Reclaiming good transactions from a corrupt journal

Shehbaz Jaffer
University of Toronto

1 Introduction

Journaling file systems depend on their journal to ensure a consistent on-disk image on the event of a system crash. An application calls `fsync()` to ensure writes are durably stored on the journal. On the event of a failure, the journal is sequentially replayed to restore lost meta data. We observe that a corruption of an intermediate transaction in the ext4 file system journal silently aborts journal replay causing data loss of subsequent transactions. For a meta data intensive workload that recursively creates 4000 directories, we observe that a file system may silently loose upto 94% of recoverable directories from the journal on a single bit flip during an improper shutdown. We provide a solution to the problem by creating independent replayable units called `sub-journals`, that can be replayed independently during recovery while keeping the file system image consistent.

2 Background

Journaling is a technique where the meta data operations are first recorded at a staging area called the `journal` before being written to the main file system. A `transaction` is an atomic set of meta data blocks that can be made to the main file system keeping the file system consistent. A transaction is *committed* when all corresponding meta data blocks have been successfully written to the journal. A transaction is *checkpointed* when all corresponding meta data blocks have been successfully persisted on the main file system or carry-forwarded to a subsequent transaction. If a system crash occurs between the time the transaction has been committed to the journal and before it has been checkpointed to the main file system, the contents of the journal are replayed in strict sequential order to recover meta data updates of the committed transaction. There are two types of journaling:

1. physical journaling - where the entire meta data is written on the transaction log.
2. logical journaling - where only updated meta data fields are recorded on the transaction log.

Logical journaling consumes more CPU time due to excessive bookkeeping. Physical journaling consumes more disk due to full metadata block writes. However, physical journaling may be optimized to consume less on-disk space by coalescing multiple transactions in a single `compound transaction`[10]. ext4 is one file system that uses `compound transactions`.

3 Motivation and Research Problem

A 60 second run of meta data intensive workloads generates a large number of transactions that get committed on the journal. Figure 1 shows the average number of transactions that remain uncheckpointed to the main file system. A corruption in any transaction discards all subsequent transactions[1, 2]:

1. It is not possible to determine which subsequent compound transactions can be executed while ensuring the main file system remains consistent.
2. Each compound transaction coalesces multiple file and directory updates in one meta transaction which cannot be independently replayed.

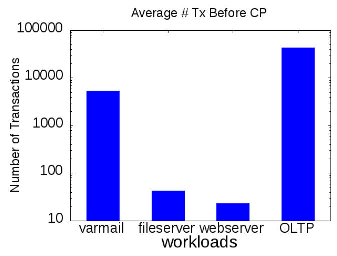


Figure 1. *Average number of transactions recorded on the journal before checkpointing to the main file system*

Avoiding journal replay prevents crash consistency problems [5] but causes severe data loss. For instance, In a recursive mkdir workload that creates 4000 directories with frequent `sync()` operations, we observe that corruption of `journal_block_header` after an improper shutdown[4] causes recovery of only 224 of 3708 recoverable directories, causing 94% directory loss. Further, journal checksums do not detect this error as they are stored at the end in a `journal_commit_block` that doesn't get accessed[3] after detecting a `journal_block_header` mismatch.

4 Approach

Our main contribution is a technique to separate transactions into independent replayable sub-journals without losing the benefits of a `compound_transaction`. If one or more sub-journals get corrupted or lost, the remaining uncorrupted sub-journals can still be replayed without compromising the consistency of the file system. Our approach is as follows:

4.1 categorize transaction updates

Each file system transaction corresponds to a particular VFS operation. VFS operations can be categorized based on the number of inodes they update. A transaction may update meta data corresponding to only one inode (eg. `write`), a parent and child inode (eg. `mkdir`) or multiple inodes (eg. `rename`) as shown in Table 1.

#Inodes	VFS Operation
0	mount, access, stat, chroot, chdir, open
1	truncate, chmod, chown, utimes, read, write
2	creat, rmdir, mkdir, link, symlink, unlink
many	sync, rename

Table 1. *Different File System VFS operations grouped on the basis of the number of inodes that get updated on each operation.*

4.2 transaction handles and inode mapping

A `transaction_handle` uniquely identifies a transaction. There are two invariants we maintain (a) All meta data blocks updated by one `transaction_handle` are written to the same sub-journal. (b) All meta data corresponding to one inode is mapped to it's sub-journal. The inode mapping may change, but the handle mapping remains constant. For `transaction_handles` that update one inode, all meta data blocks written by the `transaction_handle` are written to it's sub-journal. For `transaction_handles` that update 2 inodes, we send all the meta data blocks to the child inode's sub-journal and remap the parent inode to child sub-journal. Subsequent updates of parent inode are directed to its new sub-journal. For `rename`, we map both source parent and destination parent inode sub-journals to child inodes' sub-journal, whose mapping remains unchanged.

4.3 Recovery on crash

We consider a sub-journal as a unit of failure. When corruption or data loss occurs in one sub-journal, we discard all subsequent transaction handles mapped in that sub-journal. We sequentially replay transaction handles of all uncorrupt sub-journals.

Inode Recovery: We restore all meta data blocks corresponding to `transaction_handles` that update only one inode. While recovering from `transaction_handles` that update parent and child inodes, two cases arise:

1. If a sub-journal containing parent inode's older mapping is lost, the sub-journal containing new child inode mapping has the new parent inode. Since the lost sub-journal may contain previous child inodes that are lost, we perform directory pruning and remove older child inodes by modifying the directory `rec_len` field.
2. If a sub-journal containing new parent inode mapping is lost, the new child inode and new parent inode copy is lost and we restore the older parent inode.

Extent and Directory block Recovery: For each recoverable inode, we traverse it's extent tree and check if any blocks updated by the recoverable `transaction_handle` lie in the extent blocks range of that inode. All directory blocks within this range in the recoverable `transaction_handle` are restored to the main file system.

Bitmap Block and Group Descriptor Recovery: We rebuild the bitmap and group descriptor structures based on the number of recoverable inodes and blocks.

Super Block Recovery: We write super block updates to all sub-journals since super block maintains a list of deleted files in an orphan list.

Deleted inodes: Any deleted inode is maintained in the orphan list that starts from the super block. A list of handle and deleted inode is provided to each sub-journal. We refer this list and remove any existing inodes that were mapped in the list before replaying a recoverable `transaction_handle` update containing the same inode number. We also use this list to reset bitmap blocks.

5 Results and contribution

# Recoverable sub-journals	% Recoverable inodes
3	80.81
2	57.77
1	30.87
default journaling with corruption	6.04

Table 2. *Average recoverable inodes after sub-journal corruptions v/s single bit corruption of journal header and default replay.*

Implementation: We add 1 argument in an ext4 function to transfer inode information from the VFS layer to jbd2 during a dirty meta data write to the journal. We add 700 LOC for tracking handles, blocks, inodes and their sub-journals in jbd2. Our recovery code currently restores only inodes, extent and directory blocks and relies on e2fsck which correctly recovers group descriptor and bitmap blocks as shown in [6]. Handling delete operations is part of our future work.

Results: For a recursive mkdir workload that creates 4000 directories with 4 sub-journals, the average number of recoverable inodes after replaying a specific number of sub-journals is shown in Table 2. Since parent inodes and its relevant data structures are copied to new child inodes' sub-journal, we observe 1.66X write amplification as compared to default journaling, which we plan to reduce in future work.

6 Related Work

SpanFS [7] parallelizes journal across different domains, where each domain maps to one or more groups. Our approach does not hard-code inodes of one group to one sub-journal. Assignment of new inodes to sub-journals is uniformly distributed. Prior work[9] focuses on improving many core scalability by reducing lock contention. Both [7, 9] focus on performance and do not replay the journal on intermediate transaction corruption. Our approach is able to continue replaying subsequent transactions of uncorrupted sub-journals after an intermediate sub-journal is corrupted or lost. In [8], authors develop fine grained checkpointing to improve `fsck()` performance. Our technique does not do file level writes to disk, but instead does inode level tracking and selective replay by grouping multiple consistent `transaction_handles` into independent replayable sub-journals ensuring file system consistency.

References

[1] 2008. Responding to ext4 journal corruption. https://lwn.net/Articles/284037/. [Online; accessed 14-Aug-2019].

[2] 2019. ext4 Data Loss. Retrieved Aug 14, 2019 from https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781

[3] 2019. ext4 recovery pass exits on journal header mismatch. https://github.com/torvalds/linux/blob/master/fs/jbd2/recovery.c#L495. [Online; accessed 14-Aug-2019].

[4] 2019. Forced Reboot using /sysrq/trigger. Retrieved Aug 14, 2019 from https://www.debian.org/doc/manuals/debian-reference/ch09

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

[6] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. 2019. Evaluating File System Reliability on Solid State Drives. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 783–798. https://www.usenix.org/conference/atc19/presentation/jaffer

[7] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu, Lian Du, Shuai Ma, and Jinpeng Huai. 2015. SpanFS: A Scalable File System on Fast Storage Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 249–261. https://www.usenix.org/conference/atc15/technical-session/presentation/kang

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

[9] Yongseok Son, Sunggon Kim, Heon Y. Yeom, and Hyuck Han. 2018. High-Performance Transaction Processing in Journaling File Systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. USENIX Association, Oakland, CA, 227–240. https://www.usenix.org/conference/fast18/presentation/son

[10] Stephen C. Tweedie. 1998. Journaling the Linux ext2fs Filesystem. In *In LinuxExpoâĂŹ98: Proceedings of The 4th Annual Linux Expo.*

Reclaiming Good Transactions from a Corrupt Journal

Computer Science
UNIVERSITY OF TORONTO

Shehbaz Jaffer

Motivation

Persistent storage media is becoming larger and denser.

There is increased media corruption and hardware unreliability.

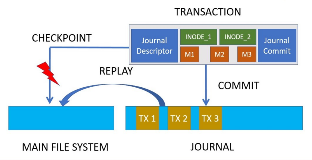
Storage systems need robust recovery techniques.

Current File System recovery techniques - replication and journaling.

Journaling

Journaling removes time consuming ***fsck*** run on unclean shutdown.

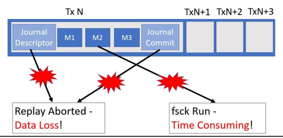
It also writes metadata to a linear log which increases performance.



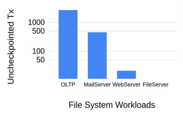
Problem

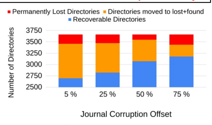
Journal header or Journal commit corruption leads to data loss.

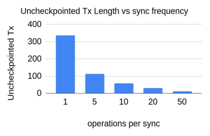
Journal metadata corruption leads to time consuming ***fsck*** run.



Sequential journal logging causes transaction dependency!







Solution: Sub-Journaling

Log atomic file system updates into different sub-journals.

Each sub-journal can be replayed ***independent*** of the other sub-journal.

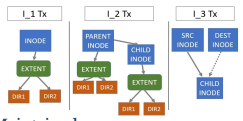
File system remains consistent after any sub-journal replay.

Categorize Transaction Updates

Number of Inodes updated (rows: File System operation)

0 inodes (I_0 Tx)	1 inode (I_1 Tx)	2 inodes (I_2 Tx)	3 inodes (I_3 Tx)
mount	truncate	create	rename
access	chmod	rmdir	
stat	chown	mkdir	
chroot	utimes	link	
chdir	read	symlink	
open	write	unlink	

Inodes updated by same transaction have parent-child relationship



Invariants Maintained

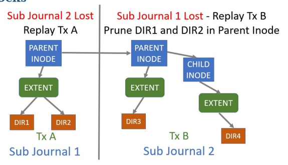
Map all metadata of one transaction to one sub-journal.

Map metadata of one inode to one sub-journal.

Remap parent inode to child inode's sub-journal for I_2 Tx and I_3 Tx.

Recovery after Crash

Inode Blocks



Extent and Directory Block

Prune ***extents*** containing lost directories.

Group Descriptor and Bitmaps

Regenerate after restoring other metadata from available sub-journals.

Deleted Inodes

re-allocation is delayed until checkpoint.

Orphan list is sent to each sub-journal.

Superblock

Record in each sub-journal

Implementation

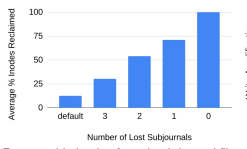
ext4 1 LOC changed

jbd2_journal_dirty_metadata (inode_no)
Register Transaction category - I_0 Tx, I_1 Tx, I_2 Tx, I_3 Tx

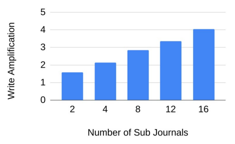
jbd2 700 LOC added

In Memory Hash Maps	On Disk Structures
<Tx_Handle - Sub-Journal>	Add to Transaction Descriptor Block
<Tx_Handle - Journal Blocks>	<Inode - Sub-Journal> Map
<Tx_handle - Inode>	<Metadata block - Sub-Journal> Map

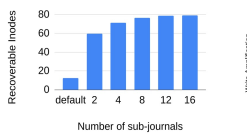
Results



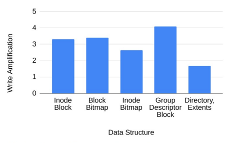
Recoverable inodes for a 4 sub-journal file system with varying lost sub-journals.



Sub-journaling technique incurs Write Amplification with more sub-journals.



Recoverable Inodes after 1 sub-journal loss with different sub-journal configurations.



Write Amplification Data Structure distribution for 4 sub-journal file system.

Future Work

Reduce Write Amplification by optimizing bitmap and descriptor block logging.

Handle delete operations by tracking orphan inodes across sub-journals.

Application crash consistency - journal one application transactions in one sub-journal.

Verify sub-journaling and journaling equivalence.

Related Work

SPANFS – Separates transactions into static domains based on Block Groups.

iJournaling – Fine Grained Journaling focusses on improving fsync() performance.

High Performance Transaction processing aims to improve multi core scalability.

Application Level Crash Consistency provides isolated streams for each application.