USENIX Association

Proceedings of FAST '03: 2nd USENIX Conference on File and Storage Technologies

San Francisco, CA, USA
March 31–April 2, 2003

yFS: A Journaling File System Design for Handling Large Data Sets with Reduced Seeking*

Zhihui Zhang and Kanad Ghose
Department of Computer Science,
State University of New York, Binghamton, NY 13902-6000
e-mail: {zzhang, ghose}@cs.binghamton.edu

Abstract

In recent years, disk seek times have not improved commensurately with CPU performance, memory system performance, and disk transfer rates. Furthermore, many modern applications are making increasing use of large files. Traditional file system designs are limited in how they address these two trends. We present the design of a file system called yFS that consciously reduces disk seeking and handles large files efficiently. yFS does this by using extent-based allocations in conjunction with three different disk inode formats geared towards small, medium, and large files. Directory traversals are made efficient by using the B*-tree structure. yFS also uses lightweight asynchronous journaling to handle metadata changes. We have implemented yFS on FreeBSD and evaluated it using a variety of benchmarks. Our experimental evaluations show that yFS performs considerably better than the Fast File System (FFS) with Soft Updates on FreeBSD. The performance gains are in the range from 20% to 82%.

1 Introduction

Recent years have seen impressive growth in CPU speeds and hard drive technology. In particular, data transfer rates of hard drives have improved quite dramatically with an increase in the performance of the data transfer paths and the rotational speeds [1]. However, processing capabilities have improved even more than disk access rates. At the same time, the demands for handling intensive I/O and large data sets have also grown noticeably [2]. These trends indicate that the I/O bottleneck will continue to be an issue in the foreseeable future.

This paper describes our effort in creating a new file system, named yFS, to alleviate this I/O bottleneck. yFS is a new file system design that brings together many of the best-known techniques for improving file system performance. By starting from scratch, we have been able to integrate proven techniques and new ideas with greater freedom and flexibility compared to either porting or patching an existing file system.

We have chosen FreeBSD as the host operating system of yFS because FreeBSD is a mature and well-pedigreed open source operating system [3]. Specifically, it has a solid merged buffer cache design and a full-fledged VFS/vnode architecture. Both features have directly influenced the design of yFS.

The rest of this paper is organized as follows. Section 2 discusses our motivation for this research. Section 3 describes the organization of yFS. Section 4 outlines the journaling model of yFS. Section 5 describes the implementation of yFS. Section 6 presents our experimental evaluation results. Section 7 discusses related work. We conclude this paper in Section 8.

2 Motivation

File system designs have always been driven by changes in two major arenas: hardware and workloads. Traditional file systems such as the Fast File System (FFS) [4] and similar file systems (e.g., Ext2 file system [5]) were designed with different assumptions about the underlying hardware and the workloads to which the file systems would be subjected. Although still used widely, there are a number of known techniques (e.g., journaling, B*-tree indices, extent-based allocation) that are not used in these systems and have been shown to perform better [5,6,7,8]. Some limitations in these file system designs are discussed below.

In classical file system designs, one block is usually allocated at a time even for a multi-block I/O request. Because traditional designs do not maintain sufficient information about the availability of extents, the preferred block is the one contiguous to a previously allocated block, even if there may be a larger extent available elsewhere. FFS tries to alleviate this problem by using a separate cluster map and a reallocation step [9], but it introduces the complexity of maintaining two bitmaps and each reallocation step must involve all the buffers in the cluster.

File systems that provide only a single allocation unit size force one to make a trade-off between performance and fragmentation. FFS divides a single file system block into one or more *fragments*, so that it can avoid undue internal fragmentation on small files and boost I/O throughput for large ones. However, FFS requires that a block consist of at most eight fragments and start at fixed fragment

*This work is supported in part by the NSF through award No. EIA 0011099.

addresses. This can lead to poor file data layout and wasted disk space. As a pathological example, if most of the files are a little more than 4096 bytes in an 8192/1024 file system, then almost half of the disk space will be wasted.

Traditional file systems translate a logical block number to its corresponding physical block numbers using file-specific metadata in the form of a highly skewed tree [4]. For large files, several indirections are needed before a data block can be accessed; this is because only a few direct pointers are stored inside the disk inode. Furthermore, although one could represent a series of contiguously allocated blocks efficiently with a single disk address, conventional block-based systems do not do so. The VIVA file system [10,11] reduces these indirections by compressing disk block addresses with partial bitmaps, but fails to address the problem of holes efficiently.

Metadata integrity is crucial in a file system. Historically, metadata was updated synchronously in a pre-determined order to ease the job of *fsck* [12]. This not only severely limits the performance of metadata updates, but also entails scanning of the entire file system for crash recovery. Log-structured File Systems (LFSs) solve both problems with the technique of logging [13,14]. LFS treats the disk as a segmented append-only log and writes both the file data and the metadata into it. Crash recovery can be performed efficiently by taking the most recent checkpoint and proceeding to the tail of the log. However, LFS introduces cleaning overhead, since the size of the file system is of finite size and the log must eventually wrap. Metadata journaling is a second technique that logs only changes to the metadata [6,7,8]. It speeds up metadata updates and crash recovery without incurring any of the cleaning cost of LFSs. However, metadata journaling introduces extra logging I/O because the metadata has to be written twice—first in the log area and then in place. Soft Updates is a third technique used to tackle the metadata update problem [15,16,17]. It maintains metadata dependency information at per-pointer granularity in the memory so that delayed writes can be used safely for metadata updates. To avoid dependency cycles, any still-dependent updates in a metadata block are rolled-back before I/O and rolled-forward after I/O. After a system crash, *fsck* is still needed to salvage any unused blocks and inodes.

Traditional file systems maintain a directory as an unsorted linear list of file name to inode number mappings called *directory entries*, which is painful to handle large directories. In addition, some old file systems lack the ability to create and deallocate new disk inodes on the fly.

To summarize, traditional file systems have various weaknesses in some or all of these subjects: disk space allocation, large directory handling, dynamic inode allocation, metadata update performance, and fast crash recovery. The goal of our work on yFS is to create a new file system that handles these issues more efficiently. Specifically, yFS has the following features:

- yFS uses extent-based allocation to maximize the chances of contiguous allocation.
- yFS makes use of B*-trees for representing large files and directories.
- yFS uses a variety of techniques to reduce the overhead of metadata logging to achieve fast crash recovery and boost metadata update performance.
- yFS allows dynamic allocation and deallocation of disk inodes.
- yFS implements fragments and inline data storage to improve the performance on small files.

Modern file systems like IBM JFS [6], SGI XFS [7,8], and FFS with Soft Updates [15,16,17] technology have also made improvements in solving design problems found in traditional file systems. In Section 7, we will compare yFS with these more modern file system designs.

3 The Organization of yFS

In this section, we discuss the major data structures of yFS and the disk space allocation strategy of yFS.

3.1 Allocation Groups

yFS divides the file system disk space into equal-sized units called allocation groups (AGs). The purpose of an AG is similar to that of cylinder groups in the Fast File System [4], that is, to cluster related data. In addition, we use AGs to increase concurrency by allowing different processes to work in different AGs. An AG consists of a superblock copy, AG group block, AG inode block, AG bitmap block, a set of preallocated inodes, and a large number of available data blocks.

yFS makes use of a bitmap to keep track of disk space usage within an AG. Like FFS, we use the idea of “sub-blocking” to trade off between space efficiency and I/O throughput. Each bit in the AG bitmap block represents one fragment, which is the smallest allocation unit. In addition, yFS also has a block size. A block is composed of one or more fragments and *can start at any fragment address*. In yFS, large files (with a size of more than 12 blocks) are always allocated in full blocks, while the last block of small files are allocated as many fragments as needed. As in FFS, fragment reallocation is needed if the last block of a small file cannot grow in place.

The AG group block and the AG inode block contain disk space and disk inode information respectively. To allocate disk space or a disk inode from a particular AG, we must first lock the buffer of its group block or its inode block. As explained in Section 4, every atomic operation performed in yFS is implemented as a transaction. Because a transaction does not release locks on metadata buffers before it commits, it should not allocate disk blocks or disk inodes from two AGs at the same time to avoid any potential deadlock. The separation of disk space and disk inode information into two different AG blocks (AG group block and AG inode block) is deliberate. It allows allocation of disk space in one AG and a disk inode in another AG within the same transaction.

3.2 Extent Summary Tree

To avoid a linear search on a bare-bones bitmap, we have adapted the IBM JFS algorithm which uses a binary buddy encoding of the bitmap and a free-extent summary tree built on top of the encoding [6]. The binary buddy representation of the bitmap and the summary tree are stored in the AG group block, separate from the AG bitmap block.

Initially, we divide the entire bitmap into 32-bit chunks called *bitmap groups*. Each group is then encoded using the binary buddy scheme (e.g., a value of 4 means that the maximum buddy group size within this bitmap group is 4 and it contains a free extent of length 2^4). The encoded value of a bitmap group can be stored in *one byte* because its maximum value is 5. After encoding bitmap groups, we merge buddies among groups if possible. During each merge, the left buddy's encoded value is incremented, while its right buddy's encoded value is *negated*. For example, if two buddies with an encoded value of 5 are merged, the left buddy will have an encoded value of 6 and the right buddy will have an encoded value of –5. If a bitmap group does not have any free fragments at all, it is assigned a special value of NOFREE (–128).

The binary buddy representation of the bitmap can be used to check the availability of free fragments without examining the underlying bitmap itself. For example, if we know the encoding value of a buddy group is 13, then we immediately know that all 2^{13} fragments starting from the first fragment of this group are free.

The extent summary tree is used to find out if and where a free extent of size 2^n is available. It is simply a 4-ary tree with each parent taking the maximum value of its four children. Each leaf in the tree takes on the binary buddy encoding of its corresponding bitmap group. This tree can be constructed bottom-up easily. Because the size of the bitmap is fixed (and hence the number of bitmap groups), the size of the tree is also fixed and can be represented using a flat array. Each node in the tree occupies one byte, capable of representing a maximum free extent of 2^{127} fragments. When we search the extent summary tree, we use the *absolute* values of its nodes to determine whether there is free space covered by a node unless, of course, the value is NOFREE.

The free-extent summary tree can locate the portion of the bitmap with enough free space quickly. However, the binary buddy encoding only records the availability of free extents of size 2^n, which must begin at a fragment address that is a multiple of 2^n. This could lead to sub-optimal allocations if we used the summary tree liberally. As a result, we consult the summary tree only when it is advantageous to do so (Section 3.3). An important note here is that we can always work on the bitmap directly and then adjust the encoding information accordingly. It is also important to know that we do not have to allocate an extent at the beginning of a buddy group. If the bitmap group from where we want to start an allocation has a negative encoding value, we must first reverse the effects of previous buddy merges (i.e., perform back split) until the bitmap group gets its free fragments back from its left buddy (or buddies).

3.3 Disk Resource Allocation

Disk resource (including disk inodes and disk blocks) allocation is the single most important issue in any file system design. yFS uses a two-step approach to perform this task. First, it chooses an AG. Second, it allocates resources from the chosen AG. To improve performance, yFS pursues two goals during disk resource allocation: *locality* and *contiguity*.

Many file systems use concepts similar to AGs to localize related data and spread data across the file system. For example, the Solaris file system achieves good temporal locality by forcing all allocations into one hot-spot allocation group [18]. The original FFS algorithm places a directory into a different cylinder group of its parent directory to ensure logical locality [4]. Both of these strategies have their weaknesses. Trying to cluster too much file data in the same AG exhausts its space quickly. This over-localization prevents future related files from being stored locally. On the other hand, switching to a new AG for each directory incurs disk seeks whenever we need to move along the directory hierarchy.

yFS adopts an algorithm similar to that used in FFS of FreeBSD 4.5 to choose an AG for non-directory files and directory files [19]. A non-directory file is preferred to be created in the same AG as its parent directory. A file system-wide parameter *maxblkpg* (maximum blocks per AG) is used to spread data blocks of a large file across AGs. Directories are laid out in a different way using the idea of "directory space reservation." Basically, a system administrator can provide two parameters when creating a file system: the average file size and the average number of files per directory. The disk space needed for each directory is reserved whenever possible (i.e., the free space in an AG should not drop well below the average amount of free space among all the AGs) using the above two parameters to avoid spreading out directories too aggressively. As a result, more directories can be created in the same AG with enough space for files to be created within them. We thus are able to take care of both temporal and logical locality in disk space allocation.

The behavior of the resource allocator can be modified by its various internal callers using bit flags. For example, the flag ALLOC_ANYWHERE is set when we want to allocate disk blocks for a file. This flag allows the allocator to search the entire file system for available disk blocks. However, if a caller wants to create more disk inodes in a particular AG, it will use the flag ALLOC_GROUP to limit the allocation within that AG.

After a usable AG is chosen, the preferred location within that AG (i.e., hint within the AG) can be determined if previous allocation information is available to indicate where contiguous allocation is possible. Allocation then proceeds as follows:

(1) If we have a hint within the AG, we attempt to use it before looking up the free-extent summary tree. This guarantees that files are allocated contiguously whenever possible. If there is a hole between two successive writes, the hint is adjusted to reserve space for the hole. This retains the possibility that when the hole is filled later, the entire file remains contiguous.

(2) Otherwise, we try the location of the first free fragment in the AG. If this fails, we then try the location following that for the last successful allocation, failing which we search the free-extent summary tree as the last resort.

yFS uses extent-based allocation to allocate more than one fragment per call whenever possible. The size of a newly allocated extent depends on the current I/O size as well as the availability of free space. It is thus possible for a large allocation request to be satisfied by several disjoint extents, possibly from different AGs. This is different from file systems that use a pre-determined extent size—VxFS is one example of such a file system [20]. As the file system fills up, the availability of large free extents decreases. Even so, yFS can locate free extents quickly with the aid of the free-extent summary tree. Note that the size of an extent is counted in fragments, not in blocks.

An extent allocated to a file is naturally described by an *extent descriptor* that is shown in Figure 1(a). Note that we may need more than one descriptor to describe one physical extent if the extent is not contiguous in terms of its logical block numbers. On the other hand, two extents can be merged if the hole between them is filled later by data from the same file.

3.4 Ubiquitous B*-tree

yFS uses B*-trees in three different contexts for: (a) disk inode management for each AG; (b) extent descriptor management for files with a large number of extents; (c) directory block descriptor management for directories with a large number of directory blocks.

The power of B*-trees lies in the fact that they are shallow and well-balanced. B*-trees give the capability of performing localized structural changes and guarantee at least 50% storage utilization [21]. These B*-tree features are indispensable in supporting any large and dynamic data set on a secondary storage device. The algorithms associated with B*-trees are complicated, so we use them only when necessary. The size of B*-tree nodes in yFS is the same as the full block size. However, the root node embedded in a disk inode is much smaller.

Each disk inode in yFS can be in one of the following three formats, as shown in Figures 1(b) through 1(d):

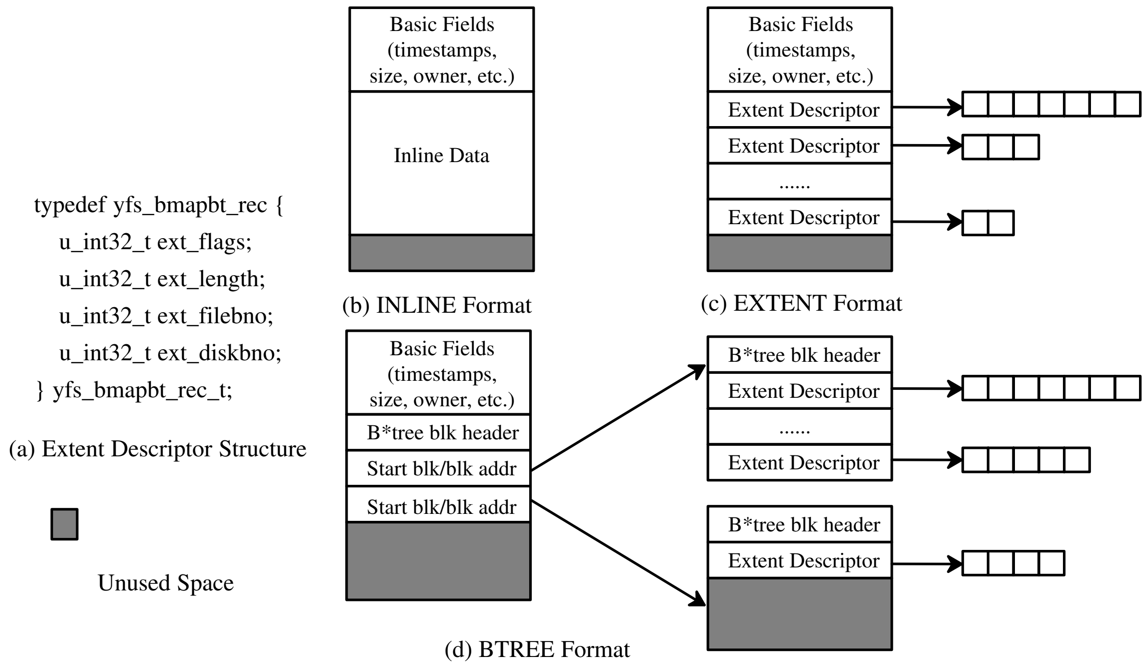
INLINE. For a small enough file, all its data can be stored inside its disk inode. This format reduces the number of seeks needed to access small files, because all metadata and data are stored in one place, namely, within the inode.

EXTENT. Here the extent descriptors of a regular file or directory block descriptors (Section 3.5) of a directory file are stored as an array that can fit into the disk inode. Note that even a large file can use this format if it has only a few non-contiguous extents.

BTREE. For a file with more descriptors than can fit into its disk inode, its descriptors are organized into a B*-tree. We always store the root block of its B*-tree in the inode, reducing the worse case search by one disk access as long as the file remains open.

The formats for directory inodes are similar except that we use directory block descriptors instead of extent descriptors. In addition to B*-trees used by files, each AG has its own disk inode B*-tree that is used to support allocation and deallocation of disk inodes on the fly within it. The root node of the disk inode B*-tree is always stored in the AG inode block. Since it is not embedded in a disk inode, its size is the block size.

In yFS, inodes are 512 bytes (one sector), which are four times larger than those of FFS. We have done this for three reasons: (1) It allows us to use sector addresses as inode numbers directly; as a result, we do not have to look up the disk inode B*-tree to find the disk address of a given inode. (2) We have better support for small files with inline format. In our prototype, at most 420 bytes of data can be stored inline; as a result, many small script files and small directories can be stored efficiently and accessed cheaply. (3) Since an inode block contains more than one disk inode, using a larger inode size reduces lock competition of the same disk inode block.



```
typedef yfs_bmapbt_rec {
    u_int32_t ext_flags;
    u_int32_t ext_length;
    u_int32_t ext_filebno;
    u_int32_t ext_diskbno;
} yfs_bmapbt_rec_t;
```

Figure 1. Regular File Disk Inode Formats

In yFS, although more I/O operations will be performed to bring in the same number of disk inodes, we feel that the above advantages outweigh this disadvantage.

3.5 Large Directory Support

The directory design of yFS aims to provide efficient support for both small and large directories. Unlike classical file system designs that implement a directory as an unsorted list of directory entries, all entries in yFS are ordered by their hash values computed from their corresponding file names using simple left rotation and XOR operations. If a directory is small, all its entries can be stored inside its disk inode (inline format). Otherwise, directory entries will be stored in *directory blocks*, which are described by *directory block descriptors*. The descriptor records three kinds of information for a directory block: address, smallest and the biggest hash values of all its entries, and space usage. All the directory block descriptors of a directory are stored in the disk inode if the number is small enough to fit into it (extent format) or organized into a B*-tree using smallest hash values as keys (Btree format). For the Btree format, all the B*-tree leaf blocks are linked together to support the *getdents()* system call easily.

yFS uses fixed 32-bit hash values as the keys for a directory B*-tree. Compared to using variable sized keys involved in techniques such as suffix compression [6,22], using fixed size keys greatly reduces complexity and enables binary search of keys within a B*-tree block. To cope with duplicate keys, any binary search result must be adjusted so that the first of a sequence of duplicate keys is always chosen at each level of a B*-tree. The rest of the directory code must also be able to deal with this situation.

Traditional file systems always use simple offsets as the directory cookies. In yFS, directory cookies are 64 bits, whose contents depend on the format of the directory inode. For a small inline directory, it is simply an offset to the next directory entry. For an extent format directory, the directory cookie consists of an index into the array of directory block descriptors stored in the disk inode and an offset into the directory block pointed to by the chosen directory descriptor. For a B*-tree format directory, the directory cookie consists of three components: the block address of a leaf B*-tree block, an index into the leaf block to select a directory block descriptor, and an offset into the directory block pointed to by the selected directory block descriptor.

The recently published design of Htree for Linux [23] also proposes the use of hash-keyed B*-trees. yFS differs from this effort in at least two aspects: (1) using a sorted linked list of entries to improve logging performance (Section 5), and (2) using physical addressing instead of logical addressing in the B*-tree to reduce one level of indirection.

4 Metadata Journaling

Like many other file systems [6,7,17], yFS uses metadata journaling to improve metadata update performance and provide fast recovery after a system failure. However, metadata journaling can rapidly become the system's performance bottleneck [17], so we take great care to make our logging system lightweight.

4.1 Transaction Considerations

A file system journaling design should be based on a good understanding of file system semantics and requires consideration of the following issues:

(1) Although a user activity (e.g., creating a file) is the ultimate source for triggering a transaction, it is up to the file system to determine when to start a transaction and what specific actions make up the transaction.

(2) A transaction in a file system is typically small. For example, a transaction used to change the owner of a file need only modify one inode block. For some large atomic operations, we can separate them into small transactions with the help of extra mechanisms (Section 4.4). As a result, all transactions run entirely within memory using the no-steal policy [22], which means that dirty buffers will not be reclaimed until the corresponding transaction commits in-core.

(3) Although we log only metadata changes, it is dangerous to sever the relationship between the file data and its associated metadata. As an example, if a metadata block is freed and then reused for file data prematurely, an untimely crash can corrupt the file system. Our solution to this problem (Section 4.4) neither restricts the reuse of freed metadata blocks [24] nor requires the use of any additional committed bitmap to check if a block is allocatable [5].

(4) We cannot afford to abort a dirty transaction for the sake of performance. In GFS [25], whenever a transaction modifies a buffer, a copy is made to preserve its old contents. If the transaction must be aborted, GFS simply restores all affected buffers by using their frozen copies. Such a scheme is expensive in terms of its memory footprint and copying overhead. As a result, we must make sure that once started, a transaction will succeed. We achieve this through the use of resource reservations.

(5) The whole purpose of using the transaction mechanism is to guarantee the integrity of the file system after a crash. In yFS, we do not attempt to provide a history of updates that allows us to roll back to some point in the past. This means that log space can be reclaimed as soon as the metadata it protects has been written in place.

The above considerations lead to a lightweight transaction model for yFS. The implementation of this model has a small impact on performance.

4.2 Transaction Model: Overview

yFS uses a variety of features to improve the efficiency of its metadata transactions. These features include: (1) fine-granularity logging, (2) dynamic incore log buffers, (3) asynchronous group commit, (4) resource reservations, (5) background daemons, and (6) a circular log area.

We first define a transaction as a group of metadata modifications that must be carried out atomically. A transaction moves the file system from one consistent state to another consistent state.

In yFS, all metadata changes are made by use of metadata buffers. Whenever a transaction wants to access a metadata buffer, a *log item* is created for the buffer if it does not already have one. A log item records if and where its associated metadata buffer is modified. Each transaction keeps track of the buffers it has locked indirectly by maintaining log item pointers.

When a transaction ends, all metadata changes made by it are first copied to incore log buffers in the form of log entries. yFS uses physical logging [22] because of its simplicity and idempotence. Each log entry is created from the information recorded in a log item and consists of: a metadata block number; start and end offsets within the block where the block was modified; new data for that region of the block; and a transaction ID. A special commit log entry is used to indicate if all log entries of a transaction have been written. After this, the transaction is said to have committed in core. Although we use the two-phase locking protocol to achieve transaction isolation, we never hold a buffer lock across an I/O operation, as in XFS [7,8]. After a transaction is committed in core, all its buffers will be unlocked immediately. However, a modified buffer is pinned in memory to enforce the Write-Ahead Logging (WAL) protocol [22]. A hot metadata buffer (e.g., a bitmap block) can be pinned by more than one transaction: a pin count is used in this case to indicate the number of non-committed transaction for this buffer. As a transaction commits on-disk, the associated buffer's pin count is decremented; a buffer is reclaimed only when its pin count is zero.

4.3 Transaction Model: Details

Figure 2 depicts the key data structures related to transaction handling in yFS. All log entries, usually from different transactions, stored in an incore log buffer constitute one *log record*. A log record is checksumed and timestamped. The size of a log record is unknown until the incore log buffer is full or flushed before it is full, and it is always rounded up to a multiple of the sector size. A 64-bit Log Sequence Number (LSN) is assigned to an incore log buffer (which contains one log record) when it is written for the first time. The LSN consists of two parts: a cycle number that is incremented each time the on-disk log wraps around and a sector address where the log record should be stored on disk. A transaction's LSN is the LSN of the incore log buffer that contains its commit log entry (in Figure 2, transaction #123 will have LSN of 5678).

All incore log buffers are organized into two queues: the idle queue and the active queue. Whenever a new incore log buffer is used, it is moved from the idle queue to the tail of the active queue. An incore log buffer is flushed by a special *log daemon*, triggered by either a log buffer becoming full, or a 30 second timer, or a synchronous operation. After an incore log buffer is flushed to the disk, transactions that have written a commit log entry into it are group committed onto the disk and the incore log buffer is moved back to the idle queue for reuse. If a flushed incore log buffer is not at the head of the active queue, these processing steps are deferred until its preceding incore log buffers are flushed and processed. This guarantees that transactions always commit on disk in the same order that they commit in core. To accommodate different rates of metadata updates, the number of incore log buffers can be adjusted dynamically.

A transaction's lifetime terminates when it is committed onto the disk. In other words, the changes made by the transaction are now made permanent. But before it dies, all the buffers dirtied by it are unpinned and their associated log items are tagged with the transaction's LSN indicating where the modified data is logged. However, if a log item already has a smaller LSN, its LSN should not be replaced with a bigger one because it must remember the first log sector that has active log data for its associated metadata buffer. Note that the pin count of a dirty metadata buffer is stored in its associated log item.

All log items that have associated "dirty but logged" metadata buffers are linked into a global *sync list* in ascending LSN order. The LSN of the first log item on the sync list indicates that the log space at and after the LSN still contains active log data. The only way to reclaim log space is to write the metadata buffers associated with the log items in place. This can be done by the log daemon if need be. After a metadata buffer is written in place, its log item can be removed from the sync list and freed.

Because we do not need to record the whole history of metadata updates, the on-disk log area can be used in a circular fashion. The log area can be either internal (in this case, we allocate log area from the middle AG to reduce disk head motion) or external. At the beginning of the log area there are two copies of *log anchors*. The log anchors are checksumed, timestamped, and updated alternately. They, in conjunction with the log area information stored in the superblock, record the current tail of the active log area.

To avoid starting a transaction needlessly, all sanity checks must be done beforehand. Furthermore, we must reserve three kinds of resources (disk space, locked buffer, and log space) to make sure that a transaction runs to completion safely. The disk space reservation is for the maximum number of disk blocks that will be requested by a transaction. The locked buffer reservation is for the maximum number of metadata buffers that will be locked by a transaction. This reservation guarantees that a transaction can continue to grab new buffers within its declared requirement. To prevent yFS from taking over all the buffers in the system, the total number of buffers locked and pinned by yFS cannot exceed a certain threshold at any given time (Section 5). The log space reservation avoids a deadlock situation that happens when the log space is low but the first log item on the sync list is locked by the committing transaction.

For example, the transaction CREATE_DINODE is used to create a chunk of disk inodes in a given AG. Let us suppose that a chunk of disk inodes contains 64 disk inodes, each inode is 512 bytes, the block size is 16 KB, and the maximum height of a B*-tree is 5. In this case, we need to reserve 7 blocks of disk space (5 blocks for each level of the B*-tree and two blocks for the inodes themselves). In addition, the transaction will modify the AG bitmap block and the AG inode block. The maximum number of buffers that can be locked by this transaction is 9 (7, as indicated above plus two more blocks—one for the

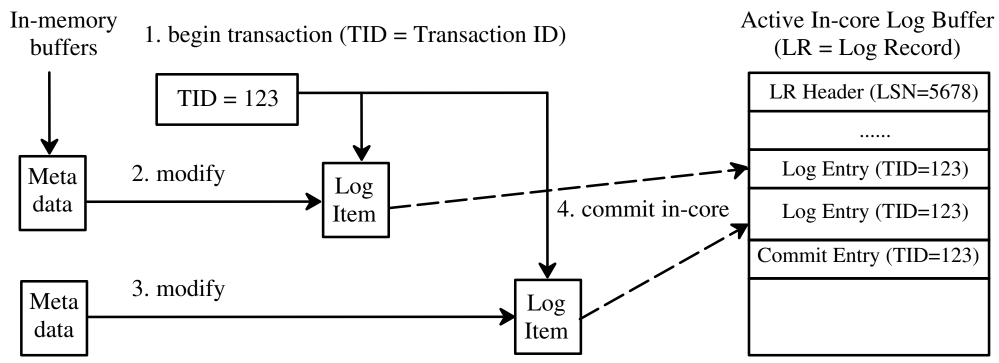


Figure 2. Data Structures Involved in a Transaction

bitmap block and another for inode block containing the root of the B*-tree and other status information). The log space needed per block is estimated to be half the block size, which is usually more than enough due to our use of fine-granularity logging. So the log space to be reserved is 73,728 bytes plus a little overhead for log entries and log record headers. Because reservations are made against the maximum possible requirements, any unused resource must be returned when the transaction commits in core.

4.4 Beyond Transactions

The transaction model described above provides us with a foundation to move the file system from one consistent state to another safely. However, we still need some extra mechanisms to deal with three special cases described below.

Inode Recovery List. POSIX semantics require that a file whose link count drops to zero continues to exist until the last reference to it is removed. This feature is often used to create temporary files. However, it poses a serious problem to journaling code because we do not know when the file will be truncated. Therefore we cannot decrement its link count and delete the file in one atomic transaction. The solution is to put such an inode on an AG Inode Recovery List (IRL) when its link count is decremented to zero. When the reference count of that file becomes zero, we truncate the file and remove the disk inode from the IRL with one transaction. If there is a crash after a file's link count drops to zero but before it is truncated, the disk inode will be found on an IRL during recovery and the intended truncation can be performed.

Extent Log Item. Locking is an important requirement in a journaling file system. To avoid deadlock, a transaction can choose to allocate disk blocks from only one AG. However, we cannot have this kind of control during deallocation because blocks to be freed could have been allocated from more than one AG. Our solution is to use more than one transaction to accomplish the job. The first transaction frees extents from the corresponding file's block map and logs the fact that these extents are to be freed from the file system bitmap using a "marked free" extent log item. After this transaction has committed on the disk, a *bitmap daemon* will start transactions to free the marked extents from the bitmap and write corresponding "unmark free" extent log items. By doing this, we not only avoid deadlocking, but also prevent any freed metadata block from being reused prematurely. Note that if a freed block was used for metadata, the first transaction has to write a special log entry indicating that the metadata block was deleted. These special log entries will be used in crash recovery (Section 4.5).

Dependent Data Block List. This last mechanism is used to make sure that a newly created file contains useful data and no file data is lost due to reallocation, making use of the concept of "dependent data blocks." Every buffer that contains newly allocated disk space is added to the dependent data block list of the ongoing transaction. These buffers are later moved over to the dependent data block list of the incore log buffer that contains the commit log entry of the transaction. Before an incore log buffer is flushed, any data buffers on its dependent data block list must be written first. This helps in enforcing the three update dependencies proposed by McKusick and Ganger in [16].

4.5 Log Recovery

Log recovery is relatively easy because of the use of physical logging. First, we read the latest valid copy of the log anchor to find the tail of the log. Then we scan forward to find all the metadata blocks that were deleted. Note that whenever we delete a metadata block, we have to write its block number into the log. If this was not done, we could end up overwriting valid user data by replaying obsolete metadata updates. This analysis scan stops at the head of the log, which is defined as the valid log record with the highest LSN. During the second scan from tail to head, we redo any changes to the metadata blocks, ignoring any stale metadata blocks. If we encounter a "marked free" extent log item that does not have a corresponding "unmark free" extent log item, we free such extents in the corresponding bitmap.

The recovery of inodes is done by the kernel after the file system is mounted. At this point, the log area is completely empty and we can use the normal transaction mechanisms to truncate any inodes on an IRL.

5 Implementation

Our prototype has been implemented as a kernel loadable module using the stackable vnode interface on the FreeBSD 4.5-Release. Several user-level utilities have also been developed to create a file system, debug a file system, and collect run-time statistics. The total source code consists of about 30,000 lines of C code. For brevity, we discuss only the key implementation details.

Our first critical decision concerned the encapsulation of metadata updates into transactions. If we follow the transaction model closely, we have to lock all metadata—including vnodes—needed to perform a transaction within its context. However, after studying the VFS layer carefully, we found out that vnode locking and unlocking are normally done within the VFS layer. Therefore, we decided to leave vnode locks outside a transaction's context. All changes to an inode are copied to its corresponding inode block when a transaction commits. This arrangement has two big benefits: (1) It keeps all transaction code within yFS while leaving the VFS layer intact; this is helpful in achieving our goal to be as unobtrusive as possible. (2) Because vnodes are managed by the VFS layer, leaving them out of the transaction context makes it possible for the VFS layer to reclaim vnodes freely.

Our second challenge was to find a way to represent metadata blocks. Because we use B*-trees and implement a directory directly on disk blocks instead of on the top of a regular file, we can no longer identify individual metadata blocks using negative block numbers as FFS does [4]. Therefore, all metadata blocks are attached to the device vnode corresponding to the disk device of the file system.

Since buffers of the device vnode use VM pages (i.e., page cache), metadata are cached longer than the lifetime of buffers that were used to access them. However, using the device vnode leads to the problem of identifying the metadata buffers that belong to each individual file. This information is needed for the *fsync()* operation. To solve this problem, we record in the inode the LSN of the transaction that made the latest changes to its file. To flush the metadata of a file, we only have to the make sure that the incore log buffer tagged with that LSN is flushed.

Our third challenge arose from the need to pin dirty metadata buffers until corresponding transactions have committed on disk. Fortunately, FreeBSD has a B_LOCKED queue that locks a buffer in the memory. All we have to do is to keep the buffer on this queue as long as its pin count is non-zero. In addition, a few fields need to be added to the buffer header to support transaction semantics. The maximum number of buffers that can be pinned down by yFS simultaneously is limited to be one half of all the available buffers in the system by default.

Our last challenge was keeping logging overhead low. Besides using fine-granularity logging, we take some additional measures to reduce the logging overhead:

(1) Log changes are made when absolutely necessary. For example, the bitmap encoding information and the summary tree do not have to be logged because these derived data can be reconstructed easily from the underlying bitmaps.

(2) Optimized data structures are used to reduce the logging I/O. An example of this is the maintenance of the directory entries in a directory block as a linked list ordered by hash values. As a result of this, the creation of an entry requires only the new entry itself and the modified pointer(s) to be logged. The deletion of an old entry requires only one modified pointer to be logged to remove the entry from the list.

Reducing logging information saves memory copies and I/O involved in logging. It also allows dirty metadata blocks to remain cached longer without being flushed because log space is consumed more slowly.

6 Experimental Evaluation

In this section we evaluate the performance of yFS against FFS enhanced with Soft Updates (FFS-SU). Our experimental comparisons are restricted to FreeBSD file systems for the following reasons. First, the effort needed to port non-FreeBSD file systems to FreeBSD for the sake of comparison against yFS is well beyond the scope of this work. Second, the comparison would not be fair because file system implementations are intimately tied to the virtual memory, scheduling, and I/O subsystems. Consequently, it would be difficult to identify what causes the performance differences between yFS and these ported file systems. Third, although the original FFS was designed almost two decades ago, it has benefited from two major improvements: clustered I/O [14] and Soft Updates [15,16,17]. The FreeBSD implementation of FFS further optimizes FFS with a variety of techniques [19]. In a nutshell, FFS-SU is a formidable file system against which to compare yFS.

The platform used in our measurements has an Intel Xeon 500 MHz. CPU, 128 Mbytes memory, an Adaptec AIC-7890 SCSI Adapter, and two 9.1 GB 7,200-RPM Seagate ST39140 SCSI disks. The first disk is used as the operating system disk. The second disk is used as the file system disk, where we created our test file systems.

All file systems are formatted with a block size of 16 KB and a fragment size of 2 KB (both are defaults under FreeBSD 4.5). The average file size parameter is 16 KB and the average number of files per directory parameter is 64. The allocation group (called cylinder group in FFS) size is 178 Mbytes for all file systems. For yFS, the log area is 32MB, which can be either inline (yFS-inline) or on a separate device (yFS-external). Because we have only two disks, the external log is created *on the operating system disk* in the case of yFS-external.

The four benchmarks used in this section are kernel build, PostMark, archive extraction, and file system aging. We start each benchmark run with a cold cache. All results are averaged over at least five runs; the standard deviation is usually less than 3% of the average, with a value as high as 6% in the aging benchmark.

We have also instrumented the kernel device driver to collect two kinds of low-level I/O statistics *on the file system disk only*: total number of I/O requests and average I/O times in milliseconds. The total number of I/Os reported (as X+Y) has two parts: the number of I/Os performed during the lifetime of a benchmark plus the number of I/Os performed after the benchmark terminates (e.g. FFS-SU issues additional I/Os for background deletion after PostMark finishes). The I/O times reported (as W+Z) also have two components: the service time (W, the time between the initiation of an I/O to the disk controller and the receipt of the corresponding interrupt) and the driver-level queuing time (Z). Note that the I/O times reported are only for the lifetime of each benchmark.

We ran all the benchmarks on FFS without Soft Updates as well. However, this version of FFS supports different semantics than both yFS and FFS-SU as all metadata operations are performed synchronously, instead of being performed asynchronously. These synchronous I/Os completely dominate the performance and make the FFS numbers uninteresting as points of comparison.

	Copy Phase Time (s)	Compile Phase Time (s)	Total I/O Requests	Avg I/O Times (ms)
yFS-inline	32	704	10505+131	5+58
yFS-external	32	703	10177+115	4+53
FFS-SU	40	701	14614+181	7+209

Table 1 Kernel build benchmark user-level and low-level I/O results

6.1 Kernel Build Benchmark

This benchmark uses a small shell script to copy kernel files from the operating system disk to the test file system created on the file system disk. It then builds the FreeBSD generic kernel and all kernel modules under the test file system. The copy phase of this benchmark is I/O intensive. During this phase, the benchmark copies files from */usr/src/sys* on the operating system disk to the test file system. The compile phase is CPU bound.

As shown in Table 1, the two yFS configurations perform 20% better than FFS-SU in the copy phase (32 seconds vs. 40 seconds). In FFS-SU, successively created small files may not be allocated contiguously to each other because a partial block cannot span across a block boundary. In addition, the last partial block of a file may not be allocated close to its preceding block [28]. In contrast, yFS is able to allocate small files adjacent to each other because it does not enforce artificial block boundaries. Since yFS does not divide disk space into blocks statically, it cannot maintain a free block count. Therefore, the directory layout algorithm described in Section 3.3 uses a free fragment count as the metric of free space. This tends to result in more locality in accessing, as compared to FFS-SU, where free space provided by fragments is ignored by the algorithm. These features certainly help yFS in other benchmarks as well.

Both yFS configurations perform quite comparably to FFS-SU in the compile phase even though yFS uses more complicated algorithms than FFS-SU. FFS-SU generates more I/Os than yFS. But this does not hurt the apparent performance of FFS-SU due to the overlap between I/O and CPU processing.

6.2 PostMark Benchmark

PostMark is a popular benchmark that simulates the working environment of a Web/News server [26]. It creates an initial pool of specified number of files. Then it performs a mix of creation, deletion, read, and append operations. Finally, all files are deleted. We use default configurations of PostMark v1.5 (the size range is between 500 bytes and 9.77 KB, both read and write block sizes are 512 bytes, the read/append and create/delete ratios are 5) except the base number of files. The results are shown in Tables 2(a) and 2(b). There are two interesting points to make at this point.

First, unlike other benchmarks used in this paper, FFS-SU performs fewer I/Os than yFS (the numbers for yFS-external do not include logging I/O). This is due to the following two reasons: (1) yFS needs four times as many I/Os as FFS-SU to write disk inodes because its inodes are four times larger than those of FFS-SU. (2) FreeBSD has a fast path deletion optimization [3], which deletes newly created files immediately if the inodes have not yet been written. yFS has to faithfully write log information for every metadata update, even if the update turns out to be cancelled before it reaches disk. Note that the average service time per I/O for FFS-SU is longer than that of yFS.

Second, the deletion rate (number of files deleted per

	50,000 Files		100,000 Files		150,000 Files	
	Total Time (s)	Deletion Rate	Total Time (s)	Deletion Rate	Total Time (s)	Deletion Rate
yFS-inline	102	1666	215	1552	320	1530
yFS-external	87	2772	177	2795	267	2560
FFS-SU	130	6962	281	5888	462	5357

Table 2(a) PostMark (v1.5) benchmark user-level results

	50,000 Files		100,000 Files		150,000 Files	
	Total I/O Requests	Avg I/O Times (ms)	Total I/O Requests	Avg I/O Time (ms)	Total I/O Requests	Avg I/O Times (ms)
yFS-inline	58084+373	2+39	117247+407	2+34	176159+358	2+33
yFS-external	56874+402	1+28	114698+462	1+24	172104+482	1+23
FFS-SU	57632+404	4+107	115603+575	4+100	173222+397	4+98

Table 2(b) PostMark (v1.5) benchmark low-level I/O results seconds) of FFS-SU is much higher than those of the two yFS configurations. This can be partly attributed to the fast path deletion optimization and the background deletion employed by FFS-SU. For example, the *df –i* command shows 22,471 inodes still in use in the case of 100,000 base files for FFS-SU after the benchmark finishes.

PostMark is the most metadata intensive test of the four benchmarks used in this paper. Let us consider the statistics for FFU-SU and yFS-internal for the 150,000 base file case. For Soft Updates, the benchmark process has to push the work item list 1,209 times itself to help the syncer daemon and sleep 7 times to slow it down. In the meantime, 13,218 metadata buffers are re-dirtied due to rollbacks. For yFS-internal, the small 32 MB log area wraps around 10 times. The benchmark process waits 108 times for incore log buffers and 10 times for log space. In comparison, yFS-inline fares better than FFS-SU by 31% (320 seconds vs. 462 seconds). Because other features do not help yFS here (e.g., yFS cannot inline files larger than 420 bytes), this benchmark clearly shows the efficiency of our journaling scheme.

PostMark writes all test files in the same directory (i.e., the root) by default, creating a large directory. The non-linear directory organization used by yFS can efficiently deal with this. FFS-SU also copes with the large directory by using its directory hashing scheme [19].

6.3 Archive Extraction Benchmark

Archive extraction is a common pre-requisite of installing a software package on a Unix-like system. This benchmark extracts the file *ports.tgz* (15,165,655 bytes, consisting of 6,470 FreeBSD 4.5 ports) with the *tar* command and then deletes these files with the *rm* command. We un-mount and re-mount the file system between the two operations to remove any impact of caching. The results of this benchmark are shown in Table 3. Both yFS configurations win in this benchmark. For example, yFS-external beats FFS-SU by 82% (89 seconds vs. 502 seconds) in the creation phase and 75% (37 seconds vs. 148 seconds) in the deletion phase.

This benchmark creates a total of 55,189 files. Out of 10,448 directory files (including the root), 10,257 are stored in the inline format in yFS. Out of 44,742 regular files, 24,368 are stored in the inline format in yFS. For all of these inline files in yFS, FFS-SU has to allocate disk space for them separately from their disk inodes. This is a major reason for the noticeable performance gap between yFS and FFS-SU. In addition, FFS-SU re-dirties 35,393 buffers due to its rollback operations, again handicapping FFS-SU against yFS. Note that deleting an inline file does not need to update disk space bitmap.

In the deletion phase, yFS even beats FFS-SU featuring background deletion. In FFS-SU, freeing a disk inode incurs an extra I/O to zero its mode field to help *fsck* identify unused inodes [12]. yFS does not do this because of its strong atomic guarantees. Furthermore, for all 55,189 files, yFS requires 1,725 I/Os to read their big 512-byte disk inodes compared to 432 I/Os needed by FFS-SU to read the smaller 128-byte inodes. The benefit is that yFS saves 10,257 I/Os that are required by FFS-SU to read the small directories.

6.4 File System Aging Benchmark

File system aging has been used to demonstrate the effectiveness of several layout optimizations in FFS [9,28]. While it is important to understand the long-term effects of the yFS allocation policy, that work is beyond the scope of this paper. The results of this test are thus not indicative of the long-term behavior of yFS.

Our file system aging benchmark performs a mix of file system operations to fill an empty file system with directories and regular files. The inclusion of directories is essential because they play an important role in file system layout. Each operation must have a working directory. Initially, there is only one root directory so the working directory is the root. As more directories are created, a working directory is selected randomly among them with a uniform distribution. Each file operation could be either a creation or a deletion. The probability that the next operation is a creation decreases gradually as the file system fills up. If we choose to create, the probability of creating a directory is 1/N (N defaults to 64), assuming that there are N files per directory on the average. For the file sizes, 93% are determined by the Lognormal distribution (μ=9.357, σ=1.318) and the rest 7% are decided by the Pareto distribution (κ=133K, α=1.1) [27]. If we choose to delete, we first order the files in the directory alphabetically before picking a victim. This guarantees that we delete the same file even if the on-disk directory structures are different for different file systems. All file names are created randomly from a set of

	Creation Phase			Deletion Phase		
	Elapsed Time (seconds)	Total I/O Requests	Avg I/O Times (ms)	Elapsed Time (seconds)	Total I/O Requests	Avg I/O Times (ms)
yFS-inline	97	26472+558	4+75	43	4683+141	12+111
yFS-external	89	25761+647	4+59	37	3929+238	11+107
FFS-SU	502	96614+201	10+645	148	22613+303	10+185

Table 3 Archive extraction benchmark user-level and low-level I/O results

characters. The results of this benchmark are shown in Tables 4(a) and 4(b).

Both yFS configurations excel in this benchmark. Since this benchmark performs a variety of operations, the performance gains for yFS come from many of the reasons mentioned for the other tests. In particular, the rollback operations in FFS-SU take a toll on its performance. Note that the performance gap grows wider as the number of operations grows (yFS-external beats FFS-SU by 30%, 40%, and 49% respectively). This shows that the metadata handling capability of yFS scales up better than that of FFS-SU. When the memory holds too many dirty metadata buffers that are constrained (by WAL or pending dependencies), large sequential I/Os, as used in yFS, offer a better way than delayed individual writes (as used in FFS-SU) to clear the backlog.

An unexpected phenomenon about this benchmark is that sometimes yFS–inline outperforms yFS-external. A closer look at the directory layout algorithm reveals that the AG of a directory *directly* under the root is chosen *randomly*. Because the working directory for each operation is also chosen randomly in this benchmark, the distances between these top-level directories can play a prominent role in determining the overall performance.

6.5 Discussions of the Results

Generally, yFS-external has an advantage over yFS-internal because it removes logging I/O from the normal I/O path. However, if a benchmark is not metadata intensive (Section 6.1) or has some other quirks (Section 6.4, for example), using a separate logging disk does not guarantee an improvement in apparent performance. Nonetheless, yFS outperforms FFS-SU on all our benchmarks except the kernel compile phase.

One of the major reasons for the performance gains of yFS is the use of a lightweight journaling scheme. Each transactions writes less than 500 bytes of log data, incurring a small overhead.

yFS uses extent-based allocation to allocate disk space for files and B*-trees to represent the extents of a file. Because we do not use large files (the largest file used in our aging benchmark is 52,242,324 bytes) and we run all benchmarks on an empty file system, the benefits gained by these features are limited. Although extent-based allocation and B*-tree algorithm should make yFS handle large data sets efficiently as other file systems with similar algorithms have claimed [6,7], it is difficult to demonstrate this advantage without proper aging on an empty file system. That work is in progress and beyond the scope of this paper.

7 Related Work

The yFS disk space management algorithm is an adaptation of the JFS scheme [6]. However, the original JFS scheme uses one allocation tree for the entire file system and two bitmaps (working and permanent) to track disk usage. We do neither of these because of their potential problem in terms of concurrency and space overhead. JFS does not use the location information inherent within the summary tree to search the tree. In JFS, when a right buddy is merged with its left buddy, its value is set to be NOFREE, disregarding the fact that the portion of the bitmap covered by this node may actually contain a free extent. As a result, JFS searches the summary tree using only the first-fit algorithm.

XFS uses dual B*-trees to track free extents by block number and by extent size respectively [7,8]. Each allocation and deallocation of disk space must update both trees. Since the two B*-trees grow and shrink on the fly, their blocks are not guaranteed to be stored contiguously. Although XFS only records free extent information, the amount of disk space used by the space management routine itself is not necessarily less than that for the bitmap solution when free disk space becomes fragmented.

Unlike JFS and XFS, yFS inherits the tradition of FFS to use fragments as the basic allocation unit. Large files (defaulting to more than 12 blocks) are allocated in full blocks. Although saving disk space is a consideration, the main motivation in yFS is to have the ability to cluster small files closer to gain better performance. Unlike FFS,

Elapsed Time	10,000 Operations (seconds)	20,000 Operations (seconds)	40,000 Operations (seconds)
yFS-inline	114	241	552
yFS-external	117	241	519
FFS-SU	168	400	1024

Table 4(a) File system aging benchmark user-level results

	10,000 Operations		20,000 Operations		40,000 Operations	
	Total I/O Requests	Average I/O Times (ms)	Total I/O Requests	Average I/O Times (ms)	Total I/O Requests	Average I/O Times (ms)
yFS-inline	20961+325	10+144	40967+454	11+150	82860+610	12+165
yFS-external	20521+216	11+148	40714+414	11+147	80034+673	12+155
FFS-SU	21817+387	15+240	47791+365	16+281	121179+452	16+335

Table 4(b) File system aging benchmark low-level I/O results we do not maintain a separate bitmap for cluster allocation because the buddy encoding and summary tree do a good job in speeding up the bitmap lookup. As a result, the number of fragments per block is not limited to eight, making the trade-off between disk space and throughput less painful.

One important difference between FFS and yFS is that a full or partial block can start at *any* fragment address in yFS. Therefore, yFS uses a free fragment count instead of a free block count to keep track of the amount of free space. In FFS, the free space provided by fragmented blocks is not considered by the directory layout algorithm, resulting in lower locality. Because there are no hard block boundaries in yFS, the external fragmentation between files is also reduced.

The use of extent maps has been suggested recently as one of the improvements planned for Ext2/3 by Ts'o and Tweedie [29]. The preferred form of this improvement avoids the use of B*-trees for storing extent maps and stores extent information inline within an inode or into a single block. Traditional indirect blocks are used when the inline and the single block storage is not enough. Fragmentation, which forces the system to revert to the old scheme, is avoided using preallocation. Performance considerations make preallocation almost mandatory in the suggested improvement. yFS implements extent maps in a different manner, allowing for three different inode formats, including one that stores *data* inline. yFS also implements a full-fledged B*-tree algorithm that is invoked only when necessary instead of altogether avoiding its use.

Our transaction model is similar to that of XFS [7,8] but with some notable differences. First, we reserve locked buffer resources before a transaction starts to avoid deadlocks. Although this is an issue that must be addressed, XFS does not appear to do so. To do this correctly, the file system must be able to detect the memory pressure. Second, each transaction in yFS only logs metadata modified by itself. XFS uses *accumulated logging* and a fixed 128 byte logging granularity allowing for the reclamation of log space without writing metadata in place. Our experiments show that the amount of logging I/O increases greatly because modifications made by previous transaction(s) to the same metadata are relogged *repeatedly.* Third, we use two queues to maintain incore log buffers instead of a closed circular queue in XFS. This makes it easy to dynamically adjust the number of incore log buffers to match the metadata update rates. Lastly, we do not lock inodes in a transaction's context. Any changes to a disk inode are copied to its corresponding inode block buffer when a transaction commits in core.

Seltzer et al. compare the journaling and Soft Updates techniques and conclude that they are largely comparable [17]. However, Soft Updates requires a non-trivial amount of memory to maintain metadata dependency at fine granularity. Although it does not enforce an order on buffer writes, it does introduce rollback and roll forward operations that increase memory and I/O overhead. For Soft Updates to work, it must have intimate knowledge of the inter-relationship between metadata that could be involved in a single operation. This dependency tracking could become tricky, if not entirely impossible, to work on complex data structures such as B*-trees, where the pointers can move within a metadata block. While Soft Updates has the ability to delay metadata updates, the memory tends to fill up. When that happens, the only way to get rid of the accumulated dependencies is to resort to synchronous writes. In case of a crash, a background *fsck* run is still needed to salvage any unused resources and both old and new names can show up due to an interrupted rename operation.

Instead of starting a transaction to accept metadata changes made by each atomic operation, Ext3 simply creates one compound transaction once in a while to receive all the changes made after a previous transaction closes [5]. This simple transaction design does not support fine-granularity logging. As a result, a modified metadata block is logged in its entirety no matter how minute the modification is. If a new transaction wants to modify a buffer that is being flushed by a committing transaction, a memory copy must be performed.

LFS is a write-oriented file system that logs both file data and metadata [13,14]. Its most glaring problem is its cleaning overhead. Although some techniques have been proposed to reduce cleaning overhead, their effectiveness depend on factors like workload, idle time, and access patterns [30,31,32]. Furthermore, LFS inherits some of the old data structures used by FFS such as the highly skewed addressing tree for each file. The maximum extent length is also limited by the size of a segment, which cannot be made large due to cleaning considerations. yFS does not have any of these problems.

8 Conclusions

This paper describes the design and implementation of yFS, a journaling file system for FreeBSD that represents a synthesis of existing and new ideas for improving file system performance. Our experimental results show that yFS works better than Soft Updates even without a dedicated logging device. For the benchmarks we have investigated, yFS's performance edge can be attributed to the use of lightweight logging, inline data storage, and the relaxation of the usual block boundary constraints.

9 Acknowledgements

We would like to thank our shepherd, Prof. Margo Seltzer, and the anonymous reviewers for their comments, which were useful in improving this paper.

10 References

[1] Edward Grochowski and Roger F. Hoyt. Future Trends in Hard Disk Drives. IEEE Transactions on Magnetics, pp. 1850-1854, Vol. 32, May 1996.

[2] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. USENIX Annual Technical Conference, pp. 41-54, June 2000.

[3] The FreeBSD open source project. http://www.freebsd.org.

[4] Marshall Kirk McKusick, et al. A Fast File System for UNIX. ACM Transaction on Computer Systems, Vol. 2, No. 3, pp. 181-197, August 1984.

[5] Stephen C. Tweedie. Journaling the Linux ext2fs Filesystem. LinuxExpo'98, May 1998.

[6] The IBM JFS open source project. http://oss.software.ibm.com/developerworks/opensource/jfs/.

[7] The SGI XFS open source project. http://oss.sgi.com/projects/xfs/.

[8] Adam Sweeney, et al. Scalability in the XFS File System. USENIX Annual Technical Conference, pp. 1-14, January 1996.

[9] Keith A. Smith and Margo I. Seltzer. A Comparison of FFS Disk Allocation Policies. In USENIX Annual Technical Conference, pp. 15-26, January 1996.

[10] Eric H. Herrin II and Raphael A. Finkel. The Viva File System. Technical Report No. 225-93. University of Kentucky, Lexington, 1993.

[11] Shankar Pasupathy. Implementing Viva on Linux. http://www.cs.wisc.edu/~shankar/Viva/viva.html, July 1996.

[12] Marshall Kirk McKusick. Fsck—The Unix File System Check Program. Computer Systems Research Group, UC Berkeley, 1985.

[13] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems, Vol. 10, No. 1, pp. 26-52, February 1992.

[14] Margo I. Seltzer, Keith Bostic, et al. An Implementation of a Log-Structured File System for UNIX. Proceedings of the 1993 Winter USENIX Conference, pp. 307-326, January 1993.

[15] Gregory R. Ganger, Yale N. Patt. Metadata Update Performance in File Systems. USENIX Symposium on Operating Systems Design and Implementation, pp. 49-60, November 1994.

[16] Marshall Kirk McKusick and Gregory R. Ganger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem, USENIX Annual Technical Conference, FREENIX Track, pp. 1-17, June 1999.

[17] Margo I. Seltzer, et al. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. pp. 71-84, USENIX Annual Technical Conference, June 2000.

[18] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast Consistency Checking for the Solaris File System. USENIX Annual Technical Conference, pp. 77-89, June 1998.

[19] Ian Dowse and David Malone. Recent Filesystem Optimisations on FreeBSD. USENIX Annual Technical Conference, FREENIX Track, pp. 245–258, June 2002.

[20] Veritas File System white papers. Available at: http://www.intel–sol.com/products/veritas/.

[21] Michael J. Folk, Bill Zoellick, and Greg Riccardi. File Structures: An Object-Oriented Approach with C++, 3nd Edition. Addison-Wesley, 1998.

[22] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, Inc., 1993.

[23] Daniel Phillips. A Directory Index for Ext2. http://people.nl.linux.org/~phillips/htree/paper/htree.html, September 2001.

[24] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System, 16th ACM Symposium on Operating Systems Principles, pp. 224-237, October 1997.

[25] Kenneth W. Preslan, et al. A 64-bit Shared Disk File System for Linux. Sixteenth IEEE Mass Storage Systems Symposium, March 1999.

[26] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR3022. Network Appliance Inc., October 1997.

[27] Paul Barford and Mark Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. Proceedings of the ACM SIGMETRICS, pp. 151-160, June 1998.

[28] Keith A. Smith and Margo I. Seltzer. File System Aging—Increasing the Relevance of File System Benchmarks. Proceedings of the ACM SIGMETRICS, pp. 203-213, June 1997.

[29] Theodore Ts'o and Stephen Tweedie. Planned Extensions to the Linux Ext2/Ext3 Filesystem. USENIX Annual Technical Conference, FREENIX track, pp. 235-244, June 2002.

[30] Trevor Blackwell, Jeffrey Harris, Margo I. Seltzer, Heuristic Cleaning Algorithms in Log-Structured File Systems, USENIX Technical Conference, pp. 277-288, January 1995.

[31] Jeanne Neefe Matthews, et al. Improving the Performance of Log-Structured File Systems with Adaptive Methods. Sixteenth ACM Symposium on Operating System Principles, pp. 238-251, October 1997.

[32] Jun Wang and Yiming Hu, WOLF—A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems. 1st Conference on File and Storage Technologies, pp. 47-60, January 2002.