Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem

Marshall Kirk McKusick

Author and Consultant

Gregory R. Ganger

Carnegie Mellon University

ABSTRACT

Traditionally, filesystem consistency has been maintained across system failures either by using synchronous writes to sequence dependent metadata updates or by using write-ahead logging to atomically group them. Soft updates, an alternative to these approaches, is an implementation mechanism that tracks and enforces metadata update dependencies to ensure that the disk image is always kept consistent. The use of soft updates obviates the need for a separate log or for most synchronous writes. Indeed, the ability of soft updates to aggregate many operations previously done individually and synchronously reduces the number of disk writes by 40 to 70% for file-intensive environments (e.g., program development, mail servers, etc.). In addition to performance enhancement, soft updates can also maintain better disk consistency. By ensuring that the only inconsistencies are unclaimed blocks or inodes, soft updates can eliminate the need to run a filesystem check program after every system crash. Instead, the system is brought up immediately. When it is convenient, a background task can be run on the active filesystem to reclaim any lost blocks and inodes.

This paper describes an implementation of soft updates and its incorporation into the 4.4BSD fast filesystem. It details the changes that were needed, both to the original research prototype and to the BSD system, to create a production-quality system. It also discusses the experiences, difficulties, and lessons learned in moving soft updates from research to reality; as is often the case, non-focal operations (e.g., **fsck** and ‘‘fsync’’) required rethinking and additional code. Experiences with the resulting system validate the earlier research: soft updates integrates well with existing filesystems and enforces metadata dependencies with performance that is within a few percent of optimal.

1. Background and Introduction

In filesystems, **metadata** (e.g., directories, inodes and free block maps) gives structure to raw storage capacity. Metadata provides pointers and descriptions for linking multiple disk sectors into files and identifying those files. To be useful for persistent storage, a filesystem must maintain the integrity of its metadata in the face of unpredictable system crashes, such as power interruptions and operating system failures. Because such crashes usually result in the loss of all information in volatile main memory, the information in non-volatile storage (i.e., disk) must always be consistent enough to deterministically reconstruct a coherent filesystem state. Specifically, the on-disk image of the filesystem must have no dangling pointers to uninitialized space, no ambiguous resource ownership caused by multiple pointers, and no unreferenced live resources. Maintaining these invariants generally requires sequencing (or atomic grouping) of updates to small on-disk metadata objects.

Traditionally, the BSD fast filesystem (FFS) [McKusick et al, 1984] and its derivatives have used synchronous writes to properly sequence stable storage changes. For example, creating a file in a BSD system involves first allocating and initializing a new inode and then filling in a new directory entry to point to it. With the synchronous write approach, the filesystem forces an application that creates a file to wait for the disk write that initializes the on-disk inode. As a result, filesystem operations like file creation and deletion proceed at disk speeds rather than processor/memory speeds in these systems [McVoy & Kleiman, 1991; Ousterhout, 1990; Seltzer et al, 1993]. Since disk access times are long compared to the speeds of other computer components, synchronous writes reduce system performance.

The metadata update problem can also be addressed with other mechanisms. For example, one can eliminate the need to keep the on-disk state consistent by using NVRAM technologies, such as an uninterruptable power supply or Flash RAM [Wu & Zwaenepoel, 1994]. With this approach, only updates to the NVRAM need to be kept consistent, and updates can propagate to disk in any order and whenever it is convenient. Another approach is to group each set of dependent updates as an atomic operation with some form of write-ahead logging [Chutani et al, 1992; Hagmann, 1987; NCR_Corporation, 1992] or shadow-paging [Chamberlin et al, 1981; Chao et al, 1992; Rosenblum & Ousterhout, 1991; Stonebraker, 1987]. Generally speaking, these approaches augment the on-disk state with additional information that can be used to reconstruct the committed metadata values after any system failure other than media corruption. Many modern filesystems successfully use write-ahead logging to improve performance compared to the synchronous write approach.

In [Ganger & Patt, 1994], an alternative approach called **soft updates** was proposed and evaluated in the context of a research prototype. With soft updates, the filesystem uses delayed writes (i.e., write-back caching) for metadata changes, tracks dependencies between updates, and enforces these dependencies at write-back time. Because most metadata blocks contain many pointers, cyclic dependencies occur frequently when dependencies are recorded only at the block level. Therefore, soft updates tracks dependencies on a per-pointer basis, which allows blocks to be written in any order. Any still-dependent updates in a metadata block are rolled-back before the block is written and rolled-forward afterwards. Thus, dependency cycles are eliminated as an issue. With soft updates, applications always see the most current copies of metadata blocks and the disk always sees copies that are consistent with its other contents.

In this paper, we describe the incorporation of soft updates into the 4.4BSD FFS used in the NetBSD, OpenBSD, FreeBSD, and BSDI operating systems. In doing so, we discuss experiences and lessons learned and describe the aspects that were more complex than was suggested in the original research papers. As is often the case, non-focal operations like bounding the use of kernel memory used to track dependencies, fully implementing the ‘‘fsync’’, system call semantics, properly detecting and handling lost resources in **fsck**, and cleanly and expediently completing an **unmount** system call required some rethinking and resulted in much of the code complexity. Despite these difficulties, our performance experiences do verify the conclusions of the early research. Specifically, using soft updates in BSD FFS eliminates most synchronous writes and provides performance that is within 5 percent of ideal (i.e., the same filesystem with no update ordering). At the same time, soft updates allows BSD FFS to provide cleaner semantics, stronger integrity and security guarantees, and immediate crash recovery (**fsck** not required for safe operation after a system crash).

The remainder of this paper is organized as follows. Section 2 describes the update dependencies that arise in BSD FFS operations. Section 3 describes how the BSD soft updates implementation handles these update dependencies, including the key data structures, how they are used, and how they are incorporated into the 4.4BSD operating system. Section 4 discusses experiences and lessons learned from converting a prototype implementation into an implementation suitable for use in production environments. Section 5 briefly overviews performance results from 4.4BSD systems using soft updates. Section 6 discusses new support for filesystem snapshots and how this support can be used to do a partial **fsck** in the background on active filesystems. Section 7 outlines the status and availability of the BSD soft updates code.

2. Update Dependencies in the BSD Fast Filesystem

Several important filesystem operations consist of a series of related modifications to separate metadata structures. To ensure recoverability in the presence of unpredictable failures, the modifications often must be propagated to stable storage in a specific order. For example, when creating a new file, the filesystem allocates an inode, initializes it and constructs a directory entry that points to it. If the system goes down after the new directory entry has been written to disk but before the initialized inode is written, consistency may be compromised since the contents of the on-disk inode are unknown. To ensure metadata consistency, the initialized inode must reach stable storage before the new directory entry. We refer to this requirement as an **update dependency**, because safely writing the directory entry depends on first writing the inode. The ordering constraints map onto three simple rules:

1) Never point to a structure before it has been initialized (e.g., an inode must be initialized before a directory entry references it).

2) Never re-use a resource before nullifying all previous pointers to it (e.g., an inode's pointer to a data block must be nullified before that disk block may be re-allocated for a new inode).

3) Never reset the old pointer to a live resource before the new pointer has been set (e.g., when renaming a file, do not remove the old name for an inode until after the new name has been written).

This section describes the update dependencies that arise in BSD FFS, assuming a basic understanding of BSD FFS as described in [McKusick et al, 1996]. There are eight BSD FFS activities that require update ordering to ensure post-crash recoverability: file creation, file removal, directory creation, directory removal, file/directory rename, block allocation, indirect block manipulation, and free map management.

The two main resources managed by the BSD FFS are inodes and data blocks. Two bitmaps are used to maintain allocation information about these resources. For each inode in the filesystem, the inode bitmap has a bit that is set if the inode is in use and cleared if it is free. For each block in the filesystem, the data block bitmap has a bit that is set if the block is free and cleared if it is in use. Each FFS filesystem is broken down into fixed-size pieces referred to as cylinder groups. Each cylinder group has a cylinder group block that contains the bitmaps for the inodes and data blocks residing within that cylinder group. For a large filesystem, this organization allows just those sub-pieces of the filesystem bitmap that are actively being used to be brought into the kernel memory. Each of these active cylinder group blocks is stored in a separate I/O buffer and can be written to disk independently of the other cylinder group blocks.

When a file is created, three metadata structures located in separate blocks are modified. The first is a new inode, which is initialized with its type field set to the new file type, its link count set to one to show that it is live (i.e., referenced by some directory), its permission fields set as specified, and all other fields set to default values. The second is the inode bitmap, which is modified to show that the inode has been allocated. The third is a new directory entry, which is filled in with the new name and a pointer to the new inode. To ensure that the bitmaps always reflect all allocated resources, the bitmap must be written to disk before the inode or directory entry. Because the inode is in an unknown state until after it has been initialized on the disk, rule #1 specifies that there is an update dependency requiring that the relevant inode be written before the relevant directory entry. Although not strictly necessary, most BSD fast filesystem implementations also immediately write the directory block before the system call creating the file returns. This second synchronous write ensures that the filename is on stable storage if the application later does an "fsync" system call. If it were not done, then the "fsync" call would have to be able to find all the unwritten directory blocks containing a name for the file and write them to disk. A similar update dependency between inode and directory entry exists when adding a second name for the same file (a.k.a. a hard link), since the addition of the second name requires the filesystem to increment the link count in the inode and write the inode to disk before the entry may appear in the directory.

When a file is deleted, a directory block, an inode block, and one or more cylinder group bitmaps are modified. In the directory block, the relevant directory entry is "removed", usually by nullifying the inode pointer. In the inode block, the relevant inode's type field, link count and block pointers are zeroed out. The deleted file's blocks and inode are then added to the appropriate free block/inode maps. Rule #2 specifies that there are update dependencies between the directory entry and the inode and between the inode and any modified free map bits. To keep the link count conservatively high (and reduce complexity in practice), the update dependency between a directory entry and inode also exist when removing one of multiple names (hard links) for a file.

Creation and removal of directories is largely as described above for regular files. However, the ".." entry is a link from the child directory to the parent, which adds additional update dependencies. Specifically, during creation, the parent's link count must be incremented on disk before the new directory's ".." pointer is written. Likewise, during removal, the parent's link count must be decremented after the removed directory's ".." pointer is nullified. (Note that this nullification is implicit in deleting the child directory's pointer to the corresponding directory block.)

When a new block is allocated, its bitmap location is updated to reflect that it is in use and the block's contents are initialized with newly written data or zeros. In addition, a pointer to the new block is added to an inode or indirect block (see below). To ensure that the on-disk bitmap always reflects allocated resources, the bitmap must be written to disk before the pointer. Also, because the contents of the newly allocated disk location are unknown, rule #1 specifies an update dependency between the new block and the pointer to it. Because enforcing this update dependency with synchronous writes can reduce data creation throughput by a factor of two [Ganger & Patt, 1994], many implementations ignore it for regular data blocks. This implementation decision reduces integrity and security, since newly allocated blocks generally contain previously deleted file data. Soft updates allows all block allocations to be protected in this way with near-zero performance reduction.

Manipulation of indirect blocks does not introduce fundamentally different update dependencies, but they do merit separate discussion. Allocation, both of indirect blocks and of blocks pointed to by indirect blocks, is as discussed above. File deletion, and specifically de-allocation, is more interesting for indirect blocks. Because the inode reference is the only way to identify indirect blocks and blocks connected to them (directly or indirectly), nullifying the inode's pointer to an indirect block is enough to eliminate all recoverable pointers to said blocks. Once the pointer is nullified on disk, all its blocks can be freed. Only for partial truncation of a file do update dependencies between indirect block pointers and the pointed-to blocks exist. Some FFS implementations do not exploit this distinction, even though it can have a significant effect on the time required to remove a large file.

When a file is being renamed, two directory entries are affected. A new entry (with the new name) is created and set to point to the relevant inode and the old entry is removed. Rule #3 states that the new entry should be written to disk before the old entry is removed to avoid having the file unreferenced on reboot. If link counts are being kept conservatively, rename involves at least four disk updates in sequence: one to increment the inode's link count, one to add the new directory entry, one to remove the old directory entry, and one to decrement the link count. If the new name already existed, then the addition of the new directory entry also acts as the first step of file removal as discussed above. Interestingly, rename is the one POSIX file operation that really wants an atomic update to multiple user-visible metadata structures to provide ideal semantics. POSIX does not require said semantics and most implementations cannot provide it.

On an active filesystem, the bitmaps change constantly. Thus, the copy of the bitmaps in the kernel memory often differs from the copy that is stored on the disk. If the system halts without writing out the incore state of the bitmaps, some of the recently allocated inodes and data blocks may not be reflected in the out-of-date copies of the bitmaps on the disk. So, the filesystem check program, **fsck**, must be run over all the inodes in the filesystem to ascertain which inodes and blocks are in use and bring the bitmaps up to date [McKusick & Kowalski, 1994]. An added benefit of soft updates is that it tracks the writing of the bitmaps to disk and uses this information to ensure that no newly allocated inodes or pointers to newly allocated data blocks will be written to disk until after the bitmap that references them has been written to disk. This guarantee ensures that there will never be an allocated inode or data block that is not marked in the on-disk bitmap. This guarantee, together with the other guarantees made by the soft update code, means that it is no longer necessary to run **fsck** after a system crash. This feature is discussed further in Section 6.

3. Tracking and Enforcing Update Dependencies

This section describes the BSD soft updates data structures and their use in enforcing the update dependencies described in Section 2. The structures and algorithms described eliminate all synchronous write operations from BSD FFS except for the partial truncation of a file and the "fsync" system call, which explicitly requires that all the state of a file be committed to disk before the system call returns.

The key attribute of soft updates is dependency tracking at the level of individual changes within cached blocks. Thus, for a block containing 64 inodes, the system can maintain up to 64 dependency structures with one for each inode in the buffer. Similarly for a buffer containing a directory block containing 50 names, the system can maintain up to 50 dependency structures with one for each name in the directory. With this level of detailed dependency information, circular dependencies between blocks are not problematic. For example, when the system wishes to write a buffer containing inodes, those inodes that can be safely written can go to the disk. Any inodes that cannot be safely written yet are temporarily rolled back to their safe values while the disk write proceeds. After the disk write completes, such inodes are rolled forward to their current values. Because the buffer is locked throughout the time that the contents are rolled back, the disk write is being done, and the contents are rolled forward, any processes wishing to use the buffer will be blocked from accessing it until it has been returned to its current state.

3.1. Dependency Structures

A soft updates implementation uses a variety of data structures to track pending update dependencies among filesystem structures. Table 1 lists the dependency structures used in the BSD soft updates implementation, their main functions, and the types of

Name	Function	Associated Structures
bmsafemap	track bitmap dependencies (points to lists of dependency structures for recently allocated blocks and inodes)	cylinder group block
inodedep	track inode dependencies (information and list head pointers for all inode-related dependencies, including changes to the link count, the block pointers, and the file size)	inode block
allocdirect	track inode-referenced block (linked into lists pointed to by an inodedep and a bmsafemap to track inode's dependence on the block and bitmap being written to disk)	data block or indirect block or directory block
indirdep	track indirect block dependencies (points to list of dependency structures for recently-allocated blocks with pointers in the indirect block)	indirect block
allocindir	track indirect block-referenced block (linked into lists pointed to by an indirdep and a bmsafemap to track the indirect block's dependence on that block and bitmap being written to disk)	data block or indirect block or directory block
pagedep	track directory block dependencies (points to lists of diradd and dirrem structures)	directory block
diradd	track dependency between a new directory entry and the referenced inode	inodedep and directory block
mkdir	track new directory creation (used in addition to standard diradd structure when doing a mkdir)	inodedep and directory block
dirrem	track dependency between a deleted directory entry and the unlinked inode	first pagedep then tasklist
freefrag	tracks a single block or fragment to be freed as soon as the corresponding block (containing the inode with the now-replaced pointer to it) is written to disk	first inodedep then tasklist
freeblks	tracks all the block pointers to be freed as soon as the corresponding block (containing the inode with the now-zeroed pointers to them) is written to disk	first inodedep then tasklist
freefile	tracks the inode that should be freed as soon as the corresponding block (containing the inode block with the now-reset inode) is written to disk	first inodedep then tasklist

Table 1: *Soft Updates and Dependency Tracking*

blocks with which they can be associated. These dependency structures are allocated and associated with blocks as various file operations are completed. They are connected to the in-core blocks with which they are associated by a pointer in the corresponding buffer header. Two common aspects of all listed dependency structures are the *worklist* structure and the states used to track the progress of a dependency.

The *worklist* structure is really just a common header included as the first item in each dependency structure. It contains a set of linkage pointers and a type field to show the type of structure in which it is embedded. The *worklist* structure allows several different types of dependency structures to be linked together into a single list. The soft updates code can traverse one of these heterogenous lists, using the type field to determine which kind of dependency structure it has encountered, and take the appropriate action with each.

The typical use for the *worklist* structure is to link together a set of dependencies associated with a buffer. Each buffer in the system has a *worklist* head pointer added to it. Any dependencies associated with that buffer are linked onto its *worklist* list. After the buffer has been locked and just before the buffer is to be written, the I/O system passes the buffer to the soft update code to let it know that a disk write is about to be initiated. The soft update code then traverses the list of dependencies associated with the buffer and does any needed roll-back operations. After the disk write completes but before the buffer is unlocked, the I/O system calls the soft update code to let it know that a write has completed. The soft update code then traverses the list of dependencies associated with the buffer, does any needed roll-forward operations, and deallocates any dependencies that are fulfilled by the data in the buffer having been written to disk.

Another important list maintained by the soft updates code is the *tasklist* that contains background tasks for the work daemon. Dependency structures are generally added to the *tasklist* during the disk write completion routine, describing tasks that have become safe given the disk update but that may need to block for locks or I/O and therefore cannot be completed during the interrupt handler. Once per second, the syncer daemon (in its dual role as the soft updates work daemon) wakes up and calls into the soft updates code to process any items on the *tasklist*. The work done for a dependency structure on this list is type-dependent. For example, for a *freeblks* structure, the listed blocks are marked free in the block bitmaps. For a *dirrem* structure, the associated inode's link count is decremented, possibly triggering file deletion.

Dependency States. Most dependency structures have a set of flags that describe the state of completion of the corresponding dependency. Dirty cache blocks can be written to the disk at any time. When the I/O system hands the buffer to the soft updates code (before and after a disk write), the states of the associated dependency structures determine what actions are taken. Although the specific meanings vary from structure to structure, the three main flags and their general meanings are:

ATTACHED
: The ATTACHED flag indicates that the buffer with which the dependency structure is associated is not currently being written. When a disk write is initiated for a buffer with a dependency that must be rolled-back, the ATTACHED flag is cleared in the dependency structure to show that it has been rolled-back in the buffer. When the disk write completes, updates described by dependency structures that have the ATTACHED flag cleared are rolled-forward and the ATTACHED flag is set. Thus, a dependency structure can never be deleted while its ATTACHED flag is cleared, since the information needed to do the roll-forward operation would then be lost.

DEPCOMPLETE
: The DEPCOMPLETE flag indicates that all associated dependencies have been completed. When a disk write is initiated, the update described by a dependency structure is rolled-back if the DEPCOMPLETE flag is clear. For example, in a dependency structure that is associated with newly allocated inodes or data blocks, the DEPCOMPLETE flag is set when the corresponding bitmap has been written to disk.

COMPLETE
: The COMPLETE flag indicates that the update being tracked has been committed to the disk. For some dependencies, updates will be rolled back during disk writes when the COMPLETE flag is clear. For example, for a newly allocated data block, the COMPLETE flag is set when the contents of the block have been written to disk.

In general, the flags are set as disk writes complete and a dependency structure can be deallocated only when its ATTACHED, DEPCOMPLETE, and COMPLETE flags are all set. Consider the example of a newly allocated data block that will be tracked by an *allocdirect* structure. The ATTACHED flag will initially be set when the allocation occurs. The DEPCOMPLETE flag will be set after the bitmap allocating that new block is written. The COMPLETE flag will be set after the contents of the new block are written. If the inode claiming the newly allocated block is written before both the DEPCOMPLETE and COMPLETE flags are set, the ATTACHED flag will be cleared while the block pointer in the inode is rolled back to zero, the inode is written, and the block pointer in the inode is rolled forward to the new block number. Where different, the specific meanings of these flags in the various dependency structures are described in the subsections that follow.

3.2. Bitmap Dependency Tracking

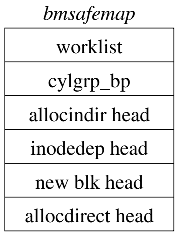

Figure 1: *Bitmap Update Dependencies*

Bitmap updates are tracked by the *bmsafemap* structure shown in Figure 1. Each buffer containing a cylinder group block will have its own *bmsafemap* structure. As with every dependency structure, the first entry in the *bmsafemap* structure is a *worklist* structure. Each time an inode, direct block, or indirect block is allocated from the cylinder group, a dependency structure is created for that resource and linked onto the appropriate *bmsafemap* list. Each newly allocated inode will be represented by an *inodedep* structure linked to the *bmsafemap* **inodedep head** list. Each newly allocated block directly referenced by an inode will be represented by an *allocdirect* structure linked to the *bmsafemap* **allocdirect head** list. Each newly allocated block referenced by an indirect block will be represented by an *allocindir* structure linked to the *bmsafemap* **allocindir head** list. Because of the FFS code's organization, there is a small window between the time a block is first allocated and the time at which its use is known. During this period of time, it is described by a *newblk* structure linked to the *bmsafemap* **new blk head** list. After the kernel chooses to write the cylinder group block, the soft update code will be notified when the write has completed. At that time, the code traverses the inode, direct block, indirect block, and new block lists, setting the DEPCOMPLETE flag in each dependency structure and removing said dependency structure from its dependency list. Having cleared all its dependency lists, the *bmsafemap* structure can be deallocated. There are multiple lists as it is slightly faster and more type-safe to have lists of specific types.

3.3. Inode Dependency Tracking

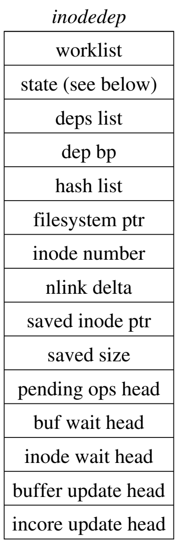

Figure 2: *Inode Update Dependencies*

Inode updates are tracked by the *inodedep* structure shown in Figure 2. The *worklist* and ‘‘state’’ fields are as described for dependency structures in general. The ‘‘filesystem ptr’’ and ‘‘inode number’’ fields identify the inode in question. When an inode is newly allocated, its *inodedep* is attached to the ‘‘inodedep head’’ list of a *bmsafemap* structure. Here, ‘‘deps list’’ chains additional new *inodedeps* and ‘‘dep bp’’ points to the cylinder group block that contains the corresponding bitmap. Other *inodedep* fields are explained in subsequent subsections.

Before detailing the rest of the dependencies associated with an inode, we need to describe the steps involved in updating an inode on disk as pictured in Figure 3.

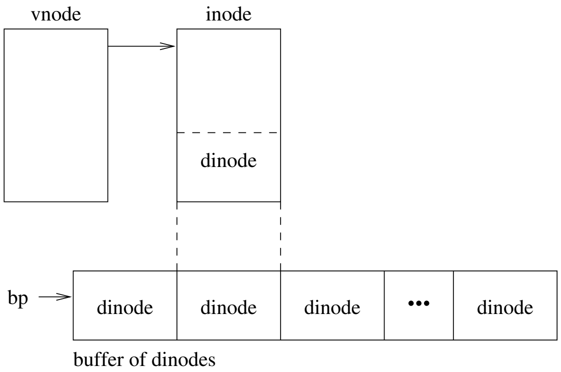

Figure 3: *Inode Update Steps*

Step 1: The kernel calls the vnode operation, VOP_UPDATE, which requests that the disk resident part of an inode (referred to as a dinode) be copied from its in-memory vnode structure to the appropriate disk buffer. This disk buffer holds the contents of an entire disk block, which is usually big enough to include 64 dinodes. Some dependencies are fulfilled only when the inode has been written to disk. For these, dependency structures need to track the progress of the writing of the inode. Therefore, during step 1, a soft update routine, ‘‘softdep_update_inodeblock’’, is called to move *allocdirect* structures from the ‘‘incore update’’ list to the ‘‘buffer update’’ list and to move *freefile*, *freeblks*, *freefrag*, *diradd*, and *mkdir* structures (described below) from the ‘‘inode wait’’ list to the ‘‘buf wait’’ list.

Step 2: The kernel calls the vnode operation, VOP_STRATEGY, which prepares to write the buffer containing the dinode, pointed to by **bp** in Figure 3. A soft updates routine, ‘‘softdep_disk_io_initiation’’, identifies *inodedep* dependencies and calls ‘‘initiate_write_inodeblock’’ to do roll-backs as necessary.

Step 3: Output completes on the buffer referred to by **bp** and the I/O system calls a routine, ‘‘biodone’’, to notify any waiting processes that the write has finished. The ‘‘biodone’’ routine then calls a soft updates routine, ‘‘softdep_disk_write_complete’’, which identifies *inodedep* dependencies and calls ‘‘handle_written_inodeblock’’ to revert rollbacks and clear any dependencies on the ‘‘buf wait’’ and ‘‘buffer update’’ lists.

3.4. Direct Block Dependency Tracking

Figure 4 illustrates the dependency structures involved in allocation of direct blocks. Recall that the key dependencies are that, before the on-disk inode points

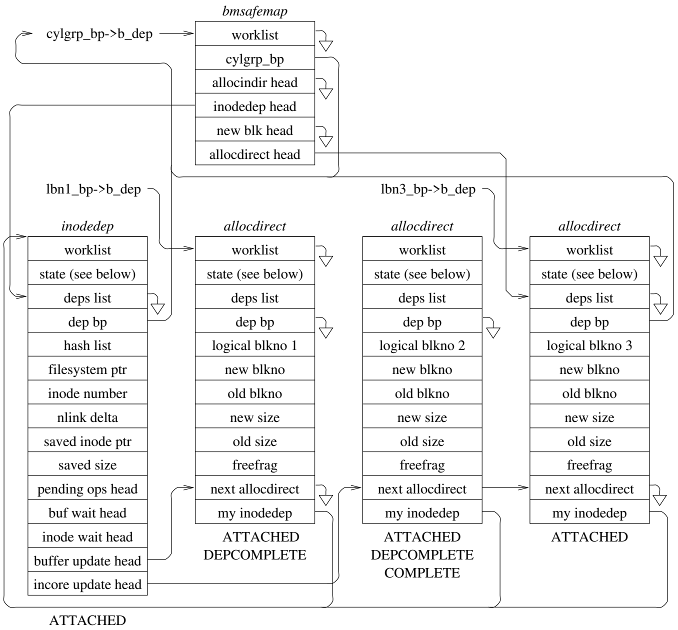

Figure 4: *Direct Block Allocation Dependencies*

to a newly allocated block, both the corresponding bitmap block and the new block itself must be written to the disk. The order in which the two dependencies complete is not important. The figure introduces the *allocdirect* structure which tracks blocks directly referenced by the inode. The three recently allocated logical blocks (1, 2, and 3) shown are each in a different state. For logical block 1, the bitmap block dependency is complete (as shown by the DEPCOMPLETE flag being set), but the block itself has not yet been written (as shown by the COMPLETE flag being cleared). For logical block 2, both dependencies are complete. For logical block 3, neither dependency is complete, so the corresponding *allocdirect* structure is attached to a *bmsafemap* ‘‘allocdirect head’’ list (recall that this list is traversed to set DEPCOMPLETE flags after bitmap blocks are written). The COMPLETE flag for logical blocks 1 and 3 will be set when their initialized data blocks are written to disk. The figure also shows that logical block 1 existed at a time that VOP_UPDATE was called, which is why its *allocdirect* structure resides on the *inodedep* ‘‘buffer update’’ list. Logical blocks 2 and 3 were created after the most recent call to VOP_UPDATE and thus their structures reside on the *inodedep* ‘‘incore update’’ list.

For files that grow in small steps, a direct block pointer may first point to a fragment that is later promoted to a larger fragment and eventually to a full-sized block. When a fragment is replaced by a larger fragment or a full-sized block, it must be released back to the filesystem. However, it cannot be released until the new fragment or block has had its bitmap entry and contents written and the inode claiming the new fragment or block has been written to the disk. The fragment to be released is described by a *freefrag* structure (not shown). The *freefrag* structure is held on the ‘‘freefrag’’ list of the *allocdirect* for the block that will replace it until the new block has had its bitmap entry and contents written. The *freefrag* structure is then moved to the ‘‘inode wait’’ list of the *inodedep* associated with its *allocdirect* structure where it migrates to the ‘‘buf wait’’ list when VOP_UPDATE is called. The *freefrag* structure eventually is added to the *tasklist* after the buffer holding the inode block has been written to disk. When the *tasklist* is serviced, the fragment listed in the *freefrag* structure is returned to the free-block bitmap.

3.5. Indirect Block Dependency Tracking

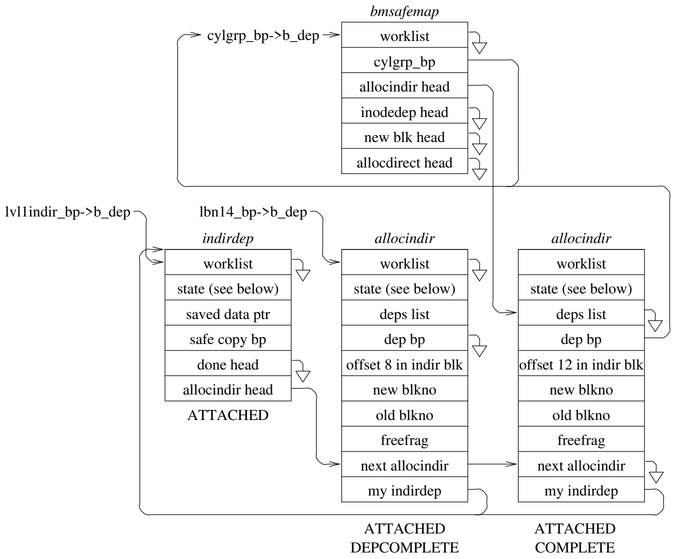

Figure 5: *Indirect Block Allocation Dependencies*

Figure 5 shows the dependency structures involved in allocation of indirect blocks, which includes the same dependencies as with direct blocks. This figure introduces two new dependency structures. A separate *allocindir* structure tracks each individual block pointer in an indirect block. The *indirdep* structure manages all the *allocindir* dependencies associated with an indirect block. The figure shows a file that recently allocated logical blocks 14 and 15 (the third and fourth entries, at offsets 8 and 12, in the first indirect block). The allocation bitmaps have been written for logical block 14 (as shown by its DEPCOMPLETE flag being set), but not for block 15. Thus, the *bmsafemap* structure tracks the *allocindir* structure for logical block 15. The contents of logical block 15 have been written to disk (as shown by its COMPLETE flag being set), but not those of block 14. The COMPLETE flag will be set in 14's *allocindir* structure once the block is written. The list of *allocindir* structures tracked by an *indirdep* structure can be quite long (e.g., up to 2048 entries for 8KB indirect blocks). To avoid traversing lengthy dependency structure lists in the I/O routines, an *indirdep* structure maintains a second version of the indirect block: the ''saved data ptr'' always points to the buffer's up-to-date copy and the ''safe copy ptr'' points to a version that includes only the subset of pointers that can be safely written to disk (and NULL for the others). The former is used for all filesystem operations and the latter is used for disk writes. When the ''allocindir head'' list becomes empty, the ''saved data ptr'' and ''safe copy ptr'' point to identical blocks and the *indirdep* structure (and the safe copy) can be deallocated.

3.6. Dependency Tracking for new Indirect Blocks

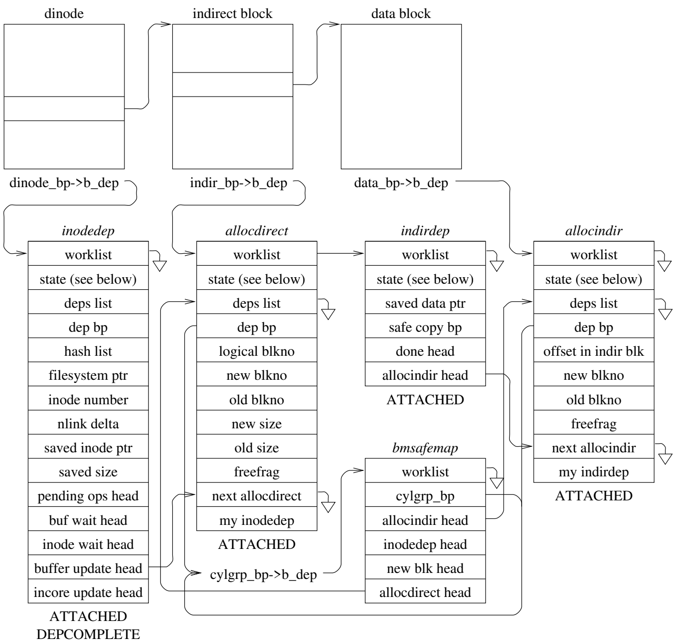

Figure 6: *Dependencies for a File Expanding into an Indirect Block*

Figure 6 shows the structures associated with a file that recently expanded into its single-level indirect block. Specifically, this involves *inodedep* and *indirdep* structures to manage dependency structures for the inode and indirect block, an *allocdirect* structure to track the dependencies associated with the indirect block's allocation, and an *allocindir* structure to track the dependencies associated with a newly allocated block pointed to by the indirect block. These structures are used as described in the previous three subsections. Neither the indirect block nor the data block that it references have had their bitmaps set, so they do not have their DEPCOMPLETE flag set and are tracked by a *bmsafemap* structure. The bitmap entry for the inode has been written, so the *inodedep* structure has its DEPCOMPLETE flag set. The use of the ''buffer update head'' list by the *inodedep* structure indicates that the in-core inode has been copied into its buffer by a call to VOP_UPDATE. Neither of the dependent pointers (from the inode to the indirect block and from the indirect block to the data block) can be safely included in disk writes yet, since the corresponding COMPLETE and DEPCOMPLETE flags are not set. Only after the bitmaps and the contents have been written will all the flags be set and the dependencies complete.

3.7. New Directory Entry Dependency Tracking

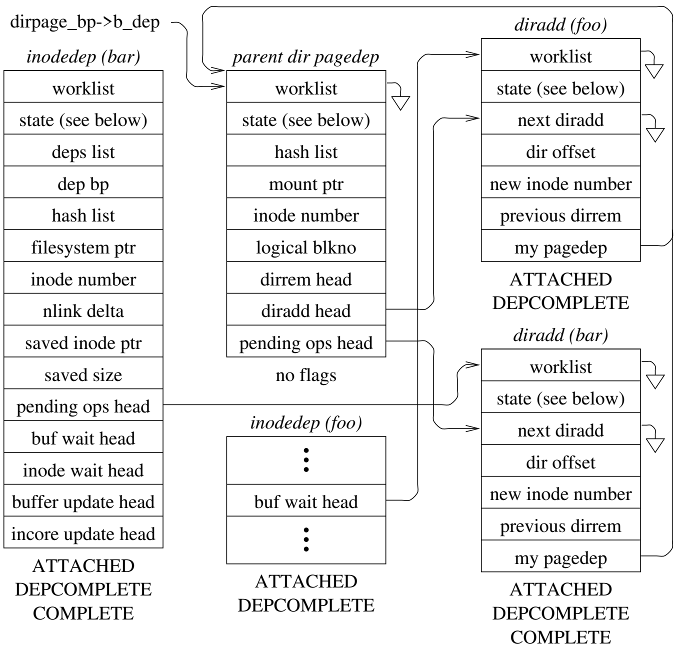

Figure 7: *Dependencies Associated with Adding New Directory Entries*

Figure 7 shows the dependency structures for a directory that has two new entries, **foo** and **bar**. This figure introduces two new dependency structures. A separate *diradd* structure tracks each individual directory entry in a directory block. The *pagedep* structure manages all the *diradd* dependencies associated with a directory block. For each new file, there is an *inodedep* structure and a *diradd* structure. Both files' inodes have had their bitmap's written to disk, as shown by the DEPCOMPLETE flags being set in their *inodedeps*. The inode for **foo** has been updated with VOP_UPDATE, but has not yet been written to disk, as shown by the COMPLETE flag on its *inodedep* structure not being set and by its *diradd* structure still being linked onto its "buf wait" list. Until the inode is written to disk with its increased link count, the directory entry may not appear on disk. If the directory page is written, the soft updates code will roll back the creation of the new directory entry for **foo** by setting its inode number to zero. After the disk write completes, the roll-back is reversed by restoring the correct inode number for **foo**.

The inode for **bar** has been written to disk, as shown by the COMPLETE flag being set in its *inodedep* and *diradd* structures. When the inode write completed, the *diradd* structure for **bar** was moved from the *inodedep* "buf wait" list to the *inodedep* "pending ops" list. The *diradd* also moved from the *pagedep* "diradd" list to the *pagedep* "pending ops" list. Since the inode has been written, it is safe to allow the directory entry to be written to disk. The *diradd* entries remain on the *inodedep* and *pagedep* "pending ops" list until the new directory entry is written to disk. When the entry is written, the *diradd* structure is freed. One reason to maintain the "pending ops" list is so that when an "fsync" system call is done on a file, the kernel is able to ensure that both the file's contents and directory reference(s) are written to disk. The kernel ensures that the reference(s) are written by doing a lookup to see if there is an *inodedep* for the inode that is the target of the "fsync". If it finds an *inodedep*, it checks to see if it has any *diradd* dependencies on either its "pending ops" or "buf wait" lists. If it finds any *diradd* structures, it follows the pointers to their associated *pagedep* structures and flushes out the directory inode associated with that *pagedep*. This back-tracking recurses on the directory *inodedep*.

3.8. New Directory Dependency Tracking

Figure 8 shows the two additional dependency structures involved with creating a new directory. For a regular file, the directory entry can be committed as soon as the newly referenced inode has been written to disk with its increased link count. When a new directory is created, there are two additional dependencies: writing the directory data block containing the "." and ".." entries (MKDIR_BODY) and writing the parent inode with the increased link count for ".." (MKDIR_PARENT). These additional dependencies are tracked by two *mkdir* structures linked to the associated *diradd* structure. The BSD soft updates design dictates that any given dependency will correspond to a single

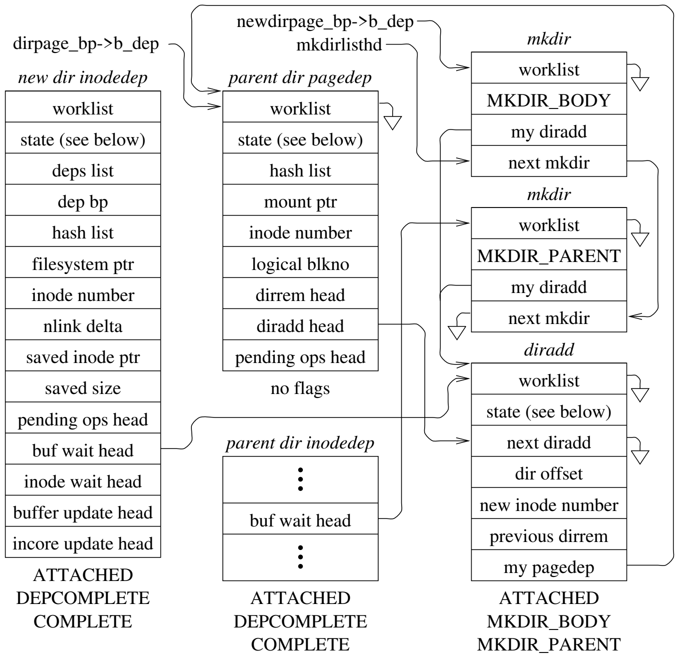

Figure 8: *Dependencies Associated with Adding a New Directory*

buffer at any given point in time. Thus, two structures are used to track the action of the two different buffers. When each completes, it clears its associated flag in the *diradd* structure. The MKDIR_PARENT is linked to the *inodedep* structure for the parent directory. When that directory inode is written, the link count will be updated on disk. The MKDIR_BODY is linked to the buffer that contains the initial contents of the new directory. When that buffer is written, the entries for "." and ".." will be on disk. The last *mkdir* to complete sets the DEPCOMPLETE flag in the *diradd* structure so that the *diradd* structure knows that these extra dependencies have been completed. Once these extra dependencies have been completed, the handling of the directory *diradd* proceeds exactly as it would for a regular file.

All *mkdir* structures in the system are linked together on a list. This list is needed so that a *diradd* can find its associated *mkdir* structures and deallocate them if it is prematurely freed (e.g., if a "mkdir" system call is immediately followed by a "rmdir" system call of the same directory). Here, the de-allocation of a *diradd* structure must traverse the list to find the associated *mkdir* structures that reference it. The deletion would be faster if the *diradd* structure were simply augmented to have two pointers that referenced the associated *mkdir* structures. However, these extra pointers would double the size of the *diradd* structure to speed an infrequent operation.

3.9. Directory Entry Removal Dependency Tracking

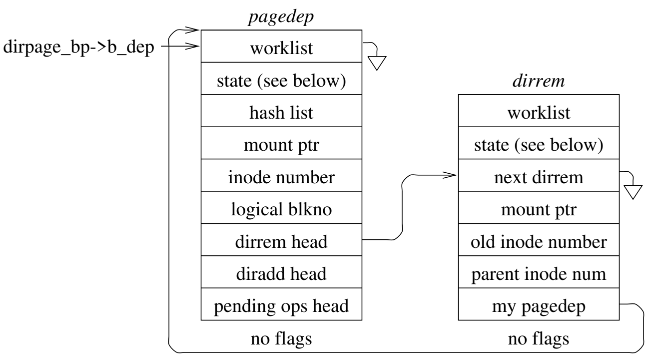

Figure 9: *Dependencies Associated with Removing a Directory Entry*

Figure 9 shows the dependency structures involved with the removal of a directory entry. This figure introduces one new dependency structure, the *dirrem* structure, and a new use for the *pagedep* structure. A separate *dirrem* structure tracks each individual directory entry to be removed in a directory block. In addition to previously described uses, *pagedep* structures associated with a directory block manage all *dirrem* structures associated with the block. After the directory block is written to disk, the *dirrem* request is added to the work daemon's *tasklist* list. For file deletions, the work daemon will decrement the inode's link count by one. For directory deletions, the work daemon will decrement the inode's link count by two, truncate its to size zero, and decrement the parent directory's link count by one. If the inode's link count drops to zero, the resource reclamation activities described in Section 3.11 are initiated.

3.10. File Truncation

In the non-soft-updates FFS, when a file is truncated to zero length, the block pointers in its inode are saved in a temporary list, the pointers in the inode are zeroed, and the inode is synchronously written to disk. When the inode write completes, the list of its formerly claimed blocks are added to the free-block bitmap. With soft updates, the block pointers in the inode being truncated are copied into a *freeblks* structure, the pointers in the inode are zeroed, and the inode is marked dirty. The *freeblks* structure is added to the ''inode wait'' list, and it migrates to the ''buf wait'' list when VOP_UPDATE is called. The *freeblks* structure is eventually added to the *tasklist* after the buffer holding the inode block has been written to disk. When the *tasklist* is serviced, the blocks listed in the *freeblks* structure are returned to the free-block bitmap.

3.11. File and Directory Inode Reclamation

When the link count on a file or directory drops to zero, its inode is zeroed to indicate that it is no longer in use. In the non-soft-updates FFS, the zeroed inode is synchronously written to disk and the inode is marked as free in the bitmap. With soft updates, information about the inode to be freed is saved in a *freefile* structure. The *freefile* structure is added to the ''inode wait'' list, and it migrates to the ''buf wait'' list when VOP_UPDATE is called. The *freefile* structure eventually is added to the *tasklist* after the buffer holding the inode block has been written to disk. When the *tasklist* is serviced, the inode listed in the *freefile* structure is returned to the free inode map.

3.12. Directory Entry Renaming Dependency Tracking

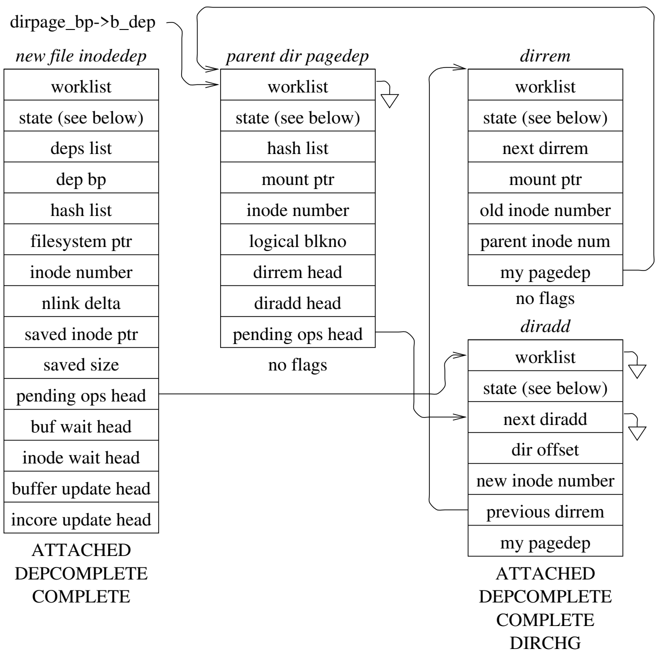

Figure 10: *Dependencies Associated with Renaming a Directory Entry*

Figure 10 shows the structures involved in renaming a file. The dependencies follow the same series of steps as those for adding a new file entry, with two variations. First, when a roll-back of an entry is needed because its inode has not yet been written to disk, the entry must be set back to the previous inode number rather than to zero. The previous inode number is stored in a *dirrem* structure. The DIRCHG flag is set in the *diradd* structure so that the roll-back code knows to use the old inode number stored in the *dirrem* structure. The second variation is that, after the modified directory entry is written to disk, the *dirrem* structure is added to the work daemon's *tasklist* list so that the link count of the old inode will be decremented as described in Section 3.9.

4. Experiences and Lessons Learned

This section describes various issues that arose in moving soft updates from research prototype to being a production-quality component of the 4.4BSD operating system. Some of these issues were evident shortcomings of the research prototype, and some were simply the result of differences in the host operating systems. Others, however, only became evident as we gained operational experience with soft updates.

The "fsync" system call. An important filesystem interface is accessed through the ‘‘fsync’’ system call. This system call requests that the specified file be written to stable storage and that the system call not return until all its associated writes have completed. The prototype soft update implementation did not implement the ‘‘fsync’’ system call.

The task of completing an ‘‘fsync’’ requires more than simply writing all the file's dirty data blocks to disk. It also requires that any unwritten directory entries that reference the file also be written, as well as any unwritten directories between the file and the root of the filesystem. Simply getting the data blocks to disk can be a major task. First, the system must check to see if the bitmap for the inode has been written, finding the bitmap and writing it if necessary. It must then check for, find, and write the bitmaps for any new blocks in the file. Next, any unwritten data blocks must go to disk. Following the data blocks, any first level indirect blocks that have newly allocated blocks in them are written, followed by any double indirect blocks, then triple level indirect blocks. Finally, the inode can be written which will ensure that the contents of the file are on stable store. Ensuring that all names for the file are also on stable store requires data structures that can determine whether there are any uncommitted names and if so, in which directories they occur. For each directory containing an uncommitted name, the soft update code must go through the same set of flush operations that it has just done on the file itself.

Although the ‘‘fsync’’ system call must ultimately be done synchronously, this does not mean that the flushing operations must each be done synchronously. Instead, whole sets of bitmaps or data blocks are pushed into the disk queue and the soft update code then waits for all the writes to complete. This approach is more efficient because it allows the disk subsystem to sort all the write requests into the most efficient order for writing. Still, the ‘‘fsync’’ part of the soft update code generates most of the remaining synchronous writes in the filesystem.

Unmounting filesystems. Another issue related to ‘‘fsync’’ is unmounting of filesystems. Doing an **unmount** requires finding and flushing all the dirty files that are associated with the filesystem. Flushing the files may lead to the generation of background activity such as removing files whose reference count drops to zero as a result of their nullified directory entries being written. Thus, the system must be able to find all background activity requests and process them. Even on a quiescent filesystem, several iterations of file flushes followed by background activity may be required. The 4.4BSD FFS allows for forcible unmounts of filesystems which allows the unmount to take place while the filesystem is actively in use, which required additional support.

Removing directories. The prototype implementation oversimplified the sequencing of updates involved with removing a directory. Specifically, the prototype allowed the removal of the directory's name and the removal of its ".." entry to proceed in parallel. This meant that a crash could leave the directory in place with its ".." entry removed. Although **fsck** could be modified to repair this problem, it is not acceptable when **fsck** is bypassed during crash recovery.

For correct operation, a directory's ".." entry should not be removed until after the directory is persistently unlinked. Correcting this in the soft updates code introduced a delay of up to two minutes between unlinked a directory and its really being deallocated (when the ".." entry is removed). Until the directory's ".." entry is really removed, the link count on its parent will not be decremented. Thus, when a user removed one or more directories, the the link count of their former parent still reflected their being present for several minutes. This delayed link count decrement not only caused some questions from users, but also caused some applications to break. For example, the ‘‘rmdir’’ system call will not remove a directory that has a link count over two. This restriction means that a directory that recently had directories removed from it cannot be removed until its former directories have been fully deleted.

To fix these link-count problems, the BSD soft updates implementation augments the inode ‘‘nlink’’ field with a new field called ‘‘effnlink’’. The ‘‘nlink’’ field is still stored as part of the on-disk metadata and reflects the true link count of the inode. The ‘‘effnlink’’ field is maintained only in kernel memory and reflects the final value that the ‘‘nlink’’ field will reach once all its outstanding operations are completed. All interactions with user applications report the value of the ‘‘effnlink’’ field which results in the illusion that everything has happened immediately.

Block Reallocation. Because it was not done in System V Release 4 UNIX, the prototype system did not handle block reallocation. In the 4.4BSD FFS, the filesystem sometimes changes the on-disk locations of file blocks as a file grows to lay the file out more contiguously. Thus, a block that is initially assigned to a file may be replaced as the file grows larger. Although the prototype code was prepared to handle upgrades of fragments to full-sized blocks in the last block of a file, it was not prepared to have full-sized blocks reallocated at interior parts of inodes and indirect blocks.

Memory used for Dependency Structures. One concern with soft updates is the amount of memory consumed by the dependency structures. This problem was attacked on two fronts: memory efficiency and usage bounding.

The prototype implementation generally used two structures for each update dependency. One was associated with the data that needed to be written and one with the data that depended on the write. For example, each time a new block was allocated, new dependency structures were associated with the allocated disk block, the bitmap from which the block was allocated, and the inode claiming the new disk block. The structure associated with the inode was dependent on the other two being written. The 4.4BSD soft updates code uses a single dependency structure (associated with the disk block) to describe a block allocation. There is a single dependency structure associated with each bitmap and each inode, and all related block allocation structures are linked into lists headed by these structures. That is, one block allocation structure is linked into the allocated block’s list, the bitmap’s list, and the inode’s list. By constructing lists rather than using separate structures, the demand on memory was reduced by about 40 percent.

In daily operation, we have found that the additional dynamic memory load placed on the kernel memory allocation area is about equal to the amount of memory used by vnodes plus inodes. For a system with 1000 vnodes, the additional peak memory load is about 300KB. The one exception to this guideline occurs when large directory trees are removed. Here, the filesystem code can get arbitrarily far ahead of the on-disk state, causing the amount of memory dedicated to dependency structures to grow without bound. The 4.4BSD soft update code was modified to monitor the memory load for this case and not allow it to grow past a tunable upper bound. When the bound is reached, new dependency structures can only be created at the rate at which old ones are retired. The effect of this limit is to slow down the rate of removal to the rate at which the disk updates can be done. While this restriction slows the rate at which soft updates can normally remove files, it is still considerably faster than the traditional synchronous write filesystem. In steady-state, the soft update remove algorithm requires about one disk write for each ten files removed while the traditional filesystem requires at least two writes for every file removed.

The fsck Utility. As with the dual tracking of the true and effective link count, the changes needed to **fsck** became evident through operational experience. In a non-soft-updates filesystem implementation, file removal happens within a few milliseconds. Thus, there is a short period of time between the directory entry being removed and the inode being deallocated. If the system crashes during a bulk tree removal operation, there are usually no inodes lacking references from directory entries, though in rare instances there may be one or two. By contrast, in a system running with soft updates, many seconds may elapse between the times when the directory entry is deleted and the inode is deallocated. If the system crashes during a bulk tree removal operation, there are usually tens to hundreds of inodes lacking references from directory entries. Historically, **fsck** placed any unreferenced inodes into the **lost+found** directory. This action is reasonable if the filesystem has been damaged because of disk failure which results in the loss of one or more directories. However, it results in the incorrect action of stuffing the **lost+found** directory full of partially deleted files when running with soft updates. Thus, the **fsck** program must be modified to check that a filesystem is running with soft updates and clear out, rather than saving, unreferenced inodes (unless it has determined that unexpected damage has occurred to the filesystem, in which case the files are saved in **lost+found**).

A peripheral benefit of soft updates is that **fsck** can trust the allocation information in the bitmaps. Thus, it only needs to check the subset of inodes in the filesystem that the bitmaps indicate are in use. Although some of the inodes marked "in use" may be free, none of those marked free will ever be in use.

Partial File Truncation. Although the common case for deallocation is for all data in a file to be deleted, the ‘‘truncate’’ system call allows applications to delete only part of a file. This creates slightly more complicated update dependencies, including the need to have deallocation dependencies for indirect blocks and the need to consider partially deleted data blocks. Because it is so uncommon, neither the prototype nor the BSD soft updates implementation optimizes this case; the conventional synchronous write approach is used instead.

5. Performance

This paper gives only a cursory look at soft updates performance. For a detailed analysis, including comparisons with other techniques, see [Ganger, McKusick, & Patt,].

We place the performance of BSD FFS with soft updates in context by comparing it to the default BSD FFS (referred to below as "normal"), which uses synchronous writes for update ordering, and BSD FFS mounted with the O_ASYNC option (referred to below as "asynchronous"), which ignores all update dependencies. In asynchronous mode, all metadata updates are converted into delayed writes (a delayed write is one in which the buffer is simply marked dirty, put on a least-recently-used list, and not written until needed for some other purpose). Thus, the O_ASYNC data provides an upper bound on the performance of an update ordering scheme in BSD FFS. As expected, we have found that soft updates eliminates almost all synchronous writes and usually allows BSD FFS to achieve performance with 5 percent of the upper bound. Compared to using synchronous writes, file creation and removal performance increases by factors of 2 and 20, respectively. Overall, 4.4BSD systems tend to require 40 percent fewer disk writes and complete tasks 25 percent more quickly than when using the default 4.4BSD fast filesystem implementation.

To provide a feeling for how the system performs in normal operation, we present measurements from three different system tasks. The first task is our ‘‘filesystem torture test’’. This consists of 1000 runs of the Andrew benchmark, 1000 copy and removes of **/etc** with randomly selected pauses of 0-60 seconds between each copy and remove, and 500 executions of the **find** application from **/** with randomly selected pauses of 100 seconds between each run. The run of the torture test compares as follows:

Filesystem Configuration	Disk Writes		Running Time
	Sync	Async	
Normal	1,459,147	487,031	27hr, 15min
Asynchronous	0	1,109,711	19hr, 43min
Soft Updates	6	1,113,686	19hr, 50min

The overall result is that asynchronous and soft updates require 42 percent fewer writes (with almost no synchronous writes) and have a 28 percent shorter running time. This is particularly impressive when one considers that the finds and the pauses involve no update dependencies, and the Andrew benchmark is largely CPU bound.

The second test consists of building and installing the FreeBSD system. This task is a real-world example of a program development environment. The results are as follows:

Filesystem Configuration	Disk Writes		Running Time
	Sync	Async	
Normal	162,410	39,924	2hr, 12min
Asynchronous	0	38,262	1hr, 44min
Soft Updates	1124	48,850	1hr, 44min

The overall result is that soft updates require 75 percent fewer writes and has a 21 percent shorter running time. Although soft updates initiates 30 percent more writes than asynchronous, the two result in the same running time.

The third test compares the performance of the central mail server for Berkeley Software Design, Inc. run with and without soft updates. The administrator was obviously unwilling to run it in asynchronous mode, since it is a production machine and people will not abide by losing their mail. Unlike the tests above, which involve a single disk, the mail spool on this system is striped across three disks. The statistics were gathered by averaging the results from thirty days of non-weekend operation in each mode. The results for a 24-hour period are as follows:

Filesystem Configuration	Disk Writes	
	Sync	Async
Normal	1,877,794	1,613,465
Soft Updates	118,102	946,519

The normal filesystem averaged over 40 writes per second with a ratio of synchronous to asynchronous writes of 1:1. With soft updates, the write rate dropped to 12 per second and the ratio of synchronous to asynchronous writes dropped to 1:8. For this real-world application, soft updates requires 70 percent fewer writes, which triples the mail handling capacity of the machine.

6. Filesystem Snapshots

A filesystem **snapshot** is a frozen image of a filesystem at a given instant in time. Snapshots support several important features: the ability to provide back-ups of the filesystem at several times during the day, the ability to do reliable dumps of live filesystems, and (most important for soft updates) the ability to run a filesystem check program on a active system to reclaim lost blocks and inodes.

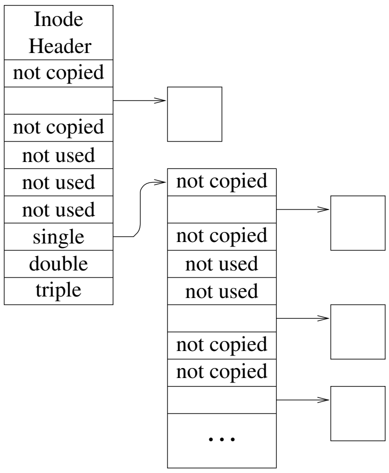

Figure 11: *Structure of a snapshot file*

Implementing snapshots in BSD FFS has proven to be straightforward, with the following steps. First, activity on the relevant filesystem is briefly suspended. Second, all system calls currently writing to that filesystem are allowed to finish. Third, the filesystem is synchronized to disk as if it were about to be unmounted. Finally, a **snapshot file** is created to track subsequent changes to the filesystem; a snapshot file is shown in Figure 11. This snapshot file is initialized to the size of the filesystem's partition, and most of its file block pointers are marked as ''not copied''. A few strategic blocks are allocated and copied, such as those holding copies of the superblock and cylinder group maps. The snapshot file also uses a distinguished block number (1) to mark all blocks ''not used'' at the time of the snapshot, since there is no need to copy those blocks if they are later allocated and written.

Once the snapshot file is in place, activity on the filesystem resumes. Each time an existing block in the filesystem is modified, the filesystem checks whether that block was in use at the time that the snapshot was taken (i.e., it is not marked ''not used''). If so, and if it has not already been copied (i.e., it is still marked ''not copied''), a new block is allocated and placed in the snapshot file to replace the ''not copied'' entry. The previous contents of the block are copied to the newly allocated snapshot file block, and the modification to the original is then allowed to proceed. Whenever a file is removed, the snapshot code inspects each of the blocks being freed and claims any that were in use at the time of the snapshot. Those blocks marked ''not used'' are returned to the free list.

When a snapshot file is read, reads of blocks marked ''not copied'' return the contents of the corresponding block in the filesystem. Reads of blocks that have been copied return their contents. Writes to snapshot files are not permitted. When a snapshot file is no longer needed, it can be removed in the same way as any other file; its blocks are simply returned to the free list and its inode is zeroed and returned to the free inode list.

Snapshots may live across reboots. When a snapshot file is created, the inode number of the snapshot file is recorded in the superblock. When a filesystem is mounted, the snapshot list is traversed and all the listed snapshots are activated. The only limit on the number of snapshots that may exist in a filesystem is the size of the array in the superblock that holds the list of snapshots. Currently, this array can hold up to twenty snapshots.

Multiple snapshot files can concurrently exist. As described above, earlier snapshot files would appear in later snapshots. If an earlier snapshot is removed, a later snapshot would claim its blocks rather than allowing them to be returned to the free list. This semantic means that it would be impossible to free any space on the filesystem except by removing the newest snapshot. To avoid this problem, the snapshot code carefully goes through and expunges all earlier snapshots by changing its view of them to being zero length files. With this technique, the freeing of an earlier snapshot releases the space held by that snapshot.

When a block is overwritten, all snapshots are given an opportunity to copy the block. A copy of the block is made for each snapshot in which the block resides. Deleted blocks are handled differently. The list of snapshots is consulted. When a snapshot is found in which the block is active (''not copied''), the deleted block is claimed by that snapshot. The traversal of the snapshot list is then terminated. Other snapshots for which the block are active are left with an entry of ''not copied'' for that block. The result is that when they access that location they will still reference the deleted block. Since snapshots may not be written, the block will not change. Since the block is claimed by a snapshot, it will not be allocated to another use. If the snapshot claiming the deleted block is deleted, the remaining snapshots will be given the opportunity to claim the block. Only when none of the remaining snapshots want to claim the block (i.e., it is marked ''not used'' in all of them) will it be returned to the freelist.

6.1. Instant Filesystem Restart

Traditionally, after an unclean system shutdown, the filesystem check program, **fsck**, has had to be run over all inodes in an FFS filesystem to ascertain which inodes and blocks are in use and correct the bitmaps.

This is a painfully slow process that can delay the restart of a big server for an hour or more. The current implementation of soft updates guarantees the consistency of all filesystem resources, including the inode and block bitmaps. With soft updates, the only inconsistency that can arise in the filesystem (barring software bugs and media failures) is that some unreferenced blocks may not appear in the bitmaps and some inodes may have to have overly high link counts reduced. Thus, it is completely safe to begin using the filesystem after a crash without first running **fsck**. However, some filesystem space may be lost after each crash. Thus, there is value in having a version of **fsck** that can run in the background on an active filesystem to find and recover any lost blocks and adjust inodes with overly high link counts. A special case of the overly high link count is one that should be zero. Such an inode will be freed as part of reducing its link count to zero. This garbage collection task is less difficult than it might at first appear, since this version of **fsck** only needs to identify resources that are not in use and cannot be allocated or accessed by the running system.

With the addition of snapshots, the task becomes simple, requiring only minor modifications to the standard **fsck**. When run in background cleanup mode, **fsck** starts by taking a snapshot of the filesystem to be checked. **Fsck** then runs over the snapshot filesystem image doing its usual calculations just as in its normal operation. The only other change comes at the end of its run, when it wants to write out the updated versions of the bitmaps. Here, the modified **fsck** takes the set of blocks that it finds were in use at the time of the snapshot and removes this set from the set marked as in use at the time of the snapshot—the difference is the set of lost blocks. It also constructs the list of inodes whose counts need to be adjusted. **Fsck** then calls a new system call to notify the filesystem of the identified lost blocks so that it can replace them in its bitmaps. It also gives the set of inodes whose link counts need to be adjusted; those inodes whose link count is reduced to zero are truncated to zero length and freed. When **fsck** completes, it releases its snapshot.

6.2. User Visible Snapshots

Snapshots may be taken at any time. When taken every few hours during the day, they allow users to retrieve a file that they wrote several hours earlier and later deleted or overwrote by mistake. Snapshots are much more convenient to use than dump tapes and can be created much more frequently.

The snapshot described above creates a frozen image of a filesystem partition. To make that snapshot accessible to users through a traditional filesystem interface, BSD uses the vnode driver, **vnd**. The **vnd** driver takes a file as input and produces a block and character device interface to access it. The **vnd** block device can then be used as the input device for a standard BSD FFS mount command, allowing the snapshot to appear as a replica of the frozen filesystem at whatever location in the namespace that the system administrator chooses to mount it.

6.3. Live Dumps

Once filesystem snapshots are available, it becomes possible to safely dump live filesystems. When **dump** notices that it is being asked to dump a mounted filesystem, it can simply take a snapshot of the filesystem and run over the snapshot instead of on the live filesystem. When **dump** completes, it releases the snapshot.

7. Current Status

The soft updates code is available for commercial use in Berkeley Software Design Inc.'s BSD/OS 4.0 and later systems. It is available for non-commercial use in the freely-available BSD systems: FreeBSD, NetBSD, and OpenBSD. The snapshot code is in alpha test and should be available in the BSD systems towards the end of 1999. Sun Microsystems has been evaluating the soft updates and snapshot technology for possible inclusion in Solaris. Vendors wishing to use soft updates for commercial use in a freely-available BSD or in their own products should visit http://www.mckusick.com/softdep/ or contact Dr. McKusick.

References

Chamberlin et al, 1981.
D. Chamberlin, M. Astrahan, & et al., "A History and Evaluation of System R," *Communications of the ACM,* 24, 10, p. 632−646 (1981).

Chao et al, 1992.
C. Chao, R. English, D. Jacobson, A. Stepanov, & J. Wilkes, *Mime: A High-Performance Parallel Storage Device with Strong Recovery Guarantees,* Hewlett-Packard Laboratories Report, HPL-CSP-92-9 rev 1 (November 1992).

Chutani et al, 1992.
S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, & R. Sidebotham, "The Episode File System," *Winter USENIX Conference,* p. 43−60 (January 1992).

Ganger, McKusick, & Patt, .
G. Ganger, M. McKusick, & Y. Patt, "Soft Updates: A Solution to the Metadata Update Problem in Filesystems," *ACM Transactions on Computer Systems* (in preparation).

Ganger & Patt, 1994.
G. Ganger & Y. Patt, "Metadata Update Performance in File Systems," *USENIX Symposium on Operating Systems Design and Implementation,* p. 49–60 (November 1994).

Hagmann, 1987.
R. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," *ACM Symposium on Operating Systems Principles,* p. 155–162 (November 1987).

McKusick et al, 1996.
M. McKusick, K. Bostic, M. Karels, & J. Quarterman,, *The Design and Implementation of the 4.4BSD Operating System,* p. 269–271, Addison Wesley Publishing Company, Reading, MA (1996).

McKusick et al, 1984.
M. McKusick, W. Joy, S. Leffler, & R. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems,* 2, 3, p. 181–197 (August 1984).

McKusick & Kowalski, 1994.
M. McKusick & T. Kowalski, "FSCK - The UNIX File System Check Program," *4.4 BSD System Manager's Manual,* p. 3:1–21, O'Reilley & Associates, Inc., Sebastopol, CA (1994).

McVoy & Kleiman, 1991.
L. McVoy & S. Kleiman, "Extent-like Performance From a UNIX File System," *Winter USENIX Conference,* p. 1–11 (January 1991).

NCR_Corporation, 1992.
NCR_Corporation, *Journaling File System Administrator Guide, Release 2.00,* NCR Document D1-2724-A (April 1992).

Ousterhout, 1990.
J. Ousterhout, "Why Aren't Operating Systems Getting faster As Fast As Hardware?," *Summer USENIX Conference,* p. 247–256 (June 1990).

Rosenblum & Ousterhout, 1991.
M. Rosenblum & J. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Symposium on Operating System Principles,* p. 1–15 (October 1991).

Seltzer et al, 1993.
M. Seltzer, K. Bostic, M. McKusick, & C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Winter USENIX Conference,* p. 201–220 (January 1993).

Stonebraker, 1987.
M. Stonebraker, "The Design of the POSTGRES Storage System," *Very Large DataBase Conference,* p. 289–300 (1987).

Wu & Zwaenepoel, 1994.
M. Wu & W. Zwaenepoel, "eNVy: A Non-Volatile, Main Memory Storage System," *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* p. 86–97 (October 1994).

8. Biographies

Dr. Marshall Kirk McKusick writes books and articles, consults, and teaches classes on UNIX- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast filesystem, and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD. His particular areas of interest are the virtual-memory system and the filesystem. One day, he hopes to see them merged seamlessly. He earned his undergraduate degree in Electrical Engineering from Cornell University, and did his graduate work at the University of California at Berkeley, where he received Masters degrees in Computer Science and Business Administration, and a doctoral degree in Computer Science. He is a past president of the Usenix Association, and is a member of ACM and IEEE.

In his spare time, he enjoys swimming, scuba diving, and wine collecting. The wine is stored in a specially constructed wine cellar (accessible from the web at http://www.mckusick.com/˜mckusick/) in the basement of the house that he shares with Eric Allman, his domestic partner of 20-and-some-odd years. You can contact him via email at `<mckusick@mckusick.com>`.

Greg Ganger is an assistant professor of Electrical and Computer Engineering and Computer Science at Carnegie Mellon University. He has broad research interests in computer systems, including operating systems, networking, storage systems, computer architecture, performance evaluation and distributed systems. He also enjoys working on the occasional filesystem project. He spent 2+ years at MIT working on the exokernel operating system and related projects as part of the Parallel and Distributed Operating Systems group. He earned his various degrees (B.S, M.S, and Ph.D.) from the University of Michigan. He is a member of ACM and IEEE Computer Society. Greg never seems to have spare time, but does very much enjoy a good game of basketball. You can contact him via email at `<ganger@ece.cmu.edu>`.