Soft Updates: A Solution to the Metadata Update Problem in File Systems

GREGORY R. GANGER
Carnegie Mellon University
MARSHALL KIRK McKUSICK
McKusick.com
CRAIG A. N. SOULES
Carnegie Mellon University
and
YALE N. PATT
University of Texas, Austin

Metadata updates, such as file creation and block allocation, have consistently been identified as a source of performance, integrity, security, and availability problems for file systems. **Soft updates** is an implementation technique for low-cost sequencing of fine-grained updates to write-back cache blocks. Using soft updates to track and enforce metadata update dependencies, a file system can safely use delayed writes for almost all file operations. This article describes soft updates, their incorporation into the 4.4BSD fast file system, and the resulting effects on the system. We show that a disk-based file system using soft updates achieves memory-based file system performance while providing stronger integrity and security guarantees than most disk-based file systems. For workloads that frequently perform updates on metadata (such as creating and deleting files), this improves performance by more than a factor of two and up to a factor of 20 when compared to the conventional synchronous write approach and by 4–19% when compared to an aggressive write-ahead logging approach. In addition, soft updates can improve file system availability by relegating crash-recovery assistance (e.g., the *fsck* utility) to an optional and background role, reducing file system recovery time to less than one second.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Design studies*; *Reliability, availability, and serviceability*; C.5.5 [**Computer System Implementation**]: Servers; D.4.2 [**Operating Systems**]: Storage Management; D.4.3 [**Operating Systems**]: File Systems Management; E.5 [**Data**]: Files; H.3.2 [**Information Storage and Retrieval**]: Information Storage

Authors' addresses: G. R. Ganger, Department of ECE, Carnegie Mellon University, Pittsburgh, PA 15217; email: greg.ganger@cmu.edu; M. K. McKusick, McKusick.com, 1614 Oxford Street, Berkeley, CA 94709-1608; email: mckusick@mckusick.com; C. A. N. Soules, Carnegie Mellon University, Pittsburgh, PA 15217; email: soules+@andrew.cmu.edu; Y. N. Patt, University of Texas, Austin, Austin, TX; email: patt@ece.utexas.edu..

General Terms: Performance, Reliability, Design, Experimentation, Measurement, Security

Additional Key Words and Phrases: File systems, metadata, integrity, failure recovery, disk caches, write-back caching, delayed writes

1. INTRODUCTION

In file systems, **metadata** (e.g., directories, inodes, and free block maps) gives structure to raw storage capacity. Metadata consists of pointers and descriptions for linking multiple disk sectors into files and identifying those files. To be useful for persistent storage, a file system 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 nonvolatile storage (i.e., disk) must always be consistent enough to deterministically reconstruct a coherent file system state. Specifically, the on-disk image of the file system must have no dangling pointers to uninitialized space, no ambiguous resource ownership caused by multiple pointers, and no live resources to which there are no pointers. Maintaining these invariants generally requires sequencing (or atomic grouping) of updates to small on-disk metadata objects.

From a performance standpoint, recent and predicted future technology trends result in a growing disparity between processing performance and disk access times. This disparity, combined with increasing main memory sizes, dictates that high-performance file systems aggressively employ caching techniques to avoid disk accesses and hide disk latencies. For metadata updates, which are characterized by strong spatial and temporal locality and by small sizes relative to the units of disk access, this means write-back caching. Write-back caching can substantially improve metadata update performance by combining multiple updates into a much smaller quantity of background disk writes. The savings come from two sources: multiples updates to a single metadata component (e.g., removal of a recently added directory entry) and multiple independent updates to a single block of metadata (e.g., several entries added to a directory block).

This article describes **soft updates**, an implementation technique for low-cost sequencing of fine-grained updates to write-back cache blocks [Ganger and Patt 1994]. The soft updates mechanism tracks dependencies among updates to cached (i.e., in-memory) copies of metadata and enforces these dependencies, via update sequencing, as the dirty metadata blocks are written back to nonvolatile storage. 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 and 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 afterward. 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.

With soft updates, the cost of maintaining integrity is low, and disk-based file system performance can be within a few percent of a memory-based file system's performance. For workloads that frequently perform updates on metadata, this improves performance by more than a factor of two (and up to a factor of 20) when compared to the conventional approach and by 4–19% when compared to an aggressive write-ahead logging approach. Also, additional update sequencing can be realized with little performance loss. So, integrity and security can be improved relative to many current implementations. Further, the on-disk state can be maintained such that the file system can be safely mounted and used immediately (without preuse consistency checking, such as the **fsck** utility [McKusick and Kowalski 1994]) after any system failure other than media corruption. This reduces file system recovery times by more than two orders of magnitude (to less than one second) when compared to fsck-like approaches.

Initially proposed and evaluated by Ganger and Patt [1994], soft updates has now been incorporated into the 4.4BSD fast file system (FFS) used in the NetBSD, OpenBSD, FreeBSD, and BSDI operating systems [McKusick and Ganger 1999]. This article briefly describes the incorporation. In doing so, it discusses experiences and lessons learned, including some not anticipated in the original research paper, and shows that achieved performance matches expectations. Specifically, using soft updates in BSD FFS eliminates most synchronous writes by allowing safe use of delayed writes for metadata updates. This results in significant performance increases, and, in most cases, the soft updates implementation is within 5% of ideal (i.e., the same file system with no update ordering). Further, soft updates allows BSD FFS to provide cleaner semantics, stronger integrity and security guarantees, and immediate crash recovery at the same time.

The remainder of this article is organized as follows. Section 2 describes the metadata update problem in more detail, discusses previous solutions, and outlines the characteristics of an ideal solution. Section 3 describes soft updates in general, the implementation of soft updates in the 4.4BSD FFS, some lessons learned from two implementations (the original research prototype and the production-quality 4.4BSD module), and the effects of soft updates on file system availability, semantics, and complexity. Then, Section 4 evaluates the soft updates implementation, comparing it to the default 4.4BSD FFS, a version that uses delayed writes for all updates, and a version that uses write-ahead logging. Section 5 summarizes the article's contributions.

2. THE METADATA UPDATE PROBLEM

Several important file system 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 file system 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 reuse 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 reallocated for a new inode).

(3) Never reset the last pointer to a live resource before a 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).

The metadata update problem can be addressed with several mechanisms. The remainder of this section discusses previous approaches and the characteristics of an ideal solution.

2.1 Previous Solutions

Synchronous Writes. Synchronous writes are used for metadata update sequencing by many file systems, including the VMS file system [McCoy 1990], the DOS file system [Duncan 1986], and most variants of the UNIX file systems [Ritchie and Thompson 1978; McKusick et al. 1984]. As a result, metadata updates in these file systems proceed at disk speeds rather than processor/memory speeds [Ousterhout 1990; McVoy and Kleiman 1991; Seltzer et al. 1993]. The performance degradation can be so dramatic that many implementations choose to ignore certain update dependencies, thereby reducing integrity, security, and availability. For example, many file systems do not initialize a newly allocated disk block before attaching it to a file, which can reduce both integrity and security, since an uninitialized block often contains previously deleted file data. Also, many file systems do not protect the consistency of the on-disk free block/inode maps, electing to reconstruct them after a system failure (e.g., with the *fsck* utility [McKusick and Kowalski 1994]).

Nonvolatile RAM (NVRAM). To eliminate the need to keep the on-disk state consistent, one can employ NVRAM technologies, such as an uninterruptable power supply for the entire system or a distinct Flash RAM device [Wu and 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. The performance of this approach

far exceeds that of synchronous writes, since ordering updates to RAM is much less time-consuming. In fact, in addition to eliminating the need for update ordering, NVRAM eliminates the need for periodic syncer daemon activity; write-backs are only required when space needs to be reclaimed. There is a concern that failing operating systems may destroy some of the contents of the NVRAM cache, but this problem can be prevented with a reasonable amount of effort and overhead [Chen et al. 1996]. The main drawbacks, of course, are the cost of NVRAM and the fact that it is only a solution for systems that are actually equipped with it. In addition, file systems that rely on distinct NVRAM devices incur additional overheads for moving data to and from them. Finally, file system recovery after a system crash requires both the NVRAM and the disk contents, which means that it is not possible to just move one component or the other from a broken system to a working one. With soft updates, NVRAM-like performance can be achieved without the extra hardware expenses.

Atomic Updates. Although update sequencing will maintain file system integrity, an alternative approach is to group each set of dependent updates as an atomic operation. Most implementations of storage update atomicity entail some form of write-ahead logging [Hagmann 1987; Chutani et al. 1992; NCR 1992] or shadow-paging [Chamberlin et. al. 1981; Stonebraker 1987; Chao et al. 1992; Rosenblum and Ousterhout 1992]. 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 file systems successfully employ write-ahead logging to improve performance compared to the synchronous write approach. However, there is still value in exploring implementations that do not require changes to the on-disk structures (which may have a large installed base) and may offer higher performance with lower complexity. In particular, this article and Ganger and Patt [1994] both show that a file system augmented with soft updates requires minimal changes to the file system proper and can deliver performance almost equivalent to having no update ordering at all. The same has not been shown for approaches based on update atomicity, and Section 4.5 indicates that logging can involve a 4–19% performance degradation.

Scheduler-Enforced Ordering. With appropriate support in disk request schedulers, a file system can use asynchronous writes for metadata and pass any sequencing restrictions to the disk scheduler with each request [Ganger and Patt 1994]. This approach has been shown to outperform the conventional synchronous write implementation by more than 30% for workloads that frequently do updates on metadata. However, with such scheduler-enforced ordering, delayed writes cannot safely be used when sequencing is required, since a disk request scheduler cannot enforce an ordering on or prevent dependency cycles among requests not yet visible to it. Also, all disk schedulers, which are generally located in disk device drivers or disk drives, must support the modified interface and the corresponding sequencing rules.

Interbuffer Dependencies. Another approach is to use delayed writes for all updates and have the cache write-back code enforce an ordering on disk writes. Tracking dependencies among buffers is straightforward, but this approach provides only a marginal reduction in the number of synchronous writes. The lack of improvement occurs because the system must avoid the creation of circular dependencies. Whenever a circular dependency is about to be created, the system must prevent the circularity (e.g., by doing a synchronous write). Unfortunately, circular dependencies quickly arise in the normal course of file system operation. For example, consider a file creation and a file deletion performed in the same directory. The file creation requires that the inode block be written before the directory. The file deletion requires that the directory be written before the inode block. For correct operation, this scenario must revert to the use of synchronous writes or some other update ordering mechanism.

2.2 Characteristics of an Ideal Solution

An ideal solution to the metadata update problem would provide immediate stability and consistency of all metadata updates with no restrictions on on-disk data organization, no performance overhead, and no special hardware support. Unfortunately, to our knowledge, no such solution exists. One must therefore choose to relax the constraints in one or more of these areas. For general-purpose file systems, we believe that consistency is not negotiable and that requirements for special hardware support should be avoided. In many environments, users are willing to compromise immediate stability and live with a small window of vulnerability for new data (e.g., 30 seconds) in order to achieve much higher performance. With these assumptions, we wish to find a software-only implementation of consistent metadata updates with the smallest possible performance penalty given a small write-back window. In approximate order of importance, the performance-related characteristics of an ideal solution are:

(1) Applications should never wait for disk writes unless they explicitly choose to do so for application-specific purposes.

(2) The system should propagate modified metadata to disk using the minimum possible number of disk writes, given the allowed window of vulnerability. Specifically, this requires aggressive write-back caching of metadata structures to absorb and coalesce writes.

(3) The solution should minimize the amount of main memory needed to cache dirty metadata and related auxiliary information. This will maximize the availability of memory for other purposes.

(4) The cache write-back code and the disk request scheduler should not be constrained in choosing what blocks to write to disk when, beyond the minimal restrictions necessary to guarantee consistency. This flexibility is important for scheduling algorithms that reduce mechanical positioning delays [Denning 1967; Worthington et al. 1994].

Soft updates provides a reasonable approximation of this ideal.

3. SOFT UPDATES

This section describes the soft updates mechanism. It consists of an overview of the approach, a description of an implementation of soft updates in a UNIX file system, and discussion of the impact of soft updates on availability, semantics, and complexity.

3.1 Overview

The soft updates mechanism allows safe use of write-back caching for metadata blocks. As discussed earlier, this improves performance by combining multiple metadata updates into a much smaller quantity of background disk writes. However, to maintain integrity in the face of unpredictable failures, sequencing constraints must be upheld as dirty blocks are propagated to stable storage. To address this requirement, the soft updates mechanism maintains dependency information, associated with any dirty in-memory copies of metadata, to keep track of sequencing requirements. When performing a metadata update, the in-memory copy of the relevant block is modified normally, and the corresponding dependency information is updated appropriately. The dependency information is then consulted when dirty blocks are flushed to disk.

When we began this work, we envisioned a dynamically managed DAG (Directed, Acyclic Graph) of dirty blocks for which disk writes are issued only after all writes on which they depend complete. In practice, we found this to be a very difficult model to maintain, being susceptible to cyclic dependencies and aging problems (e.g., blocks could consistently have dependencies and never be written to stable storage). Like false sharing in multiprocessor caches, these difficulties relate to the granularity of the dependency information. The blocks that are read from and written to disk often contain multiple structures (e.g., inodes or directory fragments), each of which generally contains multiple dependency-causing components (e.g., block pointers and directory entries). As a result, originally independent changes can easily cause dependency cycles (see Figure 1) and excessive aging. Detecting and handling these problems increases implementation complexity and reduces performance.

With soft updates, dependency information is maintained at a very fine granularity: per field or pointer. “Before” and “after” versions are kept for each individual update (e.g., the addition of a directory entry or the setting of a block pointer) together with a list of updates on which it depends. A dirty block can be written to disk at any time, as long as any updates within the in-memory block that have pending dependencies are first temporarily “undone” (rolled back). This guarantees that every block written to disk is consistent with respect to the current on-disk state. During the disk write, the block is locked to prevent applications from seeing the rolled-back state. When a disk write completes, any undone updates in the source memory block are restored before the block is unlocked. So, for

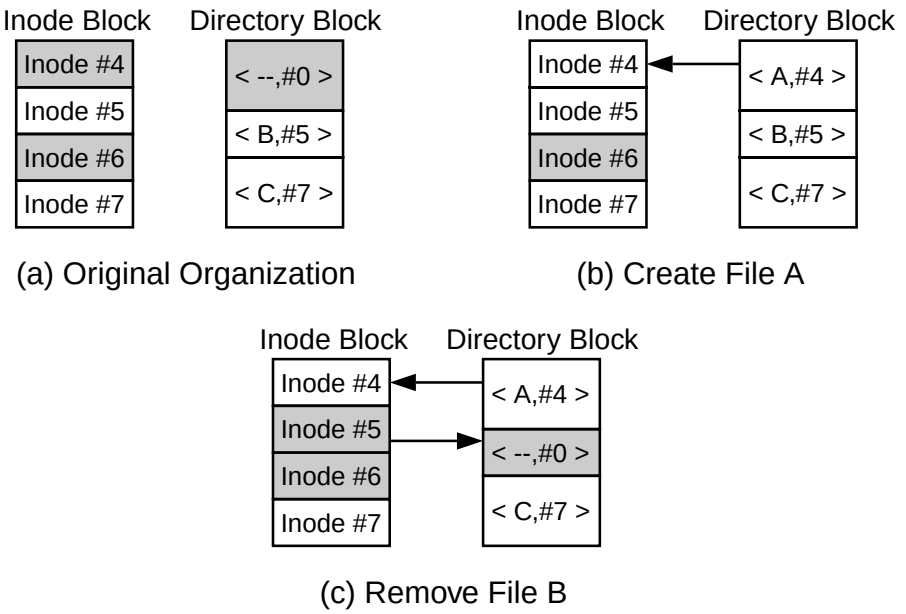

Fig. 1. A Cyclic Dependency. (a), (b), and (c) each show the same pair of in-memory metadata blocks. The shaded metadata structures are unowned and available for allocation. The arrows indicate dependencies. (a) shows the blocks before any updates. (b) shows the blocks after creating file A. When creating a new file, the newly initialized inode must be written to disk before the new directory entry. (c) shows the blocks after removing file B. When removing a file, the reset directory entry must be written before the reinitialized inode. Viewed at a block level, the two metadata blocks in (c) depend on each other. Viewed at a finer granularity, there are two independent update sequences.

example, the two metadata blocks in Figure 1(c) can be safely transferred to disk with three writes (see Figure 2). With this approach, dependency cycles do not occur because independent sequences of dependent updates remain independent and no single sequence is cyclic. Aging problems do not occur because new dependencies are not added to existing update sequences.

3.1.1 *Design Issues*. Prior to each update for which sequencing will be required, dependency information must be set up. While soft updates, in essence, employs an in-memory log of update dependencies, efficiency requires more aggressive indexing (e.g., to identify the associated block) and cross-referencing (e.g., to identify dependent updates) of dependency structures. The modules that flush dirty cache blocks must also be modified to check and enforce dependencies appropriately. Many dependencies should be handled by the undo/redo approach described above. Others can be more efficiently handled by postponing in-memory updates until after the updates on which they depend reach stable storage. This deferred update approach is only safe when freeing file system resources, since applications can not be shown out-of-date data. In a sense, deferred updates are undone until the disk writes on which they depend complete.

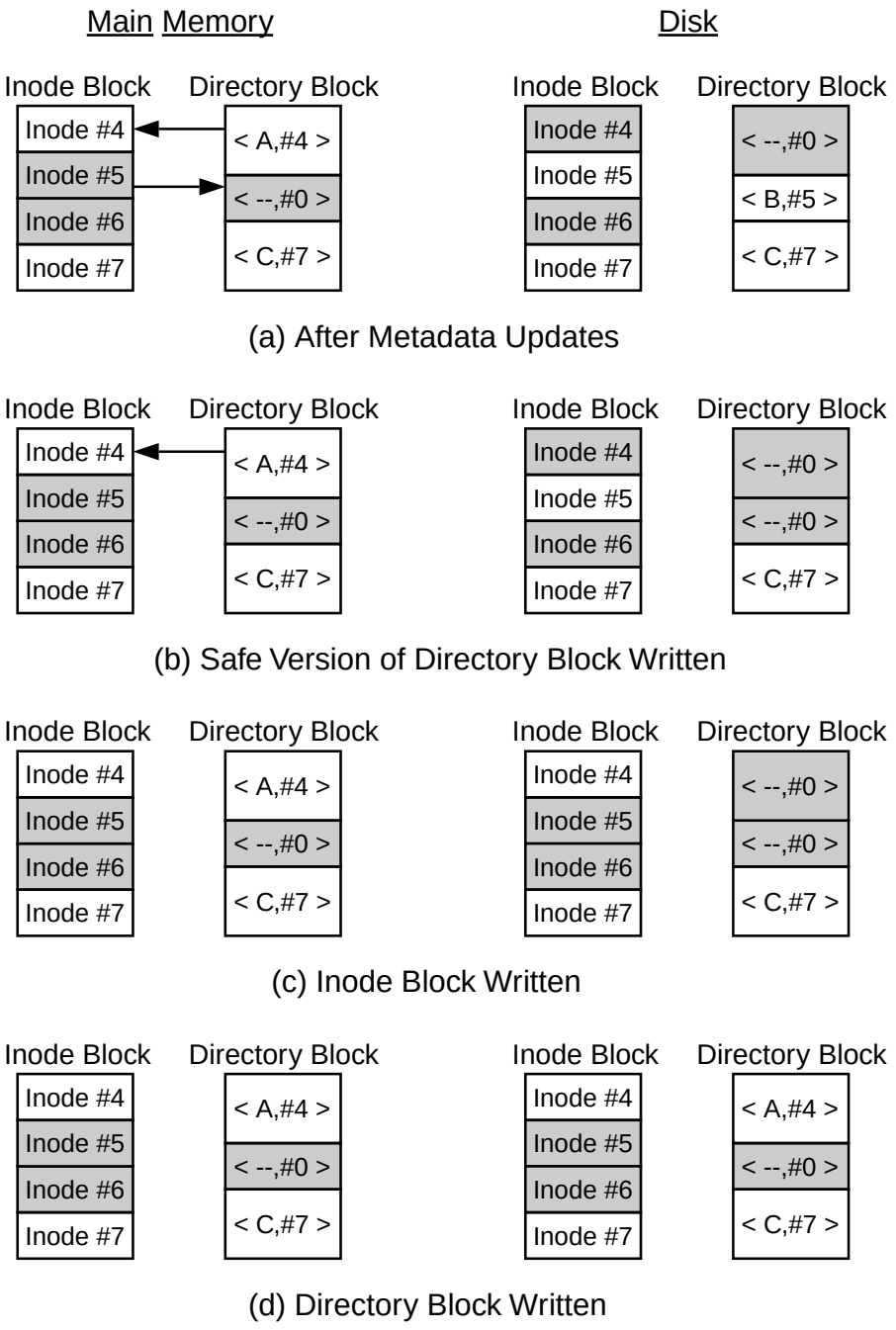

Fig. 2. Undo/redo operations in soft updates. (a) shows the in-memory and on-disk copies of the two modified metadata blocks from Figure 1(c). (b), (c), and (d) show the same blocks after each of three disk writes. For the duration of each disk write, the in-memory copy matches the resulting on-disk copy, and locks prevent any application from observing out-of-date information. As desired, the application-visible copies of metadata are always fully up-to-date, and the on-disk copy of the file system is always internally consistent.

When a disk write completes, there is often some processing needed to update/remove dependency information, restore undone changes, and deal with deferred work. An implementation of soft updates requires some method of performing these tasks in the background. Very simple changes can be made during the disk I/O completion interrupt service routine (ISR), which generally calls a predefined procedure in the higher-level module (e.g., a file system) that issued the request. However, any task that can block and wait for a resource (e.g., a lock or, worse yet, an uncached disk block) cannot be handled in this way. Such tasks must be handled outside of the ISR, preferably by a background process that executes in the near future (e.g., within a few seconds).

3.2 Soft Updates in the 4.4BSD Fast File System

Soft updates has been implemented in the 4.4BSD FFS [McKusick et al. 1984] used in the NetBSD, OpenBSD, FreeBSD, and BSDI operating systems. The basic operation of the 4.4BSD implementation of soft updates is based on and similar to the SVR4 implementation [Ganger and Patt 1994], but it is more complete, robust, and clean. This section overviews the operation of soft updates in these two systems; much more detailed descriptions are provided by Ganger and Patt [1995] and McKusick and Ganger [1999].

In both implementations, almost all of the synchronous and asynchronous metadata updates have been replaced with delayed writes. The main exceptions are: (1) when the user explicitly requests synchronous updates, as with the *fsync()* system call or the O_SYNC modifier to the *open()* system call, and (2) when mounting or unmounting a file system. Dependency-setup code has been added before each update that requires sequencing. The disk write routines have been modified to perform the appropriate undo/redo actions on source memory blocks. Background processing that cannot be handled during the ISR is performed by the syncer daemon when it next awakens.

In FFS-based file systems, there are four main structural changes that require sequenced metadata updates: (1) block allocation, (2) block deallocation, (3) link addition (e.g., file creation), and (4) link removal. The dependencies associated with each are described below together with a brief description of how they are handled in both implementations.

Block Allocation. When a new block or fragment is allocated for a file, the new block pointer (whether in the inode or an indirect block) should not be written to stable storage until after the block has been initialized.[1] Also, if the on-disk free space maps are being protected (See Section 3.3), the free

[1]Most file systems do not actually guarantee that newly allocated blocks are initialized on disk. Because it is expensive with synchronous writes (e.g., factor of two reduction in create performance [Ganger and Patt 1994]), no traditional FFS implementation with which we are familiar supports it. Because it is inexpensive with soft updates (e.g., around 2% overhead, with a maximum of 8% observed), guaranteeing initialization of new allocations is the default in our implementation. All of our experiments utilize this default.

space map from which the block or fragment is allocated must be written to disk before the new pointer. These two dependencies are independent of each other and apply to allocation of both file blocks and indirect blocks. In our implementations, both dependencies are enforced by undo/redo on the block pointer and the file size. So, for example, when an inode with an unsafe block pointer (i.e., one for which there are unsatisfied dependencies) is written to disk, the in-memory copy is first modified such that the pointer is nullified, and, if appropriate, the file size is reduced. After the disk write completes, the undone modifications are replayed so that the file size and block pointer are restored to their most current in-memory values.

Block Deallocation. A deallocated disk block must not be reused before the previous on-disk pointer to it has been reset. In our implementations, we satisfy this requirement by not deallocating a block (i.e., setting the bits in the in-memory free space map) until after the reset block pointer reaches stable storage. When block deallocation is requested, the appropriate in-memory pointers are nullified, and control returns to the caller. Only after the modified metadata block has propagated to disk are the blocks actually deallocated and made available for reuse. Of course, recently allocated blocks to which pointers have not yet been written to disk can be immediately deallocated.

Link Addition. When adding a directory entry, the (possibly new) inode, with its incremented link count, must be written to disk before the new directory entry's pointer to it. Also, if the inode is new and the on-disk free maps are being protected, the free inode map from which the inode is allocated must be written to disk before the new pointer. These two dependencies are independent of each other. In our implementations, both dependencies are enforced by undo/redo on the inode pointer field of the directory entry, since nullifying this field is sufficient to identify the entry as invalid after a system failure.

Link Removal. When removing a directory entry, the on-disk directory entry's inode pointer must be nullified before the corresponding on-disk inode's link count is decremented (possibly freeing the inode for reuse).[2] In our implementations, we achieve this by not decrementing the in-memory inode's link count until after the reset pointer reaches stable storage. So, when link removal is requested, the in-memory directory entry is nullified. Also, if the directory entry was recently added and not yet written to disk, the inode's link count is immediately decremented. (In this case, the link addition and removal are serviced with no disk writes.) Otherwise, control is returned to the caller, and the inode's link count is not decremented until after the dirty directory block is written to stable storage.

[2]The actual requirement is that the on-disk inode should not be reinitialized or pointed to by the free inode map (if the on-disk map is to be trusted after failure) before all previous on-disk directory entry pointers to it have been nullified. Our more stringent requirement simplifies the implementation and protects on-disk link counts for safe postcrash file system use.

3.3 File System Recovery

Most file system implementations minimize update sequencing in order to reduce the performance degradation caused by synchronous writes. As a result, they require time-consuming, off-line assistance (e.g., the *fsck* utility) before a file system can be safely used after any system failure. Because update sequencing costs so little with soft updates, our enhanced file systems extend the set of protected updates to guarantee that the on-disk metadata can always be used safely (except when media corruption destroys live metadata), eliminating the need for premount crash-recovery assistance. So, with our soft updates implementation, a file system can be safely mounted and used immediately after a system failure. However, it may contain several minor inconsistencies:

—Unused blocks may not appear in the free space maps.

—Unreferenced inodes may not appear in the free inode maps.

—Inode link counts may exceed the actual number of associated directory entries, which can lead to unclaimed blocks and inodes over time.

One can run the *fsck* utility on the file system, when it is convenient to have file system downtime, to reclaim unreferenced resources, and correct link counts. In the latest versions of most 4.4BSD-based OSes, a background version of *fsck* can be used to reclaim these resources while the file system is actively being used [McKusick and Ganger 1999].

Maintaining the dependencies described in Section 3.2 is sufficient to guarantee that the on-disk copies of inodes, directories, indirect blocks, and free space/inode bitmaps are always safe for immediate use after a system failure. However, FFS maintains a number of free block/inode counts in addition to the bitmaps. These counts are used to improve efficiency during allocation and therefore must be consistent with the bitmaps for safe operation. Because we know of no convenient way to guarantee postcrash consistency of these counts via update sequencing, we simply recompute them from the bitmaps when mounting a file system after a crash. By not requiring that the *fsck* utility be run on the file system after a crash, soft updates reduces file system recovery time by more than two orders of magnitude (see Section 4.4).

3.4 File System Semantics

The use of synchronous writes to sequence metadata updates does not imply synchronous file system semantics. In most implementations, the last write in a sequence of metadata updates is asynchronous or delayed. Therefore, when a file system call returns control to the caller, there is no guarantee that the change is permanent. For link addition (e.g., file creation) and block allocation, the last update adds the new name and pointer to the directory block, inode, or indirect block. As a result, such

changes are not permanent when control returns to the caller.[3] For link removal and block deallocation, on the other hand, the last update modifies the free space/inode maps. When control returns to the caller, the link is permanently removed, and/or the blocks have been deallocated and are available for reuse. With soft updates, neither is true. In particular, deallocated resources do not become available for reuse until after the reinitialized inode or indirect block reaches stable storage.

Some system calls have a flag telling the file system to guarantee that changes are permanent before returning. It may be useful to augment additional file system calls (e.g., link addition) with such a flag in order to support certain applications (e.g., those that require lock files).

It is important to note that soft updates does not significantly increase the amount of data that can be lost when the system crashes. Rather, using soft updates allows a file system to employ the same write-back strategies for metadata as it uses for file data. When file systems employ soft updates, users will continue to face the same persistence dangers that they already choose to accept with any given system (e.g., potential for loss of 30 seconds worth of information in most UNIX-derived systems).

3.5 Implementation Complexity and Lessons Learned

The original soft updates implementation consisted of 1800 lines of commented C code and only required (minor) changes to the file system and buffer cache modules. The implementation was largely straightforward, containing many procedures with similar code for dependency structure initialization, scanning, and deallocation. No changes to the on-disk metadata structures were required. Having learned key lessons from an initial implementation, a single graduate student completed a partial soft updates implementation (described by Ganger and Patt [1994]) in three weeks. Update sequencing for fragment extension and the free space/inode maps took an additional two weeks to add and debug.

The transition of soft updates from research prototype to product-quality software (for 4.4BSD) came with several lessons and problems that were more complex than was suggested in the original research papers. Some of these issues were known 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 remainder of this section describes the most significant of these issues.

The "fsync" System Call. The "fsync" system call requests that a specific file be completely committed to stable storage and that the system call not return until all associated writes have completed. 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

[3]Software locking schemes that use lock files may encounter surprises because of this.

file and the root of the file system. 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 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 storage 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 updates 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 updates 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 updates code generates most of the remaining synchronous writes in the file system.

Unmounting File Systems. Unmounting a file system requires finding and flushing all the dirty blocks that are associated with the file system. Flushing these blocks 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 file system, several iterations of file flushes followed by background activity may be required.

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 multiple structures for each update dependency: one for the "depender" and one for each "dependee." For example, each time a block was allocated, new dependency structures were associated with the disk block, the bitmap, and the inode (the "depender" in this case). The 4.4BSD soft updates code instead uses a single dependency structure to describe a block allocation. This one dependency structure is linked into multiple lists: one for the allocated block, one for the bitmap, and one for the inode. By constructing lists rather than using separate structures, the demand on memory was reduced by about 40%.

In actual operation, we have found that the additional dynamic memory used for soft updates structures is roughly equal to the amount of memory

used by vnodes plus inodes; for a system with 1000 vnodes, the additional peak memory used is about 300KB. The one exception to this guideline occurs when large directory trees are removed. In this case, the file system 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 updates code monitors the memory load for this case and prevents it from growing 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. This reduces the sustained rate of file removal to disk speeds, but does so 20 times more efficiently than the traditional synchronous write file system. In steady-state, the soft updates remove algorithm requires about one disk write for each ten files removed, while the traditional file system requires at least two writes for every file removed.

Useless Write-Backs. While soft updates allows blocks to be written back in any order, blocks with pending dependencies will remain dirty after a disk write. When we instrumented the initial BSD soft updates code, we found that 10–20% of all disk writes had pending dependencies and were immediately redirtied by the required roll-back. Many of these "useless" writes occurred because the default syncer daemon algorithm produced nearly worst-case ordering of disk writes. Specifically, it initiated all disk writes associated with particular files in a burst, which meant that all of them were initiated before any of them completed. By modifying the flush routines to roughly prioritize block write-backs based on dependency information, we eliminated over 50% of these "useless" write-backs. The revised syncer daemon initiates and waits for writes for bitmap blocks, data blocks, and other nondependent blocks. Only after these all complete does the syncer move on to metadata blocks that previously had pending dependencies—at that point, many no longer will.

Having found success with this simple technique, we set out to eliminate other causes of roll-back triggered I/O. A second place where we found success was in the cache reclamation code. By replacing the default LRU scheme with a scheme that avoids selecting a block with pending dependencies, we further reduced the number of roll-back triggered disk writes by about half (to less than one quarter of its original value). The eventually selected block is also generally much more efficient to reclaim and less likely to be reused than a dirty, dependent metadata block.

The fsck Utility. In a conventional file system 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 often 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 time that the directory entry is deleted and the time that 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 file system has been damaged by a media failure that results in the loss of one or more directories. However, it often stuffs the **lost+found** directory with partially deleted files when running with soft updates. Thus, the **fsck** program must be modified to check that a file system is running with soft updates and clear out rather than saving unreferenced inodes, unless it has determined that unexpected damage has occurred to the file system 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 file system 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.

4. PERFORMANCE EVALUATION

In this section, we compare soft updates to its upper bound and show that a file system using soft updates can achieve full metadata integrity at little cost in performance or availability. We further show that this upper bound represents a substantial increase in performance and availability when compared to the conventional synchronous write approach. We also show that soft updates compares favorably to write-ahead logging in BSD FFS.

The results of several comparisons are provided, each highlighting a different aspect of soft updates performance. Microbenchmarks are used to focus on particular activities and set expectations for more complete workloads. Macrobenchmark workloads show the impact of soft updates in multiprogramming and news/mail server environments; rough data from a real system using soft updates in the field confirm the benchmark results. Measurements of postcrash recovery time show that soft updates can improve availability significantly. Finally, comparisons with write-ahead logging indicate that soft updates provides similar or better performance for BSD FFS.

4.1 Experimental Setup

Most of our experiments compare the performance of three instances of FreeBSD's FFS file system, referred to in the article as *No Order*, *Conventional*, and *Soft Updates*. *No Order* corresponds to FreeBSD FFS mounted with the O_ASYNC option, which causes the file system to ignore ordering constraints and use delayed writes for all metadata updates. This baseline has the same performance and lack of reliability as the delayed mount option described in Ohta and Tezuka [1990]. It is also very similar to the memory-based file system described in McKusick et al. [1990]. *No Order* represents an upper bound on the FreeBSD FFS performance that can be achieved by changing only the mechanisms used to deal with metadata

integrity.[4] *Conventional* corresponds to the default FreeBSD FFS implementation, which uses synchronous writes to sequence metadata updates. *Soft Updates* corresponds to the same FreeBSD FFS modified to use soft updates. Section 4.5 compares *Soft Updates* to a write-ahead logging version of FreeBSD FFS; details are provided in that section.

With one exception, all experiments are performed on commodity PC systems equipped with a 300MHz Pentium II processor, 128MB of main memory, and two 4.5GB Quantum Viking disk drives (Fast SCSI-2, 7200 RPM, 8ms average seek for reads). One disk drive holds the root, home, and swap partitions, and the second drive is used for the file system under test. The operating system is FreeBSD 4.0, and all experiments are run with no other nonessential activity in the system. All of the file system instances benefit from the general FFS enhancements included in FreeBSD 4.0, including block reallocation [Smith and Seltzer 1996] and extent-like clustering [McVoy and Kleiman 1991]. Each experimental result is an average of numerous measurements, and metrics of variation are provided with the data.

4.2 Microbenchmark Performance Analysis of Soft Updates

This subsection uses microbenchmarks taken from Seltzer et al. [1995] to illustrate the basic performance characteristics of BSD FFS using soft updates. Specifically, the microbenchmarks measure the speed with which a system can create, read, and delete 32MB of data for files that range in size from 2KB to 4MB. The files are spread across directories, with no more than 50 files per directory, to prevent excessive name lookup overheads. Before each measurement, the file cache is emptied by unmounting and then remounting the file system under test. Intuitively, we expect to find that *No Order* and *Soft Updates* perform similarly and that they both outperform *Conventional* for the create and delete experiments. For the read experiments, which involve no metadata update ordering, we expect all three to perform identically.

Figure 3 shows the results of the create microbenchmark, which do match overall expectations: *No Order* and *Soft Updates* are similar in performance and both outperform *Conventional*. For all three, throughput improves with file size up to 64KB, since, up to this size, a contiguously allocated file can be written in a single disk operation. The performance drops after 64KB, because files larger than 64KB involve at least two contiguous data segments. Files beyond 104KB in size additionally require an indirect block. Beyond 104KB, bandwidth increases with file size again, as the cost of the indirect block is amortized over more and more data. The small drop at 1024KB for *No Order* and *Soft Updates* occurs because the file system cannot fit all 32MB of data in one cylinder group, which the benchmark and allocation routines try to do for these two cases (32 and 8

[4]Better performance could be achieved by disabling the syncer daemon, which would also eliminate the guarantee that new data blocks are written to disk soon after creation (e.g., within 30 seconds).

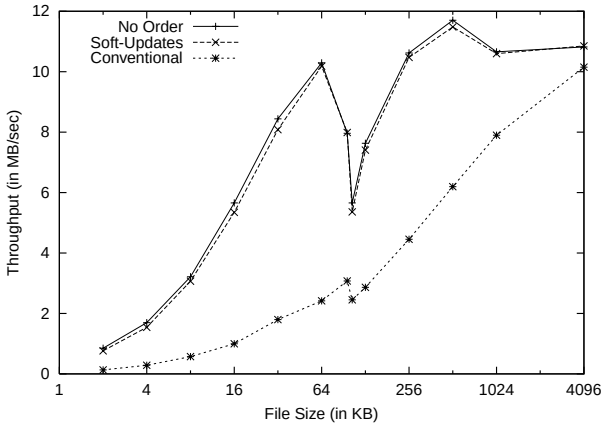

Fig. 3. Create throughput, in megabytes/second, as a function of file size. The values are averages of 25 runs, and all coefficients of variation are below 0.07.

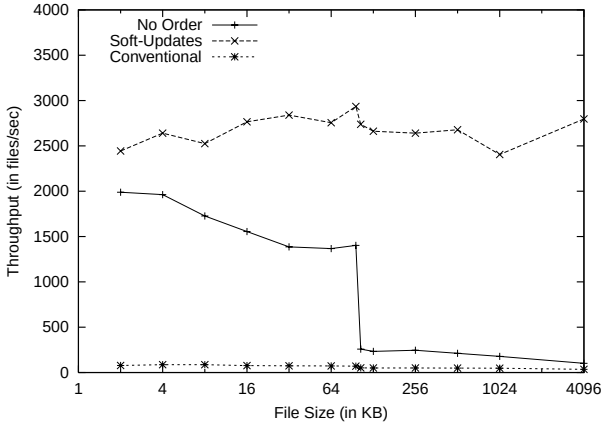

Fig. 4. Delete throughput, in files/second, as a function of file size. The values are averages of 25 runs, and most coefficients of variation are below 0.1. The one exception is the 4096KB data point for *Soft Updates*, for which the coefficient of variation is 0.13.

files, respectively); as a result, the last file created is split between two cylinder groups, which causes a performance drop mainly because the second bitmap must be read from disk. The three implementations converge as the file size increases, because the cost of writing the data begins to dominate the cost of the two synchronous writes required by *Conventional* for file creation.

Figure 4 shows the results of the delete microbenchmark. As expected, *Soft Updates* and *No Order* dramatically outperform *Conventional*. In addition, *Soft Updates* outperforms *No Order*, because *No Order* is actually removing the files as it goes whereas soft updates is simply generating work orders for the background process to do the removals. The large performance drop at 104KB results from the per-file disk read required by all schemes to fetch the indirect block. (Recall that each experiment starts

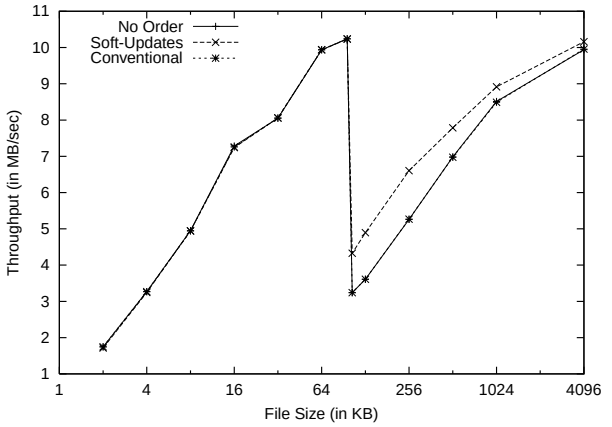

Fig. 5. Read throughput, in megabytes/second, as a function of file size. The values are averages of 25 runs, and all coefficients of variation are below 0.04.

with a cold cache.) Above 104KB, *No Order* is approximately 3 times faster than *Conventional*, because both are limited by synchronous disk I/O (one disk read per file versus one disk read and two disk writes per file). Again, by pushing actual removal to the background, *Soft Updates* avoids most of this performance drop. For all schemes, file deletion throughput tends to decrease with file size, because the work involved with each file's deletion increases.

Figure 5 shows the results of the read microbenchmark. As before, performance increases with file size but drops significantly when the file size requires an indirect block and thus an extra noncontiguous read. As expected, there are no significant differences in performance between *No Order* and *Conventional*, since there are no update dependencies associated with reading a file. The surprising difference between these and *Soft Updates* for files larger than 96KB is a microbenchmark artifact related to BSD FFS's reallocation scheme (for achieving maximum contiguity) and the use of delayed deallocation in *Soft Updates*; by delaying deallocation, *Soft Updates* prevents BSD FFS from placing indirect blocks in undesirable disk locations for this microbenchmark. Though all of the results reported in this article include BSD FFS's default reallocation schemes, we have verified that this read microbenchmark behavior does not recur in any of the other experiments; with reallocation disabled, read microbenchmark performance is identical for the three implementations, and the other benchmarks have the same relative performance as reported.

Figure 6 shows the total number of disk writes initiated during the create microbenchmark. As expected, the number of disk writes for *Soft Updates* is close to the number for *No Order* and significantly smaller than the number for *Conventional*. For 2KB thru 64KB files, *Conventional* involves approximately 3.1 disk writes per file, which includes 2 synchronous disk writes to create the file, 1 asynchronous disk write of the data, and a partial disk write for coalesced updates to bitmap blocks and

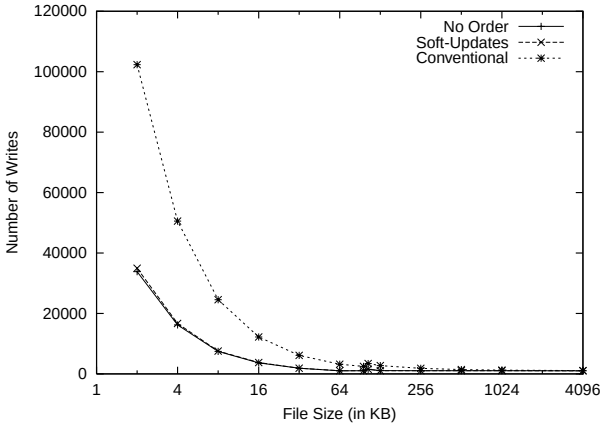

Fig. 6. Total number of disk writes for the create microbenchmark, as a function of file size. The values are averages of 25 runs, and all coefficients of variation are below 0.13.

directory inodes. For these same file sizes, *Soft Updates* and *No Order* involve approximately 1.1 disk writes per file, corresponding to 1 asynchronous disk write of the data and a partial disk write for coalesced updates to bitmap, inode, and directory blocks. Thus, *Soft Updates* both eliminates synchronous writes and reduces the number of disk writes, coalescing many metadata updates into a much smaller number of disk writes. Above 64KB, the gap between implementations closes as the file size grows, and the data block writes become a larger fraction of the total. The lines drift down between 2KB and 64KB, because the number of files created decreases as the file size gets larger.

4.3 Overall System Performance with Soft Updates

To illustrate how soft updates can be expected to perform in normal operation, we present measurements from two system benchmarks (Sdet and Postmark) and one system in real use.

Sdet. Figure 7 compares the three implementations using the deprecated Sdet benchmark from SPEC. This benchmark concurrently executes one or more **scripts** of user commands designed to emulate a typical software-development environment (e.g., editing, compiling, and various UNIX utilities). The scripts are generated from a predetermined mix of commands [Gaede 1981; 1982]. The reported metric is scripts/hour as a function of the script concurrency. As expected, the overall results are that *Soft Updates* is very close in performance to *No Order* (always within 3%) and that both significantly outperform *Conventional* (by 6X with no script concurrency and by more than 8X with script concurrency). Throughput decreases with concurrency for *Conventional* because of locality-ruining competition for its bottleneck resource, the disk. For *No Order* and *Soft Updates*, on the other hand, throughput increases with the addition of concurrency, because the portion of CPU unused by a lone script during its

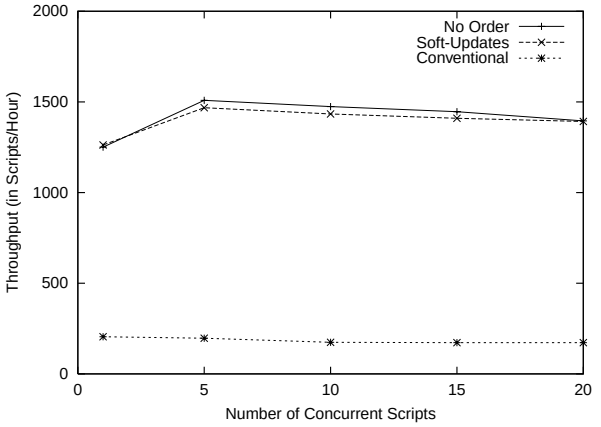

Fig. 7. Sdet results, in scripts per hour, for three script concurrency values. Each value is an average of 5 runs, and the coefficients of variation are all below 0.085.

Table I. Postmark Results, in File System "Transactions per Second." The values are averages of 5 runs, and the standard deviations are given in parentheses.

File System Configuration	Transactions per Second
No Order	165 (2.4)
Soft Updates	170 (3.0)
Conventional	45.4 (1.1)

few synchronous disk requests (mainly reads) can be used by concurrent scripts. As script concurrency increases (e.g., 10 and beyond), context switching and memory pressure slowly reduce performance for all three schemes.

Postmark. Table I compares the three implementations using a file system benchmark called Postmark [Katcher 1997]. Postmark was designed to measure file system performance for the ephemeral small-file workloads seen in environments such as electronic mail, netnews, and Web-based commerce. Postmark creates a pool of random text files of varying sizes, and then measures the time required to execute a specified number of **transactions**. Each transaction, randomly selected, consists of two of four actions: create a file of a random length (within bounds), delete a file, read a file in its entirety, or append data to an existing file. Comparative results from the benchmark reportedly match the experiences of Internet Service Providers [Katcher 1997]. Our experiments use the default benchmark settings: 30,000 transactions, equal bias across actions, file sizes between 512 bytes and 16KB, initial pool of 1000 files and 10 directories. The overall results again match expectations: *Soft Updates* performance is within 3% of *No Order*, and both outperform *Conventional* by 3.7X. *Soft Updates* outperforms *No Order* by 3% because it pushes deletion activity to the background.

Table II. Average Number of Disk Writes Observed for a Nonweekend 24-Hour Period on a Central Mail Server

File System Configuration	Disk Writes	
	Sync	Async
Conventional	1,877,794	1,613,465
Soft Updates	118,102	946,519

Central Mail Service. To show that the benchmark results correlate well with reality, we compare the performance of a machine running as the central mail server for Berkeley Software Design, Inc. run with and without soft updates (i.e., *Conventional* and *Soft Updates*). The administrator was obviously unwilling to run it in *No Order* mode, since this is a production machine and because people will not tolerate loss of their mail. The hardware is roughly equivalent to our experimental system except that the mail spool is striped across three disks. Statistics were gathered for 30 days of nonweekend operation in each mode. Table II compares the average number of disk writes for a nonweekend 24-hour period.

The normal file system 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% fewer writes, which triples the mail handling capacity of the machine. While we do not have data on the relative email loads of the mail server during the two measurement periods, we were told by the system administrators that going back to *Conventional* after the month with *Soft Updates* was not an option—the email load had grown over the experimental period to a point where the server could not keep up without soft updates. In addition, these data were collected before the write-back tuning described in Section 3.5, which could be expected to further reduce the write activity.

4.4 File System Recovery Time

Table III compares the file system recovery times of *Soft Updates* and *Conventional* for empty and 76% full file systems. *No Order* uses the same recovery mechanism (*fsck*) as *Conventional*, but often cannot fully recover to a consistent state.

For FreeBSD's default FFS on our experimental platform, the *fsck* utility executes in 5 seconds for an empty file system and requires 2.5 minutes or more for a file system that is 76% full. With soft updates, on the other hand, the same file systems can be mounted after a system failure in 0.35 seconds, regardless of how full they are. This includes the time necessary to read and modify the superblock, read all cylinder group blocks, recompute the auxiliary free space/inode counts, and initialize the various in-memory structures. Only the recomputation step is unique to the soft updates implementation.

Table III. File System Recovery Times after System Failures for Two Levels of Capacity Utilization

File System Configuration	Recovery Time	
	Empty	76% Full
Conventional	5 seconds	150 seconds
Soft Updates	0.35 seconds	0.35 seconds

To verify that soft updates correctly protects metadata integrity, we simulated unpredictable system failures by hitting the "halt button" at random points during benchmark execution. In 25 trials, we found no inconsistencies other than unclaimed free space. While not conclusive, these results give us some confidence in the implementation. For comparison purposes, we repeated this experiment for the other two implementations. For both *Conventional* and *No Order*, the on-disk file system state required off-line assistance before it could be safely used after 96% of our trials. In addition, after 30% of our trials with *No Order*, there were unresolvable inconsistencies (e.g., disk blocks pointed to by more than one file). This last datum demonstrates the need for update sequencing.

4.5 Soft Updates versus Write-Ahead Logging

While we have shown that soft updates can sequence metadata updates with near-zero impact on performance, it remains unknown how write-ahead logging compares to the performance bound of *No Order*. Here, we provide a partial answer to this question by evaluating a version of FreeBSD FFS modified to use prototype write-ahead logging software. The results indicate that the extra disk I/O required for write-ahead logging degrades end-to-end performance by 4–19% for the Sdet and Postmark benchmarks. A more complete evaluation of logging implementations (e.g., synchronous versus asynchronous logging, same disk versus separate disk logging) and their performance relative to soft updates can be found in Seltzer et al. [2000].

Logging-FFS augments FreeBSD FFS with support for write-ahead logging by linking logging code into the same hooks used for the soft updates integration. Most of these hooks call back into the logging code to describe a metadata update, which is then recorded in the log. The log is stored in a preallocated file that is maintained as a circular buffer and is about 1% of the file system size. To track dependencies between log entries and file system blocks, each cached block's buffer header identifies the first and last log entries that describe updates to the corresponding block. *Logging-FFS* uses the former value to incrementally reclaim log space as file system blocks are written to the disk; checkpoints are only explicitly performed when this mechanism does not provide free space, which is rare. *Logging-FFS* uses the latter value to ensure that relevant log entries are written to disk before dependent file system blocks. *Logging-FFS* aggressively uses a delayed group commit approach to improve performance, but always correctly writes log entries to disk before the file system updates that they

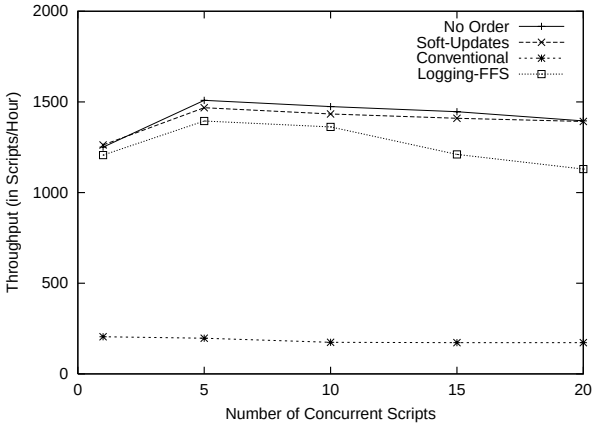

Fig. 8. Sdet results, in scripts per hour, for three script concurrency values. Each value is an average of 5 runs, and the coefficients of variation are all below 0.085.

describe. Details of *Logging-FFS*'s implementation can be found in Seltzer et al. [2000].

Sdet. Figure 8 compares the write-ahead logging implementation to the previous three implementations using the Sdet benchmark. Because of its aggressive use of delayed writes for both log and metadata updates, *Logging-FFS* follows the same basic trends as *No Order* and *Soft Updates*, including outperforming *Conventional* by 5.9–7.8X. However, the extra disk I/O required for disk-based logging results in 4–19% performance degradation relative to *No Order* and *Soft Updates*. The performance of *Logging-FFS* drops with high script concurrencies (15 and 20), as the log writes increasingly compete with foreground disk activity.

Postmark. Table IV compares the write-ahead logging implementation to the previous three implementations using the Postmark benchmark. The overall results again match expectations: *Logging-FFS* performance is 6% lower than *No Order*, 9% lower than *Soft Updates*, and 3.4X that of *Conventional*.

5. CONCLUSIONS

The use of synchronous writes and off-line crash-recovery assistance (e.g., the *fsck* utility) to protect metadata has been identified as a source of performance, integrity, security, and availability problems for file systems [Ousterhout 1990; McVoy and Kleiman 1991; Seltzer et al. 1993]. We have developed a new mechanism, soft updates, that can be used to achieve memory-based file system performance while providing stronger integrity and security guarantees (e.g., allocation initialization) and higher availability (via shorter recovery times) than most disk-based file systems. In our microbenchmark and system-level experiments, this translates into performance improvements of 100–2000% for metadata-update-intensive benchmarks and recovery time improvements of more than two orders of

Table IV. Postmark Results, in File System "Transactions per Second." The values are averages of 5 runs, and the standard deviations are given in parentheses.

File System Configuration	Transactions per Second
No Order	165 (2.4)
Soft Updates	170 (3.0)
Conventional	45.4 (1.1)
Logging-FFS	155 (2.4)

magnitude. It also represents 4–19% higher system performance than write-ahead logging.

While our experiments were performed in the context of UNIX systems, the results are applicable to a much wider range of operating environments. Every file system, regardless of the operating system, must address the issue of integrity maintenance. Some (e.g., MPE-XL, CMS, Windows NT) use database techniques such as logging or shadow-paging. Others (e.g., OS/2, VMS) rely on carefully ordered synchronous writes and could directly use our results.

A number of issues arose as soft updates moved from the research lab into the product-quality 4.4BSD operating system. As is often the case, nonfocal operations like "fsync," **fsck**, and **unmount** required some rethinking and resulted in additional code complexity. Despite these unexpected difficulties, our performance measurements do verify the results of the early research. The original soft updates code is available in Ganger and Patt [1995]. The 4.4BSD soft updates code is now available for commercial use in Berkeley Software Design Inc.'s BSD/OS 4.0 and later systems. It is available for noncommercial use in the freely available BSDs: FreeBSD, NetBSD, and OpenBSD.

ACKNOWLEDGMENTS

We thank Wilson Hsieh, Frans Kaashoek, Jay Lepreau, John Wilkes, Bruce Worthington, and the anonymous reviewers for directly helping to improve the quality of this article. We thank BSD, Inc. for providing the mail server data reported in Section 4.3. The original work at the University of Michigan was enabled by generous donations of funding, OS sources, and hardware from NCR (AT&T/GIS).

REFERENCES

CHAMBERLIN, D. D., ASTRAHAN, M. M., BLASGEN, M. W., GRAY, J. N., KING, W. F., LINDSAY, B. G., LORIE, R., MEHL, J. W., PRICE, T. G., PUTZOLU, F., SELINGER, P. G., SCHKOLNICK, M., SLUTZ, D. R., TRAIGER, I. L., WADE, B. W., AND YOST, R. A. 1981. A history and evaluation of system R. *Commun. ACM 24*, 10, 632–646.

CHAO, C., ENGLISH, R., JACOBSON, D., STEPANOV, A., AND WILKES, J. 1992. Mime: A high-performance parallel storage device with strong recovery guarantees. Tech. Rep. HPL-CSP-92-9 rev 1 (Nov.). Hewlett-Packard, Fort Collins, CO.

CHEN, P., NG, W., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. 1996. The RIO file cache: Surviving operating system crashes. In *Proceedings of the 7th International Confer-*

ence on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII, Cambridge, MA, Oct. 1–5, 1996), B. Dally and S. Eggets, Eds. ACM Press, New York, NY, 74–83.

CHUTANI, S., ANDERSON, O., KAZAR, M., LEVERETT, B., MASON, W., AND SIDEBOTHAM, R. 1992. The episode file system. In *Proceedings on Winter 1992 USENIX Conference*, USENIX Assoc., Berkeley, CA, 43–60.

DENNING, P. J. 1967. Effects of scheduling on file memory operations. In *Proceedings on AFIPS Spring Joint Computer Conference* (Reston, Va., Apr. 1967), AFIPS Press, Arlington, VA, 9–21.

DUNCAN, R. 1986. *Advanced MSDOS Programming*. Microsoft Press, Redmond, WA.

GAEDE, S. 1981. Tools for research in computer workload characterization. In *Experimental Computer Performance and Evaluation*

GAEDE, S. 1982. A scaling technique for comparing interactive system capacities. In *Proceedings of the 13th International Conference on Management and Performance Evaluation of Computer Systems*, 62–67.

GANGER, G. AND PATT, Y. 1994. Metadata update performance in file systems. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation* (OSDI '94, Monterey, CA, Nov.), USENIX Assoc., Berkeley, CA, 49–60.

GANGER, G. AND PATT, Y. 1995. Soft updates: A solution to the metadata update problem in file systems. Tech. Rep. CSE-TR-254-95 (Aug.). University of Michigan, Ann Arbor, MI.

HAGMANN, R. 1987. Reimplementing the cedar file system using logging and group commit. In *Proc. of the Eleventh ACM Symposium on Operating systems principles* (Austin, TX, Nov. 8-11, 1987), L. Belady, Ed. ACM Press, New York, NY, 155–162.

KATCHER, J. 1997. Postmark: A new file system benchmark. Tech. Rep. TR3022 (Oct.). Network Appliance.

MCCOY, K. 1990. *VMS file system internals*. Digital Press, Newton, MA.

MCKUSICK, M. AND GANGER, G. 1999. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *Proceedings of the USENIX 1999 Annual Technical Conference* (Monterey, CA, June), USENIX Assoc., Berkeley, CA, 1–17.

MCKUSICK, M. AND KOWALSKI, T. 1994. Fsck—The UNIX file system check program. In *4.4 BSD System Manager's Manual* O'Reilly & Associates, Inc., Sebastopol, CA, 3–21.

MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. 1984. A fast file system for UNIX. *ACM Trans. Comput. Syst. 2*, 3 (Aug. 1984), 181–197.

MCKUSICK, M., KARELS, M., AND BOSTIC, K. 1990. A pageable memory-based filesystem. In *Proceedings of the Summer USENIX Conference* (June), USENIX Assoc., Berkeley, CA, 137–144.

MCVOY, L. AND KLEIMAN, S. 1991. Extent-like performance from a unix file system. In *Proceedings on 1991 Winter USENIX Conference* (Jan. 1991), USENIX Assoc., Berkeley, CA, 1–11.

NCR. 1992. Journaling file system administrator guide, release 2.00. NCR Doc. D1-2724-A (Apr.). NCR Knowledge Lab, National Cash Register Co., London, UK.

OHTA, M. AND TEZUKA, H. 1990. A fast /tmp file system by delay mount option. In *Proceedings of the Summer USENIX Conference* (June), USENIX Assoc., Berkeley, CA, 145–150.

OUSTERHOUT, J. 1990. Why aren't operating systems getting faster as fast as hardware?. In *Proceedings of the Summer USENIX Conference* (June), USENIX Assoc., Berkeley, CA, 247–256.

RITCHIE, D. AND THOMPSON, K. 1978. The unix time-sharing system. *Bell Syst. Tech. J. 57*, 6 (Jul/Aug), 1905–1930.

ROSENBLUM, M. AND OUSTERHOUT, J. K. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 26–52.

SELTZER, M., BOSTIC, K., MCKUSICK, M., AND STAELIN, C. 1993. An implementation of a log-structured file system for unix. In *Proceedings of the Winter Usenix Conference* (Jan.), USENIX Assoc., Berkeley, CA, 201–220.

SELTZER, M., GANGER, G., MCKUSICK, M., SMITH, K., SOULES, C., AND STEIN, C. 2000. Logging versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the USENIX Technical Conference* (June), USENIX Assoc., Berkeley, CA.

SELTZER, M., SMITH, K., BALAKRISHNAN, H., CHANG, J., MCMAINS, S., AND PADMANABHAN, V. 1995. File system logging versus clustering: A performance comparison. In *Proceedings of the 1995 Winter USENIX Conference* (Jan.), USENIX Assoc., Berkeley, CA, 249–264.

SMITH, K. AND SELTZER, M. 1996. A comparison of ffs disk allocation algorithms. In *Proceedings of the on 1996 USENIX Technical Conference* (San Diego, CA, Jan.), USENIX Assoc., Berkeley, CA, 15–25.

STONEBRAKER, M. 1987. The design of the Postgres storage system. In *Proceedings of the 13th Confererence on Very Large Data Bases* (Brighton, England, Sept., 1987), VLDB Endowment, Berkeley, CA, 289–300.

WORTHINGTON, B., GANGER, G., AND PATT, Y. 1994. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 conference on Measurement and modeling of computer systems* (SIGMETRICS '94, Vanderbilt Univ., Nashville, TN, May 16–20, 1994), L. Dowdy, R. Bunt, and B. D. Gaither, Eds. ACM Press, New York, NY, 241–251.

WU, M. AND ZWAENEPOEL, W. 1994. Envy: A non-volatile, main memory storage system. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS VI, San Jose, CA, Oct. 4–7), F. Baskett and D. Clark, Eds. ACM Press, New York, NY, 86–97.

Received: August 1999; revised: February 2000; accepted: February 2000