CJFS: Concurrent Journaling for Better Scalability

Joontaek Oh, Seung Won Yoo, and Hojin Nam, *KAIST;*
Changwoo Min, *Virginia Tech;* Youjip Won, *KAIST*

https://www.usenix.org/conference/fast23/presentation/oh

This paper is included in the Proceedings of the
21st USENIX Conference on File and
Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings
of the 21st USENIX Conference on
File and Storage Technologies
is sponsored by

CJFS: Concurrent Journaling for Better Scalability

Joontaek Oh* Seung Won Yoo* Hojin Nam* Changwoo Min † Youjip Won *
*KAIST †Virginia Tech

Abstract

In this paper, we propose CJFS, *Concurrent Journaling Filesystem*. CJFS extends EXT4 and addresses the fundamental limitations of the EXT4 journaling design, which are the main cause of the poor scalability of EXT4. The heavy-weight EXT4 journal suffers from two limitations. First, the journal commit is a strictly serial activity. Second, the journal commit uses the original page cache entry, not the copy of it, and subsequently any access to the in-flight page cache entry is blocked. To address these limitations, we propose four techniques, namely Dual Thread Journaling, Multi-version Shadow Paging, Opportunistic Coalescing, and Compound Flush. With Dual Thread design, CJFS can commit a transaction before the preceding journal commit finishes. With Multi-version Shadow Paging, CJFS can be free from the transaction conflict even though there can exist multiple committing transactions. With Opportunistic Coalescing, CJFS can mitigate the transaction lock-up overhead in journal commit so that it can increase the coalescing degree – *i.e.*, the number of system calls associated with a single transaction – of a running transaction. With Compound Flush, CJFS minimizes the number of flush calls. CJFS improves the throughput by 81%, 68% and 125% in filebench `varmail`, `dbench`, and `OLTP-Insert` on `MySQL`, respectively, against EXT4 by removing the transaction conflict and lock-up overhead.

1 Introduction

Filesystem scalability gets emphasized further as the computer system is loaded with hundreds of CPU cores [6,7,17,22,28,32,33,35]. A single server machine can simultaneously run hundreds of containers [2,43,50], each of which may frequently synchronize its local filesystem state to the disk [13,31]. The throughput of the server machine hinges upon the scalability of the filesystem journaling of the host filesystem.

In this paper, we address the scalability issue of the EXT4 journaling. EXT4 journaling uses page granularity physical logging [47,48]. EXT4 journaling suffers from two critical drawbacks; serial commit and committing the original page cache entry. In EXT4, journal commit is strictly serial activity. It can commit the following journal transaction only after the preceding journal commit finishes. As a result, in EXT4, there can be at most one running transaction and at most one committing transaction at a time. Moreover, EXT4 uses the original page cache entry in committing the updated contents to the disk. It does not create a copy of the updates for the journal commit. In this paper, we address both issues and propose a new filesystem, CJFS, by *Concurrent Journaling Filesystem*.

A fair amount of works have been dedicated to addressing the scalability issue of the filesystem journaling [6,8,15,17,24,44]. A few works proposed to maintain the multiple running transactions in EXT4 so that contention on the global running transaction is mitigated. ScaleFS [15] allocates a running transaction per each CPU core [6,8], where each core is allocated the separate filesystem partition. Son et al. [44] adopts a lock-free data structure to the journal transaction. Another body of works proposed to maintain the multiple committing transactions. IceFS [24] and SpanFS [15] partition the filesystem into multiple regions. They allocate a separate journal area for each region. The journal commit operations to each journal area can proceed in parallel.

Despite all the sophisticated approaches mentioned above, these variants of EXT4 journaling still fail to address the fundamental limitations of the EXT4 journaling; serial commit and committing the original page cache entry. In these works, the journal commit operations for the same journal region are still serialized [15,24]. Multiple running transactions and multiple committing transactions can conflict with each other and if the concurrent transactions conflict with each other, they are serialized.

In this paper, we address the fundamental limitations of the EXT4 journaling mechanism; serial commit and using the original page cache entry in the journal commit. The contribution of CJFS can be summarized as follows:

- **Dual Thread Journaling**: We separate the journal commit operation into two separate tasks; transferring the log blocks to the disk and making them durable. We allocate a separate thread for each operation. With Dual Thread Journaling, CJFS can commit a transaction while the preceding journal commit is still in progress.
- **Multi-Version Shadow Paging**: CJFS adopts multi-version shadow paging to resolve the transaction conflict. With multi-version shadow paging, CJFS uses the "copy" of the updated page cache entry in journal commit, so the transaction is free from the transaction conflict.
- **Opportunistic Transaction Coalescing**: CJFS adopts *opportunistic coalescing* to mitigate the trans-

action lock-up overhead. To increase the compound degree of the journal transaction, CJFS releases the running transaction from the LOCKED state when it finds that the running transaction conflicts with one of the committing transactions.

- **Compound Flush**: CJFS creates a large number of flush commands since it creates the multiple committing transactions in-flight, each of which issues a flush command separately to make its journal transaction durable. To relieve the overhead of servicing the flush commands, CJFS compounds multiple consecutive flushes from the concurrent transactions into a single flush. Compound flush significantly reduces the latency of the individual `fsync()` calls.

We implement the CJFS in Linux 5.18.18. CJFS yields superior performance not only to Vanilla EXT4 but also to the other recent works including BarrierFS [49], SpanFS [15] and FastCommit [42] in `varmail`, `dbench` and `OLTP-Insert` workloads.

2 Background and Motivation

2.1 Journaling in EXT4

Block granularity physical logging. A journaling filesystem logs the updated metadata either in a block granularity, e.g. EXT4 [36] or in a metadata granularity, e.g. XFS [45]. *Physical block granularity logging* of EXT4 is not only expensive but also unable to scale.

EXT4 maintains a set of page cache entries that need to be logged to the disk for journaling. It is called *running transaction*. When the system call updates the filesystem metadata, it first acquires a lock on the associated kernel object (e.g., directory mutex) and obtains the journal handle. A journal handle is a kind of ticket-like permission to add page cache entries to the running transaction. After the application is granted with the lock and the journal handle, the application modifies page cache entries. After modifying page cache entries, it inserts the updated page cache entries to the running transaction.

EXT4 commits the running transaction either periodically or by the explicit request from the application, e.g., `fsync()`. EXT4 commits the original page cache entry of the modified data. The application that needs to update the filesystem state is blocked if the associated page cache entry is being committed to the disk. We call this situation *transaction conflict* described in §2.2. Block granularity logging leaves the EXT4 journaling under frequent transaction conflict and subsequently under scalability failure.

Serial journal commit. In EXT4, journal commit is strictly serial activity. EXT4 allocates a separate thread for journal commit, *JBD thread*. JBD thread can commit the following journal transaction only after the preceding journal transaction becomes durable.

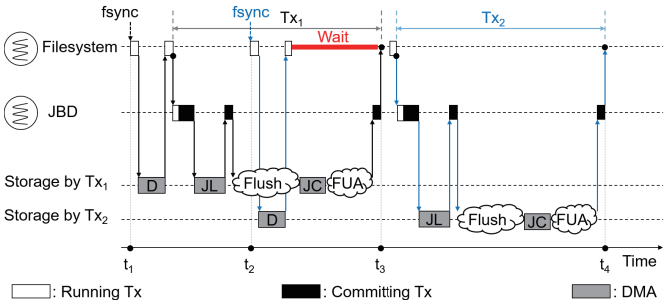


Figure 1: `fsync()` in EXT4

We illustrate the behavior of an `fsync()`. Let `D`, `JL`, and `JC` be file dirty data pages, journal log blocks and journal commit block. When an application thread calls `fsync()`, it writes the dirty data pages (`D`) to the storage, and then it wakes up the JBD thread. The JBD thread changes the transaction state from *running* to *committing* and writes the log blocks (`JL`) to the storage. Once all the log blocks are transferred (*i.e.*, DMA-ed) to the storage, the JBD thread writes the journal commit block (`JC`) to the storage with `REQ_PREFLUSH` and `REQ_FUA` [1] flags to ensure that the journal commit block is made durable only after the dirty data pages and the journal log blocks do. `REQ_PREFLUSH` flag instructs the storage controller to flush the writeback cache in a storage device before servicing the associated write command. `REQ_FUA` command writes the associated data block directly to the storage media bypassing the writeback cache of the storage. Once the write command for commit block returns, the JBD thread finishes committing a transaction.

Figure 1 illustrates the timing diagram of servicing two consecutive `fsync()`'s. The first `fsync()` and the second `fsync()` are issued at t_1 and at t_2, respectively. We mark the journal transactions for the preceding `fsync()` and the following `fsync()` as $\texttt{Tx}_1$ and $\texttt{Tx}_2$, respectively. JBD thread starts committing $\texttt{Tx}_2$ (at t_3) only after it finishes committing $\texttt{Tx}_1$.

2.2 Concurrency Control in Filesystem Journaling

To partly address the drawback of serial journal commit, EXT4 journaling adopts compound journaling and the shadow paging to increase its concurrency. The compound journaling commits multiple file operations with a single journal commit. The shadow paging allows the file operation and the journal commit operation to proceed in parallel while they share the same page cache entry. Compound journaling inevitably accompanies transaction lock-up phase when it needs to commit the running transaction. Journal commit operation should exclusively lock the page cache entry when it needs to write the page cache entry to the storage. Transaction lock-up and

page granularity exclusive locking temporarily blocks the file operations and can severely interfere with the overall system performance. Let us explain the details of individual phases of journaling.

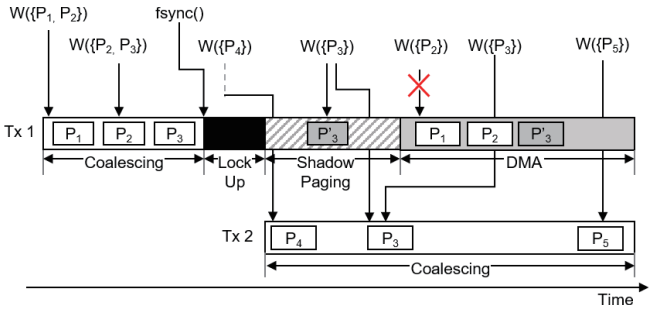


Figure 2: Dissection of EXT4 journaling phases

(i) Coalescing in Running Transaction. In coalescing phase, the application can modify the metadata and insert the associated page cache entry to the running transaction. EXT4 journaling adopts compound transaction which is also known as a group commit [12] to increase the throughput. In Figure 2, a file operation modifying page cache entries P_1 and P_2 and another file operation modifying page cache entries P_2 and P_3, sharing the commonly modified page cache entry P_2. EXT4 creates a compound transaction of P_1, P_2, and P_3.

(ii) Transaction Lock-Up in Running Transaction. When the JBD thread needs to commit the running transaction, the JBD thread stops issuing the journal handle to prohibit the new file operation to modify the running transaction. Then, it waits for the outstanding file operations which already have a journal handle to finish. Otherwise, starting the journal commit can be postponed indefinitely. When all file operations which already have a journal handle finish, JBD thread changes the transaction state from running to committing. We call this time period during which the JBD thread stops issuing the journal handle as *transaction lock-up* phase. Any file operations that update the metadata are blocked when the running transaction is in lock-up phase. In Figure 2, during the transaction lock-up phase, the file operation that modifies P_4 is blocked. The file operation wakes up when the lock-up phase of Tx_1 is released and adds P_4 to the new running transaction, Tx_2.

(iii) Shadow Paging in Committing Transaction. After the transaction state is changed to committing, JBD thread prepares the page cache entries for DMA transfer. EXT4 journaling adopts *very limited* form of *Shadow Paging* to handle the transaction conflict during this time interval. It allows only one shadow page and does not allow more multiples versions. When there occurs transaction conflict when the JBD thread prepares the page cache entries for DMA transfer, JBD thread creates the shadow copy of the conflict page and uses a shadow copy of the original page cache entry for DMA transfer [47]. With shadow paging, a file operation can modify the original page cache entry in the committing transaction without waiting for the completion of the transaction commit. EXT4 journaling can create up to only one shadow page. If two or more transactions attempt to update the same page, only one can proceed and the others are blocked until the associated page is committed to the storage. In Figure 2, during the shadow paging phase, the file operation tries to modify P_3, which is in the committing transaction, Tx_1. The file operation creates the shadow page, P'_3, and adds the original page cache entry, P_3 to the new running transaction, Tx_2.

(iv) DMA in Committing Transaction. When the log block of the committing transaction is transferred to the storage (DMA), the host establishes an exclusive lock on the associated page cache entry. This is to prohibit the file operation from blindly updating the page cache entry of the committing transaction that is being transferred and from migrating it to the running transaction compromising the atomicity of the journal commit. During DMA phase, any file operations that update the locked page cache entry are blocked. In Figure 2, the DMA phase is marked in gray. While a compound transaction of P_1, P_2, and P'_3 is under DMA, an attempt to modify page P_2 will be blocked. Because P'_3 is the shadow page and the shadow page is being transferred, the file operation which modifies P_3 is not blocked and modifies the original page cache entry, P_3. An attempt to modify a page cache entry, P_5, which is not in a committing transaction will successfully add the page to the new running transaction, Tx_2.

2.3 Existing Solutions to Scale Journaling

A number of approaches have been proposed to increase the concurrency in the filesystem journaling [15, 17, 24, 32, 45, 49]. They can be categorized with respect to the number of threads that are used to handle a single journal commit; (i) single-threaded journal commit and (ii) multi-threaded commit. They also can be categorized with respect to how they allocate the journal transaction in the filesystem; per-core basis or per-region basis. Table 1 summarizes the approaches in the existing scalable filesystem journaling techniques.

Filesystems	Concurrent Transactions		Multi-Threaded Commit
	Per-core	Per-region	
Z-journal [17]	○	○	
SpanFS [15]		○	
IceFS [24]		○	
MQFS [23]		○	
BarrierFS [49]			△
XFS [45]			○
iJournaling [32]	○		
ScaleFS [6]	○		

Table 1: Categories of existing scalable filesystems

In the concurrent transaction approach, the journaling filesystem allows multiple running transactions, multiple committing transactions or both, to proceed in parallel. In per-core basis approach, they allocate the transaction for each CPU core (per-core basis) [32] . In per-region basis approach, filesystem is partitioned into multiple regions and allocates dedicated journal area for each filesystem region [15, 17, 24]. In per-region basis approach, the journaling filesystem maintains the running transaction and/or committing transaction in per-region basis. The filesystem can commit the multiple transactions concurrently for each filesystem region. Per-region approach requires changing the on-disk layout of the existing filesystem partition [15, 17, 32]. In per-core approach, the transactions may conflict with each other, i.e. they modify the same page cache entry. Resolving the transaction conflict accompanies substantial overhead, e.g. [17] compromises `fsync()` durability or journal commit is subject to excessive tail latency [15].

In multi-threaded journal commit approach, the filesystem divides a journal commit operation into multiple phases and allocates the separate threads for handling each of the journal commit phases. With this multi-threaded organization, a thread can start processing the following journal transaction before the preceding journal commit finishes in pipelined manner. In XFS, one thread is responsible for making the journal transaction durable and the other thread is responsible for ensuring that all preceding transactions are durable after the journal transaction becomes durable [16]. In BarrierFS, one thread is responsible for issuing the IO requests for journal commit, and the other thread is responsible for making the journal transaction durable [49]. Both XFS and BarrierFS can start a new journal commit without waiting for the preceding journal commit to finish. In BarrierFS, the journal commit operation is serialized in most cases due to frequent transaction conflict.

To ensure the storage order between the log blocks and the journal commit block in committing a journal transaction, the filesystem interleaves the write requests for the log blocks and the write request for commit block with the FLUSH command. Recently, a number of works have been proposed order-preserving IO stack to mitigate the FLUSH overhead associated with ensuring the storage order in journal commit operation [9, 21, 23, 49]. The order-preserving IO stack consists of order-preserving block layer [23, 49] and order-preserving FTL [9, 21, 49]. Order-preserving FTL can be implemented via exploiting the cache-barrier command [49], via imposing a global sequence number on the IO commands [9] or via exploiting non-volatile cache at SSD [21]. These works show that order-preserving FTL can be realized without substantial overhead and renders the identical performance as legacy FTL.

3 Scalability of EXT4 Journaling

3.1 Workloads

We used four filesystem macro benchmarks – two variants of varmail (`varmail-shared` and `varmail-split`) in `filebench` [26], `dbench` [46], and `OLTP-Insert` [19] – to cover wide variety of real-world application behaviors. Each benchmark has a different mix of file operations (Table 2) and stresses various parts of the filesystem (Table 3).

Benchmarks	`create()`	`unlink()`	`write()`	`read()`	`fsync()`	`rename()`
`varmail`	7.7%	7.7%	15.4%	15.4%	15.4%	0%
`dbench`	16.6%	3.5%	8.6%	27.1%	5.2%	0.7%
`OLTP-Insert`	0%	0%	77.8%	12.2%	10.0%	0%

Table 2: Ratio of filesystem operations in benchmarks

Benchmarks	Directory contention	In-memory logging	On-disk logging
`varmail-shared`	High	Moderate	High
`varmail-split`	No	Moderate	High
`dbench`	No	Moderate	Moderate
`OLTP-Insert`	No	low	low

Table 3: Filesystem contention in benchmarks

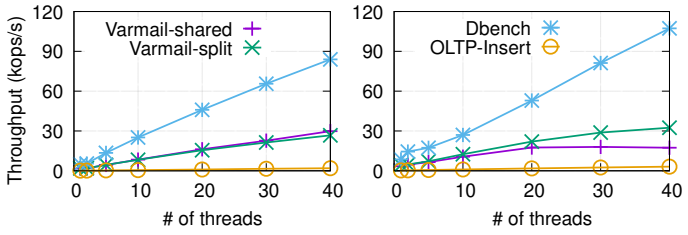
Mail server: varmail [26]. The varmail benchmark simulates the behavior of mail server. In the varmail workload, each thread repeats a set of `create()`, `unlink()`, and `fsync()` operations. Varmail is known for intensive `fsync()` calls. In the original `varmail` workload, all threads share the same directory yielding the lock contention on the shared directory. We call it `varmail-shared`. We modify the varmail workload so that each thread works on its own directory. We call it `varmail-split`. We use `varmail-split` how the filesystem journaling scales in the absence of the contention on the shared directory.

File server: dbench [46]. The dbench simulates the behavior of the fileserver. It is metadata-intensive workload calling `unlink()` and `rename()` followed by `fsync()` (with a `-sync-dir` option enabled). In `dbench`, `fsync()` calls account for 5.2% of all filesystem calls. Dbench calls `read()` and `write()` with various IO sizes; 4KB IO accounts for 60% of the `read()` and `write()`.

OLTP: OLTP-Insert on MySQL [19] OLTP-Insert simulates the server for online transaction processing. In this workload, `write()` followed by `fsync()` is frequently invoked. The write size ranges from 8KB to 32KB; 8KB write accounts for 81%. Among the four workloads, the contention (or transaction conflict) degree of this workload is the lowest. We use this workload to test the behavior of journaling under the circumstances that there is only little contention (or transaction conflict).

3.2 Scalability Results

We compare the performance of the four benchmarks under EXT4 and BarrierFS [49]. BarrierFS is the variant



(a) EXT4 (b) BarrierFS

Figure 3: Scalability of EXT4 and BarrierFS

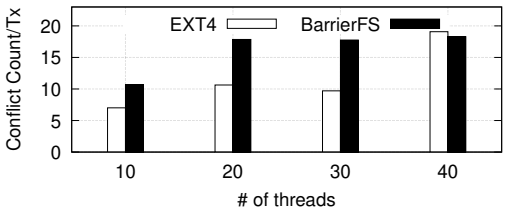


Figure 4: The average number of conflicts in a transaction (EXT4 and BarrierFS, `varmail-shared` workload)

of EXT4 which can commit multiple transactions concurrently. We used two SSDs – Samsung 860 Pro (MLC Flash, SATA interface) and 970 Pro (MLC Flash, NVMe interface) in this experiment. However, we omitted the results with 860 Pro since the performance trends on these two SSDs are almost identical. Please refer to §6.1 for the details of our evaluation setup.

As Figure 3 shows, the performance and scalability of both filesystems get worse as `fsync()` accounts for more dominant fraction of the entire system calls. The `dbench` which renders the least significant `fsync()` calls is the most performant and scalable.

As shown in Figure 3(b), BarrierFS increases the performance of `dbench`, `OLTP-Insert` and `varmail-split` by 28%, 61% and 21% against EXT4 in forty threads, respectively, thanks to its concurrent journaling scheme. However, the performance of `varmail-shared` is not at all scalable and moreover is even worse than EXT4. We found that the main problem is the transaction conflict. As presented in Table 3, `varmail-shared` has contention on a shared directory. When the modified shared directory pages are under DMA, the other concurrent transaction cannot make progress, significantly limiting scalability until the IO completes.

3.3 Analysis on Scalability Bottleneck

We examine the scalability bottlenecks in filesystem journaling with EXT4 performing serial journal commit and BarrierFS performing concurrent journaling. We identify four main components that affect the performance scalability in EXT4 and BarrierFS; *transaction conflict* (§3.3.1), *serial flush* (§3.3.1), *length of a transaction lock-up interval* (§3.3.2) and *coalescing degree of compound journaling* (§3.3.3). We present `varmail-shared` results only since the other workloads show the similar performance behavior.

3.3.1 Transaction Conflict

EXT4. Figure 4 shows the number of transaction conflicts (`varmail-shared`). The number of transaction conflicts – the number of file operations trying to modify the log blocks that are under DMA. At `varmail-shared`, the number of blocked file operations ranges from 6,360 to 15,809. It accounts for 4.7% of all file operations. Despite the shadow paging feature of EXT4 to resolve the transaction conflict, EXT4 journaling still suffers from a significant amount of transaction conflicts.

BarrierFS. BarrierFS renders significantly worse performance than EXT4 in `varmail-shared` workload (Figure 3(a) vs. Figure 3(b)). We found that the concurrent journaling design of BarrierFS increases the number of transaction conflicts substantially and it causes the scalability meltdown. BarrierFS can start committing the following transaction before the preceding transaction commit finishes. Technically, there can be multiple committing transactions in-flight in BarrierFS. In reality, BarrierFS fails to commit multiple transactions concurrently. There are two reasons; *transaction conflict* and *serial flush*. We find that most journal transactions share some pages in common, *e.g.*, inode block and bitmap, and is subject to the transaction conflict [17]. The following journal transaction cannot be committed till the preceding transactions which it conflicts with are made durable at the storage. In BarrierFS, the flush thread issues the flush command of the following committing transaction only after the preceding transaction becomes durable. Even though BarrierFS commits multiple transactions concurrently, it flushes each of them with a separate flush command. Since each journal commit yields a separate flush at the storage device, the benefit of concurrent journaling design of BarrierFS is marginal.

Moreover, when running transactions are trying to modify log blocks under flush, they all are conflicted and blocked. Shadow paging (inherited from EXT4) does not help because it can create only one version in a certain condition. As a result, higher concurrency in committing transactions and limited shadow paging causes nearly 100% of file operations suffering from transaction conflicts in all threads.

3.3.2 Transaction Lock-up

One of the main causes of scalability failure in concurrent journaling is the extended lock-up interval.

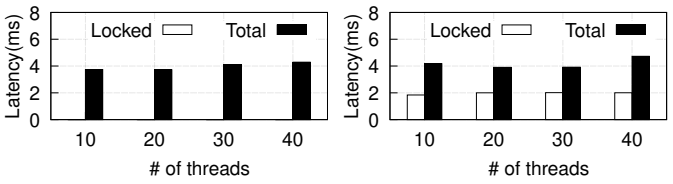
EXT4. In EXT4, the length of transaction lock-up interval is negligible as in Figure 5(a). In EXT4, the lock-up period is just a duration waiting for outstanding file operations to finish, which is very short in general.

Also, as Figure 6(a) shows, `fsync()` latency is high but the latency of `create()` and `unlink()` is still low. In other words, the short lock-up period does not interfere other file operations, `create()` and `unlink()`.

BarrierFS. In BarrierFS, the transaction lock-up latency accounts for approximately half of the entire transaction commit latency (Figure 5(b)). We found that transaction conflict and concurrent journaling negatively interfere with each other and significantly extend the transaction lock-up period. Because the running transaction waits for resolving of transaction conflict in LOCKED state.

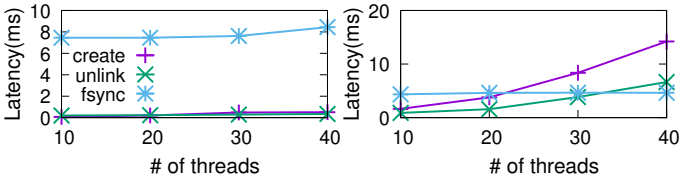
In both EXT4 and BarrierFS, JBD thread first places a running transaction in the *LOCKED* state when it starts committing the running transaction. There is a critical difference between EXT4 and BarrierFS from the aspect of the LOCKED state. In EXT4, when JBD thread places the running transaction in the LOCKED state, the running transaction is guaranteed to be free from transaction conflict. That is because, in EXT4, journal commit is strictly serial activity. In EXT4, the running transaction can be released from the LOCKED state if all outstanding filesystem operations finish.

In BarrierFS, the running transaction can be placed in the LOCKED state while the preceding journal commit is still in flight. BarrierFS can prematurely place the running transaction at the LOCKED state before the running transaction becomes free from the transaction conflict. BarrierFS waits to release the running transaction from the LOCKED state till all outstanding file operations finish and till all conflicts are resolved. As a result, a running transaction stays at the LOCKED state in much longer interval in BarrierFS than in EXT4.



(a) EXT4 (b) BarrierFS

Figure 5: Transaction lock-up interval in `varmail-shared`



(a) EXT4 (b) BarrierFS

Figure 6: Latency of `unlink()`, `create()` and `fsync()` in `varmail-shared`

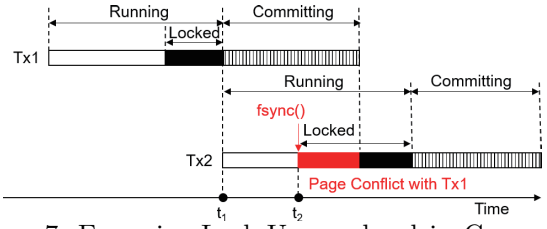


Figure 7: Excessive Lock-Up overhead in Concurrent Journaling (BarrierFS)

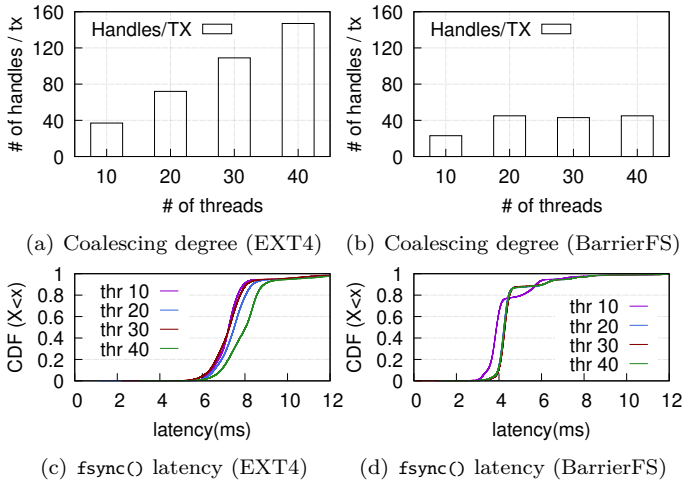
Figure 7 illustrates this situation. The running transaction, $\texttt{Tx}_2$, is created at t_1. The application calls `fsync()` at t_2. $\texttt{Tx}_2$ is placed on the LOCKED state immediately without waiting for the current committing transaction $\texttt{Tx}_1$ is made durable. If $\texttt{Tx}_2$ conflicts with $\texttt{Tx}_1$ (in most cases it does), $\texttt{Tx}_2$ can be released from the LOCKED state only after $\texttt{Tx}_1$ is committed to the storage.

3.3.3 Limited Coalescing Degree

The key ingredient that governs the performance scalability of the filesystem journaling is the *coalescing degree* of the journal transaction – the number of filesystem operations in a journal transaction.

EXT4. EXT4 scales well in `varmail-shared` workload (Figure 3). Ironically, the strict serial nature of EXT4 journaling actually helps itself to increase the coalescing degree of the compound journaling. EXT4 can start committing the running transaction only when the preceding journal commit finishes. When the journal commit is in progress, all updates associated with the incoming file operations are inserted at the running transaction. Therefore, there is a higher coalescing opportunity as the number of threads increases. Figure 8(a) confirms that the number of handles (*i.e.*, file operations) in a transaction increases linearly with the number of threads. At the same time, we observe that the journal commit latency increases with the number of threads. This is because journal transaction tends to get larger as the number of threads increases. As shown in Figure 8(c), median and 99.99% latencies increase 11% and 7%, respectively, from 10 to 40 threads.

BarrierFS. BarrierFS fails to scale in `varmail-shared` workload (Figure 3) due to its *limited coalescing degree*. This is because BarrierFS places the running transaction into LOCKED state prematurely and leaves less chance to coalesce the multiple file operations into a single journal transaction. Figure 8(b) confirms that in BarrierFS the coalescing degree remains the same while the number of threads increases. Since the coalescing degree does not increase, the latency of journal commit remains the same irrespective of the increase in the number of threads (Figure 8(d)).



(a) Coalescing degree (EXT4) (b) Coalescing degree (BarrierFS)

(c) `fsync()` latency (EXT4) (d) `fsync()` latency (BarrierFS)

Figure 8: Coalescing degree and CDF of `fsync()` latency in EXT4 and BarrierFS for `varmail-shared` workload

4 Design

In this section, we present the design of CJFS, a *Concurrent Journaling Filesystem*. CJFS consists of four key technical ingredients; (1) *dual thread journaling* (S4.1), (2) *multi-version shadow paging* (S4.2), (3) *opportunistic coalescing* (S4.3), and (4) *compound flush* (S4.4) – to overcome all the bottlenecks discussed in S3.3 and to scale filesystem journaling.

4.1 Dual Thread Journaling

For concurrent journaling, we separate the journal commit procedure into two phases, the commit phase and the flush phase and allocate separate threads, namely *commit thread* and the *flush thread*, for each phase. The commit thread is responsible for issuing the write requests for journal transaction to the storage. Once this completes, the storage device sends an interrupt to the host notifying about the completion of servicing the requests. The flush thread is responsible for making the log blocks and the commit block durable. Once the interrupt arrives, the flush thread wakes up and issues the flush command to the storage to make the log blocks and the commit block durable. Via separating the commit thread and the flush thread, CJFS can commit the following transaction without waiting for the preceding journal commit to finish.

Figure 9 illustrates the mechanism of Dual Thread Journaling. CJFS maintains a single running transaction. In `fsync()`, the flush thread waits till all dirty pages, log blocks, and the commit block are transferred to the disk. Once this completes, it issues the flush command to the storage. Our journaling module leverages the cache barrier command [14, 36, 49], which efficiently preserves the partial order between the issue order and the persist order in a commodity storage device.

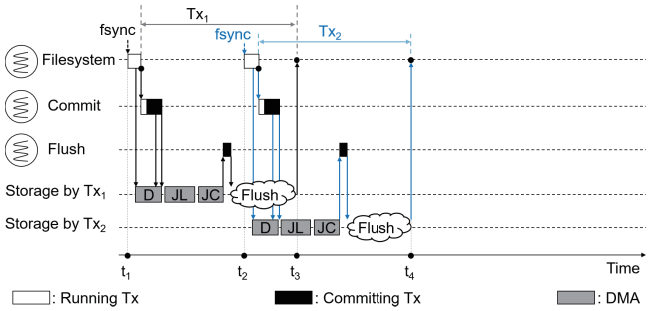


Figure 9: Concurrent Transaction Commit in Dual Thread Journaling. CJFS performs $\texttt{Tx}_1$'s flush phase and $\texttt{Tx}_2$ commit phase concurrently

4.2 Multi-Version Shadow Paging

Most filesystems cluster the filesystem metadata together in their filesystem partition. This is to exploit the spatial locality of the disk access. The filesystem operations, e.g.,`create()` or `write()`, access a few common blocks which contain the popular filesystem metadata, e.g., the allocation bitmap or inode.

EXT4 adopts the page granularity physical logging and uses the original page cache entry. When it commits the journal transaction, it establishes an exclusive lock on the page cache entry associated with the journal transaction till the journal transaction becomes durable. Transaction conflict is particularly harmful to concurrent journaling since it serializes the journal commits. If the transaction conflict happens frequently, the concurrent journaling of CJFS becomes barely effective and resorts to serial journal commit as in EXT4.

To address the transaction conflict, we propose *Multi-Version Shadow Paging (MVSP)*. In multi-version shadow paging, when the commit thread starts the journal commit, it creates the shadow copy of all pages in the journal transaction. In committing the journal transaction, the commit thread uses the shadow copy of each page in the transaction for transferring the journal transaction to the storage device instead of using the original one. Since the journaling module uses the shadow page for the journal commit, the subsequent file operation can update the original page.

There can be multiple shadow copies for a given page cache entry. Assume that the shadow copy of page `P` is being committed to the storage. An application updates the `P` to $\texttt{P}'$ and calls `fsync()`. Then, the commit thread creates the shadow copy of $\texttt{P}'$ and commits the shadow copy of $\texttt{P}'$ to the storage. While the shadow copy of $\texttt{P}'$ is being transferred by the journal commit, another application may update the $\texttt{P}'$ to $\texttt{P}''$ and calls `fsync()`. Then, there exist two shadow copies for `P`, $\texttt{P}'$ and $\texttt{P}''$. CJFS defines the maximum number of shadow pages that can be associated with a single page. The maximum number of versions is an administrator-configurable parameter

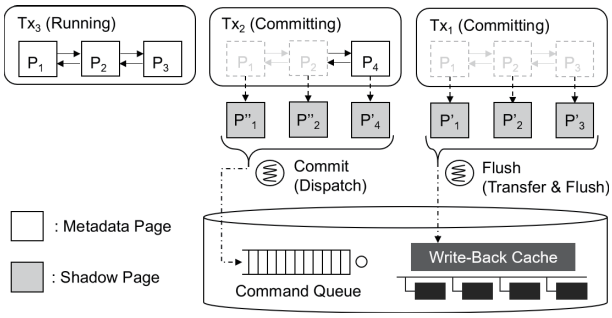


Figure 10: Multi-Version Shadow Paging

and initialized when CJFS is mounted.

Figure 10 illustrates the behavior of CJFS when the transaction conflict exists. Tx_1, Tx_2 and Tx_3 are created in order. Each of them is in a different phase. There are one running transaction, Tx_3, and two committing transactions, Tx_2 and Tx_1. Tx_1 has three pages P_1, P_2 and P_3. When the commit thread commits Tx_1, CJFS creates the shadow copies P'_1, P'_2 and P'_3 for the pages in Tx_1. The subsequent filesystem operation updates P_1, P_2 and P_4. Then, the filesystem triggers another journal commit. The following transaction, Tx_2, consists of P_1, P_2 and P_4. In committing Tx_2, the commit thread creates the shadow copies P''_1 (second shadow copy of P_1), P''_2 (second shadow copy of P_2) and P'_4 for each page in Tx_2. Two transactions, Tx_1 and Tx_2 are being committed to the storage. Subsequent file operations update P_1, P_2 and P_3. Since these pages are available for the update, the file operations update these pages and insert them to the running transaction.

Multi-version shadow paging in CJFS is a variant of versioning which is widely used in transaction concurrency control [18, 30, 51]. Multi-version shadow paging of CJFS is different from the versioning in Copy-On-Write filesystems [20, 37–40]. These filesystems retain the history of updates for individual file blocks to make the IO workload sequential and/or to construct the filesystem snapshot easily.

4.3 Opportunistic Coalescing

CJFS pre-allocates a fixed number of pages for shadow paging. Since the number of shadow pages is limited, the transaction conflict can still occur if all pre-allocated shadow pages are used to hold the logs. If the transaction conflict occurs, the running transaction is put in the LOCKED state and *all* subsequent file operations that modify the filesystem state are blocked. To resolve this problem, we propose the *Opportunistic Coalescing*. The proposed opportunistic coalescing shares the same idea with `try_lock` [4].

Algorithm 1 shows the pseudo-code for opportunistic coalescing. At first, the commit thread puts the running transaction at the LOCKED state (Line 4), After the

Algorithm 1: Opportunistic Coalescing

```
1  function journal_commit_transaction(journal)
2      while true do
3          tx = journal→running_tx;
4          tx → state = LOCKED;
5          if outstanding system calls > 0 then
6              wait (outstanding system calls == 0);
7          end
8          if transaction conflict then
9              tx → state = RUNNING;
10             wakeup(user thread waiting on LOCKED);
11             wait(preceding transaction to commit);
12             continue;
13         end
14         break;
15     end
16     journal → committing_tx = tx;
17     journal → running_tx = NULL;
18     submit_bio_tx(tx);
19     insert_committing_tx_list(tx);
20 end
```

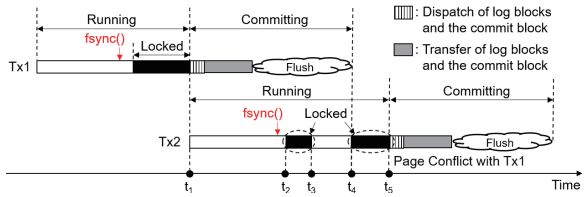


Figure 11: Illustration of Opportunistic Coalescing

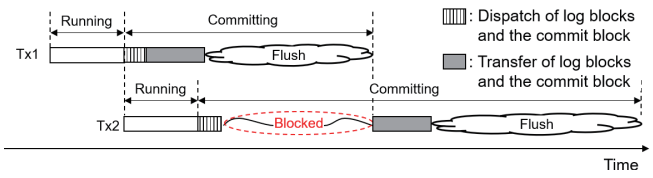
running transaction is put at the LOCKED state, the commit thread waits for the outstanding file operation which already has a journal handle to finish (Line 6). When all outstanding file operations finish, the commit thread checks if there exists any conflict (Line 8). If there exists a conflict, the commit thread places the transaction back to the RUNNING state and is blocked (Line 9-11). The running transaction can continue accommodating the newly incoming log blocks while the commit thread is blocked. Each time when the transaction commit finishes, the flush thread wakes up the commit thread. When the commit thread wakes up, it checks if the running transaction is free from the conflicts. If it is free from the conflicts, it changes the state of the transaction to LOCKED state again (Line 12).

Figure 11 illustrates how the opportunistic coalescing works. There arrive two consecutive transactions (Tx_1 and Tx_2). In Figure 11, Tx_2 is put into LOCKED state twice; at t_2 and at t_4. Tx_2 is in `RUNNING` state during the period between two `LOCKED` states. After the state of the running transaction becomes `RUNNING` state, all pending file operations, which were blocked waiting for the journal handle, are issued the journal handles. With Opportunistic Coalescing, CJFS can coalesce larger number of file operations into the running transaction.

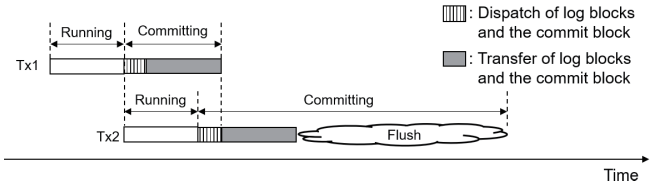
4.4 Compound Flush

CJFS splits the journal commit operation into two phases; (i) transferring the log blocks and the commit block (commit thread) and (ii) making them durable (flush thread). For journaling of CJFS to work in a fully concurrent fashion, both the commit thread and the flush thread should be able to handle the associated tasks in a concurrent manner. In CJFS, the commit thread handles the transaction concurrently; it can commit the following transaction while the preceding transaction is in-flight. However the flush thread handles the transaction in serial fashion; it can flush the following transaction only after the preceding transaction is flushed.

To ensure that the journal transactions are made durable in order, the flush thread issues the flush command for the following transaction only after the flush command for the preceding transaction returns. As a result, the behavior of the flush thread is serial, which makes the concurrent journal mechanism of CJFS only partially complete. Figure 12(a) illustrates the concurrent journaling with serial flush. Commit thread can start committing the following transaction $\texttt{Tx}_2$ before the preceding transaction $\texttt{Tx}_1$ commit finishes. However, the flush thread can flush the following transaction $\texttt{Tx}_2$ only after the transaction $\texttt{Tx}_1$ is flushed to the storage.



(a) without compound flush



(b) with compound flush

Figure 12: Comparison of the flush procedure with and without Compound Flush

To address the serial flush issue of CJFS, we propose *Compound Flush*. Compound Flush exploits the cache barrier command [14, 36]. *Compound Flush* works as follows. When the flush thread is about to send the flush command, it checks if there exist any following committing transactions. If following committing transaction does not exist, it sends the flush command. If the following committing transaction exists, it sends the cache barrier command instead. *Compound Flush* delegates the task of persisting the transaction to the following transaction commit request. An `fsync()` returns only when the associated journal transaction becomes durable. To prevent the *Compound Flush* from delaying the transaction commit indefinitely, we limit the number of transactions that can be flushed with a single flush command. When the number of transactions waiting for the flush reaches its limit or when there is no more committing transactions in-flight, the flush thread sends a flush command to the storage. With cache barrier commands, the storage controller ensures that the log blocks of the individual transactions are made durable in order. When the flush command returns, the flush thread wakes up all application threads that are waiting for their `fsync()` to return.

Figure 12(b) illustrates how Compound Flush works. When the flush thread finishes transferring the transaction $\texttt{Tx}_1$, the flush thread starts transferring the transaction $\texttt{Tx}_2$ instead of calling flush for flushing the transaction $\texttt{Tx}_1$. When the flush thread finishes transferring the transaction $\texttt{Tx}_2$, it finds that there are no other committing transactions in flight. Then, it calls flush to make the transaction $\texttt{Tx}_1$ and transaction $\texttt{Tx}_2$ durable.

5 Discussion

We compare CJFS with the closest filesystem of this sort, BarrierFS [49]. Dual Thread Journaling of CJFS and Dual Mode Journaling of BarrierFS are similar in that both allocate separate threads for transaction commit and transaction flush, respectively. However, BarrierFS's dual thread design is to efficiently support the two journaling modes; "ordered" mode and the "durability" mode. It is not designed for concurrent journaling.

There are three key differences between CJFS and BarrierFS. First is how to handle the transaction conflict. BarrierFS cannot commit the running transaction if the running transaction conflicts with any of the ongoing committing transactions. CJFS can commit the running transaction even if there is a conflict. CJFS uses multi-version shadow paging to resolve the conflict between the running transaction and the committing transactions. The second is how to handle the transaction lock-up. In BarrierFS, transaction lock-up is non-preemptive. Once the running transaction is locked-up, it waits for all committing transactions that it conflicts with to finish. In CJFS, transaction lock-up is preemptive. When a running transaction is locked-up, CJFS checks if the running transaction conflicts with any of committing transactions. If it finds a conflict, the running transaction is unlocked. The third is how to flush the committing transactions. For a set of committing transactions that proceed concurrently, BarrierFS flushes each of them separately. CJFS flushes a number of concurrent transactions together, reducing the flush overhead substantially.

The Opportunistic Coalescing and Compound Flush can be used in the other journaling filesystems such as

XFS [45]. In journal commit, XFS copies the logs in the log list to the log buffer and then flushes the log buffer to the log area in storage. With Opportunistic Coalescing, XFS can insert more logs to the log list by releasing the lock on the log list. With Compound Flush, XFS can flush the multiple log buffers with a single flush command. Dual-Thread Journaling and Multi-Version Shadow Paging are already used widely in other filesystems [20, 37, 38, 45, 49] or DBMS [18, 29, 30].

6 Evaluation

6.1 Experiment Setup

We implemented CJFS [49] on Linux Kernel 5.18.18. We used a 40-core server (two Intel Xeon Gold 6230 processors and 512 GB DRAM) and Samsung 970 Pro SSD (MLC Flash, NVMe) for our experiment. We assume that the SSD supports cache barrier command as a mobile flash products (eMMC) support cache barrier command [3,5]. They do not render any significant performance deficiency against the ones without cache barrier support. Also, previous studies [9, 49] showed the FTL overhead of supporting the cache barrier command is less than 2%. Given all these, we carefully believe that it is reasonable to assume that SSD can support cache barrier command without significant performance overhead. We compare CJFS against BarrierFS [49], SpanFS [15], Vanilla EXT4, and EXT4 with Fast-Commit [42]. We used three macro benchmarks; `varmail` for mail server, `dbench` for file server, and `OLTP-Insert` on `MySQL`. Please refer to S3.1 for details of the benchmarks. We set the maximum number of versions in CJFS to five[1].

6.2 Effect of Individual Techniques

Dual Thread Journaling. We examine the command queue depth of the JBD thread (Figure 13) at `varmail-shared`. In result, CJFS shows higher command queue depth than EXT4 and BarrierFS. Because of the serial transaction commit in EXT4, the maximum queue depth of EXT4 is one. Although BarrierFS adopts dual thread design, its maximum queue depth is two due to the transaction conflict. CJFS fully exploits the queue depth of the storage with Dual Thread design. While BarrierFS suffers from the transaction conflict, CJFS resolves the transaction conflict with Multi-Version Shadow Paging. The performance effect of the Multi-Version Shadow Paging is described separately. With the higher queue depth, it renders higher SSD IO utilization.

Multi-Version Shadow Paging. We vary the maximum number of versions in MVSP and examine the throughput, the latency, and the number of conflicts per transaction. We examine the effectiveness of MVSP under three different maximum numbers of versions; one (EXT4 and BarrierFS), three (noted as CJFS-V3), and five (noted as CJFS-V5). Note that EXT4 and BarrierFS can have up to one shadow page. In the absence of any versioning feature, BarrierFS is subject to frequent transaction conflicts. Transaction conflict becomes more harmful when the filesystem allows the concurrent journal commit (BarrierFS) since it extends the transaction lock-up interval. As a result, BarrierFS renders worse performance than EXT4. With forty threads, the performance of BarrierFS is 60% of EXT4.

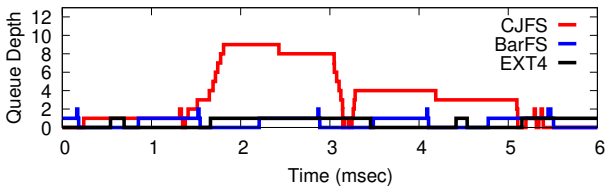


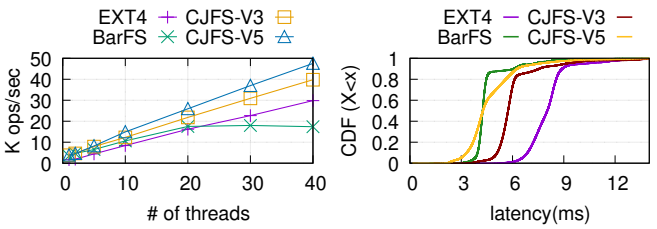
Figure 13: Queue depth of JBD thread in CJFS, BarrierFS (BarFS), and EXT4

Multi-version shadow paging brings additional memory pressure and the overhead of preparing the shadow page. The total memory pressure for multi-version shadow paging corresponds to the sum of the shadow pages associated with the concurrent transactions. The average transaction size is 33 blocks in `varmail-shared` (40 threads). CJFS with five versions (CJFS-V5) consumes 660KByte (5*33*4KByte) additional memory. According to our physical measurement, preparing the shadow page takes approximately 80 usec for a transaction in `varmail-shared` (40 threads). The average transaction commit latency decreases from 4.4 msec in EXT4 to 2.2 msec in CJFS in `varmail-shared` (40 threads). In CJFS, the reduction in the journal commit latency far outweighs the overhead of shadow paging.

We examine the `fsync()` latency of four filesystems (Figure 14(b)). CJFS and BarrierFS yield the shortest latency. The average latency of EXT4, BarrierFS CJFS (V3), and CJFS (V5) are 8.1ms, 4.6ms 6.1ms, and 4.7ms, respectively. CJFS yields the shortest tail latency (99.9%) among the four filesytstems. The tail latency of EXT4, BarrierFS, CJFS (V3), and CJFS (V5) are 17.0ms, 13.5ms, 16.3ms and 11.8ms, respectively. BarrierFS and CJFS-V5 has similar latency but CJFS-V5 has a better throughput than BarrierFS because of the transaction conflict. In BarrierFS, file operations are blocked when the transaction conflict occurs. However, CJFS-V5 is free from the transaction conflict. File operations return without waiting for the transaction conflict.

We examine the number of conflicting blocks. The average number of conflicted blocks in a transaction is eleven or larger in EXT4 and BarrierFS but less than two in CJFS (Figure 15). The number of conflicts per

[1]We found that we do not need more than five shadow pages to eliminate transaction conflict in our experiment setup.



(a) Throughput (b) `fsync()` latency (40 threads)

Figure 14: Throughput and Latency of `varmail-shared`: CJFS, BarrierFS (BarFS), and Vanila EXT4 (EXT4)

transaction is inversely proportional to the benchmark performance. CJFS with five versions (CJFS-V5) outperforms EXT4 1.7× at 40 threads.

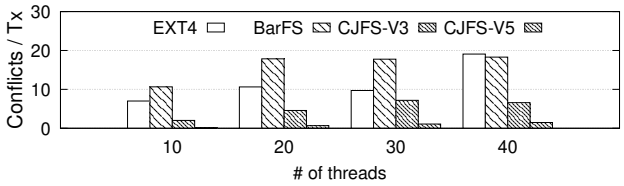
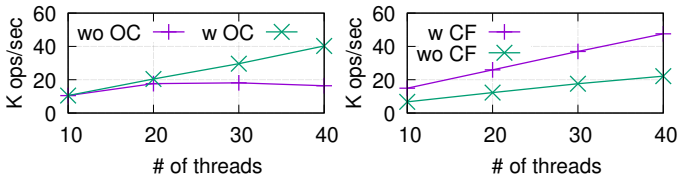


Figure 15: Average number of conflicts per transaction

Opportunistic Coalescing. Opportunistic Coalescing improves the filesystem performance by 2.5× (Figure 16(a)). With Opportunistic Coalescing, the coalescing degree of the journal transaction increases by 3.3× (Figure 17).



(a) Opportunistic Coalescing (b) Compound Flush

Figure 16: Effect of Opportunistic Coalescing and Compound Flush for `varmail-shared` in CJFS

Compound Flush. We ran `varmail-shared` to see the performance impact of Compound Flush. We set the maximum version number to five. Figure 16(b) shows that Compound Flush improves throughput up to 2.14×. By merging the multiple flush commands into one, Compound Flush reduces the average `fsync()` latency from 11.8ms to 4.7ms.

6.3 Macro Benchmarks

Mail server: `varmail-shared` and `varmail-split`. Figure 18(a) and Figure 18(b) shows the throughput of `varmail-shared` and `varmails-split`, respectively. Since all threads share the same directory in `varmail-shared`, the transaction conflicts occur much more frequently.

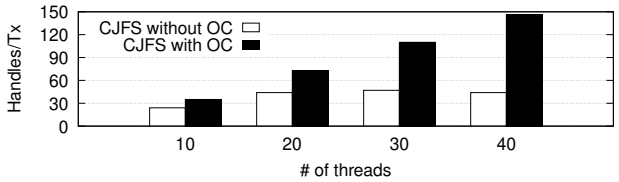


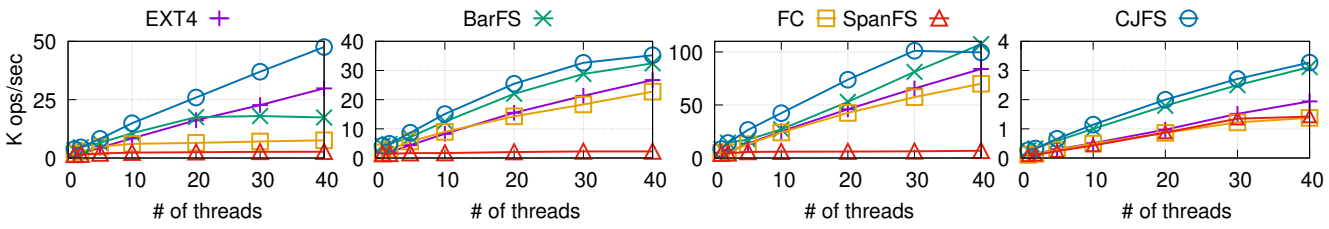
Figure 17: Comparison of coalescing degree with and without Opportunistic Coalescing for `varmail-shared`

For `varmail-shared`, CJFS outperforms EXT4 and BarrierFS by 62% and 173%, respectively. For `varmail-split`, the throughput of CJFS is 82% and 15% higher than that of EXT4 and BarrierFS, respectively. CJFS manifest itself in `varmail-shared` because MVSP of effectively handles the transaction conflict. In `varmail-shared`, BarrierFS becomes subject to the severe performance degradation due to frequent transaction conflicts.

File Server: `dbench`. As Figure 18(c) shows, CJFS increases throughput 68% over EXT4. `Dbench` does not have the directory contention (similar to `varmail-split`) and it is less `fsync()`-heavy than `varmail-shared`, incurring less transaction conflicts. Hence BarrierFS scales better in `dbench` than in `varmail` workload.

`OLTP-Insert` on `MySQL` : Here, the transaction conflict rarely occurs. CJFS scales well even when there is little or no transaction conflict. As Figure 18(d) shows, CJFS increases throughput up to 2.25× over EXT4 in ten threads. Moreover, CJFS increases the throughput by 15% compared to BarrierFS in ten threads.

Analysis : Fast commit [42] uses metadata-granularity physical logging. Despite its data structural elegance, Fast commit yields second to lowest throughput among the five. Fast commit trades the `fsync()` throughput with the `fsync()` latency. Due to its finer transaction granularity, Fast commit tends to make the smaller journal transaction. As a result, the `fsync()` latency becomes shorter in Fast commit. However, we observe that the number of flushes, i.e., the number of journal commits, increases significantly when EXT4 employs Fast commit. As a result, in terms of journaling throughput and scalability, Fast commit in EXT4 leaves substantial room for improvement. Fast commit is particularly detrimental to the journaling performance when there are a large number of threads. SpanFS [15] yields the worst performance among the five due to its serial journal commit. SpanFS defines the running transaction for each filesystem region. In SpanFS, these transactions can be committed in parallel. However, when the two or more transactions modify the shared filesystem metadata, e.g. root directory, the following running transaction can only be committed after the preceding running transaction is made durable. When there exists multiple concurrent running transactions (SpanFS), the performance becomes actually worse



(a) `varmail-shared` (b) `varmail-split` (c) `dbench` (d) `OLTP-Insert`

Figure 18: Throughput: EXT4, BarrierFS (BarFS), Fast commit (FC), SpanFS, and CJFS

than when there allows only one running transaction (EXT4). This is because SpanFS creates large number of small running transactions and all small running transactions are committed in serial fashion. On the other hand, EXT4 commits a large amount of the filesystem updates with a single running transaction.

6.4 Crash Consistency

CJFS uses the same on-disk structure and the recovery routine with EXT4. We use `CrashMonkey` [25] to examine if CJFS recovers the filesystem properly under unexpected system crashes. `Crashmonkey` generates a number of crash scenarios and checks if the filesystem recovers correctly. We use two scenarios, `rename_root_to_sub` and `create_delete`. CJFS passed all 10,000 test cases. We also generate sudden-power-off condition and examine if CJFS recovers the filesystem state into a consistent one. We confirmed that the recovery routine of CJFS correctly replays the transactions in the journal region and places the filesystem state into the consistent state.

7 Related Work

Multiple journal regions. IceFS [24] creates multiple journal regions in a filesystem partition for better isolation. ScaleFS [6], SpanFS [15], and Z-journal [17] manage multiple (or per-core) journal regions to reduce contention on journaling and to achieve high scalability. However, they still serially commit a journal transaction for each journal region and they are subject to transaction conflict when multiple threads access the same storage region. Note that Z-journal compromises the durability of `fsync()` for scalability.

Per-core running transaction. ScaleFS [6] and MQFS [23] maintain per-core running transaction to avoid contention in concurrent journaling. While these works can concurrently commit the multiple transactions in different cores, they commit the transactions in serial fashion in each core. In addition, while this approach minimizes the contention on journaling, it also loses the chance of transaction coalescing, which we found critical in achieving high performance and scalability.

Parallel journal commit. BarrierFS [49] and XFS [45] process a journal transaction commit in a separate thread to make a single journal commit parallel. However, BarrierFS serializes the journal commit when there is a conflict between a running transaction and committing transactions. Also, XFS suffers from excessive flush calls for guaranteeing a write order or a durability [16].

Reducing flush overhead. There have been efforts to reduce costly flush overhead in filesystem journaling. RFLUSH [52] specifies an `fsync()` range and iJournaling [32] performs per-file journaling. IRON filesysem [34] omits flushing the journal commit block by using transactional checksum. BarrierFS [10] leverages a cache barrier command to reduce flush overhead.

Soft Updates. Soft Updates [11,27,41] is an alternative to the filesystem journaling. It enforces write ordering with an expensive transfer-and-flush mechanism [49]. CJFS can guarantee the storage order without using transfer-and-flush mechanism.

8 Conclusion

We propose CJFS, Concurrent Journaling Filesystem. CJFS overcomes the scalability limitations of the heavy-weight EXT4 journaling mechanism with four novel techniques, namely Dual Thread Journaling, Multi-Version Shadow Paging, Opportunistic Coalescing, and Compound Flush. At a high level, CJFS parallelizes the journaling activity (Dual Thread Journaling) and avoids a page under IO being a bottleneck (Multi-Version Shadow Paging). Whenever the contention is inevitable, CJFS actively lowers the overhead by coalescing concurrent requests at thread level (Opportunistic Coalescing) and storage device level (Compound Flush). Our extensive evaluation shows CJFS achieves the significant throughput and latency improvement with multicore scalability and high storage device utilization against the state-of-the-art filesystems.

Acknowledgements We are deeply indebted to our shepherd, Yu Hua, for helping shaping the final version of this paper. We are also grateful to the anonymous reviewers for their comments. This work was supported by NRF, Korea (grant No. NRF-2020R1A2C3008525), and Samsung Electronics (HiPER SCOUT).

References

[1] block: update documentation for REQ_FLUSH / REQ_FUA. https://patchwork.kernel.org/project/dm-devel/patch/20100826095413.GA9750@lst.de/.

[2] The docker containerization platform. https://www.docker.com/.

[3] eMMC5.1 solution in SK hynix. https://www.skhynix.com/kor/product/nandEMMC.jsp.

[4] pthread_mutex_trylock(3) - Linux man page. https://linux.die.net/man/3/pthread_mutex_trylock.

[5] Toshiba Expands Line-up of e-MMC Version 5.1 Compliant Embedded NAND Flash Memory Modules. http://toshiba.semicon-storage.com/us/company/taec/news/2015/03/memory-20150323-1.html.

[6] 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 *Proc. of 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

[7] Silas Boyd-Wickizer, Austin T Clements, Yandong Mao, Aleksey Pesterev, M Frans Kaashoek, Robert Morris, Nickolai Zeldovich, et al. *An Analysis of Linux Scalability to Many Cores.* In *Proc. of 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[8] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. *OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR-2014-019*, 2014.

[9] Yun-Sheng Chang and Ren-Shuo Liu. *OPTR: Order-Preserving Translation and Recovery Design for SSDs with a Standard Block Device Interface.* In *Proc. of 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[10] Jonathan Corbet. *Barriers and journaling filesystems.* http://lwn.net/Articles/283161/, 2010.

[11] Gregory R. Ganger, Marshall K. McKusick, Craig A. N. Soules, and Yale N. Patt. *Soft updates: a solution to the metadata update problem in file systems. ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, 2000.

[12] Robert Hagmann. *Reimplementing the Cedar file system using logging and group commit.* In *Proc. of the 11th ACM Symposium on Operating Systems Principles (SOSP)*, 1987.

[13] Sooman Jeong, Kisung Lee, Seongjin Lee, Seoungbum Son, and Youjip Won. *I/O stack optimization for smartphones.* In *Proc. of 2013 USENIX Annual Technical Conference (ATC)*, 2013.

[14] JEDEC Standard JESD84-B51. *Embedded Multi-Media Card (eMMC) Electrical Standard (5.1).* 2015.

[15] 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 *Proc. of 2015 USENIX Annual Technical Conference (ATC)*, 2015.

[16] Dohyun Kim, Kwangwon Min, Joontaek Oh, and Youjip Won. *ScaleXFS: Getting scalability of XFS back on the ring.* In *Proc. of 20th USENIX Conference on File and Storage Technologies (FAST)*, 2022.

[17] Jongseok Kim, Cassiano Campes, Joo-Young Hwang, Jinkyu Jeong, and Euiseong Seo. *Z-Journal: Scalable Per-Core Journaling.* In *Proc. of 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[18] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youji Won. *Resolving Journaling of Journal Anomaly in Android I/O:Multi-Version B-tree with Lazy Split.* In *Proc. of 12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[19] Alexey Kopytov. Sysbench manual. *MySQL AB*, pages 2–3, 2012.

[20] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. *F2FS: A New File System for Flash Storage.* In *Proc. of 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[21] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. *Write Dependency Disentanglement with HORAE.* In *Proc. of 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[22] Xiaojian Liao, Youyou Lu, Erci Xu, and Jiwu Shu. *Max: A Multicore-Accelerated File System for Flash Storage.* In *Proc. of 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[23] Xiaojian Liao, Youyou Lu, Zhe Yang, and Jiwu Shu. *Crash Consistent Non-Volatile Memory Express.* In *Proc. of 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[24] 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 *Proc. of 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[25] Ashlie Martinez and Vijay Chidambaram. *Crashmonkey: A framework to systematically test file-system crash consistency.* In *Proc. of the 9th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage)*, 2017.

[26] Richard McDougall and Jim Mauro. *FileBench.* http://www.nfsv4bat.org/Documents/nasconf/2004/filebench, 2005.

[27] Marshall K. McKusick and Gregory R. Ganger. *Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem.* In *Proc. of 1999 USENIX Annual Technical Conference (ATC)*, 1999.

[28] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. *Understanding Manycore Scalability of File Systems.* In *Proc. of 2016 USENIX Annual Technical Conference (ATC)*, 2016.

[29] Erik T Mueller, Johanna D Moore, and Gerald J Popek. *A nested transaction mechanism for LOCUS.* In *Proc. of 9th ACM Symposium on Operating Systems Principles (SOSP)*, 1983.

[30] Shojiro Muro, Tiko Kameda, and Toshimi Minoura. *Multi-version concurrency control scheme for a database system. Journal of Computer and System Sciences*, 29(2):207–224, 1984.

[31] Jiaxin Ou, Jiwu Shu, and Youyou Lu. *A high performance file system for non-volatile main memory.* In *Proc. of 11th European Conference on Computer Systems (EuroSys)*, 2016.

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

[33] Jeoungahn Park, Taeho Hwang, Jongmoo Choi, Changwoo Min, and Youjip Won. *LODIC: Logical Distributed Counting for Scalable File Access.* In *Proc. of 2021 USENIX Annual Technical Conference (ATC)*, 2021.

[34] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. *IRON File Systems.* In *Proc. of 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[35] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. *HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM.* In *Proc. of 28th ACM Symposium on Operating Systems Principles (SOSP)*, 2021.

[36] Nikilesh Reddy. *Using Cache barriers in lieu of REQ_FLUSH and REQ_FUA for emmc 5.1.* https://lists.openwall.net/linux-ext4/2015/09/15/5.

[37] Ohad Rodeh, Josef Bacik, and Chris Mason. *BTRFS: The Linux B-tree filesystem. ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[38] Ohad Rodeh and Avi Teperman. *zFS-a scalable distributed file system using object disks.* In *Proc. of 20th IEEE/11th Conference on Mass Storage Systems and Technologies (MSST)*, 2003.

[39] Mendel Rosenblum and John K Ousterhout. *The design and implementation of a log-structured file system. ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[40] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An implementation of a log-structured file system for unix. In *Proc. of 1993 USENIX Winter*, 1993.

[41] Margo I. Seltzer, Gregory R. Ganger, Marshall K. McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. *Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems.* In *Proc. of 2000 USENIX Annual Technical Conference*, 2000.

[42] Harshad Shirwadkar. *ext4: add fast commits feature.* https://lwn.net/Articles/826620/.

[43] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. *BabelFish: Fusing address translations for containers.* In *Proc. of 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

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

[45] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. *Scalability in the XFS File System.* In *Proc. of 1996 USENIX Annual Technical Conference (ATC)*, 1996.

[46] Andrew Tridgell and Ronnie Sahlberg. *DBENCH*. https://dbench.samba.org/.

[47] Stephen Tweedie. *Ext3, journaling filesystem*. In *Proc. of Ottawa Linux Symposium*, 2000.

[48] Stephen C. Tweedie et al. *Journaling the Linux ext2fs filesystem*. In *Proc. of 4th Annual Linux Expo*. Durham, North Carolina, 1998.

[49] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. *Barrier-Enabled IO Stack for Flash Storage*. In *Proc. of 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[50] Haitao Wu, Guohan Lu, Dan Li, Chuanxiong Guo, and Yongguang Zhang. *MDCube: a high performance network structure for modular data center interconnection*. In *Proc. of 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*, 2009.

[51] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, mar 2017.

[52] Jeseong Yeon, Minseong Jeong, Sungjin Lee, and Eunji Lee. *RFLUSH: Rethink the Flush*. In *Proc. of 16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.