Not Just a Log - Activate Your File System Journaling

***Abstract*— Journaling has been widely used in many file systems to speedup the recovery after a system crash. At running time, updates are saved in journals temporally and later these updates are written to target locations. This journaling process generates write-twice overhead and significantly degrades system performance. This paper proposes an active journaling scheme that exploits the journal to improve the I/O performance and increase the reliability and availability. While the journals in conventional systems are only used during the crash recovery, our active journaling uses log-structures and free-write techniques to make the journals accessible during running time. The experiments based on trace simulations shows that our scheme can reduces average response time 14-39% for random access workloads and 0-14% for sequential access workloads.**

I. Introduction

Journaling file systems have been widely used in most modern operating systems. Journaling originates from database logging techniques developed for ensuring atomic commits of transactions. Essentially, journaling is a write-ahead log for improving the speed of recovering a file system to a consistent state. Since the system only need to scan from the last checkpoint recorded through journaling, a crash recovery typically takes a few minutes. Without journaling, crash recovery might take hours, days or even weeks to scan all the metadata on disks. For example, the main server of kernel.org suffered a file system corruption and the recovery took over a week to run *fsck* [1].

Journaling, however, provides fast crash recovery with considerable cost. A basic implementation of journaling consists of a *transaction manager* which wraps updates into an atomic unit, and a *circular log buffer* which records updates before actual modifications reach disks. Consequently, an extra disk write is associated with update operations like *write*, *rmdir*, and even *read* that updates the last access time. We call this as write-twice overhead.

To reduce this write-twice overhead, modern file systems such as SGI's XFS [2][3], IBM's JFS [4][5], Windows NTFS [6][7], and Solaris UFS [8] only log updates on metadata. Linux ext3/4 [9][10] and ReiserFS [11] file systems provide *journal* mode which performs both data and metadata logging but the default mode is *ordered* which only logs metadata updates and ensures data were written to disk before metadata. Even though, metadata logging is still heavy in consideration that (1) in many cases metadata operations take more than 50% of total [12][13], and (2) metadata operations are write intensive. To further alleviate journaling overhead, efforts have been made on logging to a separate device, merging transactions [9], ignoring less important metadata updates, designing lightweight transaction manager [14], wandering logs [11], etc. A comparison between Linux ext2 and ext3 file system serves a good example to show the journaling overhead because ext3 directly derived from ext2 with the journaling feature. The results in [12] shows that ext3 (ordered mode) is 4.36 times slower than ext2 on metadata write micro-benchmark. It is reported that ext3 (ordered mode) takes 90% more time to complete the PostMark benchmark [15]. Our evaluation shows further tricky performance impacts of journaling in ext3 (Section IV).

All the journaling file systems discussed above have one common characteristics - the journal is "write-only" until the systems crash. We define this journaling category as **passive journaling** which improves system availability and reliability in the sense of fast crash recovery with the cost of twice-write overhead.

This paper aims to design a new journaling scheme which turns the journaling overhead into a performance gain. Specifically, instead of journaling as just a log, consistently paying write-twice overhead, waiting and awaiting to be summoned for system crash, could we release the leash, activate journaling in normal operations, and make journaling an active force instead of a drag? To answer this question, we first analyze the fundamental traits of journaling (based on ext3 file system), and the opportunities to make the prestige:

1) *Up-to-date Data:* We write file system changes to the journal before in-place updating. Therefore, the data in the journal are the latest and "readable".
2) *Free Copies:* After in-place updating (checkpointing), replicas are made for "free". This allows I/O scheduler to read data from replica copies that are physically closer to the current disk arm position.
3) *Sequential Writes:* We always journal sequentially which is much faster than updating the blocks to their original locations separately. However, if we loose the constraint of journaling on the fixed circular log buffer on disk, we could further speed up journaling by writing to the closest free space. (Section III-A)
4) *Non-Sequential Reads:* Fundamentally, most modern file systems handle sequential workloads well because the layout is designed based on logic and spatial locality. However, non-sequential workloads typically suffer. As temporal locality of writes is recorded during journaling, we have an opportunity to record non-sequential reads in the journaling disk write operations with negligible overhead based on "freeblock scheduling" [16][17]. This obsevation motivates us to extend the role of journaling

from a “logger” to a “learner”. Specifically, not only logging the file system updates, the journaling process could also record what current file systems have unfavorably handled, i.e. the mismatch between user requests and current disk layout. This paper proposes an automatic analysisscheme to better match disk layouts and workload patterns. The reconstruction process could be done along with journaling writes or using system idle time. (Section III-B)

This paper presents a new journaling category as **active journaling**. We propose to change the fundamental role of journaling from a “logger” to a “learner” and a “constructor”. A “learner” means to record what the current file system has adversely handled on runtime. A “constructor” means to analyze and adapt. This new journaling scheme will not only provide performance gain, but also improves system reliability and availability. Conventional passive journaling improves system reliability and availability in the sense of fast crash recovery. However, we could obtain higher reliability from active journaling with the ability to be self-healing from silent data corruptions. (Section III-F)

The rest of the paper is organized as follows. Section II describes the background of file system journaling. Then, we present our active Journaling design in section III. Section IV presents our evaluation methodology and simulation results. Section V discusses prior related work. Section VI concludes the paper and gives future work.

II. Passive Journaling

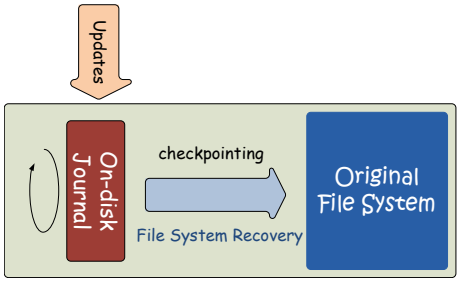


Fig. 1. **A conventional journaling file system first writes updates to on-disk journal (transaction committing), then modifies the target locations (transaction checkpointing), and reuses the space as a circular buffer.**

This section describes how a conventional journaling file system works. The basic journaling procedure in most modern file systems, such as ext3, ReiserFS, XFS, JFS, and NTFS, are similar. Figure 1 and 3 show a conventional journaling file system and its on-disk layout. To simplify our discussion, we define several terms below.

- *log, journal, and log-structured*: We use log and journal interchangeable in this paper. They both mean that a record of changes is made on disk before actually in-place updating. For the approach of using logs as an on-disk structure to store file content (log-structured file system (LFS) [18][19]), we use the term log-structured or journal-structured.
- *original file systems*: Typically journaling as a feature that can be enabled or disabled. In this paper, original file system is intended to describe a file system with journaling disabled, or the other parts of a file system except journaling itself as shown in Figure 1.

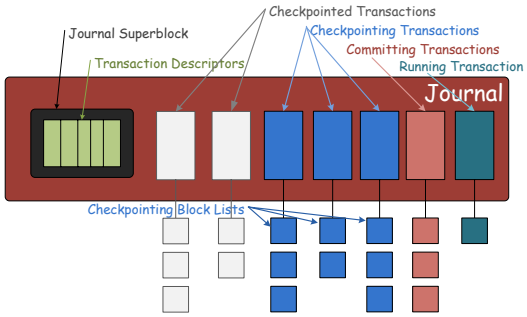


Fig. 2. **A logic view of journal which shows four stages of a transaction: a running transaction (stage 1) handles new updates; a committing transaction (stage 2) writes atomic file system modifications to the on-disk journal; a checkpointing transaction (stage 3) moves committed updates to actual locations; a checkpointed transaction (stage 4) is a transaction finished checkpointing, dead, and the on-disk space related to this transaction is reusable.**

Transaction: In a journaling file system, we use transaction to group related updates and make sure either all or none of these updates are written to the disk. This guarantees the atomicity of a transaction. For example, a file *deletion* generates a series of disk updates. First of all, its inode and directory entries are removed. Second, the disk blocks occupied by this file are released by updating related allocation table or bitmap. If only part of these updates are written to the disk, the disk is then in an inconsistent state. A journaling file system aims to minimize the chance such an inconsistency occurs. Specifically, all related updates are recorded in the on-disk journal. After the transaction is committed successfully, we can then safely write these updates to their actual locations. If the system crashes, there are two possible cases. The first case is that the system fails before the transaction is committed. Then no actual updates are written to disks and the file system is still consistent. The second is that the system after after committing to the journal. Then we need to redo this transaction based on what recorded and make the system consistent.

This procedure causes a write-twice overhead. A common optimization is compound transaction used in ext3. Instead of considering each operation as an individual transaction, we could contain a batch of operations into one transaction. This has advantages over fine-grained transactions both in clustered I/O and in frequently updated data in a short period of time (write it once instead of multiple times). Note compound transaction could only combine operations occurred in a short period of time (e.g. 5 second). This is limited by the commit policy.

Commit Policy: Journaling does not guarantee durability of modification, which means that only the committed transaction can be recovered after a system crash. To ensure a reasonable

guarantee of durability, a timer is generally used to enforce transaction committing. However, if the timer is too short, file system would act as FFS [20] synchronous metadata write that has severe performance penalty. In practical, the default timer is 5 seconds for metadata updates in NTFS [7] and ext3 file systems.

Checkpointing: The process of flushing journal updates into their actual locations in original file system is known as checkpointing. Normally, this is triggered immediately after transaction committing (*immediate checkpointing*), or by on-disk journal space shortage. In practical, the on-disk journal size is around a few dozens of MB. For example, the default maximum on-disk journaling size is 32MB in JFS, 64MB in SolarisOS UFS, and 102400 file system blocks in Linux ext3. After checkpointing, the space occupied by this transaction is reused. Thus, if the usable space left in the on-disk journal is below some threshold, checkpointing is then triggered. In the latest Linux kernel 2.6.20.4, whenever a single atomic update to ext4fs starts, the Journaling Block Device Layer 2 (JBD2) checks and ensures 1/4 of the total journal size plus current update size available. Figure 2 shows a logic view of the journaling process.

Journaling Mode: File system needs to update both data and metadata. The safest but slowest mode is to journal all data and metadata updates. This approach is called *journal* or *data* mode which works unfavorably with write intensive workload.

Since file system consistency is mainly maintained by file metadata and file system metadata, journaling only metadata is another approach that is more widely used. The fastest journaling mode is *writeback* mode. SGI's XFS [2] acts in this mode. However, this mode is not safe if the system crashes when the metadata has been journaled but related data have not been saved to disks. For example, we append data to a file and the system crashes after the corresponding meatadata are journaled. In system recovery, the information recorded in the on-disk journal shows the transaction is correctly committed. As a result, we can potentially see blank, or even worse, random data at the end of this file.

To overcome this issue, file systems like JFS and ext3 adopt *ordered* mode as the default. It demands that all the data change must be updated to actual disk locations before corresponding transaction commits.

On-disk Layout: Figure 3 shows a typical disk layout of a journaling file system. Followed by the reserved blocks for boot loader, a disk partition is divided into equal-sized groups (except the last group) named as *Cylinder Group* (CG) in Berkley Fast File System, *Block Group* (BG) in Linux extended file systems and *Allocation Group* (AG) in SGI XFS. In this paper, we prefer the term *Allocation Group*. Each group places related file data, inode and directory entries physically together. In addition, concurrent accesses to different groups are allowed to improve the degree of I/O concurrency. The first AG (AG0) stores the *superblock*, including critical system-wide information. Each AG maintains a allocation table or bitmap for both inode and blocks.

Usually a reserved inode is used to index the on-disk journal, such as, aggregate inode #3 in IBM's JFS [5], system file #2 in Windows NTFS [7], and a special file *.journal* in Linux ext3. The on-disk journal can be simply viewed as a regular file. On-disk journal is a write-ahead log that can be located in a separate device. But most file systems normally allocate consecutive blocks at the beginning of a file system for the journal (i.e., AG0). The rationale behind this is that both on-disk journal and the superblock are both frequently updated and thus they should be stored physically together to avoid long disk seeks.

The on-disk journal consists of a journal superblock and a series of transactions. The critical fields in the superblock are the last checkpointed transaction ID and the first unfinished transaction offset. During a crash recovery, this checkpoint is used to locate where we start to scan the journal. Each transaction is composed of a description block, a series of updated blocks and a commit block. The description block keeps the location mapping of the updated blocks. The commit block is used to mark the successful commit of a transaction. The transaction ID in it is the same as its corresponding description block.

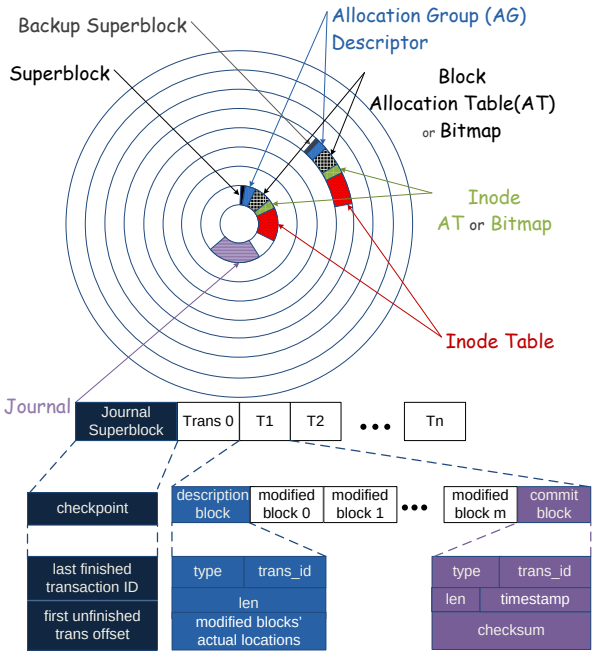


Fig. 3. **A Typical Journaling File System On-disk Layout**

Crash Recovery: With the help of transaction and on-disk journal, crash recovery is straightforward in journaling file system. We first check if the file system was safely unmounted (a flag in superblock). If not, we find the journal superblock, start from the first unfinished transaction offset, and sequentially scan the unfinished transactions. For the transactions missing commit block, we simply ignore them. For committed transactions, we redo the transaction: write the journaled blocks inside into their original locations. If several journaled blocks point to the same disk location, the latest will

be written.

III. Active Journaling

The constant effort of journaling on twice-write was observed as an overhead. However, we notice free replicas have been made along with this "overhead". The only active usage of the journal in system crash is an apparent waste. Figure 4 illustrates a common case we can potentially benefit from reading the journal in normal operations. The rationale is, when we have more than one identical blocks in the disk, we choose to access it from the closest location.

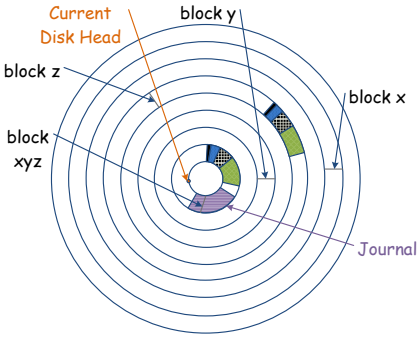


Fig. 4. **The upcoming requests read block x, y, z in a sequence. In conventional journaling file systems, disk head seeks to the original locations of x, y, z respectively instead of directly fetching x, y, z from the journal within one sequential read.**

This section details our active journaling design. We first describe the on-disk layout. Then we introduce how we capture the access patterns current file systems poorly handled, where we store them, and how we improve system performance from reconstructing the disk layout. In addition, this work introduces opportunities to enhance sequential prefetching efficiency on non-sequential workloads. Here an important note is the extra resources active journaling uses to improve system performance is just free disk space, thus boosting system without an extra penny. Then the corresponding issue is how we release the free space when needed. Being elegant, active journaling can adaptively grow and smoothly shrink without mentionable cleaning overhead. In the extreme case of no free space available, our design can shrink and work in the same way as passive journaling. But we suggest a little more space should be saved for the stable and reused non-sequential access patterns we captured and stored. Crash recovery is the basic motivation of journaling. Active journaling provides mechanism to self-heal from silent data corruption and better guarantees data reliability and durability.

Figure 5 depicts the on-disk layout of our active journaling design. Compared with passive journaling, there are two notable differences: extra group-journals and a virtual map (*vmap*) file. Note that we keep the original file system mainly untouched. Therefore, our design is well compatible with current journaling file systems. We can easily upgrade a file system from passive to active journaling or vice versa.

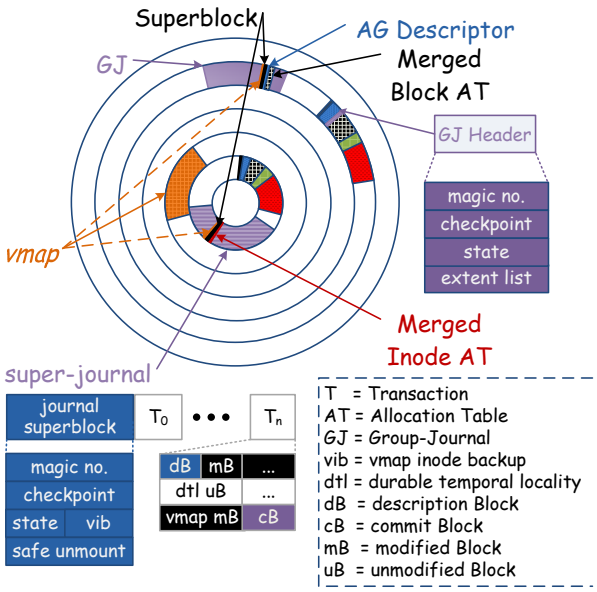


Fig. 5. **Active Journaling File System On-disk Layout**

A. Journaling "Anywhere"

The location and size of the super-journal is described in the file system superblock which is the same as passive journaling. Inside the journal superblock, we add a magic number field to identify the journaling type. Another important new field is the state which marks the super-journal itself as *clean* or *dirty*. When all the transactions including the transactions in the group-journals are checkpointed, we mark this field in the super-journal as *all clean*.

The difference lies in the group-journals. The header of each group-journal is indexed in the AG descriptor and located in the fixed location of each allocation group. The contents of group-journals are not fixed in each AG. We add extent list field in each group-journal header to track the space used by each group-journal.

As we introduced, we can improve journaling speed and reducing interference with other file system operations by commiting transactions to a close available place instead of seeking back to AG0. The group-journals serve this purpose. Each time we need disk space to commit a transaction, we start from the closest group-journal. If the space for the group-journal is full, we reused the checkpointed space or allocate new space. Thus, we boost system performance by journaling "anywhere", which has two apparent advantages on both journaling itself and normal write operations. First, writing to a close place largely decreases the seek time of the journaling itself. Second, journaling "anywhere" decreases the interference of journaling writes with user demanding operations. The place to commit transactions is followed by how read and other operations move the disk head. From the point of a user or application, read is more important because in general read is synchronous and blocking intended while write is asynchronous and delayable. The user does not care

when a write actually finishes in disk. They only need the system to make sure the durability of the data. Therefore, checkpointing is not enforced by the tiny free on-disk journal space in passive journaling, but by the entire free disk space. If we have enough free space, journaling would cache the writes in disk and postpone checkpointing until the system workload is light. Thus, we overcome the shortcoming of passive journaling which enforces checkpointing even if the system is very busy and we have plenty of free disk space to buffer and delay checkpointing operations.

Note that journaling serves the original file system as a crash-resistant safeguard. We need to guard internal inconsistency of the journals. We also use transactions to manage the operations modifying important journal structures. For example, the operation to allocate free space for a group-journal is critical. We make sure these operations are journaled in already allocated journal space.

The adaptable active journaling structure boosts write speed, aggregates write operations into sequential workload, and exploits disk free space to avoid contention between checkpointing (low priority) and requests from users (high priority). Moving further, the major motivation of this active journaling structure is to turn journaling from a performance overhead into a gain by exploiting the free replicas.

B. Durable Temporal Locality

Temporal locality is the major principle of cache replacement algorithm design. Least Recently Used (LRU) algorithm is commonly used in file and database systems. Normally, temporal locality or a special access pattern could take effect before it is replaced out of the main memory. A definition to temporal locality is: in a given cache architecture, a particular data word exhibits useful temporal locality if it will be accessed again before it is replaced from the cache[21]. In our design, the life cycle of the useful temporal locality is extended to disk level by storing the poorly handled access patterns in the journaling processing. We define this as *Durable Temporal Locality*. Specifically, three steps of action are required to make this work: *find*, *record*, and *retrieve*.

First, to find the poorly handled access patterns, we simply need to distinguish them form sequential access patterns. We adopt two views to differentiate sequential accesses from the non-sequential. The first view is the logic view on semantic level in consideration that the organization of the original file system is based on spatial(logic) locality. Access patterns match this logic locality should be viewed as sequential. For example, a request to read a big file from the start to the end is considered as sequential even if the file might be fragmented as pieces on disk. We assume this would be handled by background defragmentation tools. Similarly, operations like *ls -l* are sequential from the logic view. The second view is the physical view. We consider read requests temporally close while physically distant as the targets which could degrade system performance. In our design, we consider the size of a single allocation group as the threshold, i.e. if the physical block distance between two consecutive reads is smaller than a single allocation group, we consider them as "sequential" and gather them in a sequence; otherwise, we separate them into different groups. We also call this threshold as "zero" seek distance. Based on this simple rule, we filter the read requests into *sequence groups*.

Remember we are trying to find non-sequential access patterns based on the principle of temporal locality. If there is a time gap (0.1s in our evaluation) between two consecutive sequence groups, we consider the temporal connection between them weak. Consequently, sequence groups are further separated as batches of sequence groups. We call each batch of sequence groups as a *sequence window*.

Now we focus on each sequence window. Simply, if the group size is small, we deem it as a mismatch between current disk layout and user access patterns. This mismatch would cost considerable disk positioning time. What if these small-sized sequence groups are located close to the big-sized sequence groups on disk, we could process these requests much faster. In our evaluation, the sequence groups with size larger than 15 blocks are called the *docks*. We keep a block preference location table to track those sequence groups having a preference to locate close to their temporally previous *dock*. This is a primitive learning algorithm.

Second, to record the non-sequential access patterns, we write it down along with transaction committing as freeblock scheduling [16][17] or use system idle time. We simply check the block preference location table for the blocks with the preference locations close to current transaction committing place. The transaction structure itself is mainly unchanged as depicted in Figure 5. The difference lies between the descriptor block and commit block. Originally, only modified blocks are recorded. Now we need to store replicas and track them, which requires a new data structure called *vmap*.

Third, to retrieve the recorded data, we use a B+ tree called *vmap* to trace the blocks. Figure 6 depicts the details of *vmap*. It is keyed by physical block number. When we commit a transaction, we also record information to track these blocks in *vmap*. If a block is updated, then the original blocks are out-of-date, we use a version field to identify them. Here we use the transaction ID as the version number. The benefits of this is we can trace how the blocks were updated based on the transaction ID. Because we also write down some unmodified blocks, a physical block may have multiple replicas in the disk. As a matter of fact, the *vmap* provides a block virtualization mechanism.

With the *vmap* to track the blocks, we need to make a slight improvement in I/O scheduler to benefit from the replicas. The basic elevator scheduling algorithm is unchanged. We always choose the closest request to schedule based on last issued request location and the elevator direction. Here when we add a read request to the I/O scheduler, we add all the replicas and let the elevator sort them based on physical locations. When a certain block is chosen based on the original elevator algorithm, we need to do two extra steps. First, check if we have a closer replica in the opposite direction. If so, we issue the closer one and keep the elevator direction unchanged.

Second, we need to erase all the replicas related to a request in the elevator after issuing it.

The memory cost of *vmap* is considered. First, assuming the file system block size is 4KB, 5MB memory suffices for tracking 1GB blocks. Considering most of today's computer systems equip with 1 Gigabytes or larger memory, it is usually not a problem to keep the *vmap* in the memory. Second, the purpose of *vmap* is to filter out repeatable durable temporal locality mismatched from spatial locality. The sequential bulk workloads are not traced. Third, we mainly just need to keep the indexes to frequently referenced and newly updated blocks in memory.

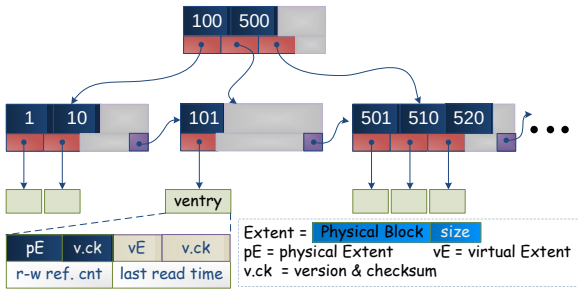


Fig. 6. **A detail view of the *vmap* B+ tree structure keyed by physical block number. *vmap* is crucial in active journaling with three roles: (1)block virtualization, (2)repeatable non-sequential access patterns filter, and (3) silent data corruption guard.**

C. Journaling - a "logger" or a "constructor"?

File systems could be generally categorized as in-place and out-of-place updating file systems. For example, ext2 is a complete in-place updating file system. On the contrary, log-structured file system is a complete out-of-place updating file system [18]. The benefits of out-of-place updating is write speed since we do not need to seek to the original places and all the updates are written in sequential. The problems of out-of-place updating file systems are (1) disarrangement of logic locality, (2) cleaning overhead, and (3) sensitivity to available free space. These three problems are connected. Because sequential access patterns complying with logic locality are common in file systems, the read performance is degraded. A generally used method is background defragmentation. Journaling file systems are a combination of in-place and out-of-place updating. We first do out-of-place updating and then in-place updating by checkpointing. Comparing a journaling file system with a LFS or WAFL with background defragmentation, we could observe the similarity of out-of-place update first and then in-place update (checkpointing or background defragmentation). These file systems actually both involve twice-writes.

However, do we need to write twice all the time?

From the performance perspective, the only reason we write blocks back to original locations is to preserve the spatial locality. If the data have no such locality, we have no reason to do. Looking back to passive journaling file system, journaling acts as a "logger" to the file system. The original file system's structure is faithfully untouched. We always write twice without considering the performance degradation. This overhead severely degrades system performance in many cases.

Two reasons motivate us to turn the role of journaling from a "logger" to a "constructor". First, journaling has been set up as default in large. If merge journaling into file system could make it work more efficiently, we should do it. It's a historical factor that journaling is treated as a optional mode. For example, Solaris OS 2.4 UFS implemented metadata logging option in 1994. Solaris OS 7 integrated metadata logging into base UFS. In 2004, Solaris OS 10 turns on logging by default.

Second, modifications go through journaling before actually updating the file system. We view all the updates and accesses as stimuli showing us in certain cases the file system poorly performs or acts well. In our active journaling design, we expect to extend the function of journaling: let it learn from the stimuli, let it re-construct the file system based on the designated workloads, and let it improve system performance and reliability.

Here we describe how we merges journaling with the original file system and produces the synergy of in-place and out-of-place updating. We consider if certain file system data or metadata (1) have no strong spatial locality to preserve, (2) have high priority to stay in memory, or (3) highly write intensive, we should adopt the write "anywhere" policy and save the overhead by always doing an extra in-place updating.

We take the file system superblock as an example, which conceptually represent the file system. Because it's important, it's always cached in memory. Also it is write intensive in consideration of recording system-wide changes like free blocks and inodes left. From the view of performance, instead of writing it in the journal first and then checkpointing it back to the original place, we should write it "anywhere" and record the new locations. This is much faster and more efficient than twice-writes all the time. From the view of reliability, it's better to keep a track of different versions of updates. This also eliminates the chance of silent data corruption by in-place writing many times. Therefore, both performance and reliability are improved. This analysis also apply to other system metadata like AG descriptors.

To support write "anywhere" in our design, *vmap* introduces a block virtualization layer. We could simply view it as an indirection map. For example, physical block *i* is the superblock's original location. Now we need to update the superblock and we journaled it in a new physical block *j* along with transaction committing. In the same transaction, we also update the leaf entry keyed by *i* in the *vmap* as a out-of-date version. Block *j* is recorded in this entry as a new version of block *i*. we call block *j* as a virtual block. All of this information are written in the same transaction so that we could track the updates to the superblock. Note we could have multiple virtual blocks to the same physical block. For system critical data, we track more versions. For normal blocks, we only track two versions which is enough for general reliability requirements.

In summary, active journaling synergizes in-place and out-of-place updating. We keep this discussion in next subsection on merging allocation tables into the journal.

D. Journal-structured Block Allocation Table

Three reasons drives our design to merge block allocation table into the journal. First, all free space in each allocation group is viewed as potential journal space. The journal could grow for tracking modifications to the file system, and shrink for space shortage. Thus, the functionality of journaling and block allocation table is overlapped. Second, updates to the block allocation table are recorded into journal first and then write back. This is inefficient and unnecessary. Third, the most important reason is the concern of performance and how we effectively allocate and reclaim free space. There are two common ways to manage disk free space - bitmap and extent.

Bitmap is the common and simple way to represent disk block usage. We could simply treat bitmap as an array of bits and the Nth bit indicating whether the Nth block is allocated or free. The first problem of bitmap is spatial overhead. If the file system block size is 4K, we need 32M to represent 1T disk space. The capacity overhead on disk is not considerable, but the cost of keeping this in memory is a waste. The fundamental problem of bitmap happens when we free blocks. To allocate free space, we normally get a sequence of blocks close to each other. We only need to change consecutive bitmap blocks to mark these blocks as taken. But, when we free blocks, a operation like "*rm -rf *.log*" forces the file system to update a series of non-consecutive bitmap blocks.

Extent is an improvement to bitmap which has been generally used in modern file systems. An extent could effectively express a contiguous region of free space using the starting address and length. This solves the problem of spatial overhead for expressing large consecutive free spaces. Extent is usually organized by B tree keyed by the starting address. Thus contiguous block allocation is efficient. However, we still need to update a lot of extents when users randomly delete files.

In passive journaling, we first write these updates to the journal and then checkpoint them back to the original locations. Here, we use the journal itself to handle the block allocation table. Each allocation group has its group-journal (super-journal for AG0) and the free space are managed by the extent list. Each time we allocate or free disk space, we keep the updated extent in the journal and indexed using the *vmap* for simplicity. By doing this, runtime allocation and free are largely speed up even if we free a series of random blocks. Only one sequential write along with transaction committing is necessary for managing free space changes in a period of time.

Note that in the active journaling structure, we still keep the space for the superblock, AG descriptors, allocation tables as the original file system. We do this for simplicity and compatibility. We can free these blocks and let the journal superblock and group-journal header index the location of these data structures. Here we keep the original locations of these data structures and let *vmap* handle the block virtualization. Then we can put the blocks inside the journal. Besides this system-wide metadata, we can also apply block virtualization to the small files which normally have weak spatial locality.

E. Checkpointing Policy

In passive journaling, we initialize checkpointing immediately after a transaction is successfully committed. This operation has a performance penalty in consideration that metadata are usually dispersed. A committed compound transaction may contain hundreds of scattered metadata blocks . As we discussed, writing twice is unnecessary if the data have no spatial locality to preserve. Metadata are such candidates [12]. In our design, *vmap* tracks the new locations of metadata. Therefore, we only need to checkpoint the updates to *vmap*. This is similar to the idea called *wandering log* in Reiser4 [11]. Basically, we update the pointers to the data instead of writing to the original locations. The advantage of this optimization is that pointers are normally much smaller and denser. We call this as *virtual checkpointing* which has another benefit on recovery from users unintentional operations.

Note that this optimization turns *vmap* to be the most critical data structure by which the system and file level metadata are tracked. The changes to *vmap* are always tracked along with transaction committing. We also make backup of it.

F. Self-healing from Silent Data Corruption

With the increasing of disk capacity, the occurrence rate of data corruption becomes non-negligible. Based on recent evaluation [22], the probability of data corruption is 0.86% on nearline disks and 0.065% on enterprise-level disks. Therefore, we not only need to improve performance but also reliability of the file system. A passive journaling file system could recover from a system crash. However, for disk level silent data corruption, passive journaling is impotent. For the case that some sectors are inaccessible, the file system could mark them as bad. But for some latent errors responding corrupted data, a passive journaling file system is unware of it and returns corrupted data to the users.

In our active journaling design, *vmap* tracks the updates to file system. Nearly all updated blocks have at least two versions or copies. To make file system self-healing from data corruption, we keep checksums for the versions of blocks. Each time we read a block from the disk, we also check the integrity by verifying the checksum. If the checksum does not match, we could recover the block from a copy or a older version.

Note that data corruption often occurs during write process somewhere in the IO path [23]. Therefore, the benefit of active journaling on reliability is two-fold. First, the *vmap* block virtualization diminishes in-place repeating writes, thus decreasing the chance of data corruption rate. Second, multiple versions of data tracked by *vmap* provide a mechanism for system self-healing from silent data corruption. Based on the IRON Recovery Taxonomy in [24], this is the highest level reliability guarantee.

Therefore, not only boosted performance from active journaling, we also improved system reliability to a higher self-healing level.

G. Cleaner

Sensitivity to available free space is an issue to active journaling. In fact, all the out-of-place update file systems like LFS [18], WAFL [25][26], and ZFS [27] have the same issue. If the free space is limited, the advantage of write "anywhere" turns out to be write "nowhere". A common way to solve this issue is to run a background defragmentation process. The blocks are read again into memory, pick up the live data, sort and write them back to disk. This could be a considerable overhead.

In active journaling design, cleaning is much easy and efficient because we provide a block virtualization for out-of-place updating. The original places are preserved. Note that we may write down many replicas but only a fair percentage are useful. To filter out the useful, we include a reference count and a last read time field in each leaf entry of *vmap*. In the cleaning process, we keep highly re-referenced and newly updated blocks if possible. The process of cleaning is as follows:

1) Scan *vmap*. Sort and group virtual blocks into each AG. Mark virtual blocks whose corresponding physical blocks are not the latest version.
2) Identify AGs without out-of-date physical block. Update physical blocks if necessary.
3) Check identified AG's group-journal header. The journal space taken by transactions previous to last checkpoint in this AG is ready to reclaim.
4) Update *vmap*. Keep the highly re-referenced and newly updated blocks if possible, or pack and relocate them.
5) Execute these updates as a system transaction.

H. Crash Recovery

Active journaling is basically the same as passive journaling in crash recovery. We scan from the checkpoint, redo the committed transaction, and discard the unfinished transactions. However, there are two notable differences. First, we have multiple pieces of on-disk journals. Unless the state field in the super-journal is *all clean*, we need to check the journal state field in each allocation group. If it is clean, we skip it. If it is dirty, we scan from the checkpoint and recover the file system to a consistent state.

The second difference is more fundamental. Because we track updates with *vmap*. Then the crash recovery process is to redo the updates to *vmap* recorded in the committed transactions instead of all the block updates. After we have a consistent vmap, the file system is consistent and ready to work.

IV. Experimental Evaluation

In this section we first describe our methodology of simulation and experimental configuration. Then we present our evaluation results on traces [28] representing sequential access workloads, random access workloads, and concurrent applications workloads.

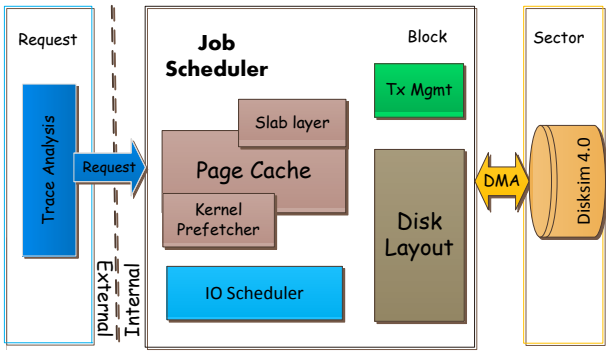


Fig. 7. **The Simulator**

A. Simulator

Our simulator is approximately 16,000 lines of C++ code, built upon C++ STL and Boost Libraries [29]. The simulator structure is depicted in Figure 7. Our simulator is trace-driven and the requests from traces are processed on open model, i.e. the request arrival time is fixed and we use average response time to measure the sytem performance. The simulator consists of five components: trace, page cache, I/O scheduler, transaction management, and disk component.

The disk component consist of the Disksim 4.0 [30] validated disk simulator and our disk layout modules. We also simulate the behavior of DMA to disk. Any disk request need to acquire the DMA channel resource before issuing to disksim. In our evaluation, we simulate ext2, ext3, LFS, DualFS [12], hFS [31], and our active journaling (AJ) disk layout. All the disk layouts are inherited from a single C++ class. We could easily expand the code and simulate more disk layouts. The disk layout are built from all the traces [28] used in our evaluation. The process of building disk layout is as follows. First, we format the block group or segment, i.e. reserve the space for system level metadata. Second, we add file system specific system level files such as on-disk journal file and LFS checkpointing region. Third, we start from the directory with the smallest inode number, and add the directory and the files recursively. Files are placed to their parent directories as close as possible.

Transaction Management component is vital because our major simulation target is journaling file systems. We faithfully emulate the details of ext3 compound transaction processing model and three journaling mode based on vast available documents [9][10][32] and latest Linux kernel code 2.6.20.4. We extend this module to our active journaling transaction processing module.

In consideration that I/O scheduler has a profound impact on file system performance, we wrote a I/O scheduler to simulate the behavior of FIFO, elevator, deadline, and anticipatory scheduler. It also simulates the subtleties of logging behaviors in LFS, DualFS, and hFS.

The page cache component includes page cache, kernel prefetcher and the slab layer module. These modules actually preform different functions. We view them together here just because they are highly related to each other. The page cache and kernel prefetcher modules are modified from accuSim [28]. The purpose of the slab layer module is to manage objects smaller than one block like inodes and dentries.

The trace component performs the function of interpreting different trace formats to our internal simulator request format. In our simulation, file level metadata including directory, inode and indirect blocks accesses are simulated. System level metadata accesses like block allocation are ignored. A tricky issue is the update on the last access time while reading a file. In our simulation, we mark the block containing this inode as *minor dirty*. We perform normal transaction processing or writeback for this block but it is still "readable" by user requests.

B. Experimental Setup

Our simulator allows a multitude of parameters to be varied. To guarantee the fairness of the performance comparison, we fix most of the parameters. The differences between target file systems are mostly confined in disk layout and write strategies: in-place writeback (ext2), transaction processing (ext3, aj), or log-structured writeback (LFS, DualFS, hFS). Unless explicitly explained, the parameters are listed in Table I:

TABLE I
EVALUATION CONFIGURATION

Common Configuration			
Replacement Alg.	LRU	Kernel Prefetch	off
Read Expire	0.5s	Write Expire	5s
Disk Model	atlas10k	DMA Channel	2
Inode	128 bytes	DEntry	256 bytes
Block	4096 bytes	Warm Up	off
ext2		LFS	
Data&Metadata	in-place	Data&Metadata	log-stru
Writeback Timer	5s	Writeback Timer	5s
Block Group	32768 blocks	Segment	1024 blocks
DualFS		hFS	
Log Partition	10%	Log Partition	10%
Metadata	log-stru	Metadata	log-stru
File$\leq$1 block	in-place	File$\leq$1 block	log-stru
File$>$1 block	in-place	File$>$1 block	in-place
Writeback Timer	5s	Writeback Timer	5s
Group	32768 blocks	Allocation Group	32768 blocks
Segment	512 blocks	Segment	512 blocks
ext3		aj	
Journal Size	8192 blocks	Journal Size	free space
Journal Mode	ordered	Journal Mode	ordered
Metadata	transaction	Metadata	transaction
Data	in-place	Data	in-place
Commit Timer	5s	Commit Timer	5s
Checkpointing	immediate	Checkpointing	immediate
Block Group	32768 blocks	Allocation Group	32768 blocks

Active journaling design improves from traditional passive journaling in many aspects. To focus on the performance improvement coming from learning and re-constructing the file system layout, we use the same *immediate* checkpointing policy as ext3 instead of *vitrual* checkpointing. This eliminates the performance gain from less writes. In addition, the experimental results are tested with zero overhead of writing the replicas to the preferred locations captured by our primitive algorithm discussed in Section III-B.

Note that our focus is to faithfully simulate journaling file systems. For LFS, DualFS and hFS, our simulation is limited. First, the disk layout are initialized based on directory affinity. Then the disarrangement of spatial locality is generally negligible because we did not test on write-intensive traces. Also, the overhead of background cleaner is not simulated. Second, many optimization strategies such as online relocation [12] and dynamic segment size [31] are not simulated. But we do simulate the hole-plugging [19] logging strategies in our simulator. Third, DualFS and hFS have a separate log partition. Because we did not find specific size recommendation in [12] and [31], We simply choose 10% as the log partition size. This can result notable performance degradation. We shall discuss this issue based on our evaluation results. In a word, our evaluation on LFS, DualFS and hFS is limited on disk layout and write strategies. This may vary from these file systems' actual performance.

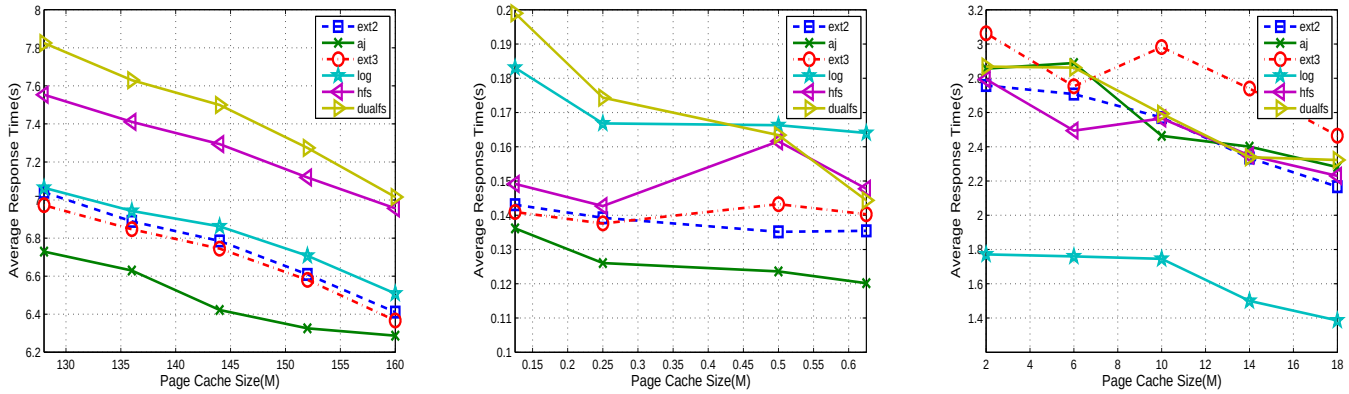
TABLE II
DISK SEEK DISTRIBUTION

time distribution (ms)				*cscope* cache size 160 MB			
	0	$<$2	$<$4	$<$6	$<$7	$<$8	$>$8
ext3	0	7769	3441	29	3	**210**	**656**
aj	0	7150	3382	6	1	**8**	**560**
distance distribution (cyl.)				*cscope* cache size 160 MB			
	0	$<$21	$<$81	$<$321	$<$641	$<$1281	$>$1281
ext3	7096	647	293	601	882	1000	**894**
aj	6550	514	296	496	797	986	**584**

C. Sequential Access Workloads

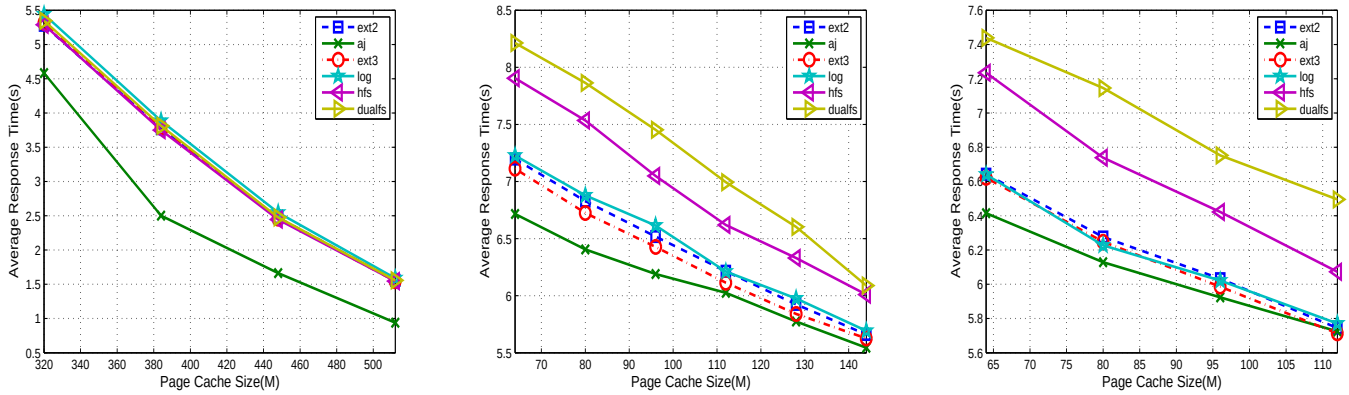
We choose three traces [28] to represent the category of sequential access workloads in our evaluation: (1) *cscope* (examining the Linux kernel 2.4.20 code), (2) *gcc* (building Linux kernel 2.4.20), and (3)*viewperf* (a SPEC benchmark which measures the performance of a graphic station.). In these traces, files are mostly read entirely and sequentially. Figure 8 shows our evaluation results on which the following observations can be made.

First and the most important, non-sequential access patterns on file level is common even though we access each individual file sequentially. For example, user applications often interact with system and library files. Applications frequently access files located in different directories. Being specific in *cscope* trace, we observed that 98.9% requests are gathered in two portions of ext2/3, LFS and aj disk layouts. The first portion is nearby the AG0. The other one is in AG[48-57]. Therefore, seeking back and forth is inevitable. Our primitive learning algorithm in Section III-B shows 0.25s was saved for each request response time in average. For example, Table II compares the disk seek distribution between ext3 and aj based on the output from disksim, which confirms this performance improvement. However, our algorithm is not smart enough to



(a) *cscope* Trace (b) *gcc* Trace (c) *viewperf* Trace

Fig. 8. **Sequential Access Workloads Simulation Results**



(a) *tpc-r* Trace (b) *multi1* Trace (c) *multi2* Trace

Fig. 9. **Random and Concurrent Applications Workloads Simulation Results**

directly make copies of the files in the first portion to the second. There is great potential for further improvements.

Second, the performance of DualFS and hFS is lower than other targets in general. A possible reason is DualFS and hFS separate data and metadata in different partitions on disk. This separation is against the *cylinder group* design originated from Berkeley Fast File System (FFS) [20], and followed by many modern file systems including ext3, XFS, JFS, etc. The basic principle is spatial locality. File system *read* and *write* operation to a file usually need the metadata to access the data. Therefore, separating the metadata and data of the same file could cause a long distance seek for each operation.

Third, ext3 performs slightly faster than ext2 in *cscope* trace. This happens for three reasons: (1) metadata updates only account a tiny amount of all the requests. (2) the on-disk journal is located close to the first requests gathering portion of this trace. Therefore, transaction committing mostly does not introduce extra seeking overhead. (3) Transaction committing interval is 5s. Checkpointing traffic does not interfere with requests in the beginning and ending of the trace. The holding and delaying of metadata updating accelerates the demanding requests.

Finally, LFS performs extremely fast in *viewperf* by a coincidence. The requests of this trace is 99% composed of references to consecutive blocks within a few large files. Therefore, there is no chance of finding non-sequential access patterns. All other targets file systems perform similarly. The key point here is the write traffic which interferes the bulk reads between whiles. However, only for LFS can it gather all metadata and data updates and write to a new place. The coincidence in our evaluation is the available segments exactly follow the area where read requests gather. Consequently, all reads and writes aggregate in a small area in disk, which eliminates almost all the seek time as a ideal disk layout for this workload. This coincidence also demonstrates our efforts of finding non-sequential access patterns and rebuilding suitable disk layouts are worthwhile.

D. Random Access Workloads

The *tpc-r* trace [28] was collected by running TPC-R benchmark [33] on MySQL database system. Only 3% of references are issued to consecutive blocks - serving as a good condidate for random access workloads. Note that the workload is much heavier compared with the traces we used in sequential access workloads. We use the Seagate Cheetah15k5 disk model in disksim 4.0 only for this trace.

Figure 9(a) shows the performance improvement by active journaling is distinct. For example, the average response time

of 15,982,481 requests is decreased from 3.7882s to 2.5012s with page cache size 384MB in comparison with ext3. This demonstrates how random access workloads are inefficiently handled in conventional file systems, and how improvement can be made from learning and capturing block level non-sequential access patterns.

E. Concurrent Applications Workloads

Two traces [28] we used in this section were collected by concurrent executions of previous applications: (1) *multi1* (*cscope*, *gcc*), and (2) *multi2* (*cscope*, *gcc*, *viewperf*).

Figure 9(b) 9(c) presents our evaluation results on these traces. Our primitive learning algorithm works less effective on concurrent applications in comparison to previous evaluation results. A possible reason is our algorithm does not differentiate requests from different processes. Therefore, the captured patterns are less accurate.

V. Related Work

A. Journaling File Systems

Currently available journaling file systems passively use journaling. Section II describes how Linux ext3 file system works [9]. In the newest upgrade to ext4 file system, journaling has no major change [10]. Two minor changes include upgrading Journaling Block Layer (JDB) to JDB2 for handling 64bit block numbers, and adding metadata checksum to ensure better reliability with the proof of IRON file system paper [24].

ReiserFS [11][32] journaling differs from ext3 in minor ways. However, Reiser4 [11] largely optimizes journaling overhead by introducing *wandering log*. Because the whole file system on-disk structure is organized as a big B+ tree, we can journal the modified blocks first, and update the block pointers instead of checkpointing the blocks back to their original locations. The benefits of *wandering log* are twofold. First, the pointers are small and highly concentrated. Therefore, the update overhead is low. Second, the locations of logged blocks disperse which can result in multiple seeks for normal checkpointing. However, *wandering log* might causes disarrangement of spatial locality. Reiser4 uses repacker to handle this issue which is the same approach as the cleaner in LFS [18] and background defragmentation in WAFL [25][26]. Note that the foundation of applying *wandering log* lies in the special B+ tree on-disk structure of Reiser4. Compared with our *active journaling* design, *vmap* performs the same functionality. The difference is *wandering log* is used only for reducing the overhead, our design aims at turning the overhead as an performance and reliability gain.

Journaled File System(JFS) is the first journaling file system released by IBM in 1990. The notable difference between JFS and ext3 is JFS's journal record is on semantic level instead of block level [4][34]. Updates to different types of metadata (inode, directory map, allocation map, etc.) are recorded differently and handled with different routines. The effect of this fine-grained journaling is twofold. First, it reduces the size of on-disk journal because we only need to record what exactly changed. Meanwhile, the complexity of locking and transaction processing increases. Instead of using compound transaction, JFS adopts group commit for records on the same log page. The reliability guarantee of Journaling in JFS is similar to ext3 *ordered* mode.

NTFS [7][6] is the default file system used in most Windows Operating Systems. The journaling in NTFS is performed by the Log File Service (LFS) inside NTFS system file driver. Similar to JFS, NTFS adopt ordered journaling mode and semantic level record. Different from JFS, NTFS includes not just redo log but also undo information in each on-disk transaction record. NTFS also groups transaction commits by "batching of log records".

Journaling in other file systems like SGI's XFS [2][3] and Solaris OS UFS [8] vary in their specific design. But the features are included in our previous descriptions.

B. Out-of-Place Updating File Systems

LFS [18][19] buffers a sequence of file system changes and then write them to disk sequentially in a single disk write operation for the purpose of write speedup. This simple change profoundly impacts the design of modern file systems.

In WAFL [25], metadata live in files. File system snapshot can be easily made by duplicating the root inode. Write allocation can be done anywhere in the file system by simply change the metadata in files. Recent improvement in WAFL is the FlexVol Architecture [26]. An extra level of indirection or virtualization between logical and spatial storage space is introduced to support new features like volume mirroring and cloning.

ZFS [27] is a 128-bits file systems which support similar functionality as FlexVol with lower overhead by using storage pool address instead of the two-level block addressing in WAFL. ZFS ensures data integrity by the copy-on-write transactional model. ZFS can self-heal data in a mirrored or RAID configuration. This differs from our design by self-healing from replicas tracked in a single disk.

C. Multi-structured File Systems

In the pursuit of a synergy of the in-place and out-of-place updating file systems, hFS [31] implemented a complementary disk layout with separated log and data partitions. DualFS [12] proposed a similar disk layout based on the rationales that metadata are write intensive and mostly accessed non-sequentially. DualFS put only metadata in the log partition. By considering small files' weak spatial locality, hFS also place them into the log partition. Besides, hFS locates the log partition in the middle whereas DualFS organize the log and data partition side by side.

Because the contents inside the journal are indexed, our active journaling design can also be viewed as a multi-structured file system with a same purpose of synergizing the in-place and out-of-place updating. Differing from DualFS and hFS, our design has no separated log partition. Therefore, the performance penalty of separating the data and metadata of same files are eliminated.

The hybrid file system layout, HyLog [35], was proposed from the angle of differentiating hot and cold pages. Hot pages are written in place whereas cold pages out of place.

D. Utilization of Free Disk Space

FS2 [36] exploits free disk space for both improving system performance and saving energy consumption. Motivated from the angle on journaling, we share the view on using free disk space, and further exploit higher system reliability from duplication. Besides, our design has notable difference on file system layout and metadata management.

VI. Conclusion

Read world workloads are diverse. However, a large majority of modern file systems are organized based on spatial locality. Therefore, non-sequential access patterns are poorly coped with. We expect an adaptive file system applicable to the diversities of workloads. After deploying to a specific environment for a period of time, we presume the file system could perform better with the recognition of the workloads and access patterns in the environment. One approach to this objective is to learn from past workloads and adaptively reorganize the file system. With the help of journaling, the overhead involved in the process of learning and reorganization is diminished as "freeblock scheduling" [16][17]. By extending the role of journaling from a "logger" to a "learner" and a "constructor", we can turn the journaling twice-write overhead inside out as a gain on system performance and adaptivity. Even though the algorithms used in this paper are still primitive, our evaluation results exhibit great potential for this active journaling approach, which reduces average response time 14-39% for random access workloads and 0-14% for sequential access workloads based on the evaluation results from a wide range of workloads.

Beyond performance enhancement and system adaptivity, our active journaling design upgrades system reliability on a higher level: self-healing from silent data corruption.

Besides, we made a series of optimizations on active journaling file system layout in the pursuit of a synergy between in-place updating and out-of-place updating. A notable enhancement is our journal-structured block allocation design.

Furthmore, We refine the journaling transactional model from committing transactions to fixed circular locations to journaling "anywhere". We also exploit the benefit of block virtualization on *virtual checkpointing*.

In our future work, we plan to refine the learning algorithm, and implement a prototype of active journaling file system in Linux kernel.We hope our work could plant a seed for the revolution to next generation journaling file systems.

References

[1] A. G. Val Henson, Arjan van de Ven and Z. Brown, "Chunkfs: Using divide-and-conquer to improve file system reliability and repair," in *2006 Hot Topics in Dependability Workshop*. USENIX, 2006.

[2] "XFS Overview and Internals," XFS team, 2006. [Online]. Available: http://linux-xfs.sgi.com/projects/xfs/training/index.html

[3] B. Naujok, "XFS Filesystem Structure," SGI Whitepaper, 2006. [Online]. Available: http://oss.sgi.com/projects/xfs/publications.html

[4] S. Best, "JFS log: How the journaled file system performs logging," in *Proceedings of the 4th Annual Showcase & Conference (LINUX-00)*. Berkeley, CA: The USENIX Association, Oct. 10–14 2000, pp. 163–168.

[5] S. Best and D. Kleikamp, "JFS Layout," May 2000. [Online]. Available: http://jfs.sourceforge.net/project/pub/jfslayout.pdf

[6] M. E. Russinovich and D. A. Solomon, *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000* , 4th ed. Microsoft Press, 2005.

[7] "How NTFS Works," MIcrosoft TechNet, March 2003. [Online]. Available: http://technet2.microsoft.com/windowsserver/en/library/8cc5891d-bf8e-4164-862d-dac5418c59481033.mspx?mfr=true

[8] R. McDougall and J. Mauro, *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*, 2nd ed. Prentice Hall, July 10 2006.

[9] S. Tweedie, "Journaling the Linux ext2fs filesystem," in *LinuxExpo '98*, 1998. [Online]. Available: ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/journal-design.ps.gz

[10] M. Cao, S. Bhattacharya, and T. Ts'o, "Ext4: The Next Generation of Ext2/3 Filesystem," in *2007 Linux Storage & Filesystem Workshop*, 2007. [Online]. Available: http://www.usenix.org/event/lsf07/tech/cao_m.pdf

[11] R. Hans, "Reiser4 file system," namesys, Tech. Rep., 2004. [Online]. Available: http://en.wikipedia.org/wiki/Reiser4

[12] J. Piernas, T. Cortes, and J. M. Garcia, "The design of new journaling file systems: The dualfs case," *IEEE Transactions on Computers*, vol. 56, no. 2, pp. 267–281, 2007.

[13] A. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and analysis of large-scale network file system workloads," in *Proceedings of the 2008 USENIX Technical Conference*. USENIX, 2008.

[14] Z. Zhang and K. Ghose, "yFS: A journaling file system design for handling large data sets with reduced seeking," in *Proceedings of the FAST '03 Conference on File and Storage Technologies (FAST-03)*. USENIX, 2003. [Online]. Available: http://www.usenix.org/events/fast03/tech/zhang.html

[15] L. B. Soares, O. Y. Krieger, and D. D. Silva, "Meta-data snapshotting: A simple mechanism for file system consistency," in *SNAPI'03 (International Workshop on Storage Network Architecture and Parallel I/O)*, 2003.

[16] C. R. Lumb, J. Schindler, G. R. Ganger, D. Nagle, and E. Riedel, "Towards higher disk head utilization: Extracting "free" bandwidth from busy disk drives," in *OSDI*, 2000, pp. 87–102.

[17] C. R. Lumb, J. Schindler, and G. R. Ganger, "Freeblock scheduling outside of disk firmware," in *Proceedings of the FAST '02 Conference on File and Storage Technologies (FAST-02)*. Berkeley, CA: USENIX Association, Jan. 28–30 2002, pp. 275–288.

[18] M. Rosenblum and J. Ousterhout, "The design and implementation of a log-structured file system," in *sosp*, Oct. 1991, pp. 1–15.

[19] J. M. Neefe, D. S. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson, "Improving the performance of log-structured file systems with adaptive methods," in *SOSP*, 1997, pp. 238–251.

[20] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Transactions On Computer Systems*, vol. 2, no. 3, pp. 181–197, 1984.

[21] M. Prvulovic, D. Marinov, Z. Dimitrijevic, and V. Milutinovic, "Split temporal/spatial cache: A survey and reevaluation of performance," *Computer Architecture Technical Committee Newsletter (TCCA)*, May 07 1999. [Online]. Available: http://citeseer.ist.psu.edu/214830.html; http://galeb.etf.bg.ac.yu/ vm/ieee90/paper1-sts-tcca.ps.gz

[22] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "An Analysis of Data Corruption in the Storage Stack," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, February 2008.

[23] A. Riska and E. Riedel, "Idle Read After WriteIRAW," in *USENIX Annual Technical Conference (USENIX'08)*, July 2008.

[24] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "IRON File Systems," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, United Kingdom, October 2005, pp. 206–220.

[25] D. Hitz, J. Lau, and M. A. Malcolm, "File system design for an NFS file server appliance," in *USENIX Winter*, 1994, pp. 235–246.

[26] J. K. Edwards, D. Ellard, R. F. Craig Everhart, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, K. A. Smith, and E. Zayas,

"FlexVol: Flexible, Efficient File Volume Virtualization in WAFL," in *USENIX Annual Technical Conference (USENIX'08)*, July 2008.

[27] F. Zlotnick, "ZFS: The last word in file systems," in *The Conference on High Speed Computing*. Salishan Lodge, Gleneden Beach, Oregon: LANL/LLNL/SNL, Apr. 2006, p. 25, lA-UR-06-?

[28] A. R. Butt, C. Gniady, and Y. C. Hu, "The performance impact of kernel prefetching on buffer cache replacement algorithms," *IEEE Trans. Computers*, vol. 56, no. 7, pp. 889–908, 2007. [Online]. Available: http://dx.doi.org/10.1109/TC.2007.1029

[29] "Boost C++ Libraries," BOOST.ORG. [Online]. Available: http://www.boost.org/

[30] G. Ganger, J. Bucy, J. Schindler, and S. Schlosser, "The DiskSim Simulation Environment (v4.0)," June 2008. [Online]. Available: http://www.pdl.cmu.edu/DiskSim/

[31] Z. Zhang and K. Ghose, "hFS: a hybrid file system prototype for improving small file and metadata performance," in *EuroSys*. ACM, 2007, pp. 175–187. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273016

[32] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems," in *USENIX Annual Technical Conference, General Track*. USENIX, 2005, pp. 105–120. [Online]. Available: http://www.usenix.org/events/usenix05/tech/general/prabhakaran.html

[33] "Transaction Processing Performance Council," tpc.org, 2005. [Online]. Available: http://www.tpc.org/

[34] S. Best, "JFS Overview," Jan 2000. [Online]. Available: http://www-128.ibm.com/developerworks/library/l-jfs.html

[35] W. Wang, Y. Zhao, and R. Bunt, "Hylog: A high performance approach to managing disk layout," in *FAST*. USENIX, 2004, pp. 145–158. [Online]. Available: http://www.usenix.org/events/fast04/tech/wang.html

[36] H. Huang, W. Hung, and K. G. Shin, "FS2: dynamic data replication in free disk space for improving disk performance and energy consumption," in *SOSP*, A. Herbert and K. P. Birman, Eds. ACM, 2005, pp. 263–276. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095836