hFS: A Hybrid File System Prototype for Improving Small File and Metadata Performance*

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

Two oft-cited file systems, the Fast File System (FFS) and the Log-Structured File System (LFS), adopt two sharply different update strategies—*update-in-place* and *update-out-of-place*. This paper introduces the design and implementation of a hybrid file system called hFS, which combines the strengths of FFS and LFS while avoiding their weaknesses. This is accomplished by distributing file system data into *two partitions* based on their size and type. In hFS, data blocks of large regular files are stored in a data partition arranged in a FFS-like fashion, while metadata and small files are stored in a separate log partition organized in the spirit of LFS but *without incurring any cleaning overhead*. This segregation makes it possible to use more appropriate layouts for different data than would otherwise be possible. In particular, hFS has the ability to perform *clustered I/O* on *all* kinds of data—including small files, metadata, and large files. We have implemented a prototype of hFS on FreeBSD and have compared its performance against three file systems, including FFS with Soft Updates, a port of NetBSD's LFS, and our lightweight journaling file system called yFS. Results on a number of benchmarks show that hFS has excellent small file and metadata performance. For example, hFS beats FFS with Soft Updates in the range from 53% to 63% in the PostMark benchmark.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management — Allocation/deallocation strategies; D.4.3 [**Operating Systems**]: File System Management — Access methods, Directory structures, File organization

General Terms

Algorithms, Design, Performance

Keywords

File systems, metadata journaling, disk inodes, update strageties

EuroSys'07, March 21–23, 2007, Lisboa, Portugal.

1. INTRODUCTION

The widening gap between the advancement of the CPU, the memory system, and the hard disk makes a file system the limiting factor in the overall system performance. The performance of a file system depends heavily on its ability to reduce disk head positioning time, which in turn relies on its data layout. Since a file system usually allocates disk space during a write, where it writes data directly affects its data layout. Depending on where a file system chooses to update its data, it belongs to one of two categories: update-in-place or update-out-of-place.

An update-in-place file system usually fixes data block locations after initial allocations. Updating an existing block requires a seek to its original location if the disk head is elsewhere. Unused blocks are freed explicitly to avoid the need for a garbage collector. Update-in-place file systems have been used widely; examples of such file systems include FFS [1], IBM's JFS [2], SGI's XFS [3], ReiserFS [4], and Ext3 [5].

In contrast, an update-out-of-place file system commits every update to a new location using the so-called no-overwrite policy. This obviates the need to seek back to its original position when writing an existing block but entails a garbage collector to clean its old disk space sometime later. Update-out-of-place file systems do not seem to have enjoyed much success, with LFS [6,7] being the most well-known example. Another example is a NFS filesystem supporting multiple snapshots/versions of files, permitting out-of-place updates to avoid the overhead of maintaining the snapshots through the use of a "write anywhere file layout (WAFL)" [8].

Both in-place and out-of-place update strategies have their strengths and weaknesses as discussed in the next section. The question is whether these two update strategies can be combined effectively in a single design to work better than using only one update strategy. This idea makes more sense in light of the observation that file system data are not accessed equally [9].

The design of hFS as presented in this paper shows that such a combination is indeed feasible. hFS incorporates attractive features of the two update strategies by using *two* partitions within *one* file

* This work is supported in part by the NSF through award No. EIA 0011099 and CNS 0454298.

** Currently at Panasas Inc.

system. The key is to use different update strategies for different data. However, since it is difficult, if not entirely impossible, for a file system to cope with all access patterns with equal efficiency, tradeoffs are inevitable.

The rest of this paper is organized as follows. Section 2 explores the differences between two update strategies using FFS and LFS as examples. Section 3 describes the on-disk structures used in hFS. Section 4 explains the segment write algorithm of hFS and its related issues. Section 5 discusses some implementation issues of hFS. Section 6 presents our benchmark results. Section 7 outlines related work and we offer our concluding remarks in Section 8.

2. UPDATE STRATEGIES

In this section, we review update-in-place and update-out-of-place strategies using FFS and LFS as their respective examples. Note that a file system that usually updates in-place can update out-of-place occasionally for the sake of performance or space efficiency. The reverse can also be true.

2.1 The Design of FFS

In an update-in-place file system, all updates to existing data must be propagated back to their originally assigned locations. In FFS, a new piece of data must be allocated a block or one or more fragments individually before its first write. After that, the allocated block or fragment(s) become the only legitimate location used to receive future updates of that data. Two exceptions exist: (1) A fragment may be relocated when it cannot grow in-place; (2) A full block can also be moved to participate in a cluster relocation during a write [10].

Disk space de-allocation is done explicitly. When a data block is deleted or relocated, its original space is marked as free in the corresponding bitmap. By allocating disk space in advance and deallocating disk space explicitly, FFS completely avoids the cost of garbage collection.

The update-in-place design dictates that every I/O must be preceded by a seek to the location of the data being accessed if the disk head is not already there. To reduce positioning time, FFS makes use of *logical locality* and *clustered I/O*. Logical locality places semantically related data and metadata in close proximity to each other to reduce the postioning length. Because locations of data are determined on their very first writes and not on their most recent writes, using temporal locality to cluster data is questionable at best. FFS uses the concept of Cylinder Groups (CGs) to achieve logical locality. It tries to gather related data in the same CG, while scattering un-related data across CGs. Clustered I/O makes each seek more productive by doing multi-block I/Os if more than one block within the same file is laid out contiguously. If a file's data and metadata blocks are initially allocated in an optimal manner in terms of contiguity and locality, any future random updates on the file will not disrupt its ideal layout. On the other hand, a poor layout cannot be improved cheaply even if more disk space has been freed. To ensure good performance, some file systems like VxFS allow users to pre-allocate large chunk of contiguous blocks for a specific file [11]. Even with these allocation strategies in place, FFS itself does not handle the following two scenarios efficiently.

First, FFS does not handle small files as efficiently as it does large files. To conserve disk space, FFS supports the idea of "sub-blocking" by using fragments to represent tails of small files. This entails a separate *fragment bitmap* to track the availability of each fragment in addition to a *cluster bitmap* where one bit represents one full block. Extension of the tail of a small file is complicated because it can either grow in-place or relocate. In addition, small files must be accessed individually and are a potential source of fragmentation. Ganger and Kaashoek proposed the use of embedded inodes and explicit grouping to exploit disk bandwidth for small files [12]. Their scheme introduces an extra *on-disk* data structure to map inode locators to their physical addresses. It enlarges a directory and complicates the handling of hard links.

The second inefficiency in FFS has to do with metadata updates. Due to the fact that metadata are potentially scattered all over the CGs, it is difficult to update several pieces of metadata atomically and efficiently. In case of a crash, there is no way to know where the last updates were going on to avoid the scanning of the entire file system. Fortunately, this problem can be resolved or alleviated by use of metadata journaling [2, 3, 4, 5] or the Soft Updates [13,14,15] technique. Both techniques are not without their limitations, as discussed in Section 7.

Note that the clustered I/O technique in FFS does not apply to metadata and files less than one block in size. For metadata and small files, the disk bandwidth is always under-utilized. Therefore, metadata and small files are more susceptible to a disk seek because its cost cannot be amortized by a subsequent large I/O.

2.2 The Design of LFS

In an update-out-of-place file system, an update to existing data can happen at any unused location without the need to seek back to their originally allocated locations. The freedom to relocate data comes with a price: their indexing metadata (i.e., block pointers) must be updated accordingly. LFS takes advantage of this freedom to boost write performance for all kinds of data. It organizes the file system disk space into a logical log by threading large segments together. A write is done in large segments and always happen at the logical end of the log. A segment is a contiguous extent of disk space and contains a summary followed by various dirty blocks. Because data locations are determined by their most recent writes, *temporal locality* is the only choice for LFS. Using logical locality would be awkward because logically related data are not necessarily modified at approximately the same time. Although the segment writes are not strictly sequential, the seek time spent between large segments is negligible. This is the reason why LFS can achieve log-like write performance.

The no-overwrite feature of logging also provides a natural safeguard to crash recovery because old data remain intact until after new data are written. In other words, a crash recovery mechanism is already inherent for LFS. Furthermore, crash recovery in LFS is fast

regardless of the file system size. This is due to the fact that updates of an LFS file system always happen at the end of its log. As a result, crash recovery needs only to locate the last checkpoint, whose location is stored in the superblock, and then roll forward until the last valid partial segment is found. In a nutshell, no additional mechanisms such as Soft Updates [13,14,15] or metadata journaling [2,3,4,5] are needed to achieve fast recovery.

The metadata update performance is also excellent in LFS because file data and all related metadata are gathered together to be written in the same segment batch [7]. This avoids the need to seek between the file data and their associated metadata to perform small writes for an atomic operation. However, because file data are always written to new locations, some metadata (e.g., an indirect block) may be updated simply to reflect an address change. This is not a serious concern because incremental I/O cost is small in a bulk data transfer. Furthermore, file data and their metadata can be stored close to each other.

Despite these great advantages, LFS has two serious shortcomings. First, the run-time cleaning cost can degrade the file system performance significantly. Because LFS does not de-allocate blocks explicitly, it relies on a garbage collection process called cleaning to provide a constant supply of clean segments. While cleaning empty segments is cheap, cleaning non-empty segments is costly because these segments must be first read from the disk in order to single out and relocate live blocks in them. Although some work based on simulations has been done in this direction [16,17,18], it remains to be seen that the cleaning overhead can be curbed in a real implementation. In fact, one simulation result shows that the cleaning cost can skyrocket in some situations [18].

The second problem of LFS is related to the read traffic. LFS makes use of temporal locality to lay out data, which means that data written at about the same time will be stored in the same segment. As a result, read performance can suffer if disk blocks are read in an order different from the one in which they were written. Originally, the authors of LFS assumed that read traffic could be absorbed effectively by a large cache [6]. However, a later study does not seem to support this assumption [18]. It is also doubtful that write traffic is more important than read traffic [19].

Using different layout strategies does not prevent FFS and LFS from sharing some common building blocks. For example, inode blocks are used in both FFS and LFS to co-locate potentially related inodes. However, an inode block in FFS always has the same inodes statically assigned to it, while an inode block in LFS contains whatever inodes that are dirty at the time it is written. An inode block in LFS can be sparse if there are not enough dirty inodes to fit into it. An inode block in FFS can be sparse if some inodes in it are not used. This once again illustrates that FFS and LFS exploit different kinds of localities.

2.3 The Alternative Approach

The above two case studies show that an update-in-place file system like FFS and an update-out-of-place file system like LFS have different strengths and weaknesses. Although there was a debate about whether FFS or LFS is a better design, we do not want to re-kindle that dispute. However, a full understanding of both file systems has indeed inspired us to pursue the alternative approach used in hFS, which updates both in-place and out-of-place. In particular, the appeal of doing large sequential writes on *any* kind of data without sacrificing large sequential reads is irresistible.

3. THE ON-DISK STRUCTURE OF hFS

One key idea behind hFS is to store and treat file system data differently based on their size and usage. In hFS, all metadata and regular files no more than one block are stored in its *log partition*, while data blocks of large regular files are stored in its *data partition.* This division of storage is shown in Figure 1. Notice that to reduce seeking between the log partition and the data partition, the former is embedded in the middle of the latter.

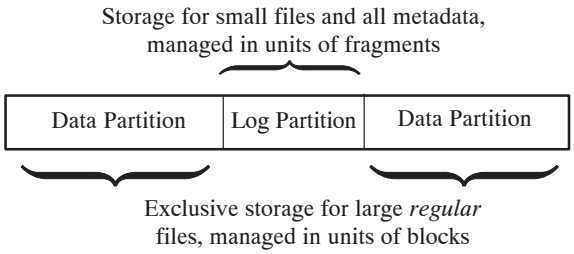

Figure 1. Two Logical Partitions of hFS

In this section, we describe the on-disk data structures used by these two partitions. For the sake of brevity, a small file is defined to be no more than one block in the following discussions. If the log partition reserved by a system administrator proves to be too small, hFS will create small files on the data partition with some loss of performance. Due to the abundance of disk space on modern drives, this should not be a limitation.

3.1 The Log Partition

All metadata and small files are stored in the log partition of hFS. The smallest allocation unit on the log partition is a *fragment*. A *block* consists of one or more fragments and is used to store metadata as well as file data. The log partition is organized in the spirit of LFS. In other words, it is treated as a log of large segments and dirty data are always written at the end of the log to optimize write performance. However, there are major differences between the design of hFS and LFS, which we will discuss when appropriate.

In hFS, system-wide metadata are stored in three different places. The superblock stores a few pieces of critical information needed to manage the file system. Such information can be either static (e.g., block size) or dynamic (e.g., free block count). The most recent values of dynamic fields in the superblock are stored in the segment summaries (see Section 4). The bulk of the system-wide metadata are stored in two special files: the inode file (IFILE) and the bitmap file (BFILE). The latter does NOT exist in LFS [7].

The IFILE Structure

The main function of the IFILE is to map an inode number to the disk address of the corresponding inode. Because hFS avoids cleaning by using the BFILE (see below), its IFILE no longer contains any cleaning and segment usage information as its counterpart in LFS does [7].

Each entry in the IFILE corresponds to a block of inodes. The IFILE can grow to add more inode address mappings when needed. The actual inode storage will be allocated when a write to a dirty inode occurs. The structure of the IFILE is depicted in Figure 2.

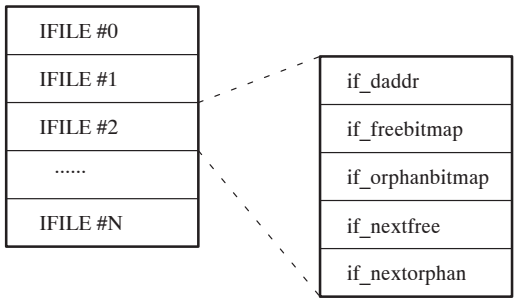

Figure 2. Inode File (IFILE) Structure

In addition to address mapping information stored in the *if_daddr* field, an IFILE entry also includes the following four fields: *if_freebitmap* indicates which inodes in the inode block are live, *if_orphanbitmap* indicates which inodes are orphaned in the inode block, *if_nextfree* points to the next inode block with free inode numbers, and finally *if_nextorphan* points to the next inode block with any orphaned inodes. An orphaned inode corresponds to a file that is still in use although its link count is already zero. Therefore, the IFILE maintains two lists within itself: a *free list* and an *orphan list*. The free list used for allocating and de-allocating free inode numbers, while the orphan list to used to reclaim orphaned inodes after a system crash (see Section 4.4). The heads of these two lists are maintained as two dynamic fields in the superblock.

Obviously, the structure of an IFILE entry is quite different from that of LFS [6,7]. Each IFILE entry in hFS maps an entire block of inodes instead of one inode to a disk address. This design reduces the size of the IFILE significantly, increasing the chance that an IFILE entry is cached when needed. Suppose one inode block contains 32 inodes, then the amount of mapping information itself maintained by the IFILE is reduced by a factor of 32. To accommodate this change, hFS follows the tradition of FFS in grouping inodes into inode blocks statically. For example, if an inode block contains 32 inodes, then inodes numbered from 0 through 31 will always be stored in the inode block 0. The *if_freebitmap* field in an IFILE entry is used to determine if the corresponding inode block has any live inodes. This tracking is required because unused space (include inode blocks) must be explicitly deallocated in hFS—no subsequent cleaning will be involved.

Each inode in hFS is 512 bytes, giving a total of 428 bytes for block pointers. hFS also supports an *inline* format that stores the data of a very small file directly into its disk inode, replacing the space used by block pointers. This improves the performance of these files because only one I/O is needed to access them. However, compared to 128-byte inodes used by both FFS [1] and LFS [7], hFS needs four times as many I/Os to read and write the same number of inodes.

The BFILE Structure

The BFILE consists mainly of bitmap information for both the log partition and the data partition. Its structure is depicted in Figure 3.

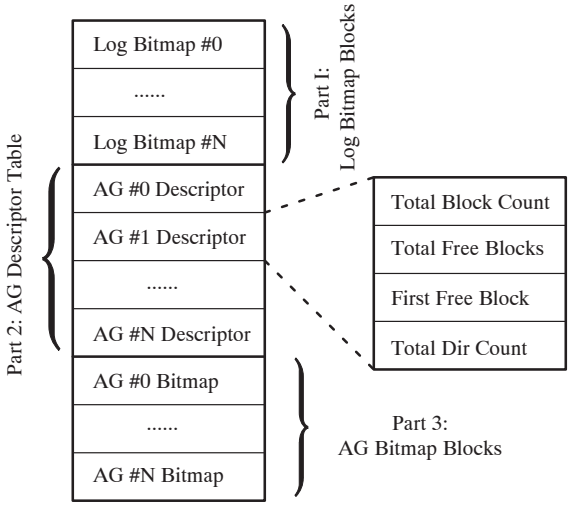

Figure 3. Bitmap File (BFILE) Structure

As shown in Figure 3, the BFILE has three parts. The first part is log bitmap blocks. In these bitmap blocks, one bit represents one fragment (default is 2,048 bytes). We will discuss the other two parts of the BFILE in Section 3.2.

The use of bitmap to track disk space usage is against the original LFS design [6] and *is done deliberately to eliminate the cleaning overhead involved in LFS*. To locate free extents within the log bitmap quickly, hFS first encodes individual bitmap words. The encoded value of a bitmap word is the length of the longest free extent that can start from anywhere in the bitmap word and can cross one or more bitmap word boundaries. On top of individual bitmap encodings, hFS then builds 4-ary *extent summary trees*, similar to those used in IBM's JFS [2]. In our prototype, an extent summary tree has 4,096 leaves and each leaf corresponds to one 64-bit bitmap word. If all the underlying bitmap words of a tree are entirely free, its leftmost leaf should have the maximum possible encoded value of 262,144 (i.e., 64*4096). Each parent node in the tree takes the maximum value of its four children. By examining the value of a tree's root node, hFS can quickly determine the largest extent available under the tree. The extent summary trees are created as needed in the memory. They are not on-disk data structures. In addition to system-wide metadata contained in the IFILE, the BFILE, and the superblock, the log partition also stores two more kinds of data: (1) indirect blocks of large regular files; (2) small files

no more than one block in size. A small file on the log partition occupies as many fragments as needed to save I/O bandwidth and disk space.

The biggest benefit from this storage arrangement is that hFS can now fully exploit the disk bandwidth for small files and metadata. Like LFS, crash resilience is inherent in hFS because metadata updates are done in a no-overwrite and consistent way. Since a small file on the log partition is no larger than one block, its performance will not suffer from any mismatch between read and write patterns. In addition, it is easy to migrate a small file from the log partition to the data partition if its size exceeds one block. Similarly, it is trivial to move a large file back to the log partition if it again shrinks within one block. However, this latter case is unlikely because partial file truncation is so uncommon [15]. File migration is transparent to file system users. Note that moving a large file would be more costly.

3.2 The Data Partition

The data partition of hFS (see Figure 1) is dedicated to storing large *regular* files (except their inodes and indirect blocks). On this partition, files are allocated in units of full blocks—no FFS-style sub-blocking is used because saving disk space is not our major concern here.

The use of a separate data partition has the following advantages:

- Read performance degradation due to random updates is avoided on a large file when its read pattern does not match its write pattern.
- Placement of large files can be optimized to use clustered I/O as FFS does. Note that in this case, a clustered I/O consists of blocks of the *same* file that have been allocated contiguously.
- Any possible fragmentation that would otherwise be incurred by small files and metadata on the data partition is eliminated.

Despite these advantages, hFS now needs to seek between its log partition and its data partition to access the metadata and file data of a large file. We hope this penalty will be outweighed by benefits of using two partitions, at least for workloads that hFS is designed for. Furthermore, using a separate disk for the log partition can reduce this penalty (see Section 6.4).

Currently, hFS uses the old indirect block addressing scheme [1] for the purpose of fast prototyping. This is inferior to the extent-based addressing scheme [2,3]. For a large file whose data are allocated contiguously, extent-based addressing scheme can effectively cut down the amount of metadata needed by such a file. As a result, a large file's metadata are more likely to be cached, reducing the seek between its data and metadata.

hFS divides its data partition into equal-sized chunks called *allocation groups* (AG) similar to CGs used in FFS [1]. Descriptors to each allocation group appear in the second part of the BFILE structure, as shown in Figure 3. An AG descriptor contains information similar in nature to those found in the cylinder group control blocks in FFS [1]. Although each AG descriptor contains only four pieces of information in our prototype, it can be expanded to include extra information needed by a more intelligent disk space allocator. Currently, hFS uses a straightforward block allocation strategy. Moving the AG control information into the BFILE removes the physical boundaries between AGs because the BFILE is stored on the log partition. As a result, the size of an extent is no longer limited by the size of an AG. This helps to provide unbroken contiguous space for very large files.

The disk space on the data partition is represented by the last part of the BFILE, which consists of AG bitmap blocks. Each bit in the AG bitmap block represents one block (default is 16,384 bytes). This is different from the log bitmap blocks, where one bit represents one fragment.

The algorithm used to cluster and distribute file data across the AGs in the data partition is similar to the *dirpref* algorithm used in recent versions of FreeBSD [20]. Basically, the algorithm tries to avoid scattering directories across AGs too eagerly as the original FFS does [1]. It achieves this by reserving space for a directory so that files under it are likely to be created in the same AG as its parent. The expected size of a directory can be calculated from the average file size and the average number of files per directory given by a system administrator. By tracking the amount of free space and the number of directories in an AG, hFS can compute the actual average directory size. The larger of the expected directory size and the actual directory size is used to decide whether to create a new directory in the current or a different AG.

Directory inodes, as other inodes in hFS, are physically stored in the log partition. To remember the preferred AG to store its files, each of them is augmented with an *anchor* field. Note that the disk space occupied by small files and indirect blocks are not counted towards the directory size because they are not stored in the AGs.

3.3 Large Directory Support

hFS uses four different formats to support small and large directories efficiently. The need for multiple directory formats has been evidenced by IBM's JFS [2] and SGI's XFS [3]. With the exception of the first inline format described below, all directory entries are stored in directory blocks ordered by non-decreasing hash values. To make deletion of an empty directory easier, the hash values of "." and ".." are hardcoded to be 1 and 2 respectively. If the hash value computed from a file name is no greater than 2, it is adjusted to be 3. The four directory formats are as follows:

- **Inline Format.** Entries of a small directory are stored inside its disk inode directly. This format is different from the *immediate file* scheme proposed by Mullender and Tanenbaum [21], which abandons co-location of potentially related inodes entirely.
- **Single Block Format.** Entries of a relatively small directory file are stored in one directory block. This single block is assigned the logical block number of 2.
- **Extent Format.** Entries of a directory file are stored in several directory blocks. A single leaf node with logical block number of 1 contains one directory block descriptor (see Figure 4) for each

of these directory blocks. The extent format is a degenerated form of the following B+tree format.

- **B+tree Format.** In this format, the block with logical block number of 0 contains the root node of the B+tree. Each leaf node of the B+tree contains directory block descriptors, which point to directory blocks.

```
typedef struct  hfs_dirbt_rec {
  u_int32_t   dir_minhash;
  u_int32_t   dir_maxhash;
  daddr_t     dir_blkaddr;
  u_int32_t   dir_freecnt;
}  hfs_dirbt_rec_t;
```

Figure 4. Structure of a Directory Descriptor

Each successive format listed above supports an increasing number of directory entries. hFS changes a directory's format automatically when necessary. In fact, notice how the logical block numbers 0-2 are reserved to make a format transition easier. The format being used depends on the number of entries as well as the total size of entries. In the B+tree format, B+tree nodes as well as directory blocks can merge and split as necessary.

Large directory support is especially important for hFS because the read and write pattern of a large directory blocks does not necessarily match each other [19]. As a result, a linear scan of directory would lead to very poor performance if the directory becomes large. Even with B+tree indexing, large directory performance of hFS does not necessarily have a edge over the in-core DIRHASH algorithm [20] when memory is ample.

In the current design, each new directory block or a directory B+tree node will take the next logical block number available. Such a design could make a directory sparse if directory entries in a block are all deleted before it can be refilled with new entries or merged with its neighbors. Such a scenario happens rarely and should not be a problem due to the large number of usable logical block numbers (i.e., 2^{31}).

4. SEGMENT WRITE ALGORITHM

In this section, we describe the algorithm used to write segments in the log partition and other issues related to segment writes. Each segment in hFS begins with one fragment worth of *segment summary* followed by a segment body consisting of inode blocks, data blocks, and metadata blocks. The structure of a segment is depicted in Figure 5. As shown in Figure 5, the segment summary starts with a segment header that indicates it is a segment summary of hFS. We will explain the remaining eight fields within this section.

4.1 Dynamic Segment Size

Disk fragments are allocated to a segment with the help of extent summary trees built on top of the bitmap word encodings (see Section 3.1). They are represented by *an array* of segment extent

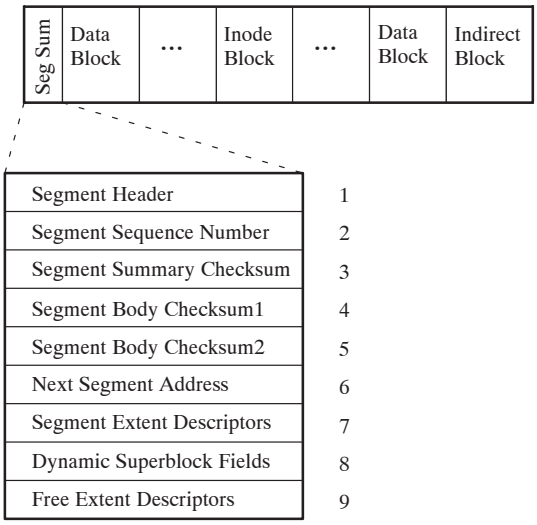

Figure 5. Structure of a Segment

descriptors (the 7th field in Figure 5), each consisting of a start address and a length in fragments. The use of more than one extent descriptor indicates that a segment can consist of mutiple extents of different sizes, making it adaptable to the disk utilization and fragmentation. When the disk utilization is low, hFS can take full advantage of large sequential writes by creating large segments. When the disk is highly utilized or fragmented, the size and contiguity of a segment will scale down accordingly. In the worst case, the performance of segment writes should be similar to FFS, which always write small files and metadata individually. In contrast, LFS insists on using fixed-size segments with no regard for disk utilization or fragmentation. Incidentally, the segment size in LFS cannot be made too large due to cleaning considerations [18]. hFS does not have this limitation. But for a fair comparison, we artifically limit the size of a segment in hFS to be the same as LFS (see Section 6).

By allowing a segment to have disjoint extents, hFS automatically switches to the behavior somewhat similar to the *hole–plugging* scheme suggested by Matthews et al. [18] when disk utilization and fragmentation are high. Under such a circumstance, the LFS cleaner starts to move live blocks from a partially empty segment into holes found in other segments to produce a free segment *using small I/Os*. Two approaches are suggested in [18] to adopt this new cleaning strategy, but both impact performance adversely. In contrast, hFS can create a new segment in the holes of existing segments directly. These holes can be located simply by looking up the log bitmap in hFS.

Because the size of a segment is finite, hFS must keep track of the amount of dirty data constantly and close the current segment when its capacity is reached. When disk space is scarce, hFS reserves enough free blocks to make sure that even the largest operation can run to completion. In this case, only operations that will not cause a net space loss can proceed. They include reads, truncations, and deletions.

4.2 Atomic Segment Write

In hFS, each segment write represents an atomic transition of the file system state. A segment gathers dirty data from mutiple updates, like a *compound transaction* does in Ext3 [5]. Similar to Ext3, hFS needs a brief quiescent state to capture a consistent snapshot of dirty data into a segment. This is achieved by using a segment lock and resource reservation. There is no concept of partial segments as used in LFS [6,7]. While hFS collects all dirty data during a segment write, LFS can choose to write dirty data from selected vnodes.

To guarantee the integrity of a segment, three checksums (fields 3-5 in Figure 5) are used to protect the segment summary and its body. After a system crash, hFS can reassemble a segment's contents by reading from the locations given by its extent descriptors. A segment is said to be *fully valid* when all three checksums are correct. A segment is said to be *partially valid* if only the summary checksum is correct. Such a segment must have holes written with new data.

A new segment is created when its capacity is reached. However, this can happen earlier in the following three scenarios:

(1) File synchronization. This happens when a user issues a *fsync()* call.

(2) Resource shortage. This happens when buffers or vnodes locked by hFS exceed a certain limit (see Section 5).

(3) Sync activity. This is triggered every 30 seconds to push backlogged dirty data to the disk to cap the window of vulnerability.

To create a segment, all dirty buffers are copied to one or more large *segment buffers* (each buffer is 128KB in our prototype) allocated on the fly. After a dirty buffer is copied into a segment buffer, it is marked as clean but pinned in the memory until the segment is committed on-disk. Similarly, when a dirty inode is copied into a segment buffer as part of its inode block, it is also pinned down to avoid reclamation before the segment is written.

A special *flush daemon* is used to flush segments. A segment cannot be flushed if any metadata it contains points to new and dirty blocks on the data partition. To enforce this constraint, each segment maintains a list of new data buffers that must be written before it can be flushed. This dependency tracking is necessary to make sure that a piece of metadata always refers to valid file data as proposed by McKusick and Ganger [14]. However, it can impose a severe performance penalty because hFS has to seek to the data partition to write any unresolved dependent buffers before flushing a segment. A solution is discussed in Section 6.4.

In hFS, dirty buffers for the log partition are never written individually. To keep track of whether dirty buffers of a vnode are flushed or not collectively, each segment is assigned an increasing 32-bit Segment Sequence Number (SSN, the 2nd field in Figure 5). hFS always records the SSN of a segment that contains the most recent updates of a vnode. This SSN is saved in the inode associated with the vnode. To flush a vnode, a process first pushes the vnode's dirty buffers into a segment if this has not been done yet. Then it waits until the SSN of the most recently flushed segment is no less than the one recorded in the inode corresponding to the vnode. Of course, if the vnode has any dirty buffers belonging to the data partition, the process must flush them as well. Only this time it has to deal with them one by one.

4.3 Delayed Deallocation

Disk space deallocation in hFS is done individually and explicitly without resorting to an inefficient cleaning process. Because hFS uses the no-overwrite strategy, a block's original location is freed whenever it is first modified. This requires hFS to modify a corresponding bitmap block stored in the first part of the BFILE (see Figure 3) just as an update-in-place file system like FFS would do. However, there are two potential problems that must be handled carefully.

First, fragments freed by a segment cannot be reused before the segment is committed on-disk to maintain system integrity. Second, freeing a fragment will modify a bitmap block, which will in turn free the bitmap block's original location, and so on. It is difficult, if not entirely impossible, to handle recursive deallocations. Note that even allocating a fragment frees a block, which is the original location of the corresponding bitmap block.

To handle these two issues simultaneously, hFS records in a segment's summary all the extents that are freed by the segment in the form of free extent descriptors (the 9th field in Figure 5). After a segment is committed on-disk, a special *bitmap daemon* will free the extents recorded in its free extent descriptors. This algorithm avoids both premature re-use of deallocated space and recursive deallocations. As a side effect, with the exception of a newly-created file system, log bitmap blocks themselves no longer contain the complete information of log space usage. The missing information is stored in free extent descriptors of committed segments.

The disk fragments used by segment summaries are freed in a different way. All segment summaries are linked together through the next segment address field in a summary (the 6th field in Figure 5). The head of this forward chain is stored in the superblock. After a certain number (64 in our prototype) of segments have been committed, the flush daemon will free the space occupied by their summaries using the following steps.

First, the daemon frees those summary fragments from bitmap blocks. The space freed by updating bitmap blocks themselves will be recorded in the free extent descriptors of the next segment. Second, the daemon writes the next segment synchronously. Finally, the daemon updates the superblock directly so that the summary list head stored in it points to the newly-committed segment. A segment disappears when its summary is freed, but its live blocks remain accessible. Because there are only a limited number of active segments at any time, a 32-bit SSN is more than adequate.

Incidentally, hFS maintains two copies of the superblock that are checksumed and timestamped. They are updated alternately so that

there is always at least one good copy of superblock even if a crash interrupts an in-place superblock update. Superblock updates are usually piggybacked in segment writes (the 8th field in Figure 5) to avoid in-place updates that induce disk seeking.

4.4 Crash Recovery

Crash recovery in hFS is easy to understand. Because a segment cannot free fragments used by itself, a subsequent segment never overwrites any part of its preceding segment. Therefore, hFS can always find at least one fully valid segment after a system crash on the list of segment summaries, whose head is stored in the superblock. A segment's contents can be reassembled by reading its summary first and then its body from where its extent descriptors point to. Note that a segment with SSN of X could be overwritten by a later segment with SSN of X+2 or larger. However, its summary remains intact (i.e., the segment is partially valid) as long as it has not been reclaimed using the algorithm as described in Section 4.3.

To perform a crash recovery, hFS follows the list of segment summaries until a fully valid segment with the largest SSN is found. It saves segment extent descriptors and free extent descriptors of each segment in the process. Any remaining corrupted segments are discarded. After this, hFS frees fragments recorded in free extent descriptors if they are not used by a later segment. Finally, hFS scans the orphan list in the IFILE, deleting inodes on the list if any.

5. IMPLEMENTATION ISSUES

hFS is written in C under FreeBSD 4.8-Release [22]. The bulk of the code is a kernel loadable module using the standard VFS/vnode interface. The number of lines of code is about 20,000. Although we considered implementation issues early in the design phase, we still encountered more obstacles than we expected. Unfortunately, only three of them can be described here due to space limitation. All of them are related to the log partition.

5.1 Inode Relocation

In hFS, disk inodes belonging to the same inode block are always stored together. This means that even if only one inode in its inode block is modified, all other live inodes in the same inode block must also be moved along with it. To achieve this effect, hFS actually scans a list of *inodeblock* structures instead of a list of vnodes when it creates a segment. Here we must deal with two scenarios:

(1) If one of these migrant inodes is not being used (i.e., it does not have an associated vnode), it must still be pinned until the transition is done. Otherwise, a process may attempt to read the inode from its new location before it is safely written there. To allow such a process to find the inode in the memory without doing a premature I/O, the traditional inode hash table (see kernel file *ufs_ihash.c*) is augmented to contain inodes without associated vnodes.

(2) If one of these migrant inodes is referenced by a free vnode, then hFS must be prepared for the possibility that the supposedly file system independent VFS layer can reclaim the vnode at any time by calling *getnewvnode()*. When that happens, hFS should leave the inode intact if its pin count is still non-zero.

In short, an inode being written or moved must be pinned down until it is safely stored into its new location. An inode without an associated vnode will be freed by a special *inode daemon* when its pin count drops to zero.

5.2 Vnode Recycling

An issue related to inode relocation is the control of vnode recycling. Whenever a vnode is reclaimed by the VFS layer, all its dirty data must be immediately flushed, at which time hFS may not have enough dirty data to write large segments. To prevent the VFS layer from recycling vnodes used by hFS at will, hFS artificially bumps up the reference count of any vnode that has been dirtied. At the same time, it keeps a count of such vnodes. When a threshold is reached, a segment write is triggered. In our prototype, the maximum number of vnodes that can be locked down by hFS simultaneously is one fourth of the total number of system-wide vnodes (as in LFS).

Because a process holds the segment lock when creating a segment, it cannot remove the artificial reference count of a vnode after it has collected the dirty data of the vnode. This is because the reference count could be the last one to an unlinked file. Removing the last reference to an unlinked file needs to update the file system state. But this is temporarily barred because a segment creation is already in progress. To avoid this potential deadlock, hFS lets the inode daemon take away the artificial reference count of a written vnode. This is done after the pin count of its associated inode drops to zero.

5.3 Buffer Recycling

A buffer for the log partition—not just a metadata buffer—cannot be reclaimed until the segment that contains its most recent updates is written. To prevent such a buffer from being released prematurely, it is put on the QUEUE_LOCKED queue. The pin count of a buffer is incremented each time it copies its newest data into a segment and is decremented each time a segment contains its dirty data is written. When the pin count of a buffer is zero, it can be released from the QUEUE_LOCKED queue. In our prototype, the maximum number of buffers that can be locked down by hFS simultaneously is one fourth of the total number of buffers available in the system minus a flat count of 10 (as in LFS).

Because buffers are a limited resource, each operation (e.g., create a file) must declare its maximum possible use of buffers before it can start. This makes sure that an operation, once started, will always run to completion without the need to perform a segment write. hFS also calculates the maximum log space usage of each operation to make sure that the current segment has enough room to contain any new dirty data to be generated by the operation.

6. EXPERIMENTAL EVALUATIONS

In this section we evaluate the performance of hFS running on FreeBSD. Because a file system has complex interplay with various subsystems in an operating system, especially with the virtual memory subsystem, we did not attempt to compare hFS with file

systems running in a different operating system. The following three file systems are compared against hFS:

(1) FFS: Although the original design of FFS was done more than a decade ago, it has received substantial improvements over the years [20]. Therefore, FFS is a formidable file system to compare against.

(2) LFS: LFS is ported from NetBSD 1.6-Release [23] and enhanced with the DIRHASH algorithm [20]. Because it is difficult to factor in realistic cleaning overhead, LFS runs *without* the cleaner process in all benchmarks. LFS is known to have superior small file write performance.

(3) yFS: yFS is our lightweight journaling file system that supports extent-based allocation and B+tree indexing [24]. It has been compared favorably with FFS [24].

The system under test is a Dell Dimension 5000 with one 2.4Ghz CPU, 512MB memory, and one Seagate ATA/100 Barracuda 80GB hard disk (ST380021A). All file systems are created in the same 30,720MB physical partition of the disk. FFS uses a fragment size of 2,048 bytes, a block size of 16,384 bytes, and a cylinder group size of 178MB. These values are all defaults under FreeBSD 4.8-Release. hFS, yFS, and LFS use the same fragment and block sizes as FFS does. The allocation group size of hFS and yFS are also 178MB. Both hFS and LFS use a segment size of 1MB. The log partition in hFS is 4,096MB. The size of the inline log area of yFS is 32MB.

We started 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 file system aging benchmark. We have also instrumented the kernel device driver to collect two kinds of low-level I/O statistics: total number of I/O requests and average I/O time in milliseconds. The total number of I/O requests has two components: the number of read I/Os (denoted as R) and the number of write I/Os (denoted as W) performed during the lifetime of the benchmark. They are reported as R+W in parenthesis after the total number of I/O requests. The average I/O time also has two components: the driver-level queuing time (denoted as Q) and service time (i.e., the time between the initiation of an I/O to the disk controller and the receipt of the corresponding interrupt, denoted as S). They are reported as Q+S in parenthesis after the average I/O time. Note that all these I/O statistics reported are only for the lifetime of each benchmark. The number of remaining I/Os after a benchmark terminates is insignificant and does not help to explain the user-level timing results.

6.1 Kernel Build Benchmark

This benchmark copies all kernel files from */usr/src/sys* to each of our test file systems and then builds the generic kernel and all kernel modules under it. The copy phase of this benchmark is I/O intensive, during which it copies 4,296 files. The compile phase is CPU bound. The results of this benchmark are shown in Table 1.

All four file systems performs comparably to each other in this benchmark. LFS spends the least amount of time in the copy phase because it writes all data sequentially. hFS is the runner-up because it has to seek to its data partition to write data blocks for 836 regular files larger than 16,384 bytes (i.e., one block). yFS performs better than FFS because the former does not impose artificial block boundaries. For example, if two files of 2 and 7 fragments respectively are written one after another in FFS, there may be a gap of 6 fragments between them (assuming one block has 8 fragments). In yFS, the two files can be laid out next to each other.

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

	Elapsed Time (s) (Copy+Compile)	Total I/O Requests (Read+Write)	Avg I/O Time (ms)
hFS	228 (21+207)	4011 (59+3962)	8 (5+3)
FFS	230 (27+203)	12887 (562+12325)	90 (89+1)
LFS	221 (17+204)	1920 (1025+895)	43 (40+3)
yFS	226 (23+203)	8981 (291+8690)	41 (40+1)

During the compile phase of the benchmark, both hFS and LFS consume slightly more CPU time than FFS and yFS. This is due to their use of memory copy to create large segments.

6.2 PostMark Benchmark

PostMark is a benchmark that simulates the working environment of a Web/News server [26]. It first creates a specified number of files. Then it performs a mix of creation, deletion, read, and append operations on them. Finally, all files are deleted. We used default settings of PostMark v1.5 (file size range is 500 bytes-9.77KB, read and write block sizes are both 512 bytes, the read/append and create/delete ratios are 5) except the number of files. Since file sizes are larger than 428 bytes, hFS cannot store files inline. The benchmark results are shown in Table 2.

hFS is the clear winner in this metadata-intensive benchmark. It beats FFS by 63%, 57%, and 53% in the case of 50,000, 100,000, and 150,000 files respectively. It also outperforms yFS by 56%, 42%, and 38% respectively in these cases. hFS just barely beats LFS, which runs without its cleaning process. Note that the deletion rates (number of files deleted per second) of FFS is unusually high. This is because deletions are pushed into the background—they are not finished when the benchmark terminates.

The deletion rates of hFS are better than those of LFS for two reasons: (1) One IFILE entry in hFS maps 32 inodes, while one IFILE entry in LFS maps only one inode; (2) A sparse directory block in hFS can merge with its neighbors to save I/O bandwidth. The directory blocks in FFS do not merge with each other.

6.3 Archive Extraction Benchmark

Archive extraction is used to install a software package on a Unix-like system. This benchmark extracts the file *ports.tgz* (19,079,386 bytes, consisting of 8,332 FreeBSD 4.8 ports) with the

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

	50,000 Files		100,000 Files		150,000 Files	
	Total Time (s)	Deletion Rate	Total Time (s)	Deletion Rate	Total Time (s)	Deletion Rate
hFS	19	12497	45	8031	74	6522
FFS	51	33325	104	36896	159	35003
LFS	20	9997	49	5554	75	5172
yFS	43	6248	78	7966	120	7213

Table 2(b) PostMark (v1.5) benchmark low-level I/O results

	50,000 Files		100,000 Files		150,000 Files	
	Total I/O Requests (Read+Write)	Avg I/O Time (ms)	Total I/O Requests (Read+Write)	Avg I/O Time (ms)	Total I/O Requests (Read+Write)	Avg I/O Time (ms)
hFS	4186 (286+3900)	81 (77+4)	9844 (1936+7908)	68 (64+4)	15774 (3984+11790)	56 (52+4)
FFS	54522 (572+53950)	67 (66+1)	109165 (1140+108025)	67 (66+1)	163604 (1612+161992)	69 (68+1)
LFS	4801 (1745+3056)	69 (65+4)	12931 (6746+6185)	50 (47+3)	19930 (10747+9183)	49 (46+3)
yFS	56237 (1732+54505)	49 (48+1)	112390 (3471+108919)	39 (38+1)	168577 (5100+163477)	39 (38+1)

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

	Creation Phase			Deletion Phase		
	Elapsed Time (seconds)	Total I/O Requests (Read+Write)	Avg I/O Time (ms)	Elapsed Time (seconds)	Total I/O Requests (Read+Write)	Avg I/O Time (ms)
hFS	24	2696 (3+2693)	22 (20+2)	10	2385 (2157+228)	3 (0+3)
FFS	151	105333 (507+104826)	248 (247+1)	41	27741 (13691+14050)	106 (105+1)
LFS	13	2619 (639+1980)	45 (42+3)	15	17868 (17788+80)	771us (62us+709us)
yFS	47	30509 (1988+28521)	65 (64+1)	34	7817 (2275+5542)	54 (50+4)

tar command and then mass-deletes all files with the *rm* command. We unmount and remount the file system between the two operations to remove any cache effects. The results of this benchmark are shown in Table 3. Once again, this benchmark brings out the excellent sequential write performance of both LFS and hFS. For example, hFS beats yFS by 49% (24 seconds vs. 47 seconds) and 71% (10 seconds vs. 34 seconds) in the two phases respectively. hFS is slower than LFS in the creation phase because (1) it writes 649 files outside its log partition and (2) it sleeps 19 times on average for reaching the locked buffer threshold. It beats LFS in the deletion phase partly due to the reasons we have discussed in Section 6.2. More importantly, 12,996 directories are stored using the inline format in hFS and can thus be read along with their inodes. In contrast, LFS has to perform 12,996 I/Os to read the contents of these small directories.

6.4 File System Aging Benchmark

File system aging has been used to demonstrate the effectiveness of several layout optimizations in FFS [10,25]. Since there is no standard way to age, this artificial aging benchmark is used to stress a file system. It does not indicate any long-term behavior of hFS. The aging benchmark performs a mix of file creations and deletions

to fill an empty file system with files and directories. The inclusion of directories is necessary so that they can play a role in the *dirpref* algorithm [20]. Each operation is performed under a randomly-chosen directory. It could be a creation or a deletion. The probability that the next operation is a creation decreases gradually as the test file system fills up. In case of a creation, the probability of creating a directory is 1/N (N defaults to 64), assuming that there are N files per directory on average. For 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 characters. The results of this benchmark are shown in Tables 4(a) and 4(b). LFS is the champion of this benchmark, but one has to figure out how much cleaning overhead to impose on it.

While hFS performs better than FFS, it loses against yFS. This is because this benchmark creates small files as well as many large ones. These new large files must be written first to make sure that their metadata in the log partition point to valid data. This semantic requirement causes many seeks, whose penalty is too large to be offset by using clustered I/O on small files and metadata. yFS also has to seek between its log area and its AGs to perform write-ahead logging (WAL). But the amount of logging I/O in yFS is far less than the amount of I/O done by hFS in its log partition. There are two reasons for this: (1) Unlike hFS, yFS does not write small files into its log area; (2) Unlike block-based journaling file systems such as Ext3 [5], yFS logs metadata updates at byte-level granularity. This benchmark shows that writing small files and *new* blocks of large files *simultaneously* is the Achilles' heel of hFS.

Looking back, the intent of this benchmark is to populate an empty file system with a directory hierarchy and file size distribution similar to a real file system. In doing so, it enforces a change of directory before each operation. This behavior defies the principle of locality that is used in practice. Furthermore, a separate experiment shows that the performance gap between hFS and yFS can be reduced if a second disk is used. In other words, hFS benefits more by using a separate disk for its log partition than yFS does by moving its log area to a separate disk.

It is worth noting that the *dirpref* algorithm [20] chooses the allocation group of a top-level directory (i.e., a directory directly under the root) randomly. This randomness affects the results noticeably because this benchmark changes a directory on each operation.

6.5 A Final Word on the Results

Our benchmark results show that using large sequential writes on small files and metadata boost their performance significantly. Although the choice of an update strategy has a key role in determining overall file system performance, it is certainly not the only factor to reckon with. For example, if we did not incorporate the DIRHASH algorithm into LFS, its PostMark performance would be much worse because PostMark creates all its files in one large directory. Therefore, it would be inappropriate to interpret these results *solely* in the context of in-place versus out-of-place updates.

7. RELATED WORK

In previous sections, we have already discussed the designs of FFS and LFS at length and showed how hFS is different from these two

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

	10,000 Operations (seconds)	20,000 Operations (seconds)	40,000 Operations (seconds)
hFS	68	152	324
FFS	73	169	359
LFS	21	64	124
yFS	48	119	246

Table 4(b) File system aging benchmark low-level I/O 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 Time (ms)
hFS	12870 (10+12860)	32 (27+5)	23833 (81+23752)	36 (30+6)	58432 (205+58227)	39 (33+6)
FFS	16000 (98+15902)	139 (134+5)	33888 (248+33640)	142 (137+5)	68623 (624+67999)	171 (166+5)
LFS	4920 (48+4872)	120 (116+4)	13782 (292+13490)	118 (114+4)	25230 (1510+23720)	110 (106+4)
yFS	14253 (248+14005)	80 (77+3)	30403 (424+29979)	84 (80+4)	58525 (768+57757)	95 (91+4)

file systems. Therefore, this section focuses on comparisons with some other file systems of interest.

The idea of separating different types of file system data is not new. Muller and Pasquale proposed the idea of a network multi-structured file system (MFS) [9] that separates control (i.e., metadata) and data into separate storage. Their design is different from hFS in several ways. For example, MFS is still an update-in-place design using a separate log area to improve write performance. In addition, MFS does not use clustered I/O to improve small file performance.

DualFS uses a dedicated metadata device to separate metadata and file data [28]. hFS separates both metadata and small files from large files. hFS stores small files on the log partition because one of its main goals is to exploit disk bandwidth for small files. This arrangement has some additional benefits when compared to DualFS: (1) The metadata of small files are closer to their data. (2) We remove one source of fragmentation caused by small files in the data partition. DualFS also puts bitmap information at the front of its IFILE, increasing the size of IFILE significantly if the file system has a large data partition. More importantly, DualFS still needs cleaning.

Update-in-place file systems take serious efforts to make sure that their disk allocators find the right spot quickly because an allocation may be done for each block and a sub-optimal allocation cannot be rectified cheaply. For example, IBM's JFS uses the binary buddy encoding of bitmap and free extent summary trees [2]. SGI's XFS uses dual B+trees to track free extents by start block address and size respectively [3]. Both schemes are designed to find large free extents quickly.

However, small files and metadata cannot take advantage of the ability to locate large extents quickly. Individual I/Os are still needed to update these data. In contrast, hFS aims to avoid small writes by using clustered I/O on small files and metadata. Allocations in hFS are done in large extents to hold all kinds of dirty data to be written into the same segment. Update-in-place file systems also mix metadata, small files, and large files in the same storage area, increasing the chances of disk fragmentation.

To improve metadata performance, update-in-place file systems usually resort to the technique called metadata journaling [2,3,4,5]. Metadata journaling uses the standard database approach of WAL, which first records metadata changes to an auxiliary log area before these metadata are allowed to be updated in-place. The performance of metadata journaling can be further improved by using fine-granularity logging and asynchronous group commit as shown in SGI's XFS [3] and yFS [24]. This technique trades costly in-place updates for many cheap log writes. However, it cannot eliminate small writes needed to update metadata in-place unless the metadata are deleted.

Soft Updates [13,14,15] is another way to boost metadata update performance by using delayed writes on metadata safely. However, Soft Updates requires a nontrivial 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. Soft Updates does not have as strong an atomic guarantee as hFS. In case of a crash, both old and new names could show up due to an interrupted rename operation. A background *fsck* run is still needed to salvage any unused resources [29], which interferes with normal I/O.

ReiserFS [4] stores metadata and small file data in one balanced B+tree, similar to the scheme proposed by Stonebraker [30]. In ReiserFS, a 128-bit key is used to locate various items stored in the B+tree directly. An item in ReiserFS can represent stat data (i.e., file attributes), direct data (i.e., tails of small files), indirect data (i.e., pointer to data blocks outside the B+tree), or directory data. One of the strengths often quoted about ReiserFS is its ability to cluster tails of small files in leaf nodes of the B+tree, thus boosting the performance of small files and saving disk space. However, performance drops when a tail growth forces a re-balancing of the B+tree. Users of ReiserFS may have to use its *notail* mount option to avoid this performance penalty but end up under-utilizing the disk space. hFS packs small files in units of fragments on the log partition. It does not have any problem with small file growth due to its use of the no-overwrite policy. hFS does not pack small files tighter at byte boundaries so that it can read a small file individually and avoid memory copy that would otherwise be needed to support *mmap()* operation. If a small file grows past one block, hFS moves it from the log partition to the data partition automatically.

A hybrid file system layout, HyLog, is proposed in [31] to exploit the relative advantages of updating in and out of place, similar in goal to the hFS concept and prototype implementation presented in the paper. Before pages are written to the disk (or disk array), they are classified as hot (pages that are written to frequently) or as cold. Writes to hot pages are implemented out of place, as in LFS, whereas writes to cold pages use update in place. The proposed approach is evaluated using a trace driven simulation that relies on an underlying analytical disk access model. Potential constraints/limitations of the model and subtle interactions between various systems components (VM, caching, interrupt handling etc.) that may not be accurately captured in the simulation model used in [31] precludes any realistic assessment of an actual implementation of Hylog. This is further exacerbated by the lack of critical implementation considerations in the design presented in [31]. It is thus not possible to compare Hylog and hFS head on.

8. CONCLUSIONS

hFS is a novel file system design that consciously combines two update strategies to improve small file and metadata performance. It also integrates features like efficient handling of large directories, dynamic disk inode allocation, and fast crash recovery.

We have evaluated the performance of hFS using a real implementation against three other file systems running on a contemporary platform. hFS delivers equal or better performance when compared to FFS with Soft Updates in all our benchmarks. It also enjoys a large small file performance edge over FFS and yFS. Therefore, we expect hFS to find its niche in the real world.

9. REFERENCES

[1] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. ACM Transactions on Computer Systems, Vol. 2, No. 3, pp. 181-197, August 1984.

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

[3] The SGI XFS project. http://oss.sgi.com/projects/xfs/.

[4] The ReiserFS project. http://www.namesys.com/.

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

[6] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems, pp. 26-52, February 1992.

[7] Margo I. Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. USENIX Winter Technical Conference, pp. 307-326, 1993.

[8] Dave Hitz, James Lau and Michael Malcolm. File System Design for a NFS File Server Appliance. Tech. Report TR 3002, Network Appliance Inc, updated 2005.

[9] Keith Muller and Joseph Pasquale. A High Performance Multi-Structured File System Design. Proceedings of 13th ACM Symposium on Operating Systems Principles, pp. 56-67. 1991.

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

[11] Steve D. Pate. Unix Filesystems: Evolution, Design, and Implementation. Wiley Publishing, Inc., 2003.

[12] Gregory R. Ganger and M. Frans Kaashoek. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. USENIX Annual Technical Conference, 1997.

[13] 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.

[14] 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.

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

[16] 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, 2002.

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

[18] Jeanne Neefe Matthews, et al. Improving the Performance of Log-Structured File Systems with Adaptive Methods. 16th ACM Symposium on Operating Sys. Principles, pp. 238-251, October 1997.

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

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

[21] Sape J. Mullender and Andrew S. Tanenbaum. Immediate Files. Software—Practice and Experience, Volume 14, pp. 365–368, April 1984.

[22] The FreeBSD Project. http://www.freebsd.org.

[23] The NetBSD Project. http://www.netbsd.org.

[24] Zhihui Zhang and Kanad Ghose. yFS: A Journaling File System Design for Handling Large Data Sets with Reduced Seeking. 2nd Conference on File and Storage Technologies, pp. 59-72, March 2003.

[25] 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.

[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] Juan Piernas, Toni Cortes, and José M. García. DualFS: A New Journaling File System without Meta-data Duplication. Proceedings of the 16th international conference on Supercomputing, 2002.

[29] Marshall Kirk McKusick. Running "fsck" in the Background. In Proceedings of BSDCON 2002, pp. 55-64, February, 2002.

[30] Michael Stonebraker. Operating System Support for Database Management. Communications of the ACM, Vol. 24, No. 7, July, 1981.

[31] Wenguang Wang, Yanping Zhao, and Rick Bunt. HyLog: A High Performance Approach to Managing Disk Layout, Proc. USENIX FAST 04 pp. 145-158.