# FJS: Fine-grained Journal Store

Shuanglong Zhang
Department of Computer Science
szhang@cs.fsu.edu

An-I Andy Wang
Department of Computer Science
awang@cs.fsu.edu

## Abstract

Journaling has been a popular reliability tool for file systems. However, journaling may involve writing updates twice, once to the journal, and once to potentially random update locations. The journal granularity tends to be coarse, leading to more bytes written than necessary.

We introduce Fine-grained Journal Store (FJS). Fine-grained Journal Store uses the journal as the final storage location for updates, thus eliminating double and random writes. Fine-grained journaling units are utilized to reduce the number of bytes written compared to the number of bytes updated. To limit the memory overhead of tracking small updatable units, FJS uses a range-based lookup table. In addition, FJS reuses existing journaling constructs to avoid the need for additional byte-addressable NVM storage devices.

We extended ext4 and jbd2 to prototype FJS. Our results show that FJS can outperform ext4 by up to 15x and reduce the write amplification of common metadata types by up to 5.8x.

***CCS Concepts*** **•Information systems~Record storage systems**

***Keywords***: journaling, file systems

**ACM Reference format:**
Anonymous Author(s). 2018. SIG Proceedings Paper in word Format. In *Proceedings of 13th ACM European Conference on Computer Systems, Porto, Portugal, April 2018 (EuroSys'18)*, 11 pages.
DOI: 10.1145/123_4

## 1 Introduction

Journaling is a common reliability technique used in modern file systems. The basic approach is to append all relevant updates to a log stored on stable storage before the file system propagates these updates to their final storage locations. While widely used, journaling interacts poorly with both hard disks and solid-state disks (SSD, e.g., commonly used NAND-based flash).

First, each write is performed twice; the propagation phase may involve writes to random storage locations. One extra round of writes can reduce file system performance for both disks and SSDs by 33% [Chen et al. 2016]. Second, only 10% of bytes journaled are actually updated [Chen et al. 2016]. Having many more bytes written than necessary has drastic negative reliability implications for SSDs, where each writable location can only tolerate a limited number of updates [Mellor 2016].

This paper introduces Fine-grained Journal Store (FJS), which directly stores the final updates in fine-grained units in the journal. Storing the final updates in the journal avoids double and random writes. The FJS further reduces the number of unnecessary bytes written, thereby mitigating the adverse effects of writes on the reliability of SSDs. The FJS is designed to be retrofit in an existing journaling framework; thus, it does not require special byte-addressable NVM storage devices (e.g., PCM).

## 2 Background

We will briefly review modern storage media characteristics and journaling mechanisms to better understand their interactions. We will then discuss their implications and some current alternatives.

***Hard disks***: A hard disk is a mechanical storage device with one or more rotating platters. Each platter is divided into concentric tracks, and each track is subdivided into storage sectors (e.g., 512B-4KB [Seagate 2017]). A disk request requires positioning a disk arm to a track, and waiting for the target sectors to rotate under the disk head before reading or writing the data. Due to the mechanical overhead, access to random disk locations can be two orders of magnitude slower than accessing adjacent disk locations (sequential access).

In recent years, shingled magnetic recording (SMR) drives [Wood et al. 2009] have become commercially available. SMR drives exploit the fact that a disk write head is wider than a read head. Therefore, a written track can overlap its previous track, similar to the way in which roof shingles overlap, as long as the non-overlapping region is wider than the read head for data retrieval (Figure 2.1). This also means that if a write is targeted to a track that is overlapped by another track, the write needs to rewrite the overlapped track as well. To limit the amount of rewriting, consecutive tracks are grouped and then separated by bands of non-overlapping regions. The performance implication of SMR is that disk writes can be significantly slower than reads due to the need to rewrite overlapping content.

**Figure 2.1.** Overlapping tracks on a shingled magnetic disk.

***SSDs***: Unlike disks, SSDs are electronic storage devices. Since they typically take the form of NAND-based flash, we will use the terms interchangeably. Flash is accessed in the unit of pages (e.g., 2KB). Without mechanical overhead, both sequential and random flash accesses are orders of magnitude faster than disks. However, for flash to overwrite the same page in place, a page needs to be cleared through an explicit erase operation before it can be written again, and an erase operation can be slower than a disk access. To amortize the cost, erase operations are carried out in flash blocks (e.g., 256KB), which contain many contiguous flash pages. To mask the latency of in-place writes and erase operations, a common practice is to remap the writes to pre-erased blocks.

While remapping writes works well, at some point an SSD needs to garbage collect pages with obsolete content. Ideally, an erase operation is issued to flash blocks containing all pages with obsolete content. However, the reality is that pages often hold up-to-date content that needs to be migrated to other pre-erased blocks. Such data movements amplify the amount of the data originally written. In addition, writes to random pages are more likely to trigger garbage collection events that involve flash blocks containing more in-use pages.

Another reliability constraint of flash is that each page can be written and erased for only a limited number of times (e.g., 1,000 – 100,000 [Mellor 2016]). Thus, writes, random writes, and amplified writes hurt the lifespan of flash.

***Journaling***: The idea of journaling can be traced back to the use of write-ahead logs in databases, where related updates are first grouped into transactions. A transaction is considered as *committed* when all the related updates are appended and written to a log on the stable storage. These updates are later written back to their final storage. Once all the operations have been successfully applied, the transaction is *checkpointed*, and the journal area used for the transaction can be freed. Recovery involves reapplying operations logged in the committed transactions since the last checkpoint.

To map to the context of a file system, the journal is the write-ahead log, and the updates logged are block-based (e.g., 4KB) and involve both data (e.g., file content) and metadata (e.g., storage allocation bitmaps, per-file i-node structures) operands. However, in practice, journaling file systems commonly omit data logging to trade consistency with performance since having consistent metadata provides sufficient file system integrity for most purposes [Sivathanu et al. 2005].

***Limitations of existing journaling***: As we juxtapose journaling and modern storage media, we identify the following three limitations:

*Double writes*: with journaling, each write is performed twice, once to the journal and once to the final destination. Double writing significantly increases the number of bytes written. For disks, extra bandwidth is needed for extra writes. For SSDs, which have a limited number of write-erase cycles, extra writes can significantly reduce the lifespan of the SSDs.

*Random writes*: the file-system writeback phase may trigger random writes. For disks, random writes involve slow seeks, and potential rewriting for SMR disks. For SSDs, random writes reduce their lifespans even faster as more garbage collection events trigger more writes to migrating in-use pages.

*Amplified writes*: the current block-based journaling granularity is too coarse-grained (4KB), as updates to file system metadata often involve changing a small fraction of bytes (e.g., updating the access timestamps for 256B i-nodes). The journaling of unchanged bytes leads to unnecessary writes, which can both slow down storage media and reduce the lifespan of SSDs.

***Some alternatives***: One alternative is to use variants of the log-structured file system (LFS) [Rosenblum and Ousterhout 1991; Lee et al. 2015], in which both data and metadata updates are represented and stored as log entries. LFS avoids double writes and associated random writes. However, LFS does not address the extra bytes written due to coarse-grained journaling. There are also systems that focus on fine-grained journaling [Kim et al. 2014; Hwang et al. 2015; Chen et al. 2016] by maintaining a mapping layer between the original journal and condensed representations. However, they rely on the use of byte-addressable persistent storage NVM devices such as PCM to protect the consistency of their mapping from crashes. Ideally, a framework can address double, random, and amplified writes, without resorting to the use of special hardware; this leads to our work.

## 3 Fine-grained Journal Store

We introduce Fine-grained Journal Store (FJS), which uses the file system journal to store the final metadata updates and avoid double and random writes. To mitigate amplified writes, we divide metadata blocks into fixed-sized subblocks for journaling. We also exploit the knowledge and format

information of metadata to accelerate the lookups of the offsets within subblocks, without resorting to mechanisms to compute and track deltas between blocks. To avoid the use of byte-addressable NVM persistent storage, we reuse the existing journaling mechanisms to store the mapping from journal entries to the original metadata block locations and modified offsets. The reverse mapping takes the form of soft states and can be reconstructed from the mapping stored in the journal. The following subsections detail the design of FJS.

### 3.1 Metadata Updates

FJS maintains an M2J lookup table in memory that maps <metadata block UID:modified subblock offset> to <journal block UID:journal subblock offset>. Both the metadata and journal block UIDs share the same namespace and are unique based on their persistent locations (i.e., major device number, minor device number, and block number). Thus, if a metadata block UID and subblock offset pair is mapped to itself (with the same UID and offset), the metadata subblock is not stored in the journal. Smaller subblocks mean finer-grained tracking of modified metadata regions; however, the tradeoff is having a larger M2J table. We used the square root of the block size, rounded to the nearest power of two. In this case, it comes to sixty-four 64B subblocks for 4KB metadata blocks.

When a metadata update arrives, FJS first checks to see whether the M2J table contains the metadata block entry. If not, an M2J table entry is created containing the target metadata block UID. Having the knowledge of metadata type, such as i-node, we can identify which four 64B subblocks within the metadata block contain the 256B i-node. Also, by wrapping all updates to attributes with inline macros, we can identify the 64B subblock within the i-node that is modified. The persistent journaling location is yet to be determined (TBD) at the commit time.

If an entry already exists when an update arrives, and if the update has not been committed, nothing needs to be done. If the update has been committed, the existing journal-block UID and the subblock offset will be set to <TBD:TBD>, so that the next commit will only log subblocks updated since the last commit.

In all cases, the update is still performed on the metadata cached in memory. The in-memory journaling mechanisms still track modified metadata at the granularity of a block. In addition, FJS assures that none of the modified 4KB metadata blocks cached will be flushed to the stable storage (see §3.3).

### 3.2 Range-based Metadata to Journal Subblock Mapping

Table 3.2.1 provides an example of the M2J table, sorted first by the metadata block UID, and then by its subblock offset. The first row shows that the metadata block 1 and subblock 0 pair, denoted as <1:0>, is mapped to <1:0> or itself. This means this subblock has not been updated, and is neither mapped nor stored in the journal. The subblock <1:33> has been updated, and will be mapped when this block is committed. The location of the journal block and its subblock are yet to be determined, denoted as <TBD:TBD>.

One observation is that the mapping status of a subblock in a shaded row can be deduced by the previous row, until its mapping status changes. In addition, the mapped subblock numbers are largely consecutive, unless otherwise specified. Thus, we can omit storing the shaded rows. If subblocks 0-3 of metadata block 3 are mapped to different journal blocks or nonconsecutive journal subblocks, new entries will be created to reflect these mappings. Therefore, individual metadata blocks tend to have relatively few entries. (We could further assume the common case that the first subblock is not modified to save one more row.)

**Table 3.2.1.** M2J metadata block UID to journal block UID table. The content of a shaded subblock entry can be deduced from the previous entry and is not stored.

| Metadata block UID | Metadata subblock offset | Journal block UID | Journal subblock offset |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |
| 1 | 2 | 1 | 2 |
| 1 | … | 1 | … |
| 1 | 32 | 1 | 32 |
| 1 | 33 | TBD | TBD |
| 1 | 34 | 1 | 34 |
| 1 | 35 | 1 | 35 |
| 1 | … | 1 | … |
| 1 | 62 | 1 | 62 |
| 1 | 63 | 1 | 63 |
| 3 | 0 | TBD | TBD |
| 3 | 1 | TBD | TBD |
| 3 | 2 | TBD | TBD |
| 3 | 3 | TBD | TBD |
| 3 | 4 | 3 | 4 |
| 3 | 5 | 3 | 5 |
| 3 | … | 3 | … |
| 3 | 62 | 3 | 62 |
| 3 | 63 | 3 | 63 |

### 3.3 Journal Commit

Continuing with the Table 3.2.1 example. Let us now suppose metadata blocks 1 and 3 belong to the same transaction. When the journal commits, instead of flushing modified 4KB metadata blocks 1 and 3, FJS packs only the modified 64B subblocks <1:33>, <3:0>, <3:1>, <3:2>, and

<3:3> (the ones mapped to <TBD:TBD> into a temporarily allocated 4KB memory block that is committed and flushed to the persistent journal (Figure 3.3.1). Suppose the journal block UID is 11. The in-memory M2J table will contain the states shown in Table 3.3.1.

The journal also persistently stores a per-transaction J2M mapping from <journal block UID:subblock offset> to <metadata block UID:subblock offset>, as shown in Table 3.3.2. The table is written to the journal before writing the journal block(s) containing metadata subblocks.

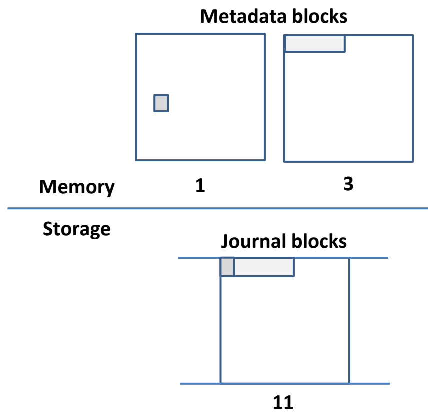


**Figure 3.3.1.** Modified metadata subblocks <1:33>, <3:0>, <3:1>, <3:2>, and <3:3> (shaded) are packed and written to journal block 11.

**Table 3.3.1.** The M2J table after committing <1:33>, <3:0>, <3:1>, <3:2>, and <3:3> subblocks to journal block 11. The TBD entries are changed (highlighted in bold font).

| Metadata block UID | Metadata subblock offset | Journal block UID | Journal subblock offset |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 1 | 33 | 11 | 0 |
| 1 | 34 | 1 | 34 |
| 3 | 0 | 11 | 1 |
| 3 | 4 | 3 | 4 |

**Table 3.3.2.** The J2M table after committing <1:33>, <3:0>, <3:1>, <3:2>, and <3:3> subblocks to journal block 11.

| Journal block UID | Journal subblock offset | Metadata block UID | Metadata subblock offset |
|---|---|---|---|
| 11 | 0 | 1 | 33 |
| 11 | 1 | 3 | 0 |
| 11 | 4 | 11 | 4 |

### 3.4 Bootstrapping and Metadata Lookups

After a reboot, the in-memory M2J table is initially missing. This indicates that the journal needs to be replayed to reconstruct the M2J table content, based on the per-transaction J2M tables persistently stored in the journal. Then, when a metadata request arrives, the target metadata block is initially not cached, and must be read from the original storage location (not from the journal). The return path of the metadata block request is intercepted with checks against the M2J table to copy the committed up-to-date subblocks stored in the journal over the target metadata block. Afterwards, the in-memory metadata block is up-to-date and can serve metadata lookup and update requests.

### 3.5 Crash Recovery

If a crash occurs when no transactions are being committed, all metadata updates since the last commit are lost. If a crash occurs while a transaction is being committed (before the commit marker becomes persistent), all metadata updates since the last commit will be discarded. For both cases, upon reboot, the FJS can repeat the steps described in §3.4 to recover the in-memory M2J table and continue with the operations.

### 3.6 Space Reclamation within the Journal

A journal is typically represented as a circular log. To enable efficient sequential logging, a journal needs to reclaim scattered, unused space and compact in-use space. However, since the journal block is the persistent storage location for metadata subblocks under FJS, a journal cannot reclaim journal blocks of a transaction until all involved metadata subblocks are obsolete. That is, the most up-to-date version of the metadata subblock must be stored in journal blocks of other transactions.

The space-reclamation process involves traversing the journal log from the oldest transaction to the most recent, in batches. Each per-transaction J2M table encountered is checked against the in-memory M2J table. Suppose according to the J2M Table 3.3.2, journal subblock <11:0> is mapped to metadata subblock <1:33>. However, if the in-memory M2J table shows metadata subblocks are mapped elsewhere, say to journal subblock <12:0>, then journal subblock <11:0> is obsolete. If all journal subblocks within a single transaction are obsolete, journal blocks belonging to this transaction can be garbage collected.

On the other hand, since journal subblock <11:1> is mapped to metadata block <3:0> according to J2M Table 3.3.2, and M2J Table 3.3.1 confirms that the subblock mapping is current, this transaction holding journal block 11 cannot be garbage collected. To enable garbage collection, we use a list to track all up-to-date journal subblocks in chronological order; this list is prepended to the list of the metadata

subblocks to be committed in the current transaction. In other words, up-to-date metadata subblocks are migrated through recommitting themselves to other journal blocks in the current transaction.

If this transaction fails, the up-to-date metadata subblocks are still in the original journal locations. The in-memory M2J table needs to be reconstructed through replaying and retrieving the per-transaction J2M tables stored in the journal. If this transaction succeeds, all blocks related to transactions prior to garbage collection can be reclaimed. The in-memory M2J table will be updated accordingly.

The garbage collection and FJS commit are in the same thread, so there is no need to coordinate them. In the event that the journal space is near exhaustion, FJS (as opposed to the file system above) needs to checkpoint the up-to-date modified metadata subblocks to their original metadata blocks.

## 4 Implementation

We have prototyped FJS under Linux by extending the jbd2 journaling layer, to leverage its handling of tricky corner cases such as committing while receiving updates. We also extended ext4, one of the most widely used file systems, to help FJS identify the metadata subblocks modified within metadata blocks. We used the popular ordered mode for jbd2, where only metadata are journaled every 5 seconds (default), while the associated data are flushed prior to metadata journaling. FJS was implemented with ~2,000 lines of C code.

***Metadata updates***: Currently, we track and handle ext4 metadata updates at the granularities of 64B for block group descriptors and bitmaps, 256B for i-nodes, and 1KB for superblocks. Other forms of metadata (e.g., content of directories, extent-tracking blocks) are still journaled and written back periodically with the semantics of the ext4 ordered mode.

The M2J table is checked for creating and updating entries when ext4 invokes `ext4_mark_inode_dirty()` and various block allocation/deallocation functions. In addition, the modified 4KB metadata blocks are marked as not dirty to prevent them from being written back to their original storage locations.

***M2J table***: We used a red-black tree to implement the M2J table, indexed by the metadata block UIDs, so that each node corresponds to a metadata block. In addition, each node stores the M2J table entries related to the metadata block.

***Journal commit***: We have intercepted the kjournald commit thread, so that only temporary buffers packed by modified metadata subblocks are flushed to the journal, instead of the `buffer_head` structures mapped by the `journal_head` structures. Also, the per-transaction journal descriptor block is modified to store the J2M mapping table.

***Bootstrapping and recovery***: We have modified the mount command to accept a `-fjs` flag to trigger journal replay and the reconstruction of the in-memory M2J table. FJS will always perform recovery at mount time, even if the system is shut down normally. For now, the FJS journal format is not backward compatible; once the journal is converted to the FJS format, all future mounts will require the `-fjs` option to be turned on. Future work will enable the option to revert to the jbd2 journal format at mount time by writing the subblocks back to their original locations.

***Metadata read***: once the M2J table is reconstructed, the return paths of `submit_bh()`,`bh_submit_read()`, and `ll_rw_block()` are intercepted. Sine a metadata block is read from the stable storage as a cache miss, FJS will consult with the M2J table to copy the corresponding modified subblocks (if any) from the persistent journal subblocks over the subblocks of the metadata block. Thus, the in-memory metadata blocks will contain the up-to-date information for future reads and updates.

***Journal space reclamation***: The journal space reclamation process is modified according to §3.6. The space reclamation process is triggered once the journal is 50% full. We have not encountered journal space exhaustion; thus, we have not implemented the handling of such cases.

**Table 5.1.** System configurations.

| | |
|---|---|
| Processor | 2.2GHz Intel® E5-2430, with 64KB L1 cache, 256KB, and 15MB cache |
| Memory | 32 GB RDIMM 1,333 MT/s |
| HDD | Seagate® Savvio ST9146853SS, 15K RPM, 146GB SAS HDD, with 64MB cache |
| SMR HDD | Seagate® Archive ST8000AS0002 8TB drive, with 128MB cache |
| SSD | Intel® DC S3700 SSDSC2BA200G3R 200GB SATA SSD |
| Operating System | Linux 4.13 |

## 5 Evaluation

We used kernel compilations and Filebench 1.5-alpha3 [Tarasov et al. 2016] to compare the performance of FJS with the baseline ext4 using the ordered mode. We used two servers with identical configurations, except that one is hosting 6 HDDs and one is hosting 6 SSDs and 2 SMR HDDs (Table 5.1). We conducted our experiments on only one storage device from each category. Each experiment

setting was repeated five times. The 90% confidence intervals tend to be narrow (< 5%) and are omitted for clarity.

### 5.1 Kernel Compilations

The purpose of the kernel compilation benchmark is to show the correctness of FJS implementation. Also, the results can assess whether the use of the journal as the final storage destination will consume journal space rapidly and impose high memory consumptions for mapping subblocks. Since compilations are CPU-bound, we anticipated no performance differences.

We compiled Linux kernel 4.13 using make `-j6` flag to allow multithreaded compilations on our multicore machine. Table 5.1.1 shows that compilation times for ext4 and FJS are the same as expected. The server hosting SSDs and SMR HDDs took 4.8% longer, but within the variations across machines of the same configurations. To confirm this hypothesis, we compiled the kernel on ramfs, which demonstrates similar performance discrepancies.

**Table 5.1.1.** Kernel compilation results.

| | Compilation time | Journal writes | Metadata writes | FJS memory overhead |
|---|---|---|---|---|
| ext4 | | | | |
| HDD | 1,547 secs | 173MB | 94MB | 0.0MB |
| SSD | 1,621 secs | 173MB | 94MB | 0.0MB |
| SMR HDD | 1,618 secs | 173MB | 94MB | 0.0MB |
| FJS | | | | |
| HDD | 1,548 secs | 60MB | 0.0MB | 8.9MB |
| SSD | 1,621 secs | 60MB | 0.0MB | 8.9MB |
| SMR HDD | 1,615 secs | 60MB | 0.0MB | 8.9MB |
| ramfs | | | | |
| HDD host | 1,552 secs | n/a | n/a | 0.0MB |
| SSD and SMR host | 1,625 secs | n/a | n/a | 0.0MB |

For the types of metadata tracked by FJS, FJS retains all metadata and reduces the amount of ext4 journal writes by 35% and eliminates their associated file-system metadata writebacks. Thus, FJS reduces the write amplification of metadata by 4.5x (173MB + 94MB / 60MB), not counting the additional amplification by the garbage collection at the flash level. The range-based M2J subblock mapping table incurred only 8.9MB. These numbers show that the overhead of FJS is sufficiently low for real-world deployment.

### 5.2 Microbenchmarks

We configured Filebench to perform six microbenchmarks, with designs similar to the ones in [Aghayev et al. 2017]. Table 5.2.1 describes each microbenchmark.

**Table 5.2.1.** Filebench microbenchmarks.

| | |
|---|---|
| make directories | Create a directory tree with 500K directories, with each directory containing 5 subdirectories on average. |
| list directories | Run `ls -lR` on the directory tree. |
| remove directories | Remove the directory tree recursively. |
| create files | Create 500K 4KB files in a directory tree, with each directory containing 5 files on average. |
| find files | Run `find` on the directory tree. |
| remove files | Remove the files and directory tree recursively. |

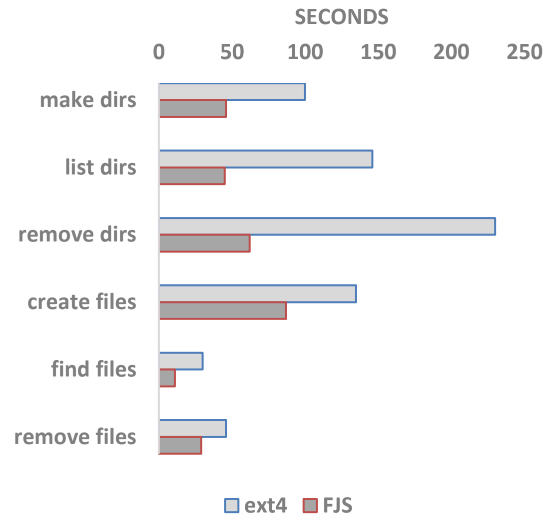


**Figure 5.2.1:** Comparison of HDD performance between FJS and ext4 running Filebench microbenchmarks.

Figure 5.2.1 compares the HDD performance of FJS with that of ext4. Creating directories involves allocating new blocks and updating metadata. For ext4, a writeback thread periodically writes the updated metadata blocks to their original storage locations, while FJS eliminates many of these writes. In addition, an ext4 directory creation involves a minimum of four 4KB journal writes to the superblock, the block group descriptor block, the i-node bitmap block, and the block containing the i-node. FJS, on the other hand, can compact subblocks involved in the creation of multiple directories into a single journal block, resulting in a 2.2x speedup for directory creation.

Creating files involves additionally writing a 4KB data block, which is not optimized by FJS, resulting in a smaller 1.6x improvement. The performance of other operations reflects how well FJS captures the locality of metadata to be accessed. The throughput improvement is as high as 3.7x for directory removals.

From a different view, Figure 5.2.2 shows that the locations of ext4 metadata writes over the duration of the make directory microbenchmark for a single run. Due to the high volume of data, for n events we only plotted every $\lceil n/2000 \rceil^{\text{th}}$ event. The '-' markers show writes to the final storage locations of metadata, and 'x' markers show writes to the journal, which is located in the middle tracks of the disk platters, to reduce the frequency of seeks to access the journal.

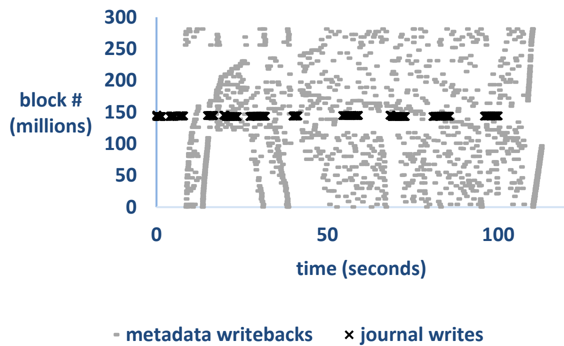


**Figure 5.2.2:** ext4 metadata write locations over time for the make directory microbenchmark.

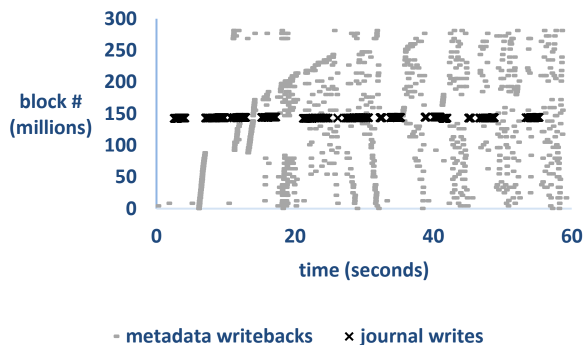


**Figure 5.2.3**: FJS metadata write locations over time for the make directory microbenchmark. FJS journal persistently stores super blocks, block group descriptors, i-nodes, and bitmaps. The remaining metadata blocks are written back according to the semantics of ext4 ordered mode.

For FJS (Figure 5.2.3), since the journal persistently stores the superblock, block group descriptors, i-nodes, and bitmaps, the writeback requests to the remaining metadata types are 1.6x fewer (Figure 5.2.6) compared to the ext4 case. On the other hand, for this microbenchmark, the journal has wrapped around three times, or once every 20 seconds, indicating that keeping the remaining metadata persistent would require a journal space much larger than the existing 1GB.

Figure 5.2.4 compares the SSD performance of FJS with that of ext4. Given that SSD has no mechanical seeks, the expected FJS performance improvement is less than that of disks. The speedup for directory creation is reduced to 41%, and the performance for file creation is almost negligible (4.1%). However, the speedup for operations such as directory removals is still improved by 1.6x.

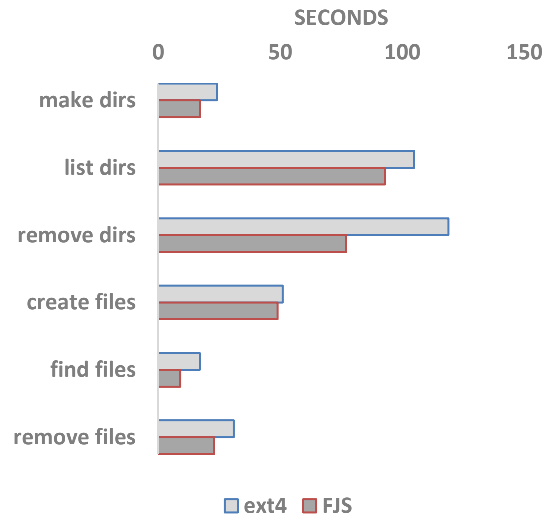


**Figure 5.2.4:** Comparison of SSD performance between FJS and ext4 running Filebench microbenchmarks.

Figure 5.2.5 compares the SMR HDD performance of FJS with that of ext4. The performance of SMR HDDs are significantly slower than conventional HDDs due to the need to rewrite overlapped track regions if updated. The FJS's approach to avoid double, random, and amplified writes can speed up directory creation by 2.9x and file creation by 1.5x. For other operations, FJS can speed up directory removals by 15x, file finds by 12x, listing directories by 9.6x, and file removals by 6.2x.

Figure 5.2.6 shows that FJS reduces the amount of metadata written over ext4 by up to 3.1x.

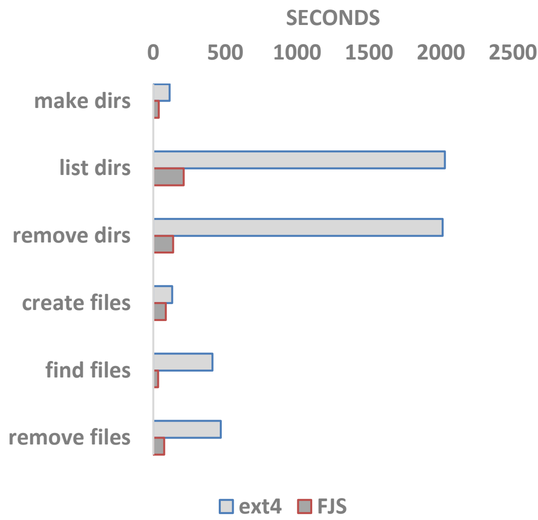


**Figure 5.2.5:** Comparison of SMR HDD performance between FJS and ext4 running Filebench microbenchmarks.

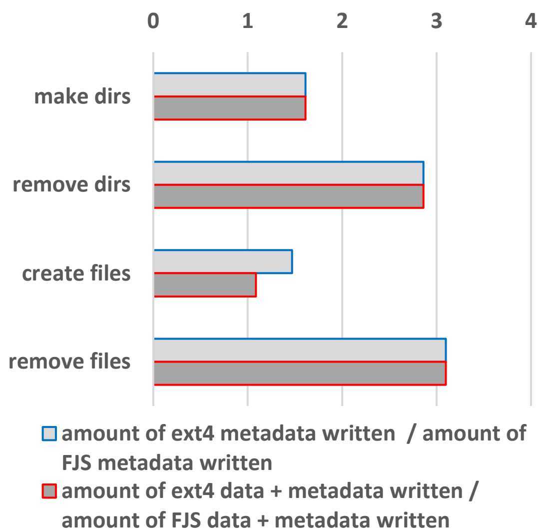


**Figure 5.2.6**: Write reduction factor of FJS over ext4 for the Filebench microbenchmarks.

## 5.3 Macrobenchmarks

We configured Filebench to emulate five workloads, each with a working set size of ~50GB.

The fileserver benchmark involves create-write-close operations, open-append-close operations, open-read-close operations, deletes, and stats. The total number of files is set to 800K, accessed by 50 threads. The mean append size is 16KB.

The varmail benchmark exercises deletions, create-append-fsync-close operations, open-read-append-fsync-close operations, and open-read-close operations. The total number of files is set to 1M accessed by 16 threads. The mean append size is 4KB.

The OLTP benchmark emulates asynchronized writes with many semaphore locks. The total number of files is 500, with an average append size of 2KB.

The webserver benchmark involves open-read-close operations and append operations. The total number of files is set to 800,000, accessed by 100 threads. The mean append size is 16KB.

The webproxy benchmark involves deletes, create-append-close operations, and open-read-close operations. The total number of files is set to 1M, accessed by 100 threads. The mean append size is 16KB.

Figures 5.3.1 and 5.3.2 show the results for traditional HDDs and SSDs. We have omitted the experiments for SMR HDDs, since they took one to two orders of magnitude longer to run, indicating that SMR HDDs are unsuitable for these workloads. In both cases, the FJS improvements are modest across the board. Even the read-mostly webserver workload gained 7.7% in throughput, and HDDs benefit more from the locality then SSDs, due to the reduction of mechanical seeks. Since FJS excels in compact logging of allocation and deallocation related updates, FJS improves web proxy throughput by up to 21%, followed by varmail improvement of up to 14%.

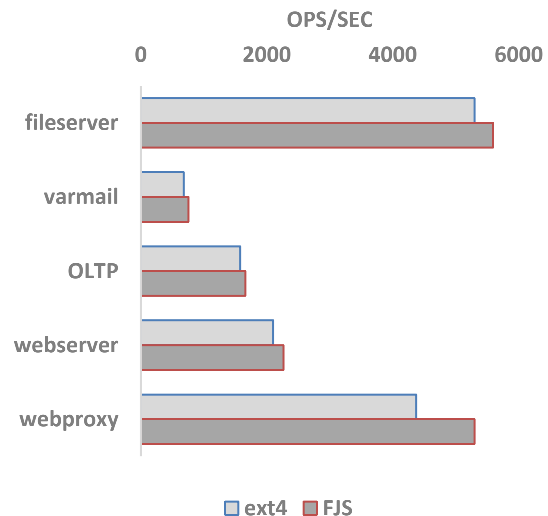


**Figure 5.3.1:** Comparison of HDD performance between FJS and ext4 running Filebench macrobenchmarks.

**Figure 5.3.2:** Comparison of SSD performance between FJS and ext4 running Filebench macrobenchmarks.

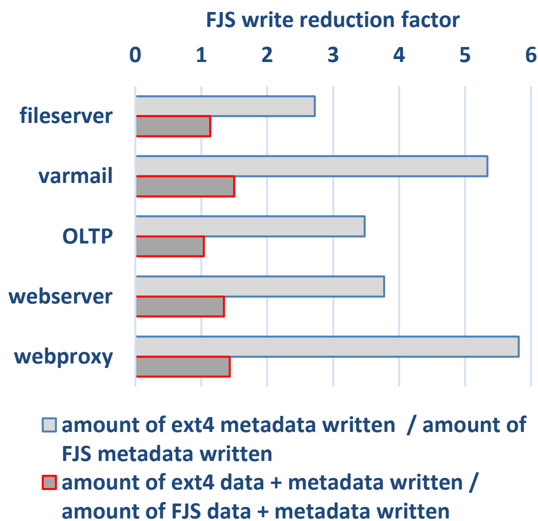


**Figure 5.3.3**: Write reduction factor of FJS over ext4 for the Filebench macrobenchmarks.

Figure 5.3.3 shows that FJS reduces the amount of metadata written over ext4 by up to 5.8x. However, when both metadata and data are considered, this write reduction factor can only be as high as 1.5x. The overall write reduction factor correlates well with the performance benefits of FJS.

## 6 Related Work

Over the years, many researchers have sought to reduce the double, random, and amplified writes associated with the journaling and file system writeback mechanisms. However, few of them address all three forms of journal writes.

***Avoiding journal double writes***: One approach to avoid journaling is to have copy-on-write semantics on file system tree nodes. Thus, whenever a leaf tree node is modified, all the updated nodes from the leaf node to the root are cloned and updated, leaving behind a consistent snapshot of the old tree. Some example file systems include ZFS [Bonwick and Moore 2009] and BTRFS [Rodeh et al. 2013].

Another approach to avoid journal writes involves the use of byte-addressable NVM. Lee et al. [2013] proposed to combine the roles of buffer cache and journal so that journal commit can be performed in place, by changing the state of a cached block instead of copying the updates to the journal.

***Avoiding double journal writes and random file-system writebacks***: The log-structured file system (LFS) [Rosenblum and Ousterhout 1991] is also a copy-on-write file system designed to optimize writes. By structuring both data and metadata as log entries and by having the log as the final storage destination for data and metadata, each update only needs to be written once. The LFS approach has been applied to many systems to optimize writes. Notably, DualFS [Piernas et al. 2002] used separate storage devices for metadata and data. The metadata storage device used an LFS layout to avoid double and random writes. hFS [Zhang and Ghose 2007] uses an LFS partition in the middle tracks of disks to store small files and metadata. F2FS [Lee et al. 2015] used a modified version of LFS to optimize random writes.

Aghayev et al. [2017] modified ext4 for SMR HDDs. Specifically, their approach allows frequently accessed metadata to be stored in the journal, while infrequently accessed metadata are written back to their original locations.

***Reducing amplified writes caused by journaling***: Many approaches to compact journals have been proposed, mostly in the context of byte-addressable NVM such as PCM. Kim et al. [2014] propose to compare the original and modified block via XOR. If the differences are small, the updates are journaled using the NVM device. Otherwise, updates are journaled using the NAND-flash device. Hwang et al. [2015] proposed the use of a two-level tracking scheme for 128B updates to memory blocks, with the tracking table stored in NVM. Chen et al. [2016] proposed a more compact journal format for byte-addressable NVM. The transaction representation removes the journal descriptor and commit block and logs only modified i-node.

***Avoiding double, random, and amplified journal writes***: Lu et al. [2013] propose an object-based flash translation layer (OFTL) object store approach to manage NAND flash.

The interface exposes the byte range so that OFTL can compact the partially updated flash pages. OFTL uses backpointers to metadata to avoid journaling. The implementation is based on log-structured merge trees, so OFTL also largely eliminates the random writes. To run legacy workloads, the authors built an object file system to interface with OFTL. Block-level traces from legacy file systems are fed into a flash translation layer simulation to generate flash read/write/erase operations comparable to those of OFTL. The write amplification is reduced by up to 89%.

Overall, the FJS framework combines and extends the techniques used in LFS, in-place commit, and journal compaction, and retrofits these mechanisms with existing journaling and storage devices.

## 7 Discussions and Future Work

The original FJS design aimed to store individual i-node attributes in separate persistent journals for improved compression. While the approach reduced double, random, and amplified writes well, different journal attributes' update frequencies introduced complications. In addition, separate journal logs did not exploit the internal parallelism of SSDs as well as anticipated, for reasons explained in [He et al. 2017].

We then decided to use fixed-size subblocks and extended the types of metadata being handled to the current incarnation of FJS, which still exploits some level of metadata knowledge to locate subblocks within a block, and reuses the journal to avoid the use of the byte-addressable NVM. FJS's benefits can be realized in various leading storage media as well.

However, FJS can be enhanced in a number of ways. For example, we can wrap all file system calls to update the i-node attributes to indicate which 64B subblock has been updated, as opposed to marking four 64KB subblocks as updated. We can also extend our tracking to handle metadata types beyond super blocks, block group descriptors, i-nodes, and bitmaps.

## 8 Conclusions

We have presented the design, implementation, and evaluation of FJS, which permanently stores updated metadata subblocks in a file system journal to avoid double, random, and amplified writes as well as the need for byte-addressable persistent storage. The resulting performance improvements span across conventional HDDs, SMR HDDs, and SSDs under a variety of workloads. The FJS research shows that it is possible to integrate the techniques used in LFS, in-place journal commit, and journal compaction to meet diverse system constraints, leading to a practical, deployable system.

## Acknowledgments

We would like to thank Brandon Stephens and Erika Dennis for reviewing an early draft of this paper. This work was sponsored by FSU. The opinions, findings, conclusions, or recommendations expressed in this document do not necessarily reflect the views of FSU or the U.S. Government.

## References

[Aghayev et al. 2017] Aghayev A, Ts'o T, Gibson G, Desnoyers P. Evolving Ext4 for Shingled Disks. *Proceeding of the 15th USENIX Conference on File and Storage Technologies (FAST)*, 2017.

[Bonwick and Moore 2009] Bonwick J, Moore B. ZFS: The Last Word in File Systems. https://www.snia.org/sites/default/orig/sdc_archives/2009_presentations/monday/JeffBonwickzfs-Basic_and_Advanced.pdf, 2009.

[Chen et al. 2016] Chen C, Yang J, Wei Q, Wang C, Xue M. Fine-grained Metadata Journaling on NVM. *Proceedings of the 32nd IEEE Symposium on Mass Storage and Technologies (MSST)*, 2016.

[Hwang et al. 2015] Hwang Y, Gwak H, Shin D. Two-level Logging with Non-volatile Byte-addressable Memory in Log-structure File Systems. *Proceedings of the 12th ACM International Conference on Computing Frontiers*, 2015.

[He et al. 2017] He J, Kannan S, Arpaci-Dusseau AC, Arpaci-Dusseau RH. The Unwritten Contract of Solid State Drives. *Proceedings of the 12th ACM European Conference on Computer Systems (Eurosys)*, 2017.

[Kim et al. 2014] Kim J, Min C. Eom YI. Reducing Excessive Journaling Overhead with Small-Sized NVRAM for Mobile Devices. *IEEE Transactions on Consumer Electronics*. 60(2):217-224, July 2014.

[Lee et al. 2013] Lee E, Bahn H, Noh SH. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[Lee et al. 2015] Lee C, Sim D, Hwang JY, Cho S. F2FS: A New File System for Flash Storage. *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[Lu et al. 2013] Lu Y, Shu J, Zheng W. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. *Proceedings of the 11th USENIX*

*Conference on File and Storage Technologies (FAST)*, 2013.

[Mellor 2016] Mellor C. QLC Flash is Tricky Stuff to Make and Use, So Here's a Primer. https://www.theregister.co.uk/2016/07/28/qlc_flash_primer/, July 2016.

[Piernas et al. 2002] Piernas J, Cortes T, Garcia JM. DualFS: A New Journaling File System without Metadata Duplication. *Proceedings of the 2002 International Conference on Supercomputing (ICS)*, 2002.

[Rodeh et al. 2013] Rodeh O, Bacik J, Mason C. BTRFS: The Linux-B-Tree Filesystem. *ACM Transactions on Storage (TOS)*, 9(3), Article No. 9, 2013.

[Rosenblum and Ousterhout 1991] Rosenblum M, Ousterhout JK. The Design and Implementation of a Log-structured File System. *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[Seagate 2017] Seagate. Transition to Advanced Format 4K Sector Hard Drives. https://www.seagate.com/tech-insights/advanced-format-4k-sector-hard-drives-master-ti/, 2017.

[Sivathanu et al. 2005] Sivathanu M, Arpaci-Dusseau AC, Arpaci-Dusseau RH, Jha S. A Logic of File Systems. *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[Tarasov et al. 2016] Tarasov V, Zadok E, Shepler S. Filebench: A Flexible Framework for File System Benchmarking, ;login, 40(1):6-12, 2016.

[Wood et al. 2009] Wood R, Williams M, Kavcic A, Miles J. The Feasibility of Magnetic Recording at 10 Terabits Per Square Inch on Conventional Media. *IEEE Transactions on Magnetics*. 45(2):917-923, February 2009.