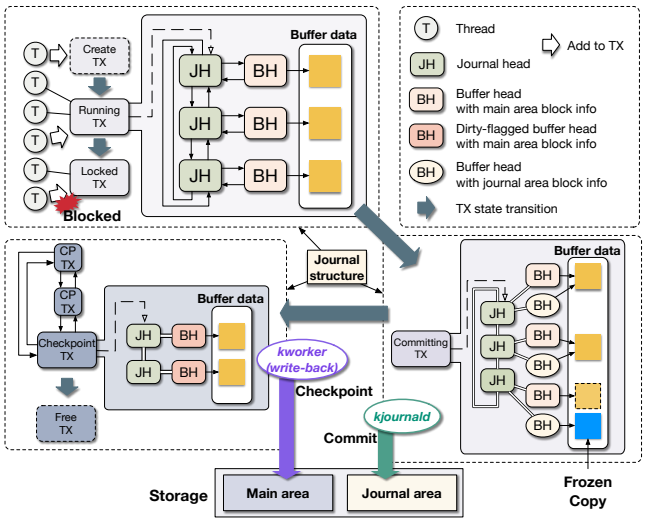


Figure 2: Organization and life cycle of a transaction in JBD2.

and ❹ *checkpoint*, respectively. (The intermediate states are omitted here for brevity.) The journal structure is the central data structure of JBD2, and has three pointers pointing to the running transaction, committing transaction and checkpoint transaction list, respectively. There can be up to a single running transaction and up to a single committing transaction at a time point. The checkpoint transaction list pointer points to the head node of the doubly linked list of transactions to be checkpointed. Its head node is the oldest transaction in the list.

The buffered image of a file system block in the main memory is called a *buffer*. A buffer in the main memory is unique for a file system block. Therefore, all cores share and access the same buffer when accessing the same file system block. A buffer has its buffer head. The buffer head contains the information about the buffer and its corresponding file system block, such as the block device, logical block number, data size, etc. A buffer head is inserted in a transaction through a journal head data structure. A journal head is allocated for and bidirectionally connected to a buffer head. In a transaction, the journal heads of modified buffers are chained in a doubly-linked circular list, as shown in the upper left side of Figure 2. The oldest journal head in a transaction becomes its list head. Modifications of buffers grouped in a transaction are considered an atomic operation.

Since JBD2 only allows a single transaction to run at a time, it is trivial to order the writeback of the updated buffers. It simply writes them back they were inserted into the list of committed transactions. Later, we will explain how Z-Journal relaxes these requirements and allows multiple transactions to run concurrently.

The running transaction accommodates the modified buffers produced by file system operations. If there is no running transaction when an operation is issued, a new transaction is created and becomes the running transaction. When a core writes to a buffer, its buffer head will attach to a new journal head, and the journal head will be inserted into the journal head list of the running transaction. At this moment, the journal space must be reserved for the inserted buffers so that later the running transaction can be committed without space allocation. If the journal space is insufficient, the user-level thread performs checkpoint to free up the journal space.

Later, `kjournald`, which is a kernel thread in charge of the commit operation, starts to commit the running transaction upon one of these three conditions: (1) the transaction timeout occurs, which is by default 5 seconds; (2) the transaction capacity, which is by default a quarter of the journal area, is exhausted; or (3) the `fsync` system call is invoked by a process. Once the commit operation begins kjournald turns the running transaction to the locked transaction.

When the transaction enters the locked state, it cannot accommodate any more updated buffers, except the ones from the file system operation accepted to join the transaction but did not finish yet. When the last modified buffer is inserted into the locked transaction, its state is changed to the committing state. A new running transaction cannot be created while the locked transaction is waiting for its closure. Therefore, a thread issuing a new file system operation should be blocked until a new running transaction becomes available after the locked transaction becomes the committing transaction.

As shown in the bottom right corner of Figure 2, kjournald creates another buffer head for each buffer. This buffer head contains the information of the journal block to which the corresponding buffer will be committed. After this, kjournald writes the buffers of the committing transaction to their assigned journal blocks.

Usually, the original buffers remain attached to the buffer heads in the committing state. However, when a thread tries to modify the buffer included in the committing transaction before kjournald begins writing to the journal, the thread makes a replica of the buffer called a *frozen copy*, and replaces the original buffer with the frozen copy. The original buffer can then be freed from the committing process, and the thread can modify the buffer.

When the commit is finished, the buffer heads will be marked as dirty to denote that their buffers are required to be written to their originated file system blocks during checkpointing. After this, kjournald finally converts the committing transaction to a checkpoint transaction and insert it at the end of the checkpoint transaction list.

The checkpoint operation is handled by the write-back kernel thread, `kworker`, every 5 seconds, or by a user-level thread performing file operations when it finds out that there is not enough space left in the journal area for the write operation. kworker moves the dirty buffers of the transactions that have stayed in the checkpoint list for longer than 30 seconds to

their originated file system blocks and mark them as clean.

Because a journal head uses a separate pointer to be connected to a checkpoint transaction, a buffer can belong to a running transaction and a checkpoint transaction at the same time. When a thread tries to modify a buffer already in a checkpoint transaction, it will be inserted to the running transaction, and at the same time, its dirty flag will be cleared so that it will not be checkpointed. In this situation, the running transaction is allowed to modify the buffer.

kjournald later frees the clean buffers from the checkpoint transaction. It also frees the empty checkpoint transactions from the checkpoint list and the corresponding commit transactions from the journal area to make free space.

In case of a system crash, kjournald initiates the recovery process. It searches for the committed transactions in the journal area, and replays them in order. This guarantees that the file system remains consistent and committed data are preserved in the file system.

2.2 Scalability Bottlenecks in JBD2

The current JBD2 design has multiple scalability bottleneck points as follows.

At a given time point, there is only a single running transaction, which is the only transaction that can accept the modified buffers. Therefore, when multiple cores perform file operations in parallel, they have to compete for the lock acquisition for the running transaction [19].

Secondly, when the running transaction is closed, a new running transaction can be created only after the current committing transaction finishes, and the closed transaction becomes the committing transaction. Therefore, when the locked transaction waits for the committing transaction to finish, all cores that issue file operations must wait altogether. The larger the number of waiting cores, the more this convoy effect adversely impacts the overall file system throughput.

Last but not least, the current JBD2 does not fully utilize the internal-parallelism provided by the modern NVMe SSDs because the kjournald thread solely issues a serialized stream of buffer writes to the storage. To utilize the high-performance of modern storage devices, the journal mechanism should be able to commit in parallel.

These problems commonly stem from that there is only a single running transaction and a single committing transaction in the system. However, blindly parallelizing the running and committing transactions or entire journal stack for achieving scalability complicates keeping the write orders of the shared blocks as stated in Section 1 because a buffer head can belong to only a single transaction in the current design. If a buffer head is allowed to simultaneously exist in multiple transactions, the coherence mechanism to guarantee the write order consistency across multiple transactions to the shared buffers is necessary. This problem was referred to as a *multi-transaction page conflict* by Won et al. [24].

An approach that makes a frozen copy of a buffer whenever the file system modifies it and inserts the copy to the journal transaction instead may allow simultaneous writes to the same buffer coming from multiple threads. However, it still requires the ordering of block copies when committing a transaction so that the preceding block modifications are guaranteed to be committed before. This will be another serialization point. In addition, adding buffer copy operations to the file system write path will notably retard the write latency. Therefore, journaling with a parallel transaction requires an efficient coherence mechanism that can preserve the write orders to the buffers, while allowing as much parallel buffer modification and independent transaction management as possible.

3 Our Approach: Z-Journal

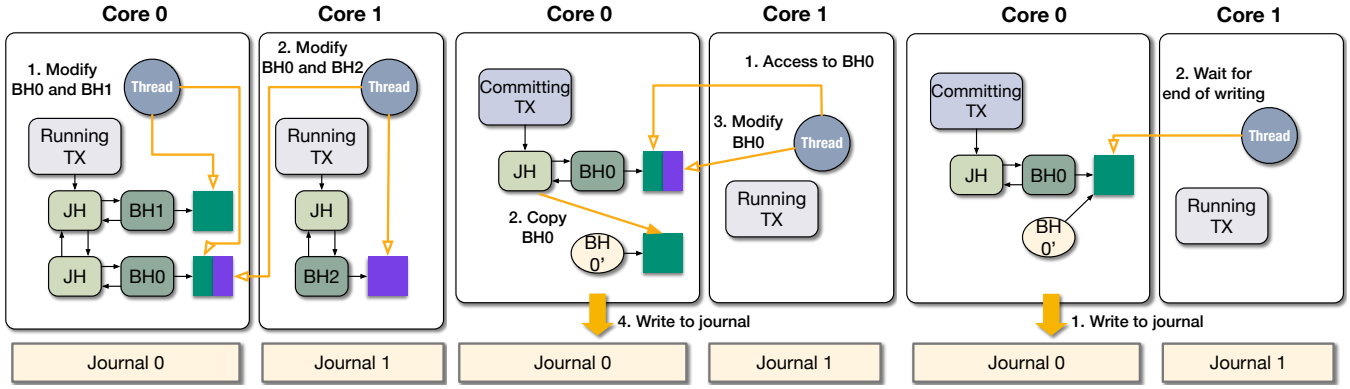
In Z-Journal, each core has its journal area on the storage device, and its kjournald, which handles the commit operation. Because kjournald is bound to each core and executed locally, this also improves the memory access locality in non-uniform memory access (NUMA) systems. The journal stack of each core has the running transaction, committing transaction, and the checkpoint transaction list the same as JBD2, and their life-cycles are identical as well.

This per-core journal approach removes the aforementioned serialization points in the journal layer and thus obtains scalability. First, because each core has its running transaction, a thread does not need to compete with the other threads for acquiring access to the running transaction. Second, when the running transaction of a core closes, and the core waits for the previous committing transaction to finish to create a new running transaction, this waiting only applies to that core. Therefore, this convoy effect is confined in the core boundary. Finally, because multiple kjournald are able to commit in parallel, Z-Journal can fully utilize modern high-performance storage devices.

However, as stated earlier, the per-core journal approach must deal with the mismatch between the in-memory buffer write order and the on-disk transaction commit order to guarantee the crash consistency.

3.1 Analysis of Journal Coherence Problem

In the per-core journal design, a thread inserts modified buffers to the running transaction allocated for the core it runs on. However, the buffer may have already belonged to a transaction of other cores as shown in Figure 1. Figure 3 categorizes this situation into three cases. In Figure 3, two cores modify two unrelated buffers, but also both modify a shared buffer. The same as JBD2, when a core tries to write to a buffer that is in a checkpoint transaction, the buffer can be inserted to the running transaction of the core without breaking the crash consistency because its last image is safely stored



(a) From a running transaction

(b) From a committing transaction before issuing commit writes

(c) From a committing transaction after issuing commit writes

Figure 3: Three different cases that a running transaction tries to access a buffer head that already belongs to another transaction in the per-core journal system.

in the journal area. Therefore, this case is not considered in the journal coherence mechanism.

In Figure 3a, both cores modify the shared buffer before either begins to commit. Since we only maintain one copy of the shared buffer, we can no longer separate the updates made by the two cores. However, if we treat both transactions of these two cores as parts of one large transaction, we can allow uninterruptible parallel access to the shared buffers while guaranteeing the crash consistency. Their commits should be considered as valid only when both are committed; otherwise, both must be voided.

In Figure 3b and Figure 3c, one core begins to commit its running transaction before the other touches the shared buffer. In Figure 3b, the buffer has not yet been scheduled for commit, so we can create a frozen copy. In JBD2, the frozen copy will be attached to the buffer head for the commit to the journal area, and the buffer head for the original buffer is kept in the committing transaction. Because the original buffer head is no longer used by the committing transaction, it can be modified by the other cores. This allows an immediate update of the shared buffer in Figure 3b. In Figure 3c, the shared buffer is already being written to the journal area by the first core. At this point, it is too late to make a copy of the buffer for concurrent modifications, since the file system page cache would continue to point to the copy that is being written. Therefore, core 1 must wait for core 0 to complete its commit operation before modifying the shared buffer. If a committing transaction always creates frozen copies for all of its buffers and use them for commit block writes, this waiting can be eliminated.

However, in both Figure 3b and Figure 3c, to guarantee crash consistency, core 1's commit must be performed after core 0's commit finishes. This serialization may suspend core 1's file system operations because a new running transaction can be created only after core 1's current running transaction turns into the committing transaction, which again can be possible only after the current committing transaction finishes. If there is a long chain of transaction dependency, this will result in poor scalability. However, if we can enforce a rule that a committed transaction can be checkpointed only after the transactions it depends on are committed, we can allow both of core 0's committing transaction and core 1's running transaction to be committed independently.

Analysis of the three conflicting cases revealed that the conditional recognition of the committed transactions, which checkpoints the committed transactions only when their dependent transactions are committed as well, allows them to be committed independently to each other. In addition, the proactive use of frozen copies for committing allows immediate writes to the buffers being shared with the committing transaction. Based on this observation, we propose the journal coherence mechanism for Z-Journal.

3.2 Journal Coherence Commit

The journal coherence mechanism of Z-Journal imposes the write order between transactions during checkpoint so that each core can commit its transactions without being interrupted by activities of other cores. In Z-Journal, the committed transaction will be considered as *valid* only after all transactions preceding the transaction in the write order are committed. Z-Journal checkpoints only the valid commits.

To realize this, Z-Journal should be able to identify the ordering relationships between transactions, and to record them in the transaction commit. We propose order-preserving transaction chaining for this.

In Z-Journal, a transaction maintains the information about the transactions having ordering relationship with it by recording their unique identifiers into its *chained-transaction lists* as shown in Figure 4. The number of lists in a transaction is

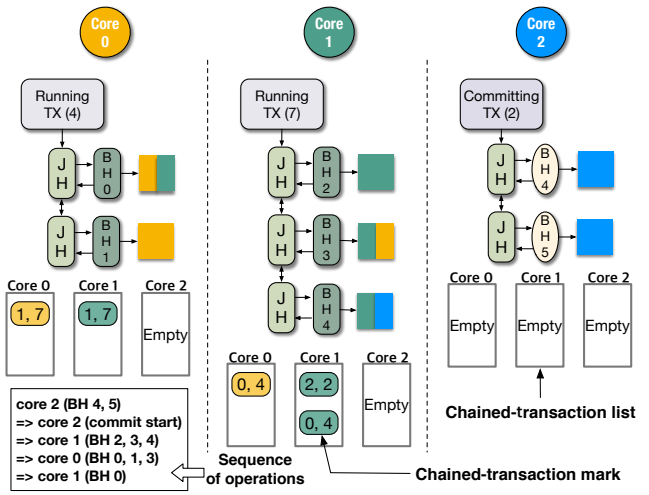


Figure 4: Example of order-preserving transaction chaining.

equal to the number of cores in a system; hence each core can update its corresponding list in any transactions without acquiring a lock. A transaction has a transaction identifier (ID), which monotonically increases in a per-core journal. Accordingly, a unique identifier of a transaction is a pair of a core ID and a transaction ID. When a transaction has an entry of (j, t) in its chained-transaction lists, the transaction is valid only after transaction t of core j is committed.

When two running transactions share a buffer, as discussed on Figure 3a, they should form a bi-directional transaction chain. If a running transaction tries to write to a buffer that belongs to a committing transaction as in Figure 3b, they should form a uni-directional chain that the running transaction follows the committing transaction.

For example, in Figure 4, buffer 4, written by transaction 2 of core 2, is about to be modified by transaction 7 of core 1. Because transaction 2 of core 2 is already in the committing state, core 1 leaves (2,2), which means transaction 2 of core 2, in core 1's list of transaction 7. By this, an uni-directional chain is formed between (1,7) and (2,2).

When core 0 writes to buffer 3, core 0 is allowed to do so while buffer 3 remains in journal 1's running transaction. In such case, these two running transactions must be considered as a single super-transaction. Therefore, core 0 leaves (0,4) in core 0's list of journal 1's running transaction and (1,7) in core 0's list of journal 0's running transaction at the same time. This forms an all-or-nothing relationship on the two running transactions. Later, when core 1 writes to buffer 0, core 1 will leave (1,7) in core 1's list of journal 0's running transaction and (0,4) in that of journal 1's because core 1 is not aware of the chained-transaction marks left by core 0 at this moment.

When each core independently modifies non-shared files, no chain will be created over their transactions. In this situation, the order in which the transactions are committed may be different from the original write operation order. For example, a process can perform metadata operations to different files on two different cores, respectively, and the second may commit when the first does not. The POSIX semantics does not guarantee the durability of write operations before finishing `fsync` of the corresponding file descriptor. Therefore, reversing the commit order between the transactions that have no ordering relationship does not violate the POSIX semantics. Even when the first invokes `fsync` before the second commits, the second may commit before the first and this is also allowed in the POSIX semantics because `fsync` is supposed to commit write operations only of a given file descriptor.

When a synchronous write from `O_SYNC` or `O_DSYNC`, is issued, the transaction chains related to the current write, if existing, should have been already formed or will be formed by the current write. Therefore, after every write operation, Z-Journal commits only the running transactions chained to the running transaction of the current write. However, when a core calls `fsync`, Z-Journal enforces all cores to commit their running transactions because `fsync` is supposed to flush all transactions related to the given file descriptor, and they can be in any core without being chained to the running transaction of the current core.

Since `fsync` does not add a new buffer head to the transaction nor allocate journal space, the commit time takes up most of the `fsync` latency. In Z-Journal, the commit operation must be performed in all cores to finish `fsync`, but the delay from it is not significant because the commit operation is executed in parallel in each core. Rather, when `fsync` is called in parallel on multiple cores, it is significantly advantageous in terms of throughput because multiple `fsync` invocations, which must be serialized in JBD2, can be parallelized in Z-Journal.

When a transaction enters the committing state, Z-Journal proactively creates frozen copies of its buffers regardless of their sharing states to prevent buffer update from waiting for finishing the commit operation as shown in Figure 3c. Through this *proactive frozen copy* approach, a committing transaction is disconnected from the original buffers and accesses only their frozen copies. Therefore, in Figure 4, when core 1 writes to buffer 4, buffer 4 can be inserted to transaction 7 without waiting because transaction 2 of journal 2 is using the frozen copy of buffer 4.

The proposed order-preserving transaction chaining scheme enables Z-Journal to keep track of write orders among transactions in a scalable and efficient way. Based on this, Z-Journal puts off the enforcement of write-order constraints to the checkpoint time and allows cores to independently commit transactions regardless of their sharing status. Because the usual checkpoint interval is a lot longer than the transaction life span, it is highly likely that almost all transactions become valid at the time of the checkpoint. Therefore, Z-Journal's journal coherence mechanism to the checkpoint duration is expected to be minimal.

The proactive frozen copy approach enables the immediate sharing of a buffer that is currently in use of a committing transaction. Combined with the order-preserving transaction chaining, this enables all transactions to simultaneously proceed to the checkpoint state without waiting. However, the proactive frozen copy generates a significant amount of memory copy operations and increases memory consumption. In addition, the order-preserving transaction chaining requires additional writes to the lists, although they are lockless. Nevertheless, because these overheads involve the in-memory structures, not the on-disk journal structures, their impact on the scalability and overall performance will be negligible compared to the expected benefits.

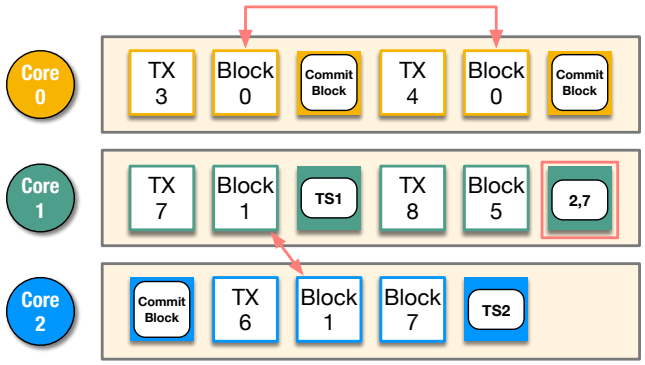


Figure 5: Snapshot of per-core journals after system crash.

3.3 Journal Coherence Checkpoint

The same as JBD2, in Z-Journal, there is a single kworker thread in the system, and it periodically performs the checkpoint operation. Also, the same as JBD2, a user-level thread can conduct checkpoint when its write operation is delayed due to the insufficient free journal space.

As stated earlier, not all committed transactions become objects of checkpointing in Z-Journal. To implement this, when kjournald changes the state of a transaction to the checkpoint state, it skips over setting dirty flags of transaction's buffers. Instead, when kjournald converts its running transaction to the committing transaction, it checks whether the transactions in its checkpoint transaction lists are valid. If a transaction turns out to be valid, its buffers will be marked as dirty. Later, they will be checkpointed by kworker, which periodically iterates and checkpoints dirty buffers in the background.

For a committed transaction to be valid, its direct preceding transactions must be not only committed but also valid. Therefore, to check the validity of a committed transaction, kjournald traverses the transaction chain graph, which was created from the chained-transaction marks of the transaction and its ancestors, and checks whether all of their ancestors are valid. Every transaction has a field that shows its validity. If every ancestor of a transaction is identified as valid during the search, kjournald sets its validity field to prevent redundant search over its ancestors in the future.

This validity check is performed not only by kjournald, but also by user-level threads. When kjournald finds out an uncommitted ancestor, it stops the search and starts processing the next checkpoint transaction. However, in such a case, a user-level thread will initiate committing the uncommitted ancestor. It continues after the commit finishes because it cannot proceed with its file system operation without freeing journal space.

A valid transaction can be checkpointed anytime independently from its chained transactions. Therefore, the checkpoint order of valid transactions may be different from their dependent transaction orders. However, the removal of a transaction from a journal can be allowed only after its chained transactions are all checkpointed because all chained transactions must be replayed together during the recovery.

The commit operation is conducted using frozen copies. However, the journal heads of a checkpoint transaction point to their original buffers. Therefore, checkpointing a valid transaction always updates the file system blocks with the up-to-date images of dirty buffers regardless of the checkpoint order.

3.4 Recovery Procedure

A committed transaction stored in a journal area has three kinds of blocks the same as JBD2: transaction descriptor blocks, data blocks, and a commit block. The transaction descriptor blocks, which store the information about the following data blocks, are located at the head of a committed transaction. Next, the data blocks are stored. Finally, The commit block is written to indicate the successful completion of the commit.

In Z-Journal, the commit block also stores the chained-transaction lists of the transaction. In addition, the commit block also has the timestamp [9] of the commit start time for the global ordering of transaction commits across multiple journals.

The recovery process first searches for the transactions with the commit block from all journals to find the valid transactions. After this, it creates the transaction order graphs based on their chained-transaction lists. Similarly to the checkpoint procedure, the recovery process traverses the graphs to find valid transactions. Finally, it updates the file system blocks with the latest buffer images from the valid transactions.

When a buffer redundantly appears in two valid transactions connected through an ordered chain, the buffer image from the latter transaction in the graph is the latest one, which will be used for recovery. If two transactions are tied together through a bi-directional chain constructed from sharing between two running transactions, a buffer cannot exist in both transactions simultaneously. Finally, when two transactions

are not chained together but have the same buffer, the buffer image of the latter transaction, which is determined by the timestamp, will be chosen because this case means that the latter transaction overwrote the buffer after the former transaction completely committed. These rules are transitively applied to the cases involving multiple transactions.

Figure 5 shows the snapshot of the per-core journals after a system crash. Transaction 8 of core 1 is invalid, although it has the commit block because the transaction 7 of core 2, which must precede it, is not written in the journal. Therefore, it is discarded in the recovery process. Block 0 is in transaction 3 and 4 of core 0, both of which are valid transactions. However, transaction 4 has the larger transaction ID, block 0 of transaction 4 will be restored. Block 1 also appears in transaction 7 of core 1 and transaction 6 of core 2 at the same time. They do not have a dependency relationship. Therefore, the recovery process compares the timestamps of both transactions to determine the latest buffer image to restore, which is that of transaction 6 of core 2 in this case.

3.5 Core-Aware Block Group Allocation

Block grouping is leverage inherited from the legacy spinning disks to provide a faster seek time [5]. The block group allocator of ext4 decides which block group a new inode or data block should be allocated in. The current block group allocator aims at increasing the access locality and minimizing seek times to obtain performance benefit from the underlying spinning disks [13].

The block group allocator of ext4 disperses the directory allocation over as many block groups as possible. However, when creating a file, it tries to place the inode of the new file in the same or nearby block group with its parent directory. It allocates file's data blocks in the same block group with the file inode when the file size is smaller than a predefined value, `stream_req`. When larger than that, data blocks will be allocated from the last block group in which the data blocks for a large file were allocated. If the block group cannot accommodate the request, the block group allocator will sequentially try the following block groups.

The current block group allocator does not benefit when using a flash SSD because it does not have seek time. On the contrary, it increases false sharing of metadata among cores because the allocated blocks are unevenly distributed over a few block groups, and the block group placement of files and directories are blind to who will access them.

In Z-Journal, as sharing between transactions gets more frequent, the lengths of transaction chains tend to be longer. The large transaction order graphs will incur large checkpoint overhead. Therefore, we propose a *core-aware block group allocator* for ext4 that allows the group of blocks requested by one core to be allocated exclusively to other cores as much as possible.

When i-th core requests a block or metadata entry, the proposed block group allocator sequentially checks from block group $\lfloor \frac{\text{number of block groups}}{\text{number of cores}} \rfloor \times i$ to find a block group that can accommodate the requested item. The data block allocation for a file is served in the same way regardless of the file size.

This simple core-aware block group allocator is proposed for analyzing the benefit from reduced false sharing, not for production use, and does not consider the long-term consequences from the core-partitioned distribution of allocated blocks and the interactions with other performance-sensitive factors. The in-depth research on core-aware or sharing-aware block group allocators is beyond the scope of this paper.

		Specification
Processor	Model	Intel Xeon Gold 6138 × 4 sockets
	Number of Cores	20 × 4
	Clock Frequency	2.00 GHz
Memory		DDR4 2666 MHz 32GB × 16
Storage		Samsung SZ985 NVMe SSD 800GB
OS	Kernel	Linux 4.14.78

Table 1: System configurations for evaluation.

4 Evaluation

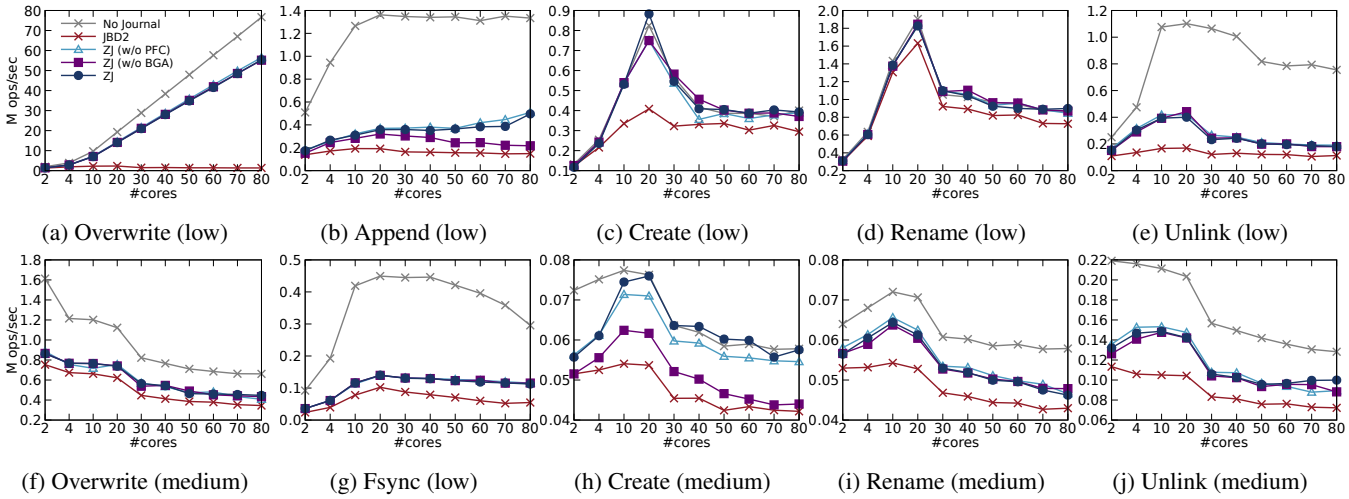
In this section, we evaluate Z-Journal to verify its performance and scalability for various file system operations under different sharing conditions. In addition, We also analyze the overhead and benefit of the proactive frozen copy scheme and Z-Journal's `fsync` handling mechanism. Finally, we show the overall file system performance and scalability of Z-Journal for benchmarks imitating real-world workloads.

4.1 Evaluation Environment

Table 1 shows the system configurations used for the evaluation. We implemented Z-Journal in the Linux kernel[2] and modified the ext4 file system to recognize per-core journals and to use Z-Journal instead of JBD2. We also modified the block group allocator of ext4 as described in Section 3.5. In addition, we modified `mke2fs` to format an ext4 file system to have multiple journals. The ext4 file system was modified so that the super block can have multiple journal control structures and the mount operation recognizes them.

We compared the performance of Z-Journal (denoted as ZJ on the graphs) with ext4 with JBD2 (denoted as *JBD2* on the graphs), and ext4 without journaling (denoted as *no-journal* on the graphs). Because we are not aiming at the scalability of the overall file system, the performance of ext4 without journaling can be considered the best possible value Z-Journal can achieve. We also measured the performance of Z-Journal without proactive frozen copy (denoted as *w/o PFC* on the graphs), and without core-aware block group allocator

[2]The source code of the Z-Journal-patched Linux kernel is available at https://github.com/J-S-Kim/journal



(a) Overwrite (low) (b) Append (low) (c) Create (low) (d) Rename (low) (e) Unlink (low)

(f) Overwrite (medium) (g) Fsync (low) (h) Create (medium) (i) Rename (medium) (j) Unlink (medium)

Figure 6: Throughput of FxMark write workloads while varying number of cores under different sharing conditions.

(denoted as *w/o BGA* on the graphs). All experiments were conducted in the data=journal mode [18] (The results obtained in the data=ordered mode are also presented in the appendix).

4.2 Scalability of File System Operations

To assess the scalability of Z-Journal when performing file operations under various block sharing conditions, we used FxMark [16]. The write workloads of FxMark consists of overwrite, append, fsync, create, unlink, and rename operations executed in the low and medium sharing level, respectively. In the low sharing condition, each core performs the target file system operation for private files in its dedicated directory. In the medium sharing level, the overwrite workload lets all cores access the same file, and the other workloads perform the given operation in the same directory. The append and fsync workloads provide only the low sharing level mode. Figure 6 shows the experiment results.

The favorable condition for Z-Journal against JBD2 is where the file system scalably performs, but the resulting performance is poor for the serialization at the journal layer. overwrite (low) barely modifies metadata and frequently writes to data blocks of files stored in non-shared per-core directories. Therefore, it is the most favorable workload in FxMark.

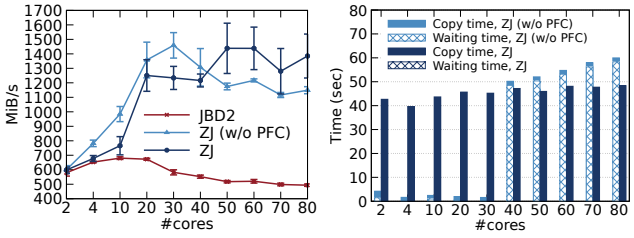
Z-Journal showed a close performance to no-journal for overwrite (low) excluding the slow down from the double writing overhead, and its performance scaled well to the number of cores. Z-Journal showed 41 times higher throughput in comparison to JBD2 at 80 cores. However, in the case of overwrite (medium), the file system scalability was poor due to sharing, and the performance of the journal layer was also poor for the same reason. Even in this case, Z-Journal gained 30% of performance improvement at 80 cores compared to JBD2 through its parallel journaling mechanism.

In the case of append (low), the difference between no journal and journal group was very large. The metadata manipulation overhead without journaling is very low because delayed allocation is possible when allocating a new data block. Z-Journal without BGA scaled gently up to 20 cores, but performance decreased after that. This was due to increased false sharing caused by the current block group allocator. When the core-aware block group allocator was used, the performance steadily increased up to 80 cores. Z-Journal showed 3.34 times the performance of JBD2 at 80 cores.

For create (low) and rename (low), the journal's scalability bottleneck was not revealed due to the scalability problem of the ext4 file system [16]. However, since commits were performed in parallel, there were 33 % and 24 % performance improvements at 80 cores, respectively. In terms of create (medium), Z-Journal without BGA obtained performance improvement of up to 15 % at 10 cores and about 4 % at 80 cores. Z-Journal showed similar performance to no-journal. Because it allocated block groups for each core, although the files were in the same directory, metadata sharing was greatly reduced, and this leads to improved performance. This performance gain was mostly from the file system, not from the journal layer. This result shows that the block group allocator in a file system is one of the major obstacles to scalability.

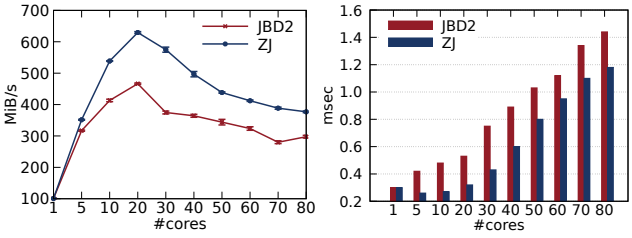
In the case of unlink, no-journal also showed scalability characteristics similar to create and rename. The big difference in performance between no-journal and the journaling group is from the heavy checkpointing operations occurring in the measurement interval caused by the preparation stage, in which the large file set was created, and their data blocks were written. For unlink (medium), Z-Journal achieved the maximum performance improvement of 42 % at 10 cores, and 38 % at 80 cores compared to JBD2.

In the fsync workload, where all cores write to their files and call `fsync`, Z-Journal also showed an average throughput improvement of 70 % in comparison to JBD2. When a core



(a) Throughput (b) Copy and waiting time

Figure 7: Throughput and journaling delay of SysBench.



(a) Throughput (b) 99th percentile latency

Figure 8: Throughput of `fsync`-invoking SysBench and tail latency of `fsync` operations observed.

called `fsync` in Z-Journal, all cores were suspended because they had to commit their running transactions together. However, the commit of each core in Z-Journal was a lot faster than that in JBD2 because per-core kjournald could commit in parallel while a single kjournald had to process alone.

Z-Journal showed a scalability pattern similar to that no-journal when the file system's scalability was good in a low sharing situation. Even when the scalability of the file system was poor, its parallel committing scheme achieved superior performance compared to JBD2. However, in the workloads where shared writes and metadata updates frequently occurred, the performance gain was diminished due to the validity check procedure during the checkpoint process.

In the experiment for each file operation, most of the performance was better when proactive force copy was not applied. Only, low sharing create and medium sharing create achieved an average of 4 % and 5 % improvement by applying proactive force copy, respectively. It was because FxMark's write workloads mainly produced metadata sharing and not much file sharing. If write accesses to the shared block do not occur frequently, proactive force copy will generate only meaningless overhead.

4.3 Analysis of Design Components

We used SysBench [12] to analyze the performance of Z-Journal and the impact of proactive force copy when data block write sharing frequently occurs. SysBench performs random overwrites on files in the file set created in advance. Therefore, when the number of cores increases, the occurrences of metadata block and data block sharing increase as well. We configured SysBench to randomly overwrite 4 KB data over 80 files, each of which has 1 GB size.

Figure 7a shows the throughput measured while increasing the number of cores, and Figure 7b shows the sum of the journal waiting time and the time to create frozen copies in Z-Journal and Z-Journal without proactive frozen copy, respectively. The waiting time refers to the time for a running transaction to wait for the committing transaction having the shared buffers to finish.

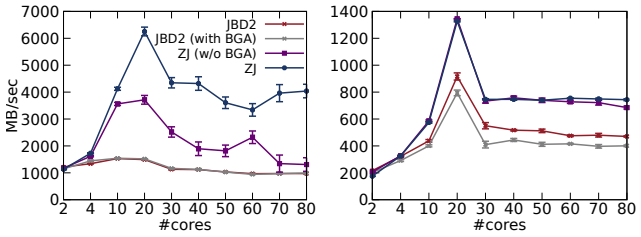
Both proactive frozen copy and waiting for shared buffers can be carried out by kjournald as well as user-level threads writing to files. However, proactive frozen copy is mostly performed by kjournald in the background. Because it is off from the write path, the proactive frozen copy has little impact on the user-perceived performance. At 50 cores in Figure 7b, the copy time in Z-Journal and the waiting time in Z-Journal without PFC are similar, but unlike Z-Journal without PFC waiting in the write path, proactive frozen copies are mostly conducted by kjournald. Therefore, Z-Journal showed a 22 % better performance than Z-Journal without PFC.

The sudden increase in waiting time at 50 cores in Figure 7b was because, as mentioned in Section 3.1, long chains of dependent transactions were created. On the other hand, the copy time did not significantly increase even though the throughput of Z-Journal increased. This is because SysBench writes to the same file set repeatedly, and therefore, the number of shared buffers is limited.

As a result, as shown in Figure 7a, up to 40 cores Z-Journal without PFC outperformed Z-Journal by an average of 13 %. However, after 50 cores, proactive frozen copy improved the average throughput by 19 %. Based on this observation, we conclude that the proactive frozen copy scheme should be applied adaptively to the degrees of parallelism and cross-core file sharing.

To assess the throughput and latency of `fsync` on Z-Journal, we altered SysBench so that it calls `fsync` once every 100 write operations, and measured its throughput and 99th percentile tail latency of the `fsync` operation. Considering that `fsync` is intensively performed while cores are writing on a shared data set, this can be regarded as a notably hostile condition for Z-Journal.

As shown in Figure 8a, Z-Journal still performed better than JBD2 at all core counts. However, as the number of cores increased, unlike the results of the original SysBench, the throughput of Z-Journal also decreased similar to JBD2 because the validity check overhead of kjournald offset a significant part of the performance improvement earned from parallelized journaling. However, as shown in Figure 8b, the tail latency of `fsync` on Z-Journal was significantly better than that on JBD2. It was 18 % shorter at 80 cores even though the difference between JBD2 and Z-Journal narrowed as the number of cores increased as expected in Section 3.2.



(a) fileserver (b) varmail

Figure 9: Throughput of Filebench workloads.

4.4 Overall File System Performance

To assess the performance impact of Z-Journal in the environment, which has a mixture of reads, writes, and metadata operations, we used the *fileserver* and *varmail* workloads of Filebench [21,23]. fileserver repeatedly conducts create, write, append, read, and deletion operations over a file set of relatively large files. varmail also continually performs create, read, append, and fsync operations over a file set of small files.

Figure 9a shows the throughput of fileserver. Since a large number of cores repeatedly created and updated files in a small number of directories, the throughput of JBD2 peaked at 10 cores and decreased as the number of cores increased. Z-Journal showed significantly better performance than JBD2 in every case. However, in the absence of BGA, false sharing increased rapidly with the increase in the number of cores, and the gap between the throughput of Z-Journal and that of JBD2 gradually shrunk as the number of cores increased. BGA significantly reduced false sharing and enabled Z-Journal to achieve 315 % performance improvement. When BGA was applied to JBD2, no throughput improvement was observed. This tells that the scalability of JBD2 was not limited only by metadata sharing. Z-Journal performed best at 20 cores, and the performance gently declined beyond that point due to the NUMA effect [7] resulting from the rapid increase in remote memory access across domains.

Figure 9b shows the throughput of varmail. varmail also showed a similar throughput change pattern with fileserver, and Z-Journal improved performance by 58 % at 80 cores. However, unlike fileserver, varmail frequently called `fsync`, and the performance improvement for `fsync` by Z-Journal is smaller than that for other file operations. Therefore, the performance gap between JBD2 and Z-Journal was smaller than that in fileserver. The effect of false sharing reduction from applying BGA was mostly concealed by the frequent suspension for processing `fsync`. As a result, BGA could obtain only 9 % performance improvement in comparison to Z-Journal without BGA. Rather surprisingly, when BGA was applied to JBD2, the performance deteriorated by up to 26 %. This was because the number of committed blocks was significantly amplified from reduced metadata sharing.

5 Related Work

Min et al. studied the scalability of file systems on manycore systems and left insights to design scalable file systems [16].

SpanFS [8] achieved scalability by partitioning files and directories into multiple domains. While this partitioning enables parallel write operations to each domain, careful data partitioning is important to achieve scalability. When two cores each write to two files in the same domain, SpanFS needs to serialize the write operations although they are independent of each other. In contrast, Z-Journal does not require explicit data partitioning and minimizes the serialization of journaling operations.

IceFS [15] also partitioned files and directories for performance isolation but its main goal was not multi-core scalability since multi-threads on a single partition (called a cube) can cause scalability bottleneck.

ScaleFS [2] decouples an in-memory file system from an on-disk file system and allows scalable file system operations since highly concurrent data structures are used in the in-memory file system. File system modifications are flushed to on-disk file system structures through per-core parallel journaling. However, the dual file system approach with the integrated journal layer of ScaleFS cannot be applied to the existing file systems. For example, journaling in ScaleFS can have multiple images of a buffer in memory by logging changes to the buffer per-core. This allows parallel journaling. However, in ext4 and most other conventional file systems, a buffer in memory must be unique and up-to-date. Z-Journal maintains this invariant while allowing parallel commits.

A few studies have addressed the scalability bottleneck caused by coarse-grained locking during write operations. They proposed to use a fine-grained range lock to increase the concurrency of write operations [10, 14]. The use of a file-grained lock is complementary to our approach.

iJournaling [17] proposed to use a per-file journal transaction for fsync. Accordingly, only necessary blocks need to be flushed to reduce the fsync latency. Since the per-file transaction uses a logical logging scheme, concurrent fsync processing can be possible. However, it also maintains the original file system journaling, causing serialization during write operations.

Son et al. [19] proposed to improve the concurrency of JDB2 by aggressively using concurrent and lock-less data structures and multi-threaded journal writes. While their approach enhanced the concurrency of journaling, the inherent serialization from committing to a single journal area remains.

BarrierFS [24] improved the concurrency of journaling by allowing multiple committing transactions in a journal file, which improves the concurrency of journaling. However, it has a limited concurrency due to waiting-based dependency handling, which is eliminated by Z-Journal's journal coherence mechanism.

The journaling mechanism for block devices is unsuitable

for NVM due to the write amplification of metadata journal. Chen et al. [3] proposed a fine-grained metadata journal mechanism optimized for journaling in NVM aiming at reduction in write amplifiction.

Sul et al. [20] proposed an NVM-optimized journaling scheme in which the use of parallel journals is similar to our proposal. However, its journaling operation aggressively exploits NVM's byte-addressable characteristic, which hinders its application to block devices.

Koo et al. [11] analyzed blocking of I/O operations due to the transaction in the locked state. To resolve this issue, they proposed a transaction lock-up elimination scheme that optimizes the transaction commit procedure.

6 Conclusion

Most modern file systems use the journal to guarantee crash consistency. However, because conventional journaling schemes are performed serially from transaction generation in memory to journal commit on disk, it acts as a serious scalability bottleneck when file system operations are run simultaneously in many cores.

In this paper, we proposed Z-Journal, a scalable journal design using per-core journals, which retains the interface of JBD2. Z-Journal includes a journal coherence mechanism, which provides complete parallelism for unshared buffer modification and guarantees crash consistency by order-preserving transaction chaining for shared buffer modification.

Our evaluation showed that Z-Journal achieves a steady increase in throughput in journal-bottlenecked workloads, showing up to approximately 4000 % improvement in comparison to JBD2. It also showed 29% throughput improvement on average even under unfavorable conditions by allowing parallel journaling.

Acknowledgements

We thank the anonymous reviewers and our shepherd, Rusty Sears, for their valuable suggestions for improving this paper.

This research was supported by Samsung Electronics, and by the Institute of Information and Communications Technology Planning and Evaluation funded by the Ministry of Science and ICT (MSIT), Korean Government, (Research on High Performance and Scalable Manycore Operating System) under Grant 2014-3-00035.

References

[1] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*, chapter 41, page 4. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[2] Srivatsa S Bhat, Rasha Eqbal, Austin T Clements, M Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*, pages 69–86. ACM, 2017.

[3] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Fine-grained metadata journaling on NVM. In *32nd Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–13. IEEE, 2016.

[4] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 266–277, 2011.

[5] Kevin D Fairbanks. An analysis of Ext4 for digital forensics. *Digital investigation*, 9:S118–S130, 2012.

[6] Mark Fasheh. OCFS2: The Oracle clustered file system, version 2. In *Proceedings of the 2006 Linux Symposium*, volume 1, pages 289–302. Citeseer, 2006.

[7] Fabien Gaud, Baptiste Lepers, Justin Funston, Mohammad Dashti, Alexandra Fedorova, Vivien Quéma, Renaud Lachaize, and Mark Roth. Challenges of memory management on modern NUMA system. *Queue*, 13(8):70–85, 2015.

[8] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: a scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference (ATC)*, 2015.

[9] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. A scalable ordering primitive for multicore machines. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[10] June-Hyung Kim, Jangwoong Kim, Hyeongu Kang, Chang-Gyu Lee, Sungyong Park, and Youngjae Kim. pNOVA: Optimizing shared file I/O operations of NVM file system on manycore servers. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSYS)*, pages 1–7, 2019.

[11] Kyoungho Koo, Yongjun Park, and Youjip Won. LOCKED-Free journaling: Improving the coalescing degree in EXT4 journaling. In *Proceedings of IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2020.

[12] Alexey Kopytov. SysBench: A system performance benchmark, 2004.

[13] Aneesh Kumar KV, Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Linux Symposium*, volume 1, 2008.

[14] Chang-Gyu Lee, Hyunki Byun, Sunghyun Noh, Hyeongu Kang, and Youngjae Kim. Write optimization of log-structured flash file system for parallel I/O on manycore servers. In *Proceedings of the 12th ACM International Conference on Systems and Storage (SYSTOR)*, pages 21–32, 2019.

[15] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Physical disentanglement in a container-based file system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 81–96, 2014.

[16] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. Understanding manycore scalability of file systems. In *2016 USENIX Annual Technical Conference (ATC)*, 2016.

[17] Daejun Park and Dongkun Shin. iJournaling: Fine-grained journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (ATC)*, 2017.

[18] Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track*, volume 194, pages 196–215, 2005.

[19] Yongseok Son, Sunggon Kim, Heon Y Yeom, and Hyuck Han. High-performance transaction processing in journaling file systems. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[20] Woong Sul, Kihwang Kim, Minsoo Ryu, Hyungsoo Jung, and Hyuck Han. Fast journaling made simple with NVM. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing (SAC)*, pages 1214–1221, 2020.

[21] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

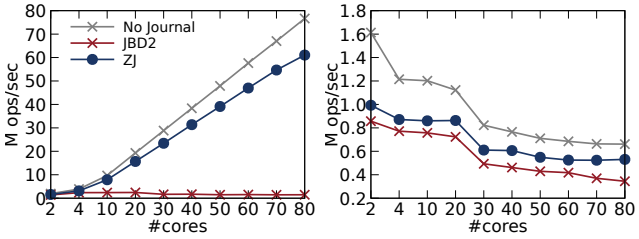
[22] Stephen C Tweedie et al. Journaling the Linux ext2fs filesystem. In *The Fourth Annual Linux Expo*. Durham, North Carolina, 1998.

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

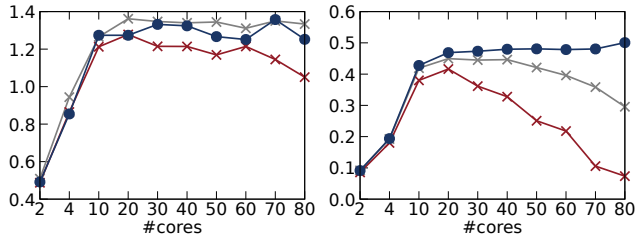
[24] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled IO stack for flash storage. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

Appendix

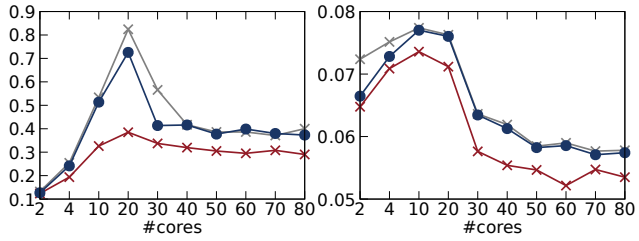
We also evaluated Z-Journal in the data=ordered mode. The performance gain from Z-Journal was less in the ordered mode than in the journal mode because the amount of buffers written to the journals was significantly smaller in the ordered mode. However, the patterns of performance improvement from applying Z-Journal in the ordered mode were similar to that in the journal mode. Figure 10 and Figure 11 show the ordered mode counterparts of Figure 6 and Figure 9, respectively. These are not included in Section 4 for the readability of the graphs and the limited space.



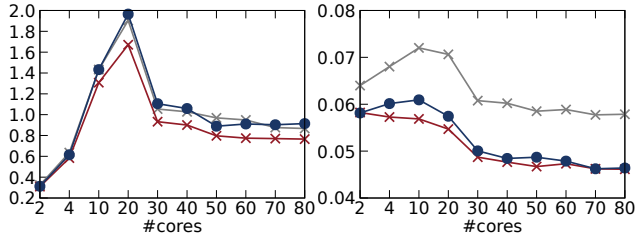
(a) Overwrite (low) (b) Overwrite (medium)



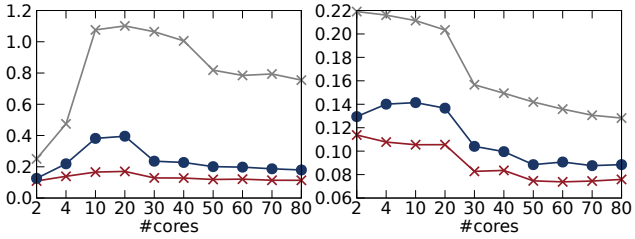
(c) Append (low) (d) Fsync (low)



(e) Create (low) (f) Create (medium)

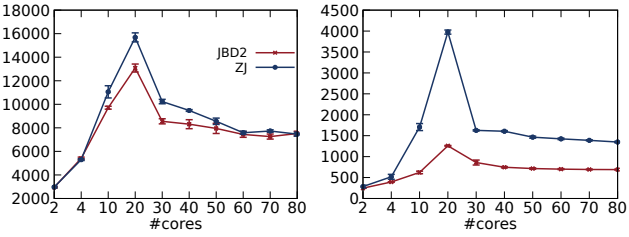


(g) Rename (low) (h) Rename (medium)



(i) Unlink (low) (j) Unlink (medium)

Figure 10: Throughput of FxMark write workloads while varying number of cores under different sharing conditions.



(a) fileserver (b) varmail

Figure 11: Throughput of Filebench workloads.