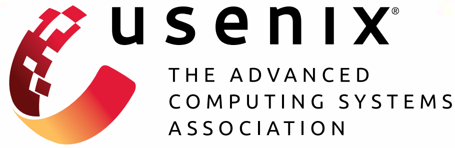

FastCommit: resource-efficient, performant and cost-effective file system journaling

Harshad Shirwadkar, Saurabh Kadekodi, and Theodore Tso, *Google*

https://www.usenix.org/conference/atc24/presentation/shirwadkar

This paper is included in the Proceedings of the
2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the
2024 USENIX Annual Technical Conference
is sponsored by

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

FASTCOMMIT: resource-efficient, performant and cost-effective file system journaling

Harshad Shirwadkar
Google

Saurabh Kadekodi
Google

Theodore Tso
Google

Abstract

JBD2, the current physical journaling mechanism in Ext4 is bulky and resource-hungry. Specifically, in case of metadata-heavy workloads, fsyncs issued by applications cause JBD2 to write copies of changed metadata blocks, incurring high byte and IO overhead. When storing data in Ext4 via NFS (a popular setup), the NFS protocol issues fsyncs for every file metadata update which further exacerbates the problem. In a simple multi-threaded mail-server workload, JBD2 consumed approximately 76% of the disk's write bandwidth. Higher byte and IO utilization of JBD2 results in reduced application throughput, higher wear-out of flash based media and increased performance provisioning costs in cloud-based storage services.

We present FASTCOMMIT: a hybrid journaling approach for Ext4 which performs logical journaling for simple and frequent file system modifications, while relying on JBD2 for more complex and rare modifications. Key design elements of FASTCOMMIT are *compact logging*, *selective flushing* and *inline journaling*. The first two techniques work together to ensure that over 80% commits are contained within a single 4KB block and are written to disk without requiring an expensive cache flush operation. *Inline journaling* minimizes context switching delays. With faster and efficient fsyncs, FASTCOMMIT reduces throughput interference of JBD2 by over 2× along with throughput improvements of up to 120%. We implemented FASTCOMMIT in Ext4 and successfully merged our code to the upstream Linux kernel.

1 Introduction

In this paper, we focus on reducing the application slowdown, resource-inefficiency and high consumer cloud costs associated with JBD2, the crash consistency journaling system used by Ext4[1]. JBD2 is a physical journal, which means that it stores entire copies of modified file system metadata blocks such as free block bitmaps, inodes, directory entries, etc. as part of a JBD2 commit on application issued fsyncs. A physical journal (including JBD2) is designed to be simple and efficient during recovery, but incurs a high byte and IO overhead during commits. For instance, JBD2 stores a minimum of three blocks (12KB), and on average 6 blocks (24KB) per commit via two separate IO operations. In a multi-threaded

[1]Ext4 is the default file system shipped with most Linux distributions.

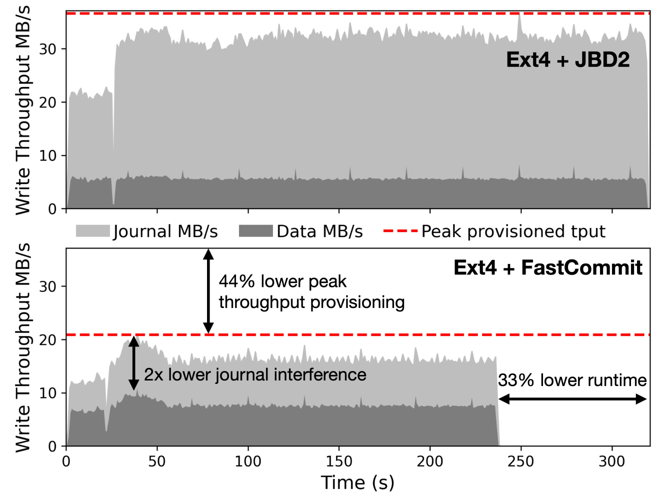

Figure 1: The breakdown of a mail-server workload write bandwidth showing what fraction of it is journal vs user data. The top plot shows the conventional setup of Ext4+JBD2 where journal bandwidth is over 4× the user bandwidth. The bottom plot shows the same workload executed on Ext4+FASTCOMMIT where the journal consumes approximately 1× user data bandwidth enabling 44% lower peak throughput provisioning and 33% lower end-to-end runtime.

mail-server workload, JBD2 reduced end-to-end application throughput by 45%, and increased fsync latency by 280%[2]. To compensate for its resource-inefficiency JBD2's design implicitly relies on fsyncs being issued rarely by applications compared to the number of file manipulations. This assumption is flagrantly violated when Ext4+JBD2 (or just JBD2) is used as a backend for the Network File System (NFS) protocol [27]. Specifically, NFS by default converts every file close to close+fsync (async mode), and also supports stricter semantics (sync mode) where every file update issued by the client becomes an update+fsync. The same mail-server workload issues 26× more fsyncs on NFS-async reducing the end-to-end throughput by 8×. Figure 1 (top) shows that 76% of the write bandwidth is consumed by JBD2 alone in the NFS-async setup.

Another popular setup is to use JBD2 atop cloud-based virtualized block storage devices (VBD) such as Google's Persistent Disk [12], Amazon Elastic Block Service [2] or Azure Disk Storage [24] that manage many exabytes of data.

[2]In this experiment we made JBD2 RAM-resident preventing it from being the bottleneck.

Resource-efficiency is paramount in these globally available services, and in fact modern VBD offerings allow independent provisioning and purchasing of capacity, throughput and IOPS for flexibility. JBD2's high overheads either result in throttling, which reduces application performance, or require the cloud consumer to purchase additional throughput and IOPS, which is costly.

A natural alternative is to build purely logical journaling in Ext4 (similar to XFS [3]), where just the operations are journaled rather than journaling bulky copies of changed metadata blocks. However, this option would require a comprehensive redesign of the current JBD2 journaling subsystem, including modifications to the on-disk format and re-instrumentation of journaling transactions, effectively making adoption impractical. Moreover, the implementation of a fully logical journal would require designing an intricate recovery process that would take years of development and refinement to attain maturity. Additionally, as users migrate their applications to public clouds, a popular approach is to replicate their on-premises setup as-is to the virtualized cloud environment. Expecting users to redesign their application as part of cloud migration is highly impractical.

We present FASTCOMMIT: a hybrid logical+physical journal that is resource-efficient and minimizes journal overhead. By default, a JBD2 physical journal commit is triggered at every fsync, and at a frequency of every 5 seconds. For fsync-heavy workloads, or in NFS, JBD2 commits are very frequent. A cornerstone of FASTCOMMIT's design is to add logical journaling between successive JBD2 commits, which logs the operation that manipulated files and directories on every fsync rather than storing the entire metadata. This essentially decouples fsyncs from bulky JBD2 commits, and thus reduces fsync latency which in turn improves performance.

FASTCOMMIT's resource-efficiency is attributed to three fundamental techniques. First, reducing byte overhead via *compact logging (FCLog)*. In a single FCLog, FASTCOMMIT packs numerous file system updates (*FCTags*), and an FCLog fits entirely within a single disk block (4KB). Second, reducing IO overhead via *selective flushing*. FClogs that fit in a single block can be written durably to the underlying storage media using the Forced Unit Access (FUA) IO command [15]. These commits do not need to issue expensive cache flush commands [36] (unless there was any data that needed to be flushed during the transaction commit). Since JBD2 by design stores at least 3 blocks to the journal on each commit, it cannot make the commit durable without a cache flush. Third, reducing context-switching delays via *inline journaling*. JBD2 commits are done using a dedicated thread. Since FASTCOMMIT commits are tiny and fast, the scheduling priority of the thread issuing the fsync is temporarily elevated to match that of the JBD2 thread to perform the commit. This prevents context switching delays and improves fsync latency. Figure 1 (bottom) shows that the fraction of application write bandwidth consumed by the journal reduces by over 44% with FASTCOMMIT. This not only improves resource-efficiency but also reduces total application runtime by over 33%. Lastly, FASTCOMMIT is carefully designed to reuse the same JBD2 hooks and APIs, while providing most of the performance and cost benefits of purely logical journaling. Thus, migrating from JBD2 to FASTCOMMIT requires no changes to the application code, or to Ext4's on-disk format. In fact, FASTCOMMIT can be enabled on a conventional Ext4+JBD2 setup even without reformatting. A testament to FASTCOMMIT's practical design and implementation is that majority of its code has already been merged in the Linux kernel and is part of mainline Ext4 [29].

We evaluate FASTCOMMIT over a variety of microbenchmarks and popular macrobenchmarks. Key design elements of FASTCOMMIT help reduce fsync latency by approximately 65% compared to JBD2. By decoupling the fsyncs with JBD2 commits, the byte, IO and cache flush overheads incurred by FASTCOMMIT reduce by up to 63%, 42% and 79% respectively. These overheads are much lower than JBD2, and in fact even lower than the purely logical journaling done by XFS. FASTCOMMIT's resource-efficient journaling minimizes interference in multi-tenant environments resulting in reducing total runtime by almost 20% while achieving increasing throughput of two simultaneously running workloads by 80% and 23% compared to JBD2.

The rest of the paper is organized as follows:

- In Section 2, we provide a background of journaling and cloud-based block storage devices.
- Section 3 contrasts FASTCOMMIT with other journaling optimizations and popular file systems.
- Section 4 motivates the design of FASTCOMMIT via detailed analyses of JBD2 commits and fsyncs.
- Section 5 describes the design and implementation of FASTCOMMIT– a hybrid logical+physical journaling approach to minimize journal interference and maximize user workload in various popular setups.
- Section 6 provides a detailed evaluation of FASTCOMMIT showing resource-efficiency, improved performance and reduced interference, cost using a variety of microbenchmarks and macrobenchmarks.

2 Background

In this section we give a brief overview of file system journaling and cloud storage offerings.

Logical vs Physical journaling. The two most common journaling approaches in file systems are logical journaling and physical journaling. A logical journal logs the file / directory manipulation operation. A physical journal maintains a copy of the changed metadata blocks such as inode bitmaps, block bitmaps, directory entries and so on, which are then re-written to their correct on-disk locations during a journal checkpointing operation or during crash recovery. Ma-

jority file systems use physical journaling since it is easy to use, is file system format independent, and easy to maintain. XFS [30] is a file system that performs logical journaling. As expected, logical journals are small in size, and journaling happens quickly, but crash recovery is often complex and slow since every operation that is logged needs to replayed. Physical journals on the other hand tend to be bulky, but also have a simple design and an efficient crash recovery protocol since the journal contains modified copies that need to be replayed.

The JBD2 journal. The Journaling Block Device v2 (JBD2) [32] is a robust, simple and popular physical journal used by multiple file systems in the Linux kernel, including the file system shipped with most Linux distributions – Ext4. JBD2 uses transactions to perform multi-block updates atomically. It is usually stored on the same block device as the data (can be stored on another device as well), and has a fixed on-disk size and format. At a later time based on multiple triggers, JBD2 performs checkpointing which synchronously writes the latest state of the metadata blocks stored in the journal to their actual on-disk locations. The write-pointer of JBD2 is then reset and JBD2 is ready to accept new commits.

NFS protocol and semantics. Network File System (NFS) is a distributed file sharing protocol. It allows sharing a local file system running on NFS server machine with clients on the network. NFS provides close-to-open cache consistency (CTO) by default (also called async mode). Typically, an NFS client opens a file, writes content to the file and then closes it. NFS guarantees that when the file is closed, it written to durable storage by issuing fsync on close. NFS also provides a stronger consistency model (sync mode) where it issues an fsync after each file system metadata update. This mode can have cause serious performance degradation, and thus is not recommended for performance sensitive applications [10]. Thus, we only use NFS async mode for all our performance evaluations.

Cloud storage block devices. Virtualized storage block devices (VBD) such as Google PD, Amazon EBS or Azure Disk Storage are three major storage offerings by large public cloud infrastructures[3]. These VBDs are designed to mimic physical storage devices (PBD) such as a hard-disk drive (HDD) or solid-state disk (SSD) albeit with different characteristics (see Section 4). Modern VBDs can charge separately for capacity, throughput (MB/s) and IOPS, thereby allowing users to remain flexible on all three dimensions.

3 Related work

Journaling has been a commonly used crash consistency mechanism in both file systems and databases for decades. Popularly used production journaling local file systems include Ext3 [5] (the predecessor of Ext4), XFS [3], Microsoft's NTFS [28] and IBM's JFS [4]. NTFS and JFS are proprietary. XFS does purely logical journaling, and therefore is one of the file systems with which we compare FASTCOMMIT's hybrid journaling approach. ReiserFS [22] and OCFS [9] are in-kernel file systems but are seldom used in production. OCFS also uses JBD2 for journaling. All other file systems use their own journaling mechanisms. There have been several research journaling file systems such as SplitFS [18], WineFS [17], BarrierFS [34], SpanFS [19], iJournaling [26] and CJFS [25]. SplitFS and WineFS use variants of JBD2 but they are designed for use on byte-addressable persistent memory. BarrierFS, SpanFS are both designed for faster storage devices, i.e. SSDs or low-latency storage media.

CJFS [25], iJournaling [26] and Fine-Grained Journaling [6] are the closest to FASTCOMMIT. FASTCOMMIT's design is loosely inspired from iJournaling, as it also maintains a hybrid journal with a focus on reducing fsync latency. However, iJournaling writes at least 3 blocks using 2 cache flushes for each commit resulting in high IO and byte overheads. More fundamentally, iJournaling changes Ext4's fsync behavior to commit only the blocks of the file issuing the fsync. Although this technically does not break the fsync contract, it breaks Hyrum's Law [16], since in practice users heavily rely on fsync committing *all* outstanding unsync'd files. These reasons make iJournaling impractical to use. CJFS focuses on scalability and trades off resource efficiency, thus resulting in high byte and IO overheads. It relies on an order-enabled IO stack [35], essentially requiring certain IOs to get written to disk before others, which is not implemented in Linux today, and therefore is also impractical to use. Fine-Grained Journaling [6] is byte-level journaling designed for byte-addressable storage media such as persistent memory. Moreover, Fine-Grained Journaling's logical journal for Ext4 requires a complete restructuring its journaling subsystem.

4 Motivation

Tiny and frequent commits expose JBD2's inefficiencies. A JBD2 commit (issued every 5 seconds, and on each fsync) has a high byte and IO overhead. Each JBD2 commit stores a minimum of 3 blocks (4KB) – a descriptor block (metadata about other blocks in the commit), at least one changed metadata block, and a commit marker block indicating end of the commit. Each JBD2 commit requires at least two write IOs – one to write the descriptor block alongwith changed metadata blocks to disk, and one to write the commit marker block. If there are data changes, the data needs to be made durable before the metadata for correctness. Using a simple microbenchmark that creates a file, appends 4KB data, and closes it, we measure the number of bytes written by JBD2, the IOs it performs and the cache flushes it requires. We vary the frequency of fsync to happen once every n operations, where n ranges from 1024 to 1 in powers of 2. Figure 2 and

[3]The other two are object storage such as Google GCS, Amazon S3 and file storage such as Google FileStore, Amazon EFS.

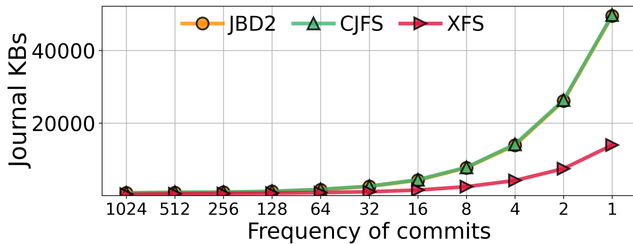

Figure 2: JBD2, CJFS have identical journal byte overheads for tiny and frequent fsyncs. XFS's logical journal has the lowest overhead.

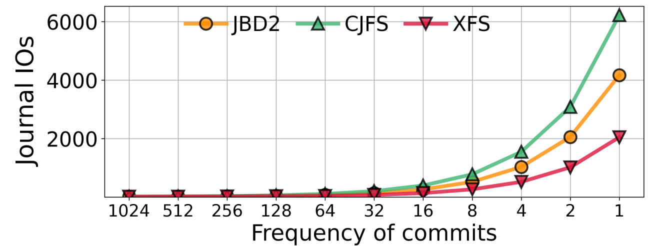

Figure 3: XFS also has the lowest journal IO overhead. CJFS has the highest IO overhead requiring 50% more IOs than JBD2.

Figure 3 shows the byte and IO overhead of JBD2 on 3 file systems – Ext4, CJFS and XFS[4]. The X axis ranges from one fsync performed every 1024 file operations (extreme left) to one fsync per file operation (extreme right). Since JBD2 performs one commit per fsync, its overhead increases steeply as the fsync frequency increases. CJFS [25]'s state-of-the-art optimization to JBD2 for scalability in fact has the same byte overhead as JBD2, but requires 50% more IOs than JBD2. XFS's byte and IO-efficient purely logical journal is unsurprisingly best in both byte and IO overheads. Hence, applications making tiny and frequent commits incur huge JBD2 overhead. Figure 4 shows the number of cache flushes needed by each file system. CJFS's compound flushing technique reduces the number of flushes compared to JBD2, but only when it is multi-threaded. While XFS is best in byte and IO overheads, it is the worst in cache flushes. Thus, there is no file system that is best at byte overhead, IO overhead and flush overhead.

NFS is a pathological case for JBD2. We conduct an experiment to calculate the JBD2 byte and IO overhead for the same microbenchmark, with the only difference being that the benchmark itself does not issue any fsyncs. NFS's async mode semantics converts each create+append+close to create+append+close+fsync. Thus, for JBD2, default NFS converts large infrequent commits to fsync-on-close. If NFS with sync mode is used, the above operations get converted to create+fsync, append+fsync, close+fsync, which is by definition its pathological case, and in fact has an overhead similar to fsync frequency of 1 in Figure 2 and Figure 3.

Pricing model in cloud does not favor JBD2. To ease cloud adoption, all major public clouds enable users to transfer their on-premises setups without any change, and indeed this is a very common case. However, there are fundamental differences between on-premises performance and cost expectations compared to the cloud. When expanding capacity in on-premises setups by purchasing PBDs, automatically expands the cluster's throughput and IOPS as well. Moreover, these resources are wasted if utilization is low. On the other hand, modern VBDs allow provisioning throughput, IOPS and capacity independently for higher flexibility. Thus, users wanting to minimize cost are incentivized to minimize journal byte and IO overheads.

[4]We could not compare with iJournaling since the code was not available.

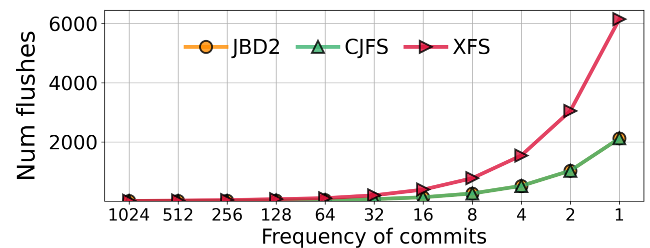

Figure 4: XFS requires the most number of cache flushes, with CJFS and JBD2 requiring the same. CJFS reduces flushes in multi-threaded workloads.

Poor resource-efficiency hurts performance and costs money. PBDs and VBDs are both designed to support a fixed bandwidth (MB/s) and IOPS. When a resource-hungry journal such as JBD2 consumes a lot of bandwidth and IOPS, it leaves less available for processing application data. Moreover, modern VBDs allow independent provisioning and purchasing of capacity, bandwidth and IOPS [11]. Thus, a resource-hungry journal can result in a high cloud provisioning cost. Finally, there is a direct impact of higher byte and IO overheads with reduced device lifetime in limited write-endurance storage devices like SSD [21].

Designing practically useful optimizations. We emphasize on two aspects of practicality: system maturity and backward compatibility of cloud migrants. Firstly, it is a well-known fact that storage systems take approximately a decade to mature. Ext4 and JBD2 are both mature, decades old software artifacts used by millions of users. While it is academically fulfilling to completely redesign a file system or its journal, it is often the case that optimizations that keep the user API unchanged and work within the framework of existing mature systems are the ones that see practical impact. Secondly, cloud migrants – users who migrate to cloud from on-premises systems – are used to certain systems. Ext4+JBD2 has a huge user base as it is the default file system shipped with majority of the Linux distributions. We thus designed FASTCOMMIT with the view of merging it with the upstream Linux kernel, and therefore choose to operate within the constraints and assumptions made by users of Ext4+JBD2.

5 FASTCOMMIT design and implementation

Overview. FASTCOMMIT is designed with the goals of minimizing journal byte and IO overheads to improve end user performance and reduce cost. FASTCOMMIT introduces a *hybrid* approach to file system journaling. In this approach JBD2 still commits every 5 seconds, but within those 5 seconds, FASTCOMMIT attempts to logically journal the file system updates and falls back to conventional JBD2 (henceforth called slow commits) when unable to perform logical journaling (mostly in case of complex and rare operations such as file system resize). Doing so, effectively *decouples* JBD2 commit from fsync.

Techniques. FASTCOMMIT employs three techniques to improve the efficiency of file system journaling. (a) *Compact logging* reduces journal byte overhead by packing multiple updates in a single FASTCOMMIT *log (FCLog)*; see Section 5.2. (b) *Selective flushing* reduces journal IO overhead by minimizing the number of flushes needed to perform commits (Section 5.3). (c) *Inline journaling* reuses the thread issuing fsync as an opportunistic journal thread instead of waking up the JBD2 thread avoiding an expensive context switch (Section 5.4). We start by explaining the design of the hybrid journal (Section 5.1) followed by the details of the various techniques, and finally discuss the crash recovery using FASTCOMMIT (Section 5.5). We discuss the salient features of our implementation that make FASTCOMMIT practical to use such as no changes to APIs (Section 5.6) and backward compatibility (Section 5.7).

5.1 Hybrid journaling

Hybrid journaling refers to the combination of logical+physical journaling performed by FASTCOMMIT.

Simultaneously supporting file-level and block-level journaling. A physical journal such as JBD2 is designed to provide journaling at the block level. On the other hand, a logical journal logs file and directory manipulations, which are at the inode level, and not the block level. Therefore, FASTCOMMIT's hybrid journal needs to support journaling and recovery at both levels without causing layering violations. The implementation details of simultaneously supporting file-level and block-level journaling are discussed in Section 5.6.

Logical journaling area. Conventionally, JBD2 reserves on-disk space for maintaining the physical journal. FASTCOMMIT marks a small fraction of that space (by default 1.5%) for its logical journal, which is called the *FC area*. As will be explained later, FASTCOMMIT's updates are very space-efficient, allowing even a 1.5% JBD2 space reservation to suffice for logging its updates. Therefore, no additional space is necessary for FASTCOMMIT, which means that the JBD2 on-disk area as seen by conventional Ext4 remains unchanged. Such design features enhance practicality as explained in Sections 5.6, 5.7.

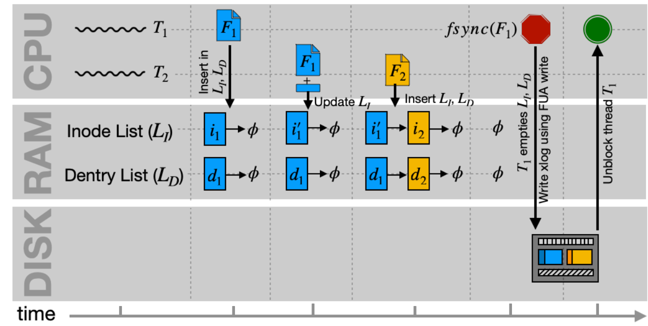

Figure 5: The different operations done in the CPU, RAM, disk during a FASTCOMMIT commit.

The FASTCOMMIT commit. Logical operations in FASTCOMMIT are only performed between successive slow commits. Recall that by default slow commits happen every 5 seconds. Let us walk through a FASTCOMMIT commit using an example shown in Figure 5. Between two slow commits, FASTCOMMIT maintains an in-memory lists of updates. L_D represents the directory entry updates list and L_I represents changed inodes list. Suppose thread T_1 creates a file F_1. Ext4 creates an inode for the newly created file, and inserts it in L_I along with a new directory entry update in L_D. At a later time, thread T_2 appends data to F_1, say one block. At this point, conventional JBD2 would have explicitly recorded copies of all the metadata blocks that were changed by this operation. Instead, FASTCOMMIT simply logs the logical offsets in the inode that were affected by this allocation, which is shown as updated i_1 to i_1' in L_I. T_2 then creates file F_2 which results in entries i_2 and d_2 in L_I and L_D respectively. At a later time T_1 issues fsync on F_1. FASTCOMMIT performs the commit operation in the context of the user thread T_1 by traversing L_D and L_I, packing all the updates in an FCLog and storing it in the FC area. In most commit operations, FCLog fits in a single disk block (4KB). Once the FCLog is written and acknowledged by the device, the commit operation is marked complete.

5.2 Reducing byte amplification via compact logging (FCLogs)

FASTCOMMIT's compact FCLogs significantly help in reducing byte overhead by tightly packing metadata updates in a single 4KB block. We first discuss FASTCOMMIT's commit format followed by an example of how FCLogs are created.

The FASTCOMMIT format. A FASTCOMMIT commit consists of a series of tiny updates (called FCTags) to multiple files. FCTags are designed to be as small as possible. Moreover, they are designed to be idempotent, which greatly simplifies recovery as discussed in Section 5.5. Figure 6 shows FCTags being part of a single FCLog. Although most FCLogs are 1 block in size, some FCLogs can occupy multiple blocks.

Each FCTag has three fields – (1) type: 2 bytes (2) length

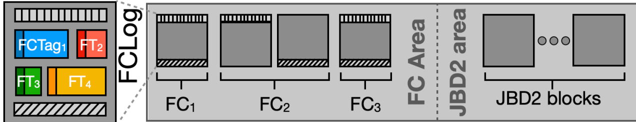

Figure 6: Several FCTags are part of a single FCLog. Multiple FCLogs are written to FASTCOMMIT area, which is next to the JBD2 area on disk.

(short int): 2 bytes and (3) value: variable length. An FCLog always starts with a head tag and ends with a tail tag, which both occupy 12 bytes. The head tag marks the start of a FASTCOMMIT commit, and contains the commit ID of the previous slow commit after which this FCLog should be replayed in case of recovery. The tail tag marks the end of an FCLog (similar to the commit block in a conventional JBD2 commit). It contains the checksum of the entire FCLog. Most of the usual file manipulations can be captured using a small set of only 8 FCTags:

1. HEAD: marking the start of an FCLog
2. ADD_RANGE: adding data to a file
3. DEL_RANGE: deleting data from a file
4. CREAT: creating a file
5. LINK: symlink or renaming a file
6. UNLINK: deleting a file
7. INODE: storing an inode
8. TAIL: end and checksum of an FCLog

We describe what gets committed in an FCTag for each of these tags. A set of 8 straightforward FCTags with a fall back to conventional JBD2 allows FASTCOMMIT to extract most of the benefits of purely logical journaling. In contrast XFS has a list of around 40 tags, many of which are very complicated [14]. This is why completely revamping Ext4 to adopt purely logical journaling is impractical. We describe the most common file operations and what FASTCOMMIT would write in their respective FCLogs below.

File creation / deletion. Creating a file creates an FCLog with two FCTags. A CREAT FCTag indicates that a new inode has been allocated, and added to a parent directory based on the file path. The INODE FCTag maintains a copy of the newly allocated inode.

Appending to a file. Similarly, let us consider an example of an append operation to a file named "foo" which adds 4KB at the end of the file. This operation would generate following FCTags:

1. HEAD FCTag (12 bytes).
2. ADD_RANGE FCTag (20 bytes) indicating that a new extent with logical block address 1, physical block address 1000, and size 1 block was added to the file.
3. INODE FCTag (136 bytes): the most recent copy of the file's inode.
4. TAIL FCTag (12 bytes).

Thus the entire FASTCOMMIT commit in this case consumes only 168 bytes. JBD2 requires 6 blocks making every append cost 24KB.

Deleting data from a file. Similarly, when certain blocks are removed from the file (such as using ftruncate or fallocate punch hole operations), FASTCOMMIT only stores the extents that were removed from the file using the DEL_RANGE tag. When blocks are removed, FASTCOMMIT does not need to store the physical block addresses of the blocks that were removed, since they can be inferred from the inode.

Renaming a file. The rename operation involves storing more than one FCTag, which is explained using an example. Suppose a file "/foo" is to be renamed to "/bar". Let us assume that the directory entry "/foo" was associated with inode i_{10} on disk. The rename operation would generate following FCTags:

1. HEAD FCTag (12 bytes).
2. LINK FCTag that records the association of "bar" with i_{10} (16 bytes).
3. UNLINK FCTag that records the disassociation of the directory entry "foo" from i_{10} (16 bytes).
4. INODE FCTag that records the most recent copy of inode i_{10} (136 bytes).
5. TAIL FCTag (12 bytes).

Thus, the entire FASTCOMMIT commit for rename is captured in 192 bytes. In JBD2, a rename operation requires storing 7 blocks of size 4KB each amounting to 28KB.

5.3 Better IO efficiency via selective flushing

Cache flush command forces the disk to completely write-out data written in volatile disk cache to non-volatile media. Flushing is widely used by file systems to ensure consistency of data. However, if a journaling subsystem is not careful in deciding when to flush, it might flush data to disk which could have safely resided in the disk cache for longer (for example, data written in parallel to inodes after a commit has started, data written to a different partition on the same disk). This deprives the data in the disk cache from being coalesced with future writes in order to be more efficient, and perform fewer disk IOs [36]. Thus, preventing unnecessary flushes not only improves IO efficiency of future IOs, but also results in making the currently issued commit faster.

JBD2 cannot avoid at least one flush per commit. In ordered mode[5], first, the data is written to the disk. Next, all the updated metadata blocks along with JBD2 descriptor block are written to the journal area. JBD2 then issues a flush to ensure that data blocks and journaled metadata blocks are made durable on the disk. Finally, JBD2 writes a commit block indicating the end of a commit. The commit block is written using Force Unit Access (FUA) [15] and thus is written all the way to the nonvolatile storage skipping past disk's write

[5]The default journaling mode for Ext4 is ordered mode where data needs to be made durable before metadata.

cache. Note that FUA can only be issued for 1 block writes, and does not constitute a full cache flush. In this way, JBD2 can avoid issuing a second flush after the JBD2 commit block. However, the first flush is simply unavoidable irrespective of FUA. In a heavily contended system, this unavoidable flush can result in flushing data from the volatile disk cache that is completely unrelated to the current commit operation. However, flushing unrelated data prematurely does not affect the safety or ordering guarantees of the flushed data in any way.

FASTCOMMIT reduces flushes required on fsync. We have seen that a flush command is needed during commits for two reasons: 1) persist user data on disk 2) persist blocks written as part of journal commit. In case of frequent commit operations (such as in case NFS), many fsyncs are called before writing user data. Also, in case of FASTCOMMIT majority of the FCLogs fit within a single 4KB block, and thus can be completed without a cache flush. Thus, FASTCOMMIT avoids sending an expensive cache flush operation when there are no data writes and FCLog is contained within 4KB.

5.4 Reducing context switches using inline journaling

As shown in Figure 7a, there are two context switches that happen during a JBD2 commit. The first is when the file system fsync function wakes up the dedicated JBD2 journal thread in order to perform the commit. The journal thread then sleeps intermittently waiting for disk IO between the successive writes of data, metadata and the commit block (work in JBD2 shown as W). Finally, the journal thread wakes up the user thread that is sleeping for the JBD2 commit to complete. JBD2's journal thread is initialized with a high IO scheduling priority, which instructs the Linux block layer to quickly finish the IO issued by JBD2. This priority also allows for a smaller delay in scheduling the JBD2 thread for the commit (shown as S_j). In contrast, when the JBD2 thread wakes up the user thread, the delay is much longer (S_f) since the user thread has a lower priority.

Using inline journaling technique, in FASTCOMMIT the thread issuing fsync performs journal commits by itself without involving the JBD2 journal thread. FASTCOMMIT temporarily lifts the user thread's scheduling and IO priority to match that of the JBD2 journal thread. This helps us ensure that while the commit operation is ongoing, kernel schedulers continue to treat fsync operations and IO with higher priority. As soon as the disk write for the FASTCOMMIT commit is completed, but before acknowledging the completion of the commit operation to the user, FASTCOMMIT reverts the priority of the user thread to its original value. This ensures that the temporary lift in priority is invisible to the file system and applications, and is also done safely within the protection rings of the kernel. Figure 7b shows no context switches in the case of FASTCOMMIT since the heavy JBD2 thread is not involved in FASTCOMMIT commit path unless there is a fallback to slow commits in the case of rare and complex operations.

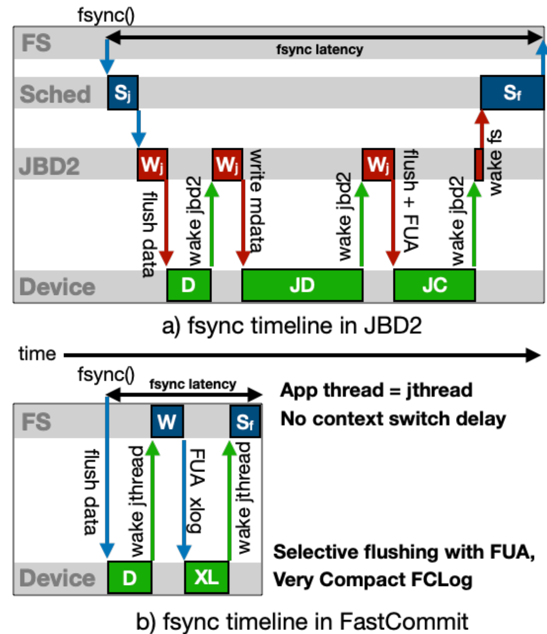

Figure 7: (a) shows fsync latency in conventional JBD2, (b) shows fsync latency in FASTCOMMIT. FASTCOMMIT's compact FCLog, selective flushing using FUA and inline journaling results in 2× faster fsyncs.

5.5 Crash Recovery

With FASTCOMMIT, the JBD2 journal area now consists of both JBD2 transactions and FCLogs. Since the FC area only holds the FCLogs since the last slow commit, during recovery, all the FASTCOMMIT commits should only be replayed *after* the recovery of the last slow commit from the JBD2 area. Within each FCLog there are several FCTags which are successively replayed during recovery.

Instead of defining new recovery logic, FASTCOMMIT reuses Linux's virtual file system layer (VFS). During recovery, FASTCOMMIT first flags the file system to be in a recovery state. It then traverses the FASTCOMMIT area, and for each FCTag, invokes one of the VFS APIs [1] to replay the logged operation.

Replay Idempotency. The FASTCOMMIT replay procedure is designed to be idempotent. FCTags achieve idempotency by storing results of file system operation rather than the operations themselves. During recovery, if the stored result is already applied, it is ignored. For example, consider an operation of file deletion achieved via "rm /dir/foo" where file foo is a hard link to inode 10. Assume that inode 10 has hard links from 2 other directory entries (thereby making inode 10's reference count 3). Instead of storing the delete operation in FCTags (delete file "foo" from "/dir"), FCTags store the series of outcomes: (1) "/dir" does not have file "foo", (2)

inode 10's reference count is 2. This guarantees that the same FCTag can be applied in an idempotent fashion to the file system. This helps to ensure that if the file system crashes during recovery itself, the FASTCOMMIT replay procedure can restart from the beginning, and the file system correctness will remain unaffected. All FCTags whose HEAD tag does not contain the commit ID of the last JBD2 slow commit that was successfully replayed, are discarded.

5.6 Hybrid journaling without API changes

Since our goal was to get FASTCOMMIT merged into the upstream Linux kernel, we pay close attention to its practicality. In FASTCOMMIT, the notion of atomic updates changes from atomically modify one or more *blocks* to atomically modify one or more *inodes*. Being a *physical* journaling system, JBD2 provides *block* based journaling APIs. However, since most of the file system functions that need to perform atomic updates to the file system are implementations of virtual file system (VFS) interface, Ext4 already masks JBD2 block mutating journal APIs behind a shim layer of inode mutating routines. FASTCOMMIT leverages these inode mutating routines to inform the construction of the FCTags. Thus, FASTCOMMIT introduces no changes to the journaling APIs of Ext4.

5.7 Ensuring backward compatibility

FASTCOMMIT is designed to support backward compatibility with disks formatted with conventional Ext4 in the following ways: (1) An old Ext4 partition should be FASTCOMMIT-compatible when mounted using a new FASTCOMMIT-enabled kernel, (2) A new FASTCOMMIT-compatible Ext4 partition should be able to function correctly when mounted using an old kernel without FASTCOMMIT provided the journal area is empty, and (3) if in the previous case, the journal is not empty (i.e. contains FASTCOMMIT commits), then the old kernel should reject mounting of FASTCOMMIT-enabled Ext4 partition. Since FASTCOMMIT will be rolled out to all Linux users in the world, these aforementioned backward compatibility criteria ensure that migration of existing Ext4 users onto FASTCOMMIT happens without any problems.

FASTCOMMIT introduces two file system feature flags to handle this: 1) FASTCOMMIT enabled (Backward compatible) 2) FASTCOMMIT present (Backward incompatible). FASTCOMMIT enabled flag is set, as the name suggests, when FASTCOMMIT is enabled on the file system. This backward-compatible flag implies that old kernels can mount the file system even if they don't have FASTCOMMIT code. The FASTCOMMIT present flag is set on first FASTCOMMIT commit after a JBD2 slow commit. This backward-incompatible flag implies that old kernels cannot mount the file system if they don't have FASTCOMMIT code. After a JBD2 slow commit, this flag is cleared as the FC area is now empty implying that the old kernel can mount new Ext4 without any problems.

5.8 Testing for Upstream Readiness

We worked closely with upstream kernel developers from early design days of FASTCOMMIT to ensure that its design is practical. We ensured upstream readiness of FASTCOMMIT code by closely following the upstream submissions process [8]. We then ran tests in XFSTests [7] FS testsuite on FASTCOMMIT. XFSTests is widely used by upstream file system maintainers to ensure quality of the upstream submissions. FASTCOMMIT passes all the tests in auto and log groups. Tests in auto group ensure overall file system correctness, while tests in log group ensure crash consistency. Moreover, as a part of regular upstream maintenance process, FASTCOMMIT undergoes continuous testing on a daily basis using XFSTestsbld [31]. This ensures that new kernel patches that get merged into the Linux kernel continue to work with FASTCOMMIT.

6 Evaluation

Experimental setup. We use virtual machines (VM) setup on the Google Compute Engine (GCE) for all our evaluations. We run our experiments on a n2-standard-32 [13] VM which has 32 vCPUs, 128GB RAM and 32Gbps of egress network bandwidth. Debian 11 distribution with Linux Kernel Version 5.19 is the OS used.

File systems. We compare Ext4+FASTCOMMIT (called FC) with 4 other file systems. The obvious comparison is with Ext4+JBD2 (or just JBD2). Next, we choose Ext4 with asynchronous journal commits (called Async) – a JBD2 optimization that performs commits asynchronously. CJFS [25] is the state-of-the-art JBD2 optimization which is aimed at improving its scalability. We could not compare with iJournaling [26] since its code is not open-sourced. We also compare with XFS [3] as it is other extreme in fine-grained journaling since it uses a purely logical journal.

Scenarios. There are three main scenarios in our evaluation: a) SSD – where the local SSD (375GiB) in the VM is formatted with the test file system (100GB partition), b) NFS+SSD – where there is a client VM that access the local SSD formatted with the test file system via NFS protocol, c) NFS+VBD– this setup is similar to the previous NFS+SSD except that instead of SSD, VBD is used. We use Google Cloud Hyperdisk [11] as our VBD. Hyperdisk allows independent provisioning of IOPS, bandwidth (MB/s) and capacity. Our VBD is a virtualized HDD for which we provision 150MB/s bandwidth, 15000 IOPS and 2TB capacity.

Microbenchmarks. In order to measure the latency breakdown of fsync, we run a controlled experiment of a single create file, followed by append, and then fsync. To measure the impact of frequent fsyncs, this same create+append+fsync workload is executed 1024 times with varying frequency of fsyncs, as described in Section 4.

Marcobenchmarks. We use four popularly used mac-

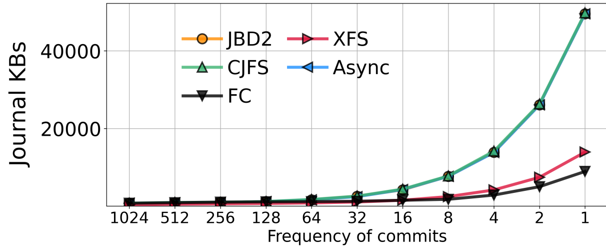

Figure 8: FASTCOMMIT has the lowest byte overhead wrt other FS.

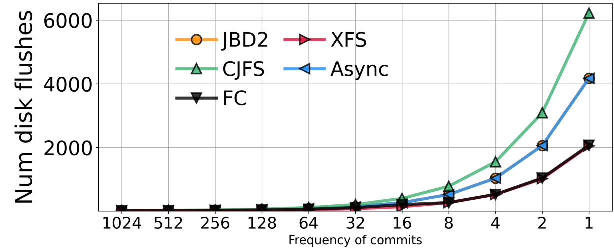

Figure 9: FASTCOMMIT has the lowest IO overhead wrt other FS.

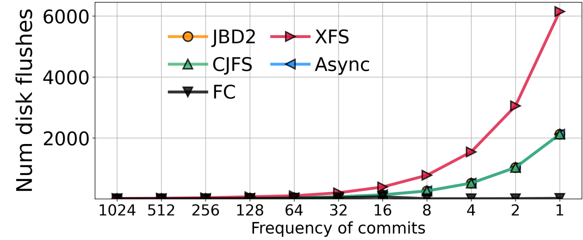

Figure 10: FASTCOMMIT has the least flushes versus other FS.

robenchmark workloads. Varmail and Fileserver belong to the Filebench testsuite [23], and as per their name emulate mail-server and file-server workloads respectively. Postmark [20] is a benchmark from NetApp that simulates NFS workloads. Finally, FSMark [33] is a metadata-heavy benchmark where the write pattern performs lots of synchronous IO operations across multiple directories.

We answer the following questions in our evaluation:

- How much do the design elements of FASTCOMMIT help in speeding up fsyncs? (Section 6.1)
- How is FASTCOMMIT's byte and IO overhead compared to other file systems? (Section 6.2)
- What are the performance improvements achieved by FASTCOMMIT? (Section 6.3)
- How much does goodput of workload improve due to FASTCOMMIT's optimizations? (Section 6.3.2)
- Is FASTCOMMIT scalable? (Section 6.3.1)
- Can cloud provisioning cost be reduced using FASTCOMMIT? (Section 6.4)

6.1 Evaluating fsync performance, overheads

FASTCOMMIT's design elements make fsync up to 2.8× faster. We begin the evaluation by measuring the effect of various design elements of FASTCOMMIT in reducing the fsync latency using the two microbenchmarks mentioned above. Table 1 shows the comparison between JBD2 and FASTCOMMIT in the different phases of fsync in a controlled one-file microbenchmark described above. Using selective flushing technique (section 5.3), FASTCOMMIT avoids unnecessary cache flushes and uses FUA writes for single block commits. Thus, FASTCOMMIT only spends 144μs doing commit related disk IO. JBD2, on the other hand, needs to first write commit descriptor and changed metadata blocks to the journal (182μs) and then issue a flush along with a FUA write of the commit marker block (99μs). FASTCOMMIT is thus $\approx 2\times$ faster in completing commits than JBD2. The context switch + misc delays capture the effect of FASTCOMMIT avoiding context switch via inline journaling. This reduces FASTCOMMIT's misc delays to 10% of JBD2. Thus, the overall fsync latency of FASTCOMMIT in this controlled experiment is 2.8× lower than JBD2.

FASTCOMMIT decouples fsync from JBD2 commit resulting in lower byte and IO overheads. Figure 8 shows the KBs written to journal with increasing fsync frequency. As explained in Section 4, JBD2 and CJFS perform one JBD2 commit per fsync. In contrast, FASTCOMMIT's efficient FCLogs effectively decouple JBD2's expensive commits from the fsync frequency. Thus, the journal byte overhead in FASTCOMMIT is lowest compared to other file systems, including XFS. This is surprising since XFS is a purely logical journal, and so is expected to incur the least byte overhead. There are multiple reasons for this result. First, FASTCOMMIT very efficiently packs updates in FCLogs, which delays JBD2 checkpointing that performs expensive random on-disk updates. Second, certain metadata operations are logged in FASTCOMMIT whereas they are directly written to disk in case of XFS. For example, creating directory entries, a very common operation, requires XFS to perform non-journal disk IOs, whereas FASTCOMMIT simply writes the directory entries to FCLogs. Figure 9 shows the number of journal IOs incurred. CJFS issues 3 IOs per fsync, which is in fact even more expensive than conventional JBD2. On the other hand, XFS and FASTCOMMIT issue almost identical number of journal IOs, which is 1 per fsync. Finally, Figure 10 shows the number of cache flushes issued by all file systems. This is

Category	JBD2 (μs)	FC (μs)
Descriptor + Metadata	182	144 (FCLog + FUA)
Commit Marker	99 (flush + FUA)	-
Cxt switch + Misc delays	177	18
Total	458	162

Table 1: Breakdown of fsync latency in JBD2 vs FASTCOMMIT. JBD2 fsync latency is 2.8× higher than FASTCOMMIT.

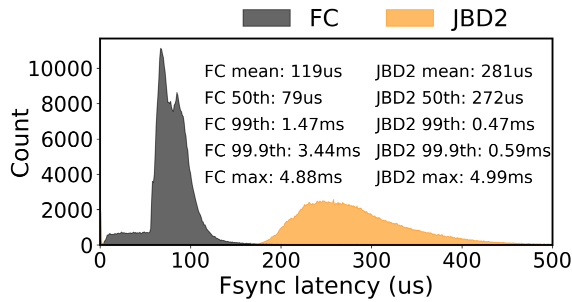

Figure 12: FASTCOMMIT fsyncs have much lower latency compared to JBD2. p99 FASTCOMMIT latency > p99 JBD2 latency.

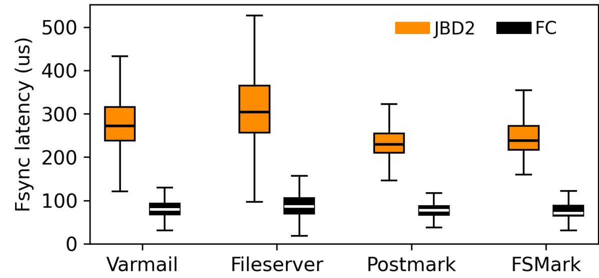

Figure 13: Median latency of fsync in FASTCOMMIT is approximately 3× as fast as JBD2 across multiple benchmarks.

where FASTCOMMIT outperforms other file systems significantly. By using selective flushing and FUA, FASTCOMMIT performs only a handful of flushes even when thousands of fsyncs are being done. This improves cache retention of data and minimizes interference of fsyncs in a multi-tenant, multi-threaded setting.

FASTCOMMIT makes fsync latency more predictable. Figure 12 shows the latency distribution of fsync in FASTCOMMIT versus JBD2 when running 1 million file operations in Varmail benchmark when running over NFS. The fsync latency distribution is closer to the origin compared to that of JBD2. The plateau of tiny fsync latencies in FASTCOMMIT from 0 through 60μs can be attributed to threads intending to perform fsync, but actually not doing any real work. Recall from Section 5.4 that an ongoing FASTCOMMIT commit will opportunistically perform outstanding commits as well. In particular, the mean fsync latency of JBD2 in this experiment is 2.3× higher than FC, whereas the 99th percentile latency is 3× lower. FASTCOMMIT fsync latencies can have a long tail because they occasionally need to fallback to performing slow JBD2 commits (approximately once every 5 seconds). When performing these slow commits, they can end up being larger than the average JBD2 commit. Overall, the latency distribution for FASTCOMMIT is much tighter in comparison to the large spread observed in JBD2, indicating that average fsync latency in FASTCOMMIT is much more predictable (except the tail). Lower FASTCOMMIT tail latency can be traded for increased mean latency in two ways: a) Reducing FASTCOMMIT journal area: A smaller FC area will cause more frequent fallbacks to slow commits, thus making each slow commit less expensive. b) Forcing frequent JBD2 commits: JBD2 can be configured at mount time to make more frequent, and therefore smaller and less expensive commits. These optimizations can help curtail the expensive tail latency of FASTCOMMIT.

Figure 13 captures the fsync latency spread (sans the long tail) in the 4 multi-threaded macrobenchmarks shown above. In all cases the FASTCOMMIT's median fsync latency is > 2.5× lower than JBD2 with a tighter distribution.

6.2 FASTCOMMIT improves resource efficiency

We now evaluate the resource efficiency of various file systems by calculating the overall bytes written and IOs performed for the four multi-threaded macrobenchmarks mentioned above in multiple scenarios: local SSD, NFS+SSD and NFS+VBD.

FASTCOMMIT has lowest byte overhead. Figure 11 shows the byte overhead compared to JBD2. In local SSD (Figure 11a) we only look at Varmail and FSMark since those are the only two workloads explicitly performing fsync, whereas in Figure 11b, Figure 11c we evaluate all four benchmarks since NFS issues fsyncs even when applications explicitly do not, as explained in Section 4. FASTCOMMIT is the most byte-efficient file system in all benchmarks and all sce-

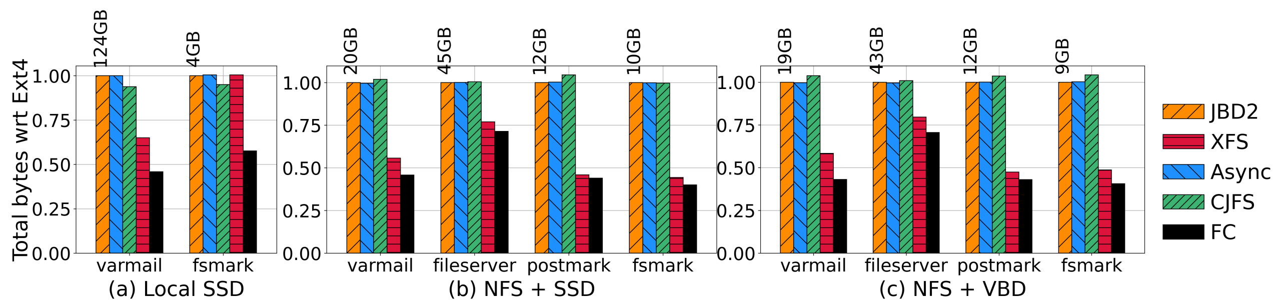

Figure 11: FASTCOMMIT achieves least byte overhead compared to all other file systems.

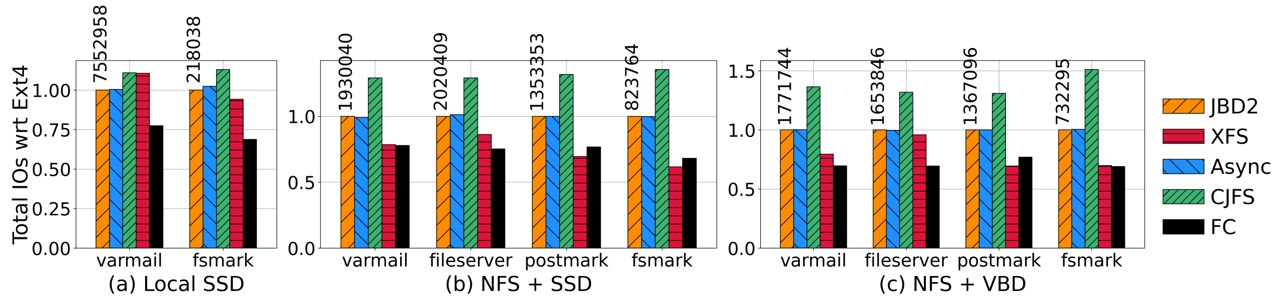

Figure 14: FASTCOMMIT and XFS have the lowest IO overhead compared to all other file systems.

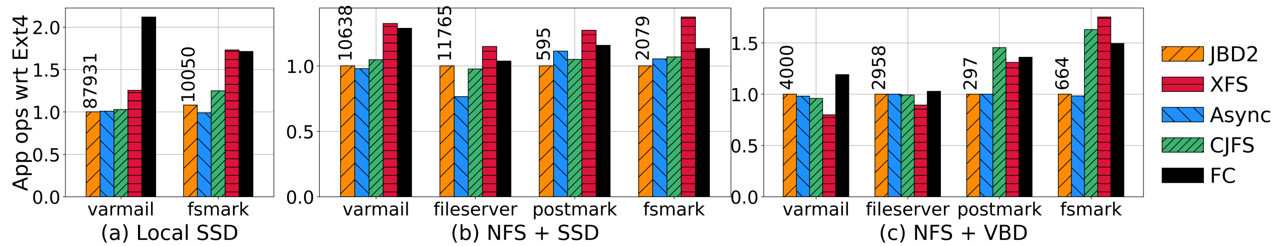

Figure 15: FASTCOMMIT has low IO amplification compared to other file systems due to fewer coalesced journal writes, due to faster fsyncs.

narios, and writes almost 2× lesser data than JBD2 in every workload except Fileserver (because Fileserver has larger and fewer files, so fewer metadata operations). Compact FCLogs significantly reduce the byte footprint of FASTCOMMIT. A lower byte overhead also has advantages other than performance, such as a longer device life time in case of SSD, reduced bandwidth provisioning costs for VBDs (see Section 6.4), and reduced bandwidth interference because of the journal implies an increase in bandwidth available for useful work (see section 6.3.2).

FASTCOMMIT and XFS have the lowest IO overhead. Figure 14 shows the total number of IOs performed when running the same macrobenchmarks. CJFS's multi-version shadow paging design aims to reduce wait-time for an fsync, but consequently performs more IOs. While XFS performs more IO locally, over NFS, there are workloads where XFS has a marginally lower IO overhead than FASTCOMMIT. In NFS+SSD and NFS+VBD, very frequent and tiny commits result in 100% of the FCLogs being 1 block in size. In local SSD, because fsyncs are few and far between, there can be multi-block FCLogs. Nevertheless, even locally, 81% of total commits were single-block commits that allowed FASTCOMMIT to perform a single IO to write both metadata and commit marker to disk. Moreover, 98.5% fsync calls result in a FASTCOMMIT commit whereas only 1.5% (includes JBD2 commits that happen every 5 seconds) fsync calls result in slow commits.

We observe that Async performs very similar to JBD2. This is because akin to JBD2, Async requires two disk IO commands. It writes the descriptor, changed metadata blocks and the commit block as part of first write request. It then issues a cache flush to ensure all of the blocks written as part of the previous write request are durably stored on the disk. Nevertheless, for completeness we continue to show Async in our evaluation since it is the closest representation of FASTCOMMIT, wherein the entire commit is performed using only 1 write operation.

6.3 FASTCOMMIT achieves high performance

FASTCOMMIT improves end-to-end application throughput in most cases. Figure 15 shows the throughput using the application reported ops/s, files/s or transactions/s (all abbreviated to app ops) on the Y axis. The throughput is reported relative to JBD2. In Figure 15a, we observe FASTCOMMIT's throughput is highest compared to all other file systems, and between 1.75–2× that of JBD2. Since the benchmarks are IO-intensive, they are bottlenecked on fsync. With FASTCOMMIT fsync latency being the lowest, it provides the best throughput. In Figure 15b and Figure 15c, every NFS client operation now contains a fixed network round-trip time in addition to the remote fsync. Since NFS has converted fsyncs from being large and rare into tiny and frequent, the latency observed by JBD2 over NFS+SSD is not very different from that observed by FASTCOMMIT. Thus, the improvements are modest in the case of NFS+SSD. For NFS+VBD, the VBD is much slower than the SSD. Therefore, FASTCOMMIT shines, since it reduces the number of disk flushes, and essentially, makes fewer costly disk accesses.

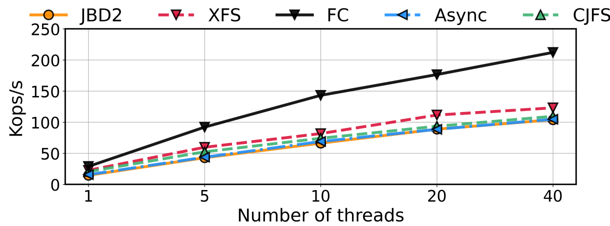

Figure 17: FASTCOMMIT scales between 1–40 threads. At 40 FASTCOMMIT provides 2× Varmail throughput versus other FS.

6.3.1 Scalability

Next, we look at scalability of FASTCOMMIT. Figure 17 shows the reported ops/s for the Varmail workload running on local SSD for 60 seconds with increasing number of threads in each run (shown on the X axis) – 1, 5, 10, 20, 40. While all file systems scale to some degree, the scaling of FASTCOMMIT is more pronounced. As the number of threads increase so does the gap between FASTCOMMIT and other file systems, and it is over 2× at 40 threads.

6.3.2 FASTCOMMIT minimizes journal interference

Figure 18 captures the interference of a heavy journal on slowing down the performance of non-journal-dependent workloads. In this experiment, we run the FIO workload for a fixed number of operations on the NFS+SSD setup. The FIO benchmark fallocates one large file and then performs random IO within that file. Therefore, FIO does not cause any journal activity during its run except for the first time it fallocates the large file. In parallel, we execute the FSMark workload which is metadata intensive and results in a lot of journal traffic. Figure 18 shows the breakdown of the device's write bandwidth (capped at 400 MB/s) among the journal and the application workload. Figure 18a shows JBD2 consuming almost 50% of the device bandwidth leaving only half for the application data. JBD2 requires about 115s for this experiment. CJFS shown in Figure 18b is better, but still its journal consumes close to 150 MB/s on average. CJFS is only about 2% faster in total runtime. XFS's logical journal significantly reduces journal bandwidth consumed to about 100 MB/s, and is 10.5% faster than JBD2. FASTCOMMIT only consumes about 60 MB/s bandwidth and finishes almost

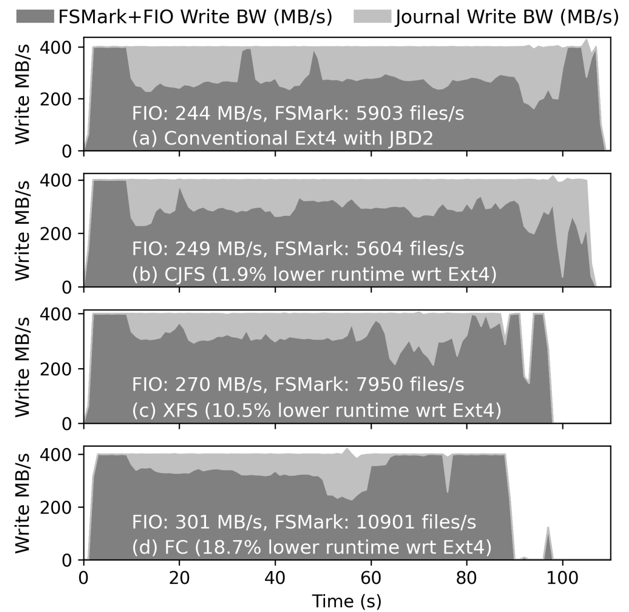

Figure 18: Impact of journal on application throughput in a multi-tenant environment. FASTCOMMIT interferes the least allowing higher application throughputs and lower runtimes.

19% faster than JBD2. More importantly, both FSMark and FIO report significantly higher throughput in FASTCOMMIT showing that reduced journal interference is critical to improved application performance, especially in a multi-tenant environment. We show only two workloads for brevity, but we have evaluated the interference in multiple cases and our results remain consistent in all cases.

6.4 FASTCOMMIT reduces cloud provisioning cost

Users often purchase cloud resources in order to meet a certain application performance requirement. For instance, a user might want their application to service 100K requests/s. In a world with independent performance provisioning, the user can estimate the provisioning value by knowing how many block storage IOs are performed by the file system per second (IOPS), along with the number of bytes transferred by the file system to/from the block storage server per second

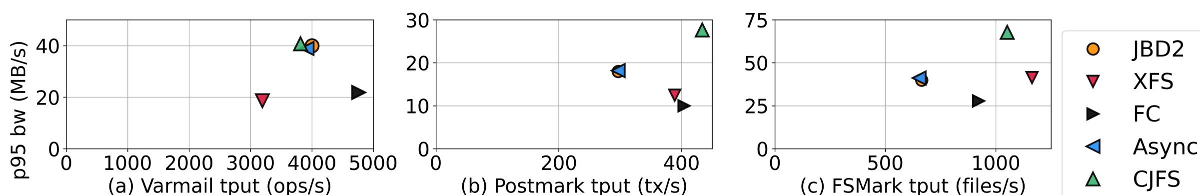

Figure 16: Scatter plot of avg. application MB/s vs p95 disk bandwidth. Except Varmail (that too marginally) FASTCOMMIT requires the least peak bandwidth provisioning. In FSMark, CJFS and XFS report higher files/s but require higher p95 bandwidth provisioning.

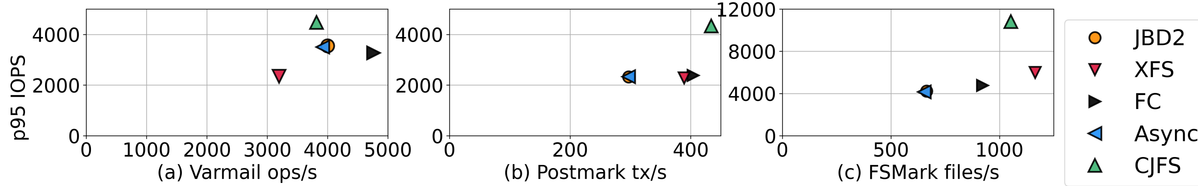

Figure 19: Scatter plot of avg. application ops/s vs p95 disk IOPS. In Varmail XFS requires least p95 IOPS, but FASTCOMMIT achieves much higher ops/s, whereas in FSMark, JBD2 and Async have marginally lower p95 IOPS but report much lower files/s.

(throughput). This experiment captures IOPS and bandwidth provisioning requirements of different workloads.

A common resource reservation approach is to provision the storage device bandwidth close to the peak bandwidth of the workload. Figure 16 is a scatter plot that captures the p95 device bandwidth needed (Y axis) to achieve the necessary user application performance (X axis). The best tradeoffs are achieved in the lower right corner, which has maximum user throughput for minimum provisioned p95 bandwidth. We show 3 workloads for brevity, but the trends hold across all workloads. FASTCOMMIT and XFS consistently achieve the highest application throughput for the lowest p95 device bandwidth. Similarly Figure 19 is a scatter plot that shows the p95 IOPS needed (Y axis) for the application performance (X axis). Owing to selective flushing, FASTCOMMIT reduces IO operations that are issued to disk by the file system. In most cases, FASTCOMMIT achieves higher application throughput while using lower IOPS. CJFS achieves higher application throughput in case of postmark and FSMark but at the cost of $2\times$ higher p95 IOPS provisioning cost. By applying publicly available pricing model of VBDs, FASTCOMMIT incurs the lowest dollar amount, in particular 22–57% lower per-month cost compared to JBD2.

7 Conclusion

This paper presents FASTCOMMIT, a hybrid logical+physical journaling for reducing byte and IO overheads, maximizing performance and minimizing provisioning cost in cloud-based block storage offerings. FASTCOMMIT is designed to reduce byte amplification via space-efficient logical journaling between successive bulky physical journaling commits, making fewer device IOs, reducing the number of cache flushes per commit, and reducing context switching and scheduling delays by reusing threads for performing commits. FASTCOMMIT reduces fsync latency by approximately $2.8\times$ and improves application throughput by up to 120%. FASTCOMMIT reduces journaling interference by $2\times$ and improves runtime of multiple simultaneously running applications by upto 84% while reducing the combined runtime by almost 20%. FASTCOMMIT's design is heavily vetted by experts in the open-source community, and majority of it has been merged to the mainline Linux kernel.

8 Acknowledgments

We thank the anonymous reviewers for their valuable feedback and suggestions. We extend special thanks to Mustafa Uysal, Larry Greenfield, Jan Kara, Ritesh Harjani, and numerous other reviewers of the Linux kernel community for their technical help and critical feedback.

References

[1] Vfs inode operations using struct inode_operations. https://www.kernel.org/doc/Documentation/filesystems/vfs.txt, 2005.

[2] Amazon. Elastic Block Store. https://aws.amazon.com/ebs/, 2023.

[3] Thomas E Anderson, Michael D Dahlin, Jeanna M Neefe, David A Patterson, Drew S Roselli, and Randolph Y Wang. Serverless network file systems. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 109–126, 1995.

[4] Steve Best, David Gordon, and Ibrahim Haddad. Kernel korner: Ibm's journaled filesystem. *Linux Journal*, 2003(105):9, 2003.

[5] Mingming Cao, Theodore Y Tso, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the ext3 filesystem. In *Proceedings of the Ottawa Linux Symposium (OLS)*. Citeseer, 2005.

[6] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. Fine-grained metadata journaling on nvm. In *IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, 2016.

[7] Jonathan Corbet. Toward better testing, 2014.

[8] Upstream Linux Kernel Developers. Submitting patches: the essential guide to getting your code into the kernel. https://docs.kernel.org/process/submitting-patches.html, 2023.

[9] Mark Fasheh. Ocfs2: The oracle clustered file system, version 2. In *Proceedings of the 2006 Linux Symposium*, volume 1, pages 289–302. Citeseer, 2006.

[10] Google Filestore. Mounting Fileshares. https://cloud.google.com/filestore/docs/mounting-fileshares, 2023.

[11] Google. Google Cloud Hyperdisk. https://cloud.google.com/compute/docs/disks/hyperdisks.

[12] Google. Persistent Disk. https://cloud.google.com/persistent-disk, 2023.

[13] Google. N2 VM Family. https://cloud.google.com/compute/docs/general-purpose-machines, 2024.

[14] Silicon Graphics. XFS Algorithms Data Structures. https://ftp.ntu.edu.tw/linux/utils/fs/xfs/docs/xfs_filesystem_structure.pdf, 2006.

[15] Christoph Hellwig. Forced Unit Access. https://patchwork.kernel.org/project/dm-devel/patch/20100826095413.GA9750@lst.de/, 2010.

[16] Hyrum. Hyrum's Law. https://www.hyrumslaw.com.

[17] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R Ganger, Aasheesh Kolli, and Vijay Chidambaram. Winefs: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 804–818, 2021.

[18] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, 2019.

[19] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A scalable file system on fast storage devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 249–261, 2015.

[20] Jeffrey Katcher. Postmark: A new file system benchmark. *TR3022*, 1997.

[21] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 257–270, 2013.

[22] Chris Mason. Journaling with reisersfs. *Linux Journal*, 2001, 2001.

[23] Richard McDougall and Jim Mauro. Filebench, 2005.

[24] Microsoft. Azure Disk Storage. https://azure.microsoft.com/products/storage/disks, 2023.

[25] Joontaek Oh, Seung Won Yoo, Hojin Nam, Changwoo Min, and Youjip Won. {CJFS}: Concurrent journaling for better scalability. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 167–182, 2023.

[26] Daejun Park and Dongkun Shin. {iJournaling}:{Fine-Grained} journaling for improving the latency of fsync system call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 787–798, 2017.

[27] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. Nfs version 3: Design and implementation. In *USENIX Summer*, pages 137–152. Boston, MA, 1994.

[28] Richard Russon and Yuval Fledel. Ntfs documentation. *Recuperado el*, 1, 2004.

[29] Harshad Shirwadkar. FastCommit Linux Kernel Code. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/ext4/fast_commit.c, 2022.

[30] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[31] Theodore Tso. xfstests-bld. https://github.com/tytso/xfstests-bld/tree/master, 2023.

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

[33] Ric Wheeler. Benchmark synchronous/async file creation. https://github.com/josefbacik/fs_mark, 2003.

[34] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-Enabled IO Stack for Flash Storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 211–226, 2018.

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

[36] Jeseong Yeon, Minseong Jeong, Sungjin Lee, and Eunji Lee. {RFLUSH}: Rethink the flush. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 201–210, 2018.

A Artifact Appendix

Abstract

This artifact includes tools and infrastructure that can be used to validate FASTCOMMIT results described in the paper. Most of the FASTCOMMIT code is already upstreamed. This artifact describes how to obtain unmerged fast commit code, setup Google Cloud backed Virtual Machines and re-run the evaluation discussed in the paper.

Scope

Using the benchmarks and setup instructions in this artifact, users are able to verify following claims (details can be found in Section 6):

- FASTCOMMIT reduce fsync latency.
- FASTCOMMIT reduce byte and IO overhead of journaling.
- FASTCOMMIT minimizes journal interference.
- FASTCOMMIT lower cloud provisioning costs.

Contents

This artifact consists of following items:

- **Unmerged FASTCOMMIT patches**. Most of the FASTCOMMIT code has already been merged upstream. This artifact provides the unmerged patches that can be applied on top of the kernel source code.
- **Benchmarking Scripts**. This artifact provides benchmarking scripts that can be used to run all the experiments discussed in the paper.
- **Documentation**. This artifact provides documentation about how to setup Google Cloud VMs to run the above mentioned benchmarks.

Hosting

Most of the fast commit code is already upstreamed. Here are 2 main files that we contributed to upstream Linux kernel:

- https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/ext4/fast_commit.c
- https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/ext4/fast_commit.h

The code that is not merged upstream can be found in the following GitHub repository: https://github.com/harshadjs/fc-perf-v2. The benchmarking scripts can be found in the following Github repository: https://github.com/harshadjs/fast-commit-atc-2024. The README file in that repository describes the steps to setup a Google cloud VM to run the benchmarking. These steps can also be used to setup any other Linux based machine for benchmarking.

Requirements

Although our evaluation in this paper was based on Google Cloud based virtual machines, fast commit feature and the benchmarking environment is capable of running on any Linux operating system. If users need to exactly reproduce the results as described in the paper, then they should use a machine with 32 cores and 128GB of RAM.