LDJ: Version Consistency Is Almost Free on Commercial Storage Devices

DONG HYUN KANG, Dongguk University-Gyeongju, South Korea
SANG-WON LEE and YOUNG IK EOM, Sungkyunkwan University, South Korea

In this article, we propose a simple but practical and efficient optimization scheme for journaling in ext4, called lightweight data journaling (*LDJ*). By compressing journaled data prior to writing, *LDJ* can perform comparable to or even faster than the default ordered journaling (OJ) mode in ext4 on top of both HDDs and flash storage devices, while still guaranteeing the version consistency of the data journaling (DJ) mode. This surprising result can be explained with three main reasons. First, on modern storage devices, the sequential write pattern dominating in DJ mode is more and more high-performant than the random one in OJ mode. Second, the compression significantly reduces the amount of journal writes, which will in turn make the write completion faster and prolong the lifespan of storage devices. Third, the compression also enables the atomicity of each journal write without issuing an intervening `FLUSH` command between journal data blocks and commit block, thus halving the number of costly `FLUSH` calls in *LDJ*. We have prototyped our *LDJ* by slightly modifying the existing ext4 with jbd2 for journaling and also e2fsck for recovery; less than 300 lines of source code were changed. Also, we carried out a comprehensive evaluation using four standard benchmarks and three real applications. Our evaluation results clearly show that *LDJ* outperforms the OJ mode by up to 9.6× on the real applications.

CCS Concepts: • **Software and its engineering** → **File systems management**; **Consistency**; • **Information systems** → *Storage recovery strategies*;

Additional Key Words and Phrases: Crash consistency, file system, compression

ACM Reference format:
Dong Hyun Kang, Sang-Won Lee, and Young Ik Eom. 2019. LDJ: Version Consistency Is Almost Free on Commercial Storage Devices. *ACM Trans. Storage* 15, 4, Article 28 (November 2019), 20 pages.
https://doi.org/10.1145/3365918

1 INTRODUCTION

Guaranteeing *metadata consistency* has traditionally been opted for in designing or configuring file systems [5], since it provides the file system consistency at high performance (e.g., ordered journaling mode in ext4 and in XFS [34]). Under *metadata consistency*, however, a file can point to

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (No. NRF-2015M3C4A7065696) and supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2015-0-00314, NVRam-based High-performance Open Source DBMS Development).
Authors' addresses: D. H. Kang, Dongguk University-Gyeongju, Gyeongju, South Korea; email: dhkang@dongguk.ac.kr; S.-W. Lee and Y. I. Eom (corresponding author), Sungkyunkwan University, Suwon, South Korea; emails: {swlee, yieom}@skku.edu.

1553-3077/2019/11-ART28 $15.00
https://doi.org/10.1145/3365918

an older version of data after a system crash or power failure. That is, the consistency in metadata does not necessarily guarantee the consistency of data itself pointed to by the metadata. This kind of weak consistency has forced many applications, such as SQLite, MySQL, and CePH, to devise their own ways to guarantee the application-level crash consistency. To mitigate the situation, some researchers have begun shifting toward a *version consistency* [5], which ensures that the metadata correctly points to its data (e.g., data journaling mode in ext4, logging in LFS [32], and copy-on-write scheme in btrfs [31]). But, the existing schemes can achieve *version consistency* only at the cost of significant performance degradation such as redundant journaling and cleansing [1, 4, 22, 25, 35, 36]. For instance, the double writes of data pages in journaling-based file system will, particularly in flash-based storage devices, have adverse effects on the performance and lifespan of storage devices.

Meanwhile, during the past decade, we have witnessed a few intriguing trends in the storage market. First, the capacity of each individual storage device has exponentially grown. As a consequence, it is not uncommon nowadays to see HDD storage devices of 12 TB and flash SSDs of 16 TB. Accordingly, the size of the file system has also been increased significantly. Second, the ever-evolving storage interfaces and related techniques have continuously made the read and write operations on storage more lightweight. For example, the SATA 3.0 interface provides a throughput of 6 Gbps, two orders of magnitude faster than SATA 2.0. Third, flash SSDs have been rapidly becoming another main-stream storage device. One intrinsic characteristic of flash SSDs is that sequential writes are preferred to random writes. Even for HDDs, the performance gap between sequential and random writes has been widening. These trends, interestingly, provide us an opportunity to revisit the data journaling mode of file systems for higher performance; the data journaling mode can be optimized to perform very comparably to, and in some cases, to outperform the ordered mode, while preserving its higher consistency level.

In this article, we propose a simple but effective optimization scheme for data journaling mode in ext4, called *lightweight data journaling* mode (*LDJ*). The main ideas of *LDJ* are as follows: (1) It highlights the crucial importance of sequential writes both in SSD and HDD and utilizes the fact that the performance overhead of sequentially writing data and metadata to the journal area would be marginal. (2) By enlarging the size of journal area (e.g., from 128 MB to 5 GB) and thus making the checkpoints triggered more lazily, we can alleviate the overhead of the so-called *forced checkpoint* [1] and thus the foreground transactions will commit quickly without being blocked any more by the sluggish checkpoints [1]. (3) By compressing journal blocks on-the-fly prior to writing them to the journal area, we can reduce the amount of data to be journaled, which in turn will shorten the write completion, prolong the lifespan of flash SSDs, and, most importantly, make checkpoints further delayed with the same size journal area. In addition, the compression enables the atomicity of each journal write without issuing an intervening `FLUSH` command between journal data blocks and commit block, thus halving the number of costly `FLUSH` calls in *LDJ*.

While limited when each idea is individually applied, the performance improvement is quite significant when all the three ideas are applied in combination. In particular, we made an observation that the problem of *forced checkpoint* could not be perfectly solved simply by enlarging journal area [1]. Instead, when combined with the compression, the enlarged journal area allows the *foreground* committing transactions to proceed their journal write operations, while perfectly overlapping them with the *background* checkpoint operations. From the technical perspective, this is in stark contrast with the recent *ext4-lazy* scheme [1], which would still suffer from the *forced checkpoints* while running applications with frequent `fsync()` calls on top of high-end SSDs. The main contributions of this article can be summarized as follows:

- We made an observation that the technological characteristics and trends in storage devices provide an opportunity to revisit the data journal mode for further performance optimization.
- We have designed and implemented the *lightweight data journaling* mode (*LDJ*) by slightly modifying the existing ext4 with jbd2 for journaling and also *e2fsck* for recovery (i.e., less than 300 lines of source code were added or updated).
- We carried out comprehensive performance evaluations by running four standard benchmarks and three real application workloads in three different journlaing modes of OJ, DJ, and *LDJ*, on top of the state-of-the-art storage devices including Seagate HDD, Samsung SSD, and Samsung NVMe SSD. Our evaluation results confirm that, in most cases, the overall performance of *LDJ* is comparable to or even better than that of ordered mode, while preserving version consistency.

The rest of the article is organized as follows. Section 2 describes the journaling mechanism and investigates the IO behaviors of each journal mode. Section 3 gives our motivation based on the experimental performance results of the traditional journal modes on the latest HDD and SSD. Sections 4 and 5 describe the details of *LDJ* design and implementation, respectively. Section 6 shows our evaluation results and Section 7 compares *LDJ* with prior work. Finally, Section 8 provides the conclusion.

2 BACKGROUND

2.1 Journaling Mechanism of Ext4

Ext4 file system is commonly used in modern computing systems, such as enterprise, desktop, and mobile systems, and its journaling layer in Linux (i.e., jbd2) is the key component to ensure crash consistency upon a system crash or power failure. As a compromise between the performance and consistency, ordered journaling (OJ) is used as the default journaling mode in Linux; it journals only metadata and guarantees only the *metadata consistency*, therefore writing less data onto the storage and performing faster than the data journaling (DJ) mode. In contrast, DJ mode guarantees higher-level data consistency, so called *version consistency*, as well as metadata consistency, by journaling data blocks too. In this section, we briefly describe how each mode works to guarantee its consistency level upon *commit*, which is triggered either by pre-defined interval (e.g., 5 seconds) or one of the synchronization operations (e.g., `fsync()`, `fdatasync()`, and `msync()`). On a commit in OJ mode, jbd2 first reflects each modified data block to its home location in a synchronous way, and then issues a series of journal blocks, including a journal descriptor (JD) block, journal data blocks (metadata only), and a journal commit (JC) block, to the journal area. We would like to note that, during commit operation in OJ mode, the write pattern onto the storage device tends to be *random*, since the logical block addresses (LBAs) of the home location of modified data blocks are likely to be scattered. This random write pattern in OJ mode may, as will be explained in Section 2.2, have an adverse effect on the block layer in Linux and the storage media.

In contrast, in DJ mode, when a commit operation is triggered, jbd2 writes a series of journal blocks, including a JD block, journal data blocks (metadata and data), and a JC block, to the journal area. Therefore, the dominant write pattern onto the storage device under the DJ mode will be *sequential*, and any random write is not in the critical path of committing transactions. All the metadata and data blocks once journaled will be asynchronously copy-backed to their home locations later when a checkpointing is triggered. In DJ mode, every metadata and data blocks are written twice in theory for higher consistency, and this write amplification is the main cause of performance degradation and will also shorten the lifespan of the flash memory. But in practice, if metadata or data block with strong update locality is committed two or more times prior to the

next checkpoint, the write amplification ratio could be lowered, because only the latest copy of the block need to be copy-backed to its corresponding home location. Unfortunately, both modes suffer from two consecutive FLUSH commands during committing a transaction; the first FLUSH is issued to the storage, for ensuring the order between journal data blocks and journal commit block, before writing the commit block, and then the second FLUSH command is issued to make journal blocks durable right after writing the commit block.

In either mode, once a metadata or data block is successfully written to the journal area by jbd2, its durability is ensured even though the block has not yet been propagated to its home location; the block is recoverable upon crash. For this reason, the propagation of blocks to their home locations can be safely delayed to the next checkpoint, which is triggered either in predefined interval (e.g., 5 minutes) or by a kernel thread (e.g., pdflush in Linux). The checkpoint will be carried out by a dedicated kernel daemon, which writes all the dirty blocks to their home location asynchronously, and this *background checkpoint* will not pose severe overhead on the committing foreground processes.

2.2 Performance-related Issues in Ext4 Journal Modes

With the background in Section 2.1 in mind, we now compare the characteristics of the two journaling modes in ext4 and discuss a few critical technical issues outstanding in each mode mainly from the performance perspective. As discussed before, although DJ mode writes more than OJ mode, the write pattern in DJ mode tends to be sequential while that in OJ mode is inherently random.

Let us first discuss the technical issues resulting from the random write pattern in OJ mode. It is well known that the random IO pattern incurs noticeable overhead on the underlying IO stack layers [1, 8, 15, 19, 23]. Therefore, the random write pattern, which is dominating in OJ mode will make the storage performance optimization hard. For example, the random writes give unintended overhead to the block layer in Linux, mainly because the block layer has to spend a lot of time in making an object of block IO (i.e., *bio* structure), and also in finding an opportunity for merging the incoming block IOs even when the IOs cannot eventually be merged due to their random LBAs. In addition, the random write pattern is harmful in both HDDs and SSDs; while it will incur numerous disk seeks in HDDs [1], it will also cause costly garbage collection overhead in SSDs [8, 13, 15]. In contrast, DJ mode does not suffer from these technical issues, because the dominating write pattern in the mode tends to be sequential.

Meanwhile, the amplified journal writes in DJ mode raises two technical issues different from the ones above. The first and most crucial one is the *forced checkpoint* problem [1]. In DJ mode, a small-sized journal area (e.g., 128 MB) will be quickly filled up with data as well as metadata before the background checkpoint is triggered, especially when the applications seldom issue the `fsync()` calls. In this case, jbd2 will force the checkpoint operation to reclaim free space in the journal area. And, unfortunately, during such a *foreground checkpoint*, a committing transaction has to wait until some amount of free space in the journal area is secured. For this reason, the checkpoints in DJ mode does not proceed in background mode; instead they are blocking the committing transactions and thus become the critical path on the performance of committing transactions. This phenomenon is called as *forced checkpoint* [1]. The second technical issue in DJ mode is that the double amount of writes can halve the endurance of flash storage devices.

3 MOTIVATION

With the tremendous growth in storage industry, new storage technologies and devices have been relentlessly developed during the last decade: SATA 3.0 and PCIe/NVMe interface, internal parallelism in flash storage, large and/or battery-backed write buffer [9], and so on. Today, they are

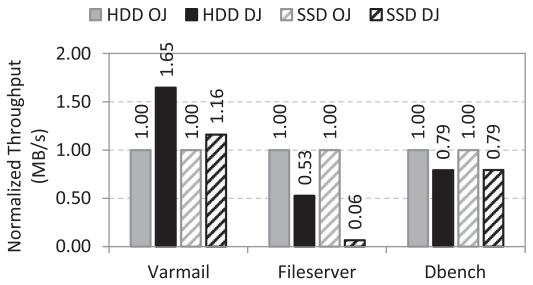


Fig. 1. Throughput of two journal modes in ext4, ordered journaling (OJ) and data journaling (DJ); the results of DJ are normalized to those of OJ.

commonly used for optimization of the storage performance [1, 12, 14–16]. To understand the performance trends of both journal modes in ext4 on the latest storage devices and also to get an insight about the limitations and opportunities of DJ mode in comparison to OJ mode, we measured the performance of both modes while running three standard file system workloads, Varmail, Fileserver, and Dbench, on top of two representative storage devices of Seagate 2TB HDD and Samsung 256GB 850 PRO SSD (see Figure 1).

The experimental results in Figure 1 reveal both the promises and challenges of DJ mode together. For both HDD and SSD, while DJ mode slightly underperforms OJ mode in case of Dbench workload, the former even slightly outperforms the latter in case of Varmail workload. This promising result of DJ mode is consistent with the observations made in other recent research [26]. Both Varmail and Dbench workloads are common in that `fsync()` calls are made frequently. Under the heavy `fsync()` calls, the random write pattern in OJ mode will have adverse effect on the performance while the sequential write pattern in DJ mode, despite of its doubled amount of writes of data blocks, makes its performance comparable to that of OJ mode. Meanwhile, from the performance result of Fileserver workload in Figure 1, we know that DJ mode is still quite inferior to OJ mode consistently for both storage devices. Fileserver workload is a write-intensive workload with almost no `fsync()` calls so that DJ mode suffers from both redundant writes and excessive *forced checkpoints* [1]. It is clear from this discussion that reducing both the amount of data to be written to the journal area and the frequency of *forced checkpoint* is critical in further optimizing DJ mode. It is also evident that the sequential write pattern in DJ mode is helpful in making DJ mode perform comparably to OJ mode. These are the main motivations behind the *lightweight data journaling* (*LDJ*) mode.

4 DESIGN OF *LDJ*

One of the key objectives in designing *LDJ* is to trade expensive IO costs in DJ mode with less-expensive CPU costs by compressing journal blocks, while preserving the *version consistency*. To achieve our design goal, we extends the DJ mode of ext4 file system and utilizes *compression-properties* of the lossless compression algorithms (e.g., Lempel-Ziv [38, 39]).

4.1 Compression-properties

Compression techniques are widely used in various fields, including storage systems and computer networks, to reduce the amount of space needed to store or to transfer data [7, 18, 40]. In storage systems, the compressed data can increase the efficiency of data transfers between the host and storage, reduce space requirement on the storage, and, in case of flash memory, can shorten the write completion time and prolong the lifespan of the storage. Now, let us elaborate on two important properties that the journal compression in *LDJ* provides in optimizing the data journaling.

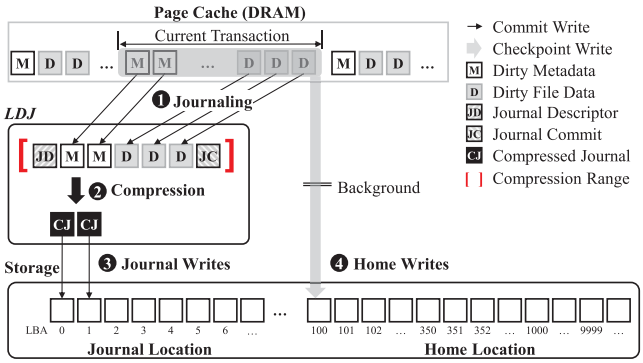


Fig. 2. The journaling procedure under ext4 with *LDJ*.

Compression-property 1. The first *compression-property* is that the compression reduces the amount of space needed to store data. Obviously, this property allows *LDJ* to reduce the amount of journal writes by compressing the whole data that need to be journaled. In turn, this reduction is obviously beneficial to the commit latency and the lifespan of flash storage devices; (1) it can shorten the commit latency in *LDJ* by reducing the number of IO operations (e.g., insert/merge/issue operations on the *bio* structure) and by mitigating the interference of IO traffic, and (2) it can prolong the endurance of flash storage devices by reducing the amount of data itself to be written to the storage and the garbage collection overhead inside flash storage.

Compression-property 2. The second property of the compression is about the *write atomicity* of the compressed journal data. This *compression-property* is inherited from the *lossless compression* algorithm that is based on Lempel-Ziv [38, 39] and fixed Huffman coding [11]:

- The compression algorithm first finds the repeated literals while scanning raw data sequentially, and the matches are encoded into the dictionary that is composed of a set of pairs of literal length and matched length.
- The decompression algorithm, while reading the dictionary from the compressed data, restores the original raw data by referencing the literal length and matched length; if one of the matches on the dictionary is omitted (or removed), decompression fails to rebuild the raw data.

An important implication of these properties is that *LDJ* can achieve the write atomicity and durability of a compressed journal spanning multiple blocks just by issuing one FLUSH command after writing all the blocks. That is, if the decompression procedure succeeds in restoring the original data from a compressed journal, it means that all the blocks of the compressed journal were successfully written. Otherwise, it means that one or more blocks of the compressed journal were not properly written to the storage.

4.2 Journaling Procedure

LDJ is simple and straightforward but it completely guarantees *version consistency* and efficiently improves the overall performance by utilizing the *compression-properties* in committing the transactions (i.e., the running transactions in Linux). Now, we describe *LDJ* under ext4 file system in detail. Figure 2 shows the journaling procedure of ldj with an example.

Compressed Commit. When a commit operation is triggered either by pre-defined interval (e.g., 5 seconds) or by one of the synchronization operations (e.g., `fsync()`, `fdatasync()`, and `msync()`), *LDJ* first collects the whole dirty data belonging to the current transaction, including modified data and its metadata (see ❶). Then, *LDJ* transforms the raw data to the *compressed journal* by compressing the data blocks along with the corresponding journal metadata blocks, journal descriptor and journal commit block (see ❷); one *compressed journal* consists of compressed data (variable length) and its size (4 bytes), and it is stored over one or more blocks aligned at a page granularity (e.g., 4 KB). Such a compression is advantageous in committing the transaction compared with the traditional OJ mode, in that *LDJ* can reduce the amount of storage writes when a large amount of raw data is transformed to a smaller amount of *compressed journal*. Finally, *LDJ* starts writing a series of blocks belonging to one *compressed journal* into the journal area on the underlying storage device (see ❸).

Note that, because the write request for each block of the *compressed journal* is made to the storage device individually at page granularity, the atomic propagation of all blocks of the *compressed journal* to the storage is not guaranteed at all either by the storage device itself or by the kernel. But, recall that *LDJ* can ensure the atomic write of each compressed journal to the journal area according to the *compression-property* 2. Therefore, *LDJ* can achieve the atomicity as well as the durability for the *compressed journal* by calling only one FLUSH command after issuing the write call for its final block. In contrast, under the DJ mode, two FLUSH commands should be called to guarantee the atomic write of a journal data [5]. The first FLUSH command should be called, prior to writing the journal commit block, to ensure the write ordering between the journal data blocks and the journal commit block. And, after writing the journal commit block, the second FLUSH command should be called to ensure the durability of the commit block. Of course, the benefits of the compression in *LDJ*, such as reduction in the amount of writes and halved FLUSH calls, do not come for free - the compression of the journal data requires CPU power and time, both of which are not negligible. That is, there exists a tradeoff between the IO benefit and the CPU overhead. In many realistic applications, as will be demonstrated in Section 6, the benefits of write efficiency obtained by compression and higher consistency level (*version consistency*) in *LDJ* far outweigh the additional CPU overhead taken for compression. In addition, the write efficiency that comes from journal compression in *LDJ* will be more outstanding when combined with the lazy checkpoint, which is described below.

Lazy Checkpoint. During the past decade, we have witnessed the capacity of storage devices growing exponentially. However, we believe that existing journaling mechanisms (i.e., jbd2) fail to take full advantage of an opportunity that comes from the ever-growing storage capacity. That is, although it is now affordable to invest much more space on journal area for better performance, the size of journal area in jbd2 is still set to 128MB by default, which we believe is too small and thus is the root cause for performance detriment in DJ mode (i.e., the *forced checkpoint*). This observation led us to extend the size of journal area in designing *LDJ*, and one obvious benefit of this enlarged journal area is that the *forced checkpoint* operations will be naturally deferred in proportion to the size of the journal area. Basically, the checkpoint procedure in *LDJ* proceeds equally as in DJ mode of jbd2. During checkpoint, a background job flushes every journaled block of either data or metadata from the buffer cache to its home location (see ❹). The enlarged journal area in *LDJ* will trigger the checkpoints in a much longer interval than the DJ mode and thus the *forced checkpoint* can be transformed to a background activity. In contrast to the baseline checkpoint in DJ mode, this lazy checkpoint in *LDJ*, though much more IOs have to be carried out upon each checkpoint, fortunately will not impose a significant burden on the file system, because the checkpoint operation is a background activity. More importantly, the compression in *LDJ* allows to further delay

the checkpoint operations, because more transactions can be journaled in a compressed format on the same size of journal area. This in turn reduces the costlier *forced checkpoints*, and thus the total amount of writes and the number of FLUSH commands can be significantly decreased.

Crash Recovery. In the event of system crash or power failure, *LDJ* can still guarantee its *version consistency*. Basically, the recovery process in *LDJ* is exactly same as the UNDO-based recovery in jbd2, only except for the additional step of *journal decompression*. As explained before, *LDJ* keeps a set of *compressed journals* in the journal area, and each *compressed journal* consists of compressed data (variable length) and its size (4 bytes). Therefore, upon recovery, for each *compressed journal* in sequence from the start of the journal (i.e., `s_start` in Linux), *LDJ* first reads its size information to find out the offset of its compressed data and then restores the original data by decompressing it. This step of decompressing each *compressed journal* repeats until the decompression of the current *compressed journal* fails due to the dictionary mismatch according to *compression-property* 2. The decompression-failed compressed journal will be simply discarded from the recovery, because its one or more blocks were lost by the crash and thus its corresponding transaction can be safely regarded as invalid. After finishing decompressing all valid compressed journals, *LDJ*, likewise the DJ mode, scans each decompressed journal until reaching its journal commit block and replays each valid transaction having its own commit block but not checkpointed yet in sequence.

4.3 False Recovery and Self-trimming

Unfortunately, since the journal data is appended to the journal area just in a round-robin fashion, *LDJ* may cause data corruption and inconsistency by recovering incorrect data. We call such a recovery as *false recovery*. For example, let us assume that the head pointer of the journal is reset to the first offset in jbd2, because the journal area was already full of the previously written *compressed journals*. Then, let us suppose that, at the next commit time, *LDJ* issues three blocks from a *compressed journal* to the journal area: **CJ1**, **CJ2**, and **CJ3**. Also, further assume that a system crash was encountered after only two of the three blocks had been stored in the journal area (e.g., CJ1 – **CJ2** – **CJ3**, **CJ1** – CJ2 – **CJ3**, or **CJ1** – **CJ2** – CJ3; Here, black box means up-to-date version and white box means out-of-date version). If the dictionary from the up-to-date durable blocks (i.e., two black boxes) coincidently matches with the dictionary from the outdated version of a *compressed journal* (i.e., one white box), then the recovery procedure in *LDJ* might recover incorrect data. This false recovery might result in data corruption and inconsistency. Unfortunately, we cannot assume that the *false recovery*, though very rare in practice, will not happen in *LDJ*. To completely prevent the *false recovery*, we decided to take a lightweight cleaning mechanism, called *self-trimming*, that erases the older versions on the journal area. By default, immediately after each checkpoint, *self-trimming* is asynchronously carried out as a background activity that issues a set of TRIM commands (or 0x00) against the checkpointed region in the journal area. With *self-trimming* enabled, *LDJ* can now detect a mismatch of the dictionary under the above scenario, because literal and matched length stored on the black boxes will now point to the trimmed blank block, filled with 0x00, instead of outdated block. In this way, the *false recovery* problem can be avoided in *LDJ*. Note that the overhead of *self-trimming* is negligible, because it runs as a background task and thus will not block other foreground operations such as committing transactions and checkpoint operations.

5 IMPLEMENTATION

We have implemented *LDJ* by extending jbd2 journaling layer in ext4 file system. Briefly, 13 lines of code (LoC) were added to ext4 to use both *LDJ* and lazy checkpoint (lazy) as mount options in ext4. Now, we will elaborate how *LDJ* is implemented on jbd2 layer.

To speed up the (de)compression, numerous hardware-assisted (de)compression techniques have been developed [29, 40], and it is well known that the hardware-based approach can outperform the software-based one by 20× or more [29]. But, in this article, we decided to use the software-based compression technique for two reasons. The first one is to implement *LDJ* easily and the second is to show that the performance of *LDJ* is comparable to that of OJ mode even on commodity servers with no extra support for (de)compression. We used the optimized compression algorithm, lz4, that presents the best compression speed and ratio among modern compression algorithms. To compress all the dirty blocks that are not yet committed, *LDJ* allocates a large amount of contiguous DRAM space (e.g., 1 GB) at boot time. The memory space is divided into two buffers: `I_BUF` and `O_BUF`: The former is used to hold raw data to be compressed and the latter to store the results of the compression (i.e., *compressed journal*). To compress the whole data of a committing transaction, *LDJ* first copies both the raw data and journal metadata (i.e., journal descriptor and commit block) to the `I_BUF` and then calls `lz4_compress()` along with `O_BUF`. Note that, in the resulting *compressed journal*, the first 4 bytes at offset 0 keeps the size of the *compressed journal* and the variable-sized real content of the *compressed journal* follows the size information. Next, likewise the existing ext4 journal modes, *LDJ* crafts log blocks to store the *compressed journal* by calling `jbd2_journal_next_log_block()` and copies the contents of the `O_BUF` to the log blocks. Finally, *LDJ* writes the *compressed journal* to the journal area. In total, we modified 171 LoC of jbd2 layer.

Meanwhile, we can embody LDJ's *lazy checkpoint* without any source code change simply by setting the size of journal area to 5 GB at mount time (i.e., The default journal size is changed by performing "*tune2fs* -J size=5120"). To implement the *self-trimming* functionality, we need to know the target range to be trimmed in the journal area. For this purpose, 11 LoC was added to the existing checkpoint code to find the range of journal LBAs covered by the current checkpoint transaction: the `Head` and `Tail` pointers on the journal area (i.e., the `Head` and `Tail` are captured by `s_start` of the journal super block at the start and end time of the checkpoint operation, respectively). Against this range, a set of `TRIM` commands (or 0x00 in case of the HDDs) is issued in sequence by *self-trimming*, which is implemented as a *kernel worker* thread; it only requires 14 LoC modification.

In general, *e2fsck* is used to verify the ext4 file system, because it can detect file system corruption from various system crashes or power failures. In addition, *e2fsck* is responsible for restoring a damaged file system to a durable and consistent state [21]. We have implemented the recovery procedure of *LDJ* by adding 85 LoC to e2fsck. As mentioned in Section 4.2, the recovery procedure of *LDJ* involves one additional step of *journal decompression*. The *journal decompression* process is carried out in the reverse order of the compression process along with `lz4_decompress_safe()`. It first reads 4 bytes to find the offset of the compressed data and then restores the raw data and journal metadata by scanning and decompressing the compressed data. To reproduce the restored journal data, the *journal decompression* process requires an extra memory space. The *journal decompression* process repeats until the decompression fails or reaches the last *compressed journal*. Finally, *LDJ* replays the data belonging to each transaction, which was committed but not yet checkpointed, to guarantee crash consistency.

6 EVALUATION

We evaluate *LDJ* by driving the following questions:

- How does *LDJ* affect the overall performance with standard benchmark and real-world workloads? And is it efficient enough in guaranteeing *version consistency*? (Section 6.2)
- How effective, in compression time and ratio, is the compression process in accelerating the performance of *LDJ*, and what are the limitations of the current version of *LDJ*? (Section 6.3)

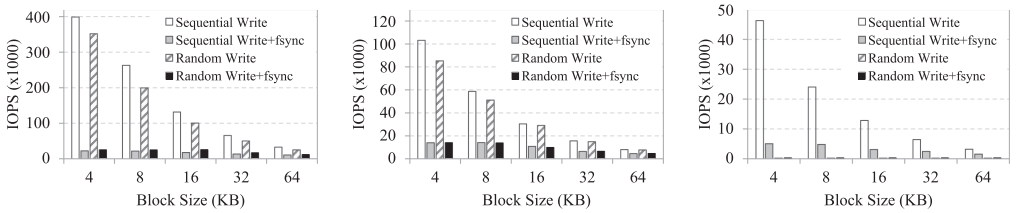


Fig. 3. The effect of write pattern and fsync() call pattern on the storage performance.

- How much recovery time does *LDJ* require, including the time of decompression of the *compressed journal*? (Section 6.5).

6.1 Experimental Setup

To perform a comprehensive evaluation, we validated *LDJ* with four standard benchmarks and three real-world applications on top of three up-to-date commercial storage devices. Now, we present our experimental setup and workload in detail.

Hardware. We conducted all experiments on a machine with Intel Xeon E5-2620 processors of 2.1 GHz with 32 cores and 128 GB DRAM running the Linux kernel 4.9 on the Ubuntu 14.04.5 LTS. To verify the performance effect of *LDJ* on different storage devices, we used three different contemporary commercial storage devices: Samsung 1TB 960 PRO NVMe M.2, Samsung 256 GB 850 PRO SSD, and Seagate 2TB Barracuda HDD. To understand the effect of different write patterns and `fsync()` call patterns on the storage performance, we performed fio benchmark using three different contemporary storage devices and repeated the same experiments while varying the block size; the parameters are configured with "–ioengine=libaio –direct=1 –size=50GB –iodepth=64 –thread=1 –fsync=0 or 100 –rw=write or randwrite" (see Figure 3). As expected, the overall performance of the NVMe SSD is much better than that of SSD or HDD, because NVMe interface provides high IO performance (e.g., multi-GB/s IO rates) using PCIe interface and multiple queues for I/O commands. However, the results in Figure 3 demonstrate that `fsync()` negatively affects the performance across all storage devices: when the `fsync()` calls are made, the write throughput was reduced by up to 17.8×, 7.3×, and 9.1× on NVMe SSD, SSD, and HDD, respectively. This significant performance drop is because `fsync()` adds extra time to the latency of IO path (e.g., transaction commit and write buffer flushing). Also, given that write requests caused by `fsync()` shortens the limited write endurance and increases the garbage collection overhead, the frequent `fsync()` calls will be harmful for the endurance of flash storage devices (e.g., NVMe SSD and SSD). Meanwhile, the performance of HDD is highly dependent on the write pattern mainly because it has a mechanical arm; it is especially bottlenecked by the seek time of head arm. As shown in Figure 3, HDD, in comparison to SSD devices, shows very low write performance against the random write patterns.

Workloads. To evaluate the effectiveness of *LDJ*, we first used two kinds of standard benchmark tools, Filebench 1.5 version and Postmark. The new version of Filebench supports to evaluate the performance of file systems along with data reduction techniques (e.g., data deduplication and compression); this new version generates different types of data unlike its earlier version, which fills data to be written with all zeros or some arbitrary values [2]. This new functionality is especially important in that *LDJ* uses the compression technique as its main feature. To demonstrate *LDJ* under various conditions, we carefully selected three workloads in the Filebench: Varmail, Fileserver, and Fileserver-MF. Varmail is one of the `fsync()`-intensive workloads, because it

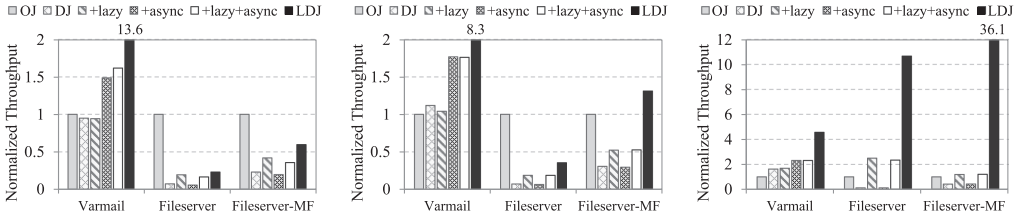
generates many small writes and issues `fsync()` system calls frequently. Meanwhile, Fileserver is a write-intensive workload that involves file operations such as create, append, delete, and so on. Finally, Fileserver-MF is processed exactly the same way as the Fileserver except that it handles more files (i.e., 50,000 files). We modified Postmark benchmark so that it issues `fsync()` after creating its working set, and then performed experiments using the modified benchmark with a configuration such as 10,000 file creations, 25,000 subdirectories, 50,000 transactions, and 1 MB read/write size.

To confirm how *LDJ* can improve the performance of real workloads, we evaluated *LDJ* using three real-world applications: Git, OLTP, and Copy Files. In general, Git is primarily used for source code management and generates a lot of IO activities when pulling source code from the server. To reduce the variance of network bandwidth, we installed Git on top of our local server and measured the time to clone data on the remote repository, where 20 GB of the Linux kernel source is stored. As for OLTP worklod, we ran Sysbench benchmark on top of MySQL. This is because it generates IO-intensive online transactions and supports sophisticated concurrency control protocol to ensure isolation among the threads. For Sysbench in an OLTP mode, we configured it with InnoDB engine, having 128 concurrent threads, and 40 GB database that is composed of 128 files with 1 million records. Both applications generate write-intensive workloads but they are different in that OLTP frequently issues fsync() calls. Finally, to demonstrate a worst-case scenario, we performed copy operations (i.e., cp in Linux) for 5,000 JPG files (total 20GB); JPG is a well-known uncompressed data type.

Methodology. For a fair comparison, we initialized the target storage by performing a series of operations, such as `fdisk()`, `format()`, and `mount()`, right before each evaluation. Especially, at format time, we enabled discard option and disabled lazy initialization options (i.e., the options are configured by passing "-E discard, `lazy_itable_init` =0, `lazy_journal_init` =0" to mkfs.ext4) and delayed allocation option (i.e., the option is disabled by passing "nodelalloc" to mount), because they would give disturbance to our experiments. Each experiment was run for 10 minutes and it was repeated three times to stabilize the results. Before running each experiment, we conducted the aging process by issuing random writes amounting to the half of the storage capacity. Basically, we compared *LDJ* with the traditional journaling modes, ordered journaling (OJ) mode and data journaling (DJ) mode. We also compared *LDJ* with the journaling modes optimized by adding additional features; "+" sign is used as the prefix for the additional feature in the rest of this section. For example, +lazy indicates that base journal mode adopted lazy checkpoint feature. This article presents how the options affect the performance of DJ mode, however, the performance and behaviors of OJ with the options are out of the scope of this article. Finally, *LDJ* was compared with the *asynchronous commit* function [4] that halves the number of `FLUSH` commands by calculating a checksum of journaled data. In case of the OLTP workload, we differently configured the option for *double-write-buffer* according to the journaling mode, because it is primarily used on top of the OJ mode to complement the limitation of *metadata consistency*; the option is enabled on the OJ mode, but it is disabled on the DJ mode.

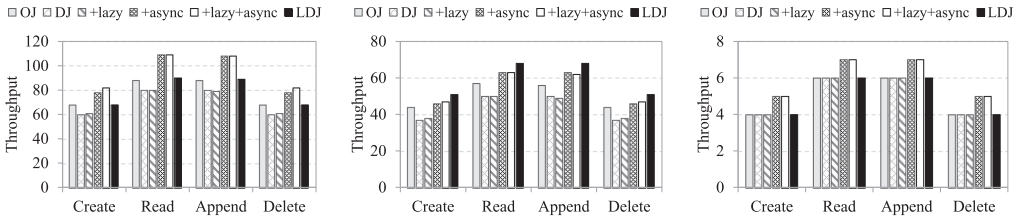
6.2 Performance Comparison

Benchmark Workloads. To emulate the IO activities in server environments, we first evaluated *LDJ* by running a set of server workloads in Filebench. Figure 4 presents the throughput of each workload on three different storage media; the results are normalized to OJ mode. The performance results in Figure 4 shows that *LDJ* achieves performance improvement over OJ mode in many cases even though *LDJ* completely guarantees *version consistency*. Unfortunately, in NVMe SSD, however, *LDJ* underperforms the OJ mode for the write-intensive workloads of Fileserver and



(a) NVMe SSD (b) SSD (c) HDD

Fig. 4. Throughput of each workload on three different storage devices; the results are normalized to OJ mode. Note that "+" sign means that an additional feature was added to the base journal mode; +lazy and +async enable the lazy checkpoint and *asynchronous commit* feature, respectively.



(a) NVMe SSD (b) SSD (c) HDD

Fig. 5. Throughput in operations per second (OPS) for each storage device. Note that "+" sign means that an additional feature was added to the base journal mode; +lazy and +async enable the lazy checkpoint and *asynchronous commit* feature, respectively.

Fileserver-MF. These results are understandable given that *LDJ* causes more checkpoints than OJ mode and the checkpoint operation, in turn, issues the FLUSH command to the underlying storage device; the FLUSH command will severely drop the write performance in NVMe SSD (see Figure 3). Meanwhile, under other storage devices, *LDJ* improves the throughputs of all workloads except for Fileserver over the existing journaling modes. These surprising results can be explained from the fact that modern storage devices prefer sequential write patterns in *LDJ* and the *compression-property* further gives *LDJ* the opportunities to accelerate storage performance. Especially, in Varmail, *LDJ* shows up to 8.3× higher throughput compared to OJ mode, and up to 7.4× compared to DJ mode. The reason behind this improvement is that *LDJ* can, according to *compression-property* 2, halve the number of FLUSH commands in every commit procedure. Unfortunately, SSD reveals a performance collapse for Fileserver workload, but we believe this is acceptable, because OJ mode journals only the metadata of the files and thus 128 MB journal area is large enough. In this experiment, our assumption can be directly confirmed via the evaluation of Fileserver-MF that handles 50,000 files; *LDJ* improves the performance of SSD by up to 31% compared with OJ mode. In case of HDD, due to the sequential write pattern of the *version consistency*, *LDJ* shows significant speedup for all workloads (Figure 4(c)).

For event-based evaluation, we ran Postmark that processes the predefined set of file operations with no time limit. Figure 5 shows throughput of each operation on three different storage devices. As we expected, *LDJ* and *asynchronous commit* (+async) perform comparable to or even better than the OJ mode in all cases even though they completely ensure the *version consistency*. This is because both approaches can halve the number of FLUSH commands in every commit. However, +async is different from *LDJ* in that it cannot reduce the amount of journal writes and it gets some help from

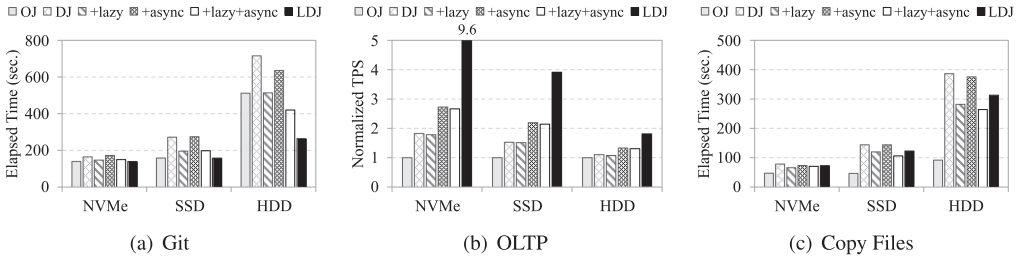


Fig. 6. The performance results of each journaling mode with three real-world applications. Note that "+" sign means that an additional feature was added to the base journal mode; +lazy and +async enable the lazy checkpoint and *asynchronous commit* feature, respectively.

hardware (e.g., crc32c intel hardware acceleration) to calculate the checksum value. We will study the difference in the following sections in more detail.

Application Workloads. Now let us investigate the effect of *LDJ* on three real-world applications, Git, OLTP, and Copy Files. Figure 6 presents the performance results of *LDJ* on the real-world applications; Figures 6(a) and 6(c) show the elapsed time that was measured while cloning data from the server to the local storage device and the total elapsed time, which was taken to copy JPEG files, respectively. Figure 6(b) reports the normalized results that were measured by querying 200,000 transactions under OLTP workload.

Unlike the previous results, Figure 6 shows that *LDJ* outperforms OJ mode in all storage devices and workloads (except Copy Files workload). Such consistent results are especially important in that the compression cost of *LDJ* is negligible and acceptable in real-world environments even though it journals both data and its metadata for *version consistency*. Also, Figure 6 clearly confirms that *LDJ* takes 0.01%–51% speedup on Git and 0.04%–9.6× performance improvement on OLTP, compared with OJ mode. We believe that this performance benefit comes from the reduced amount of writes by compressing journal data on-the-fly as well as sequentializing write patterns, which is more important in low-end storage devices, such as SSD and HDD. Meanwhile, the trends of Figure 6 is very similar to those of Figure 4; Git has similar trends to the Fileserver or Fileserver-MF and OLTP shows similar trends to Varmail. The reason behind these trends is that they have similar IO behaviors and `fsync()` intervals during running each workload. Finally, Figure 6(c) shows the worst-case scenario for *LDJ* and it performs worse than OJ mode because of low-compression ratio.

6.3 Performance Analysis

We first start our performance analysis on the amount of writes, because it is significantly correlated with the performance of *LDJ*. To achieve this, we captured whole writes of each workload, issued to the underlying storage device, by using *blktrace tool*. Table 1 lists the amount of writes captured while running each benchmark on different storage devices. As shown in Table 1, *LDJ* noticeably reduces the amount of writes compared with DJ mode while guaranteeing the *version consistency* (except Copy Files workload); *LDJ* just writes 20% more journal data than OJ mode in the best case. These results are surprising in that DJ mode always performs on average 2x more writes than OJ mode. However, we expected these results before performing the measurement, because *LDJ* can transform a large amount of raw data (i.e., data and metadata) to a small size compressed journal. In addition, such a transformation further reduces the amount of writes issued when `fsync()` is called to force data consistency and durability. The reason is that *LDJ* compresses both data and its metadata needed to be durable and then the writes *compressed journal* whose size is definitely smaller than original data. However, as we expected, the amount of

Table 1. The Amount of Storage Writes that Were Captured with the *blktrace* While Running Each Workload

	Workload	OJ	DJ	+async+lazy	LDJ
NVMe SSD	**Varmail**	15.5	17.7	28.1	52.9
	Fileserver	132.1	300.8	211.1	3.1
	Fileserver-MF	548.9	355.5	315.8	8.6
	Postmark	16.3	31.0	30.8	29.7
	Git	20.4	39.9	39.7	25.5
	OLTP	12.0	11.5	11.8	8.7
	Copy Files	20.3	40.7	40.7	40.7
SSD	**Varmail**	8.1	10.5	16.0	21.7
	Fileserver	134.8	185.8	152.1	3.1
	Fileserver-MF	249.9	208.1	216.0	8.5
	Postmark	16.4	31.1	30.8	29.8
	Git	20.3	39.9	39.7	25.2
	OLTP	12.1	11.8	12.0	9.0
	Copy Files	20.3	40.7	40.7	40.7
HDD	**Varmail**	0.9	1.8	2.3	1.8
	Fileserver	6.6	12.7	64.7	3.3
	Fileserver-MF	10.6	20.1	27.2	8.5
	Postmark	16.2	31.2	30.9	29.8
	Git	20.0	39.9	39.7	25.4
	OLTP	13.3	13.2	13.3	10.2
	Copy Files	20.3	40.7	40.7	40.7

The unit is GB.

Table 2. The Average Compression Time and Compression Ratio of *LDJ* per Transaction for Different Workloads; a Small Compression Ratio Indicates Good Compressibility

	Benchmark Workloads				Application Workloads		
	Varmail	**Fileserver**	**Fileserver-MF**	**Postmark**	**Git**	**OLTP**	**Copy Files**
Avg. Comp.Ratio	0.12	0.007	0.009	0.59	0.39	0.37	0.94
Avg. Comp.Time	107 us	109 ms	165 ms	169 us	1.4 s	153 us	393 ms

writes issued by +async is similar to that of DJ mode, since +async mode should write the whole metadata and data twice; once in the journal and once again in their home locations.

To further understand the performance variation, we measured two compression-related metrics while running each workload: the compression time and the compression ratio. The compression time was measured by placing `ktime_get()` before and after the compression function in jbd2 and the compression ratio was computed as the ratio of the size of data after compression to that before compression. As far as we know, the benefit from the compression functionality can be extremely extended when the contents to be compressed are organized into the binary-based data and the pattern repeated. Therefore, the performance gain of *LDJ* is highly dependent on the types of workloads. Table 2 summarizes the average compression time and compression ratio of *LDJ* for each workload. As shown in Table 2, in general, *LDJ* is very effective in both compression time and compression ratio. Noticeably, it seems an attractive solution for workloads using Filebench, because the workloads are organized into patterns with some arbitrary values [2]. Meanwhile,

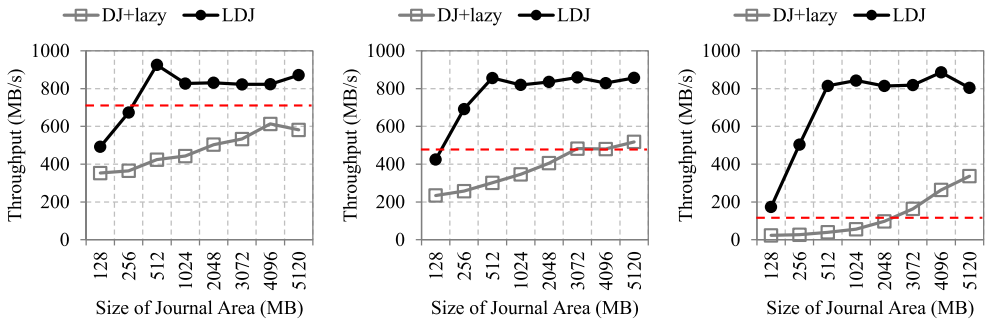


Fig. 7. Performance results of Fileserver workload according to the size of journal area. Red dot line means the performance of OJ mode.

with the real-world workloads OLTP and Git, *LDJ* can reduce the size of data to be journaled to 0.37 and 0.39 of the size of raw data before compression on average, respectively. Fortunately, such a significant reduction is important in that it can give the flash storage devices, such as NVMe SSD and SSD, an opportunity to prolong their limited endurance. However, *LDJ* should pay the cost of compression overhead (i.e., the compression time) to support the compression on-the-fly. In other words, since the commit latency will include the compression time, there exists trade-off between the IO benefit from the compression and the additional CPU overhead for the compression; if the CPU overhead is higher than the IO benefit, the performance of *LDJ* will be decreased. However, we believe that the CPU time overhead for the compression is negligible or reasonable because of two reasons; *LDJ* significantly reduces the amount of writes and the compression functionality can be offloaded to hardware-level. However, *LDJ* sometimes yielded high-compression time along with low-compression ratios in some workload (i.e., Copy Files) whose data is already compressed with JPEG standards.

6.4 Impact of Journal Size

Now, we carefully revisit the size of journal area, because if the journal size is set too small, then jbd2 frequently triggers the *forced checkpoint* that considerably exacerbate both the performance and the amount of writes. To evaluate how the size affects the performance, we repeated the same experiments while varying the journal size. In this article, we show the performance results of Fileserver and Git workload in Figures 7 and 8, respectively. The red dot lines in these figures indicate the performance of OJ mode. From these graphs, we can see that the performance results are slightly different according to the workload and storage type, but they show a similar trend in the same workload. Additionally, In both figures, as the journal size is increased, *LDJ* shows better performance. The reason behind this result is that *LDJ* uses the journal area more efficiently with less checkpoint operations. Especially, in Figure 8, when the size of journal area is extended to 5 GB, *LDJ* shows similar or slightly better performance compared to OJ mode. That is why we extend the journal area to 5 GB in this article. However, DJ mode always shows worse performance compared with *LDJ* although the journal size is extended with *Lazy checkpoint*. This result clearly confirms that *Lazy checkpoint* cannot avoid triggering checkpoint operations frequently.

6.5 Recovery Overhead

Now, we verify the recovery process of *LDJ* and compare its performance with those of existing journaling modes, OJ mode and DJ mode. Since the verification of crash recovery is possible only

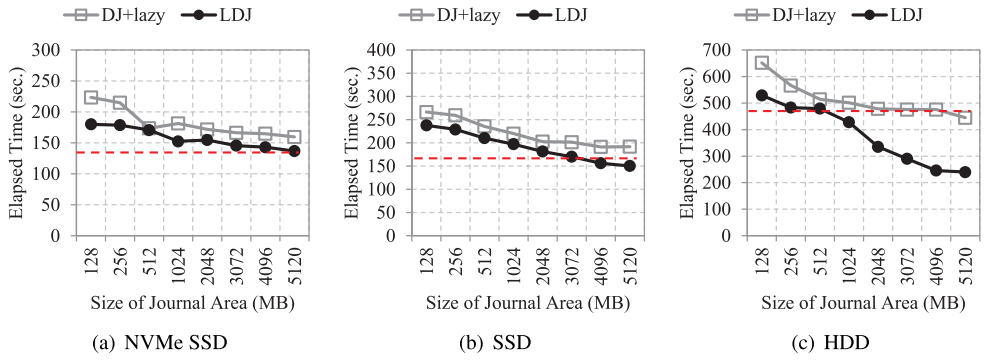


Fig. 8. Performance results of Git workload according to the size of journal area. Red dot line means the performance of OJ mode.

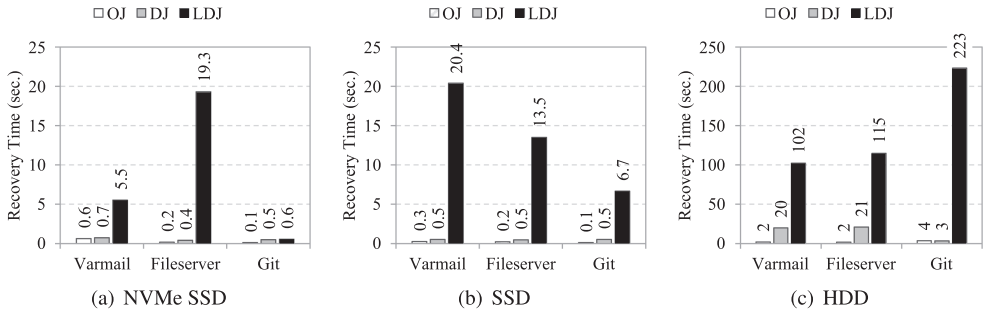
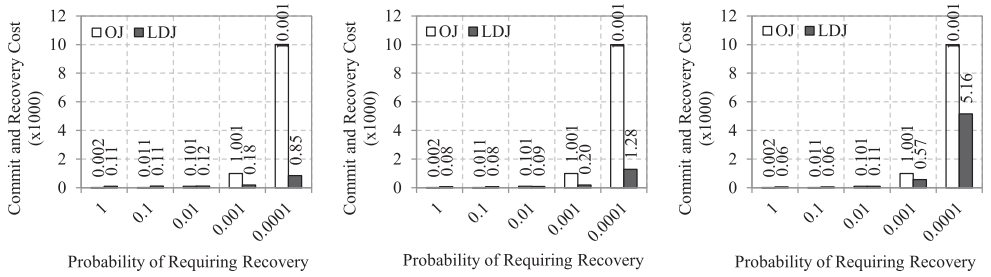


Fig. 9. Comparison of recovery time.

when a system crash or power failure occurs, we forced the system reboot by sending 'b' character to the system request key (i.e., the character is set by passing "echo b > /proc/sysrq-trigger") while running each workload. As mentioned in Section 4, *LDJ* introduced one more step of decompressing the compressed journal data in its recovery process. Therefore, it is not surprising at all that *LDJ* will take longer than default journaling modes in terms of recovery time. For comparison, we measured the time taken for crash recovery in *e2fsck*, which is modified to implement the recovery procedure of *LDJ*. Figure 9 shows the recovery time measured under different types of storage devices. In contrast to the performance results in Section 6.2, *LDJ* is quite inferior to the existing two journaling modes in terms of crash recovery time. Unfortunately, in the worst case, the recovery time of *LDJ* takes longer than that of OJ mode and DJ mode by up to 110× and 47×, respectively.

Clearly, the recovery time directly leads to performance penalty, but it only happens during the boot time, not during the normal execution. To simulate the failure recovery conditions, we made the practical scenarios based on the synthetic trace where we intentionally injected the system failure while varying the number of commit operations. Figure 10 reports the overall cost obtained by calculating how much performance gain is achieved from *LDJ* even though it takes a long time for failure recovery. In this figure, the X-axis is the frequency that the system might invoke a recovery process. For example, 0.01 means that the recovery event is raised once in 100 commit operations. Y-axis shows the overall cost calculated by adding the commit cost and the recovery cost. To simulate each cost, we first looked at the performance gap between OJ and *LDJ* on each workload and normalized the results of *LDJ* to those of OJ. Finally, we calculated the cost by multiplying the number of executed operations by the corresponding cost; the cost is proportionate



(a) NVMe SSD (Fileserver) (b) SSD (Varmail) (c) HDD (Git)

Fig. 10. Comparison of normalized I/O cost.

to the number of commit operations executed along with a recovery operation. As we expected, Figure 10 clearly shows that smaller possibility of recovery events brings better performance of *LDJ*. Furthermore, we believe the long recovery time in *LDJ* will be solved in the near future, because the decompression can be performed with the hardware-accelerator, such as GPUs and FPGAs.

7 RELATED WORK

There have been numerous efforts to improve the performance of crash consistency mechanism in both industry and academia. Most of prior efforts have focused on optimizing the traditional consistency mechanisms in the file system layer [1, 4, 12, 24, 28, 30, 36, 37] or on reducing the consistency overhead in the application layer [3, 6, 10, 17, 18, 20, 25–27, 33, 35].

File system layer. Closest to our *LDJ* is the ext4-lazy mechanism [1] that can reduce the journaling overhead by remapping the location of metadata blocks to journal area. In particular, its checkpoint approach was borrowed by *LDJ* as one of the main journaling functionalities. In recent years, many researchers have been actively working to develop low-overhead but higher-level file system consistency. For instance, the checksum-based journaling and padding-based journaling were proposed to relieve `fsync()` overhead in the journaling process [4, 12, 28, 30]. Especially, the IronFS is the first research that reduces half of the FLUSH operations in the journaling procedure by using the checksum technique; users can use the checksum functionality of IronFS by enabling the *asynchronous commit* option that is already compiled as part of ext4 file system in Linux Kernel. In the previous section, we tried to find the performance difference between *LDJ* and checksum-based journaling via the evaluation that compared *LDJ* with the DJ whose the asynchronous commit option (i.e., +async) turned on. As shown in the graphs, *LDJ* has better performance than +async in most cases. The reason behind these results is that *LDJ* not only reduces the amount of writes by compressing the raw data to be journaled, but also delays the forced checkpoint operations as long as possible. Meanwhile, Park et al. and Yeon et al. improved the traditional journaling mechanism to reduce the overhead caused by `fsync()` [24, 37]. Weiss et al. proposed the ext4-based journaling approach that improves the overall performance by reducing the redundant write overhead in journaling [36].

Application layer. Recently, to achieve application-level crash consistency as well as high performance, Pillai et al. proposed stream APIs in their CCFS work, which are used to preserve the write order of each application [26]. In the application layer, most work has focused on improving SQLite, which is an embedded lightweight database engine, but with high runtime overhead. To reduce the overhead, some researchers revisited both the consistency mechanism of SQLite and the

file system underneath, and then redesigned them for their collaboration [3, 10, 17, 20, 33]. Other researchers studied the overhead of application-level consistency in the file system layer and they suggested new interfaces that can be used for better performance [25, 27, 35]. Meanwhile, some researches utilized the compression technology to reduce the size of log data to be stored or to be transferred [6, 18]. For example, MongoDB employs the snappy algorithm to reduce the amount of journal data managed by the database, not the file systems. Therefore, MongoDB does not have the semantics of the file system, and the compressed data at MongoDB can be journaled in the file system layer once again.

In summary, *LDJ* has unique characteristics compared to prior studies. First, *LDJ* guarantees the *version consistency* along with the lazy checkpoint to efficiently use the journal area. Second, *LDJ* reduces half of the FLUSH commands at commit time by translating multiple writes of journal blocks into one write of *compressed journal*.

8 CONCLUSION

In this article, we first discussed existing journaling modes in terms of consistency-level, performance, and their own benefits. Then, we introduced a novel journaling mode, called *LDJ*, that combines the benefits of DJ mode with the compression technology for efficient journaling. In our evaluation, *LDJ* clearly shows performance improvement compared with the existing journaling modes of ext4 file system. In the best case, *LDJ* outperforms the OJ mode by up to 36.1× and 9.6× on the standard benchmark and real application workload, respectively.

REFERENCES

[1] Abutalib Aghayev, Theodore Tsâo, Garth Gibson, and Peter Desnoyers. 2017. Evolving Ext4 for shingled disks. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 105–119.

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

[3] Qingshu Chen, Liang Liang, Yubin Xia, and Haibo Chen. 2016. Mitigating sync amplification for copy-on-write virtual disk. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'16)*. USENIX Association, 241–247.

[4] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*. ACM, 228–243.

[5] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association.

[6] Xianzheng Dou, Peter M. Chen, and Jason Flinn. 2017. Knockoff: Cheap versions in the cloud. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 73–87.

[7] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. 2017. The logic of physical garbage collection in deduplicating storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 29–43.

[8] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. 2017. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of the USENIX Annual Technical Conference (ATC'17)*. USENIX Association, 759–771.

[9] Weiping He and David H. C. Du. 2017. SMaRT: An approach to shingled magnetic recording translation. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 121–133.

[10] Yige Hu, Youngjin Kwon, Vijay Chidambaram, and Emmett Witchel. 2017. From crash consistency to transactions. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS'17)*. USENIX Association, 1–8.

[11] David A. Huffman. 1952. A method for the construction of minimum-redundancy codes. In *Proceedings of the Institute of Radio Engineers (IRE'52)*. IEEE, 1098–1101.

[12] Dong Hyun Kang and Young Ik Eom. 2017. TO FLUSH or NOT: Zero padding in the file system with SSD devices. In *Proceedings of the 8th ACM Asia-Pacific Workshop on Systems (APSys'17)*. ACM, 1–9.

[13] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'14)*. USENIX Association, 1–5.

[14] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. 2014. Durable write cache in flash memory SSD for relational and NoSQL databases. In *Proceedings of the International Conference on Management of Data (SIGMOD'14)*. ACM, 529–540.

[15] Hyukjoong Kim, Dongkun Shin, Yun Ho Jeong, and Kyung Ho Kim. 2017. SHRD: Improving spatial locality in flash storage accesses by sequentializing in host and randomizing in device. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 271–283.

[16] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'16)*. USENIX Association, 1–5.

[17] Wook-Hee Kim, Beomseok Nam, Dongil Park, and Youjip Won. 2014. Resolving journaling of journal anomaly in Android I/O: Multi-version B-tree with lazy split. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX Association, 273–285.

[18] Florian Lautenschlager, Michael Philippsen, Andreas Kumlehn, and Josef Adersberger. 2017. Chronix: Long term storage and retrieval technology for anomaly detection in operational data. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 229–242.

[19] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 273–286.

[20] Wongun Lee, Keonwoo Lee, Hankeun Son, Wook-Hee Kim, Beomseok Nam, and Youjip Won. 2015. WALDIO: Eliminating the filesystem journaling in resolving the journaling of journal anomaly. In *Proceedings of the USENIX Annual Technical Conferences (ATC'15)*. USENIX Association, 235–247.

[21] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. ffsck: The fast file system checker. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX Association, 1–15.

[22] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. 2015. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*. USENIX Association, 221–234.

[23] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. [n.d.]. SFS: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association.

[24] Daejun Park and Dongkun Shin. 2017. iJournaling: Fine-grained journaling for improving the latency of fsync system call. In *Proceedings of the USENIX Annual Technical Conference (ATC'17)*. USENIX Association, 787–798.

[25] Stan Park, Terence Kelly, and Kai Shen. 2013. Failure-atomic Msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*. ACM, 225–238.

[26] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan1 Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application crash consistency and performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, 181–196.

[27] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. 2009. Operating system transactions. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'09)*. ACM, 1–16.

[28] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. ACM, 206–220.

[29] Habibelahi Rahmani, Cihan Topal, and Cuneyt Akinlar. 2014. A parallel huffman coder on the CUDA architecture. In *Proceedings of IEEE Visual Communications and Image Processing Conference (VCIP'14)*. IEEE, 311–314.

[30] Abhishek Rajimwale, Vijayan Prabhakaran, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2011. Coerced cache eviction and discreet mode journaling: Dealing with misbehaving disks. In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN'11)*. IEEE, 518–529.

[31] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-tree filesystem. *Trans. Storage* 9, 3 (Aug. 2013), 9:1–9:32.

[32] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (Feb. 1992), 26–52.

[33] Kai Shen, Stan Park, and Meng Zhu. 2014. Journaling of journal is (almost) free. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'14)*. USENIX Association, 287–293.

[34] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS file system. In *Proceedings of the USENIX Annual Technical Conference (ATC'96)*. USENIX Association.

[35] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Mannarswamy, Terence Kelly, and Charles B. Morrey. 2015. Failure-atomic updates of application data in a Linux file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 203–211.

[36] Zev Weiss, Sriram Subramanian, Swaminathan Sundararaman, Nisha Talagala, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. ANViL: Advanced virtualization for modern non-volatile memory devices. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 111–118.

[37] Jeseong Yeon, Minseong Jeong, Sungjin Lee, and Eunji Lee. 2018. RFLUSH: Rethink the flush. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, 201–207.

[38] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Info. Theory* 23, 3 (May 1977), 337–343.

[39] Jacob Ziv and Abraham Lempel. 1978. Compression of individual sequences via variable-rate coding. *IEEE Trans. Info. Theory* 24, 5 (Sept. 1978), 530–536.

[40] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. 2014. Compression and SSD: Where and how? In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW'14)*. USENIX Association, 1–10.

Received April 2019; revised July 2019; accepted September 2019