CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Harvard University - DASH

DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Logging versus Soft Updates: Asynchronous Meta-data Protection in File Systems

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

Citation	Seltzer, Margo I., Gregory R. Granger, M. Kirk McKusick, Keith A. Smith, Craig A.N. Soules, and Christopher A. Stein. 1999. Logging versus Soft Updates: Asynchronous Meta-data Protection in File Systems. Harvard Computer Science Group Technical Report TR-07-99.
Citable link	http://nrs.harvard.edu/urn-3:HUL.InstRepos:24829598
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Logging versus Soft Updates: Asynchronous Meta-data Protection in File Systems

Margo I. Seltzer, Gregory R. Ganger,
M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, Christopher A. Stein

Abstract

The UNIX Fast File System (FFS) is probably the most widely-used file system for performance comparisons. However, such comparisons frequently overlook many of the performance enhancements that have been added over the past decade. In this paper, we explore the two most commonly used approaches for improving the performance of meta-data operations and recovery: logging and Soft Updates.

The commercial sector has moved en masse to logging file systems, as evidenced by their presence on nearly every server platform available today: Solaris, AIX, Digital UNIX, HP-UX, Irix, and Windows NT. On all but Solaris, the default file system uses logging. In the meantime, Soft Updates holds the promise of providing stronger reliability guarantees than logging, with faster recovery and superior performance in certain boundary cases.

In this paper, we explore the benefits of both Soft Updates and logging, comparing their behavior on both microbenchmarks and workload-based macrobenchmarks. We find that logging alone is not sufficient to "solve" the meta-data update problem. If synchronous semantics are required (i.e., meta-data operations are durable once the system call returns), then the logging systems cannot realize their full potential. Only when this synchronicity requirement is relaxed can logging systems approach the performance of systems like Soft Updates. Our asynchronous logging and Soft Updates systems perform comparably in most cases. While Soft Updates excels in some meta-data intensive microbenchmarks, it outperforms logging on only two of the four workloads we examined and performs less well on one.

1 Introduction

For the past several decades, a recurring theme in operating system research has been file system performance. With the large volume of operating systems papers that focus on file systems and their performance, we do not see any change in this trend. Conventionally, any new file system designs or file system improvements are compared to the performance of some derivative of the 4.2BSD Fast File System (FFS) [McKusick84]. However, such comparisons ignore significant improvements that have been made to FFS during the past decade.

There are three major performance barriers in the original FFS: I/Os are issued in small (i.e., block size) units, disk seeks are often required between accesses to different files, and meta-data operations (e.g., creates, deletes) are performed synchronously. It is frequently argued that large caches address the issues around read performance (small I/Os and seeks between accesses to small files), so much of the literature has focused on addressing these first two issues in the context of writes. The first of these barriers (small I/O sizes) has largely been overcome by clustering sequential reads and writes to the same file [Peacock88, McVoy91, Seltzer93]. This solves the problem for large files. Log-structured file systems optimize all writes and avoid synchronous meta-data updates [Rosenblum92]. They also improve read performance when the read pattern matches the write pattern. The Co-locating FFS [Ganger97] solves the inter-file access problem for both reads and writes when the data access pattern matches the namespace locality; that is, when small files in the same directory are accessed together. The synchronous update problem has been addressed most directly through logging systems [Hagmann87, Chutani92] and Soft Updates systems [Ganger94]. Unfortunately, none of the previous work quantifies how much each of these barriers actually contributes to end-to-end application performance. Furthermore, there is no indication as to which approach, Soft Updates or logging, offers the superior solution to the synchronous meta-data problem.

In this paper, we focus on the performance impact of synchronous meta-data operations and evaluate the alternative solutions to this problem. In particular, we compare Soft Updates to logging under a variety of conditions and find that while their performance is comparable, each provides a different set of semantic guarantees.

The contributions of this work are: The design and evaluation of two logging file systems, both FFS-compatible; a novel logging architecture where the log is implemented as a stand-alone file system whose services may be used by other file systems or applications apart from the file system; and a quantitative comparison between Soft Updates and logging.

The rest of this paper is organized as follows. In Section 2, we discuss how to maintain the integrity of file system data structures. In Section 3, we discuss our two logging implementations. In Section 4, we describe our benchmarking methodology and framework. In Section 5, we present our experimental results, and in Section 6 we discuss related work. In Section 7, we conclude.

2 Meta-Data Integrity

File system operations can broadly be divided into two categories, data operations and meta-data operations. Data operations act upon actual user data, reading or writing data from/to files. Meta-data operations modify the structure of the file system, creating, deleting, or renaming files, directories, or special files (e.g., links, named pipes, etc.).

During a meta-data operation, the system must ensure that data are written to disk in such a way that the file system can be recovered to a consistent state after a system crash. FFS provides this guarantee by requiring that when an operation (e.g., a create) modifies multiple pieces of meta-data, it must write those data to disk in a fixed order. (E.g., Create writes the new inode before writing the directory that references that inode.) Historically, FFS has met this requirement by synchronously writing each block of meta-data. Unfortunately, synchronous writes can significantly impair the ability of a file system to achieve high performance in the presence of meta-data operations. There has been much effort, in both the research community and industry, to remove this performance bottleneck. In the following sections, we discuss some of the most common approaches to solving the *meta-data update* problem, beginning with a discussion of Soft Updates [Ganger94] and logging [Hagmann87], the two techniques under analysis here.

2.1 Soft Updates

With Soft Updates, systems maintain dependency information in memory. This dependency information identifies which pieces of data must be written to disk before which other pieces of data, and the system ensures that these ordering constraints are met [Ganger94]. This section provides a brief description of Soft Updates; much more detail can be found in other publications [Ganger95, McKusick99, Ganger00].

When using Soft Updates to maintain meta-data consistency, the file system uses delayed writes (i.e., write-back caching) for meta-data changes. Because most meta-data blocks contain many pointers, cyclic dependencies occur frequently if dependencies are recorded only at the block level. Therefore, Soft Updates tracks dependencies on a per-pointer basis instead. Each block in the system has a list of all the meta-data dependencies that are associated with that block. The system is free to use any algorithm it wants to select the order in which the blocks are written. When the system selects a block to be written, it allows the Soft Updates code to review the list of dependencies associated with that block. If there are any dependencies that require other blocks to be written before the meta-data in the current block can be written to disk, then that piece of meta-data in the current block is rolled back to an earlier, safe state. When all needed rollbacks are completed, the initially selected block is written to disk. After the write has completed, the system deletes any dependencies that are fulfilled by the write. In addition, the system restores any rolled back values to their current value so that applications inspecting the block will see the value that they expect. This *dependency-required rollback* allows the system to break dependency cycles. With Soft Updates, applications always see the most recent copies of meta-data blocks and the disk always sees copies that are consistent with its other contents.

Soft Updates rollback operations may cause more writes than would be minimally required if integrity were ignored. Specifically, when an update dependency causes a rollback of the contents of an inode or a directory block before a write operation, it must roll the value forward when the write completes. The effect of doing the roll forward immediately makes the block dirty again. If no other changes are made to the block before it is again written to the disk, then the roll forward has generated an extra write operation that would not otherwise have occurred. To minimize the frequency of such extra writes, the syncer task and cache reclamation algorithms attempt to write dirty blocks from the cache in an order that minimizes the number of rollbacks Soft Updates incurs.

If a Soft Updates system crashes, the only inconsistencies that can appear on the disk are blocks and inodes that are marked as allocated when they are actually free. As these are not fatal errors, the Soft Updates file system can be mounted and used immediately, albeit with a possible decrease in the available free space. A background process, similar to *fsck*, can scan the file system to correct these errors [McKusick99].

2.2 Logging Systems

Logging or journaling file systems maintain an auxiliary data structure that functions as a log. The contents of this log describe the meta-data operations that the file system has performed. The system ensures that the log is written to disk before any pages containing data modified by the corresponding operations. If the system crashes, the log system replays the log to bring the file system to a consistent state. Logging systems always perform additional I/O to maintain ordering information (i.e., they write the log). However, these additional I/Os can be efficient, because they are sequential and they may allow meta-data to be cached longer, avoiding multiple writes of a frequently-updated piece of meta-data.

Logging systems are based on standard database write-ahead-logging techniques [Gray93]. Before a piece of data is updated, the system writes a log record describing the update. In a database environment, this log record typically contains enough information to redo the operation if the system crashes after it completes as well as enough information to undo the operation if the application requests it or if the system crashes before the operation has been logically completed. A logging file system usually only needs to support the redo operation, since most file system operations are themselves atomic and cannot be undone once the system call has been completed. Techniques for constructing write-ahead logging file systems are well known [Hagmann87, Chutani92, Vahalia95].

In the context of building a logging file system, the key design issues are:

- Location of the log.
- Management of the log (i.e., space reclamation and checkpointing).
- Integration or interfacing between the log and the main file system.
- Recovering the log.

In Section 3, we present two alternative designs for incorporating logging in FFS, focusing on how each addresses these issues.

2.3 Other Approaches

Some vendors have addressed the meta-data update problem by throwing hardware at it, most notably non-volatile RAM (NVRAM). Systems equipped with NVRAM can avoid synchronous writes, safe in the knowledge that the modified meta-data are persistent after a failure. On a system crash, the contents of the NVRAM can be written to disk or simply accessed during the reboot and recovery process.

The Rio system provides a similar solution [Chen96]. Rio assumes that systems have an uninterrupted power supply, so memory never loses its contents. Part of the normal main memory is treated as a protected region, maintained with read-only protection during normal operation. The region is made writable only briefly to allow updates. This memory is then treated as non-volatile and used during system restart after a crash. Just as in the non-volatile case, storing meta-data in Rio memory eliminates the need for synchronous writes.

Log-structured file systems (LFS) offer a different solution to the meta-data update problem. Rather than using a conventional update-in-place file system, log-structured file systems write all modified data (both data blocks and meta-data) in a segmented log. Writes to the log are performed in large segment-sized chunks. By carefully ordering the blocks within a segment, LFS guarantees the ordering properties that must be ensured to update meta-data reliably. Unfortunately, it may be the case that all the related meta-data cannot be written in a single disk transfer. In this case, it is necessary for LFS to make sure it can recover the file system to a consistent state. The original LFS implementation [Rosenblum92] solved this problem by adding small log entries to the beginning of segments, applying a logging approach to the problem. A later implementation of LFS [Seltzer93] used a simple transaction-like interface to make segments temporary, until all the meta-data necessary to ensure the recoverability of the file system was on disk.

3 Logging Implementations

In this section, we describe two different implementations of logging applied to the fast file

system. The first implementation (LFFS-file) maintains a circular log in a file on the FFS, in which it records logging information. The buffer manager enforces a write-ahead logging protocol to ensure proper synchronization between normal file data and the log.

The second implementation (LFFS-wafs) records log records in a separate stand-alone service, a write-ahead file system (WAFS). In theory, this stand-alone logging service could be used by other clients, such as a database management system, as was done in the Quicksilver operating system [Haskin88].

3.1 LFFS-file

LFFS-file augments 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 meta-data update, which is then recorded in the log. The log is stored in a pre-allocated 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 an update to the corresponding block. The former is used to ensure that log space is reclaimed only when it is no longer needed, and the latter is used to ensure that all relevant log entries are written to disk before the block. These two requirements are explained further below.

A core requirement of write-ahead logging is that the logged description of an update must propagate to persistent storage before the updated blocks. The function LFFS-file calls during initiation of disk writes enforces this requirement. By examining the buffer headers of blocks to be written, LFFS-file can determine those portions of the log that must first be written. Specifically, all log information up to the last log entry relating to the to-be-written block must be flushed. When such log flushing is required, LFFS-file synchronously initiates the log flush, which requires a write of the file system superblock as well, and then initiates the original disk write. In most cases, the relevant log entries will already be on disk and synchronous flushes will be unnecessary.

Another core requirement is that log space must be reclaimable, since the log is implemented as a circular buffer. LFFS-file satisfies this requirement via standard database checkpointing techniques [Gray93]. Specifically, space is reclaimed in two ways. First, during the periodic syncer daemon activity (once per second), the logging code examines the buffer headers of all cached blocks to determine the oldest log entry for which a cached block has not yet been written. This becomes the new start of the log, releasing previously live space in the log. The log's start is recorded in the superblock, so that roll-forward can occur efficiently during crash recovery. While this approach is usually sufficient to keep the log from becoming full, the logging code will force a checkpoint when necessary. Such a forced checkpoint causes all blocks with updates described by some range of log entries to be immediately written to persistent storage.

3.2 LFFS-wafs

LFFS-wafs implements its log in an auxiliary file system that is associated with the FFS. The logging file system (WAFS, for Write-Ahead File System) is a simple, free-standing file system that supports a limited number of operations: it can be mounted and unmounted, it can append data, and it can return data by sequential or keyed reads. The keys for keyed reads are log-sequence-numbers (LSN), which correspond to logical offsets in the log. Like the logging file in LFFS-file, the log is implemented as a circular buffer within the physical space allocated to the file system. When data are appended to the log, WAFS returns the logical offset at which the data were written. This LSN is then used to tag the data described by the logging operation exactly as is done in LFFS-file (low and high LSNs are maintained for each modified buffer in the cache).

LFFS-wafs uses the same checkpointing scheme as that used for LFFS-file. LFFS-wafs also enforces the standard write-ahead logging protocol as described for LFFS-file.

Because LFFS-wafs is implemented as two disjoint file systems, it provides a great deal of flexibility in file system configuration. First, the logging system can be used to augment any file system, not just an FFS. Second, the log can be parameterized and configured to adjust the performance of the system. In the simplest case, the log can be located on the same drive as the file system. This will necessarily introduce some disk contention between log writes and foreground file system activity. A higher performing alternative is to mount the log on a separate disk, ideally a small, high speed one. In this case, the log disk should never seek and the data disk will perform no more

seeks than does a conventional FFS. Finally, the log could be located in a small area of battery-backed-up or non-volatile RAM. This option provides the greatest performance, at somewhat higher cost.

LFFS-wafs can also be mounted either synchronously or asynchronously, trading off performance for durability guarantees. By default, it is mounted synchronously so that meta-data operations are persistent upon return from the system call. That is, log messages for creates, deletes, and renames are flushed to disk before the system call returns, while log messages corresponding to bitmap operations are cached in memory until the current log block is flushed to disk. For higher performance, the log can be mounted to run asynchronously. In this case, the system maintains the integrity of the file system, but does not provide synchronous FFS durability guarantees.

LFFS-wafs requires minimal changes to FFS and to the rest of the operating system. The FreeBSD 4.0 operating system was augmented to support LFFS-wafs by adding approximately 16 logging calls to the ufs layer (that is the Unix file system layer, independent of the underlying file system implementation) and 13 logging calls to manage bitmap allocation in the FFS-specific portion of the code. These logging calls are writes into the WAFS file system. The only other change is in the buffer management code, which is enhanced to maintain and use the LSNs to support write-ahead logging. The buffer management changes required approximately 200 lines of additional code.

3.3 Recovery

Both logging file systems require database-like recovery after system failure. First, the log is recovered. In both LFFS-file and LFFS-wafs, a superblock contains a reference to the last log checkpoint. In LFFS-file, the superblock referenced is that of the FFS; in LFFS-wafs, the superblock is that of the WAFS. In LFFS-file, checkpoints are taken frequently and the state described in the superblock is taken as the starting state. In LFFS-wafs, superblocks are written infrequently and the log recovery code must find the end of the log. It does so by reading the log beginning with the last checkpoint and reading sequentially until it locates the end of the log. Log entries are timestamped and checksummed so that the log recovery daemon can easily detect when the end of log is reached.

Once the log has been recovered, recovery of the main file system begins. This process is identical to standard database recovery [Gray93]. First, the log is read from its logical end back to the most recent checkpoint and any aborted operations are undone. In LFFS-wafs, creates are the only potentially aborted operations. Creates require two log records, one to log the allocation of the inode and one to log the rest of the create. If the system crashes between these two operations, then the first operation must be undone (aborted) rather than rolled forward. This happens on the backward pass through the log. On the forward pass through the log, any updates that have not yet been written to disk are reapplied. Most of the log operations are idempotent, so they can be redone regardless of whether the update has already been written to disk. Those operations that are not idempotent affect data structures (e.g., inodes) that have been augmented with LSNs. During recovery, the recovery daemon compares the LSN in the current log record to that of the data structure and applies the update only if the LSN of the data structure matches the LSN logged in the record.

Once the recovery daemon has completed both its backward and forward passes, all the dirty data blocks are written to disk, the file system is checkpointed and normal processing continues. The length of time for recovery is proportional to the inter-checkpoint interval.

4 Measurement Methodology

The goal of our evaluation is twofold. First, we seek to understand the trade-offs between the two different approaches to improving the performance of meta-data operations and recovery. Second, we want to understand how important the meta-data update problem is to some typical workloads.

We begin with a set of microbenchmarks that quantify the performance of the most frequently used meta-data operations and that validate that the performance difference between the two systems is limited to meta-data operations (i.e., that normal data read and write operations behave comparably). Next, we examine macrobenchmarks. In addition to providing end-to-end performance results from the macrobenchmarks, we characterize their initial FFS performance and access pattern. We use the microbenchmark results and this characterization to predict and explain the actual end-to-end numbers that we measure.

	FreeBSD Platform
Motherboard	Intel ASUS P38F, 440BX Chipset
Processor	500 Mhz Xeon Pentium III
Memory	512 MB, 10 ns
Disk	3 9 GB 10,000 RPM Seagate Cheetahs Disk 1: Operating system, /usr, and swap Disk 2: 9088 MB test partition Disk 2: 128 MB log partition Disk 3: 128 MB log partition
I/O Adapter	Adaptec AHA-2940UW SCSI
OS	FreeBSD-current config: GENERIC + SOFTUPDATES - bpfilter - unnecessary devices

Table 1. System Configuration. We are using the "current" version of FreeBSD for submission, but will move to a stable, easily identifiable version before final copy.

	File System Configurations
FFS	Standard FFS
FFS-async	FFS mounted with the async option
Soft-Updates	FFS mounted with Soft Updates
LFFS-file	FFS augmented with a file log log writes are asynchronous
LFFS-wafs-1sync	FFS augmented with a WAFS log log writes are synchronous
LFFS-wafs-1async	FFS augmented with a WAFS log log writes are asynchronous
LFFS-wafs-2sync	FFS augmented with a WAFS log log is on separate disk log writes are synchronous
LFFS-wafs-2async	FFS augmented with a WAFS log log is on a separate disk log writes are asynchronous

Table 2. File System Configurations.

4.1 The Systems Under Test

We compared the two LFFS implementations to both FFS and Soft Updates. We also compared them to FFS-async, an FFS file system mounted with the async option. In this configuration, all file system writes are performed asynchronously. Because it does not include the overhead of either synchronous meta-data updates, update ordering, or logging, we expect this case to represent the best case performance. However, it is important to note that such a file system is not practical in production use as it may be unrecoverable after system failure. We have multiple, identical test platforms whose configuration is shown in Table 1. Table 2 shows the various file system configurations under test.

4.2 The Microbenchmarks

Our microbenchmark suite is reminiscent of any number of the microbenchmark results that appear in the file system literature [Rosenblum92, Ganger94, Seltzer95]. The basic structure is that for each of a large number of file sizes, we create, read, write, and delete either 128 MB of data or 512 files, whichever generates the most files. The files are allocated 50 per directory to avoid excessively long lookup times. The files are always accessed in the same order.

We add one microbenchmark to the suite normally presented: a create/delete benchmark that isolates the cost of meta-data operations in the absence of any data writing. The create/delete benchmark creates and immediately deletes 50,000 0-length files, with each newly-created file deleted before moving on to the next. This stresses the performance of temporary file creation/deletion.

The results of all the microbenchmarks are presented and discussed in Section 5.1.

4.3 The Macrobenchmarks

The goal of our macrobenchmarking activity is to demonstrate the impact of meta-data operations for several common workloads. As there are an infinite number of workloads, it is not possible to accurately characterize how these systems will benefit all workloads. Instead, we show a variety of workloads that demonstrate a range of effects that meta-data operations can introduce.

4.3.1 The SSH Build Benchmark

The most widely used benchmark in the file system literature is the Andrew File System Benchmark [Howard88]. Unfortunately, this benchmark no longer stresses the file system, because its data set is too small. We have constructed a benchmark reminiscent of Andrew that does stress a file system.

Our benchmark unpacks, configures, and builds a medium-sized software package (ssh version 1.2.26 [Ylonen96]). In addition to the end-to-end timing measurement, we also measure the time for each of the three phases of this benchmark:

- *Unpack* This phase unpacks a compressed tar archive containing the `ssh` source tree. This phase highlights meta-data operations, but unlike our microbenchmarks, does so in the

context of a real workload. (I.e., it uses a mix of file sizes.)

- *Config* This phase determines what features and libraries are available on the host operating system and generates a Makefile reflecting this information. To do this, it compiles and executes many small test programs. This phase should not be as meta-data intensive as the first, but because most of the operations are on small files, there are more meta-data operations than we see in the final phase.
- *Build* This phase executes the Makefile built during the config phase to build the `ssh` executable. It is the most compute-intensive phase of the benchmark (90% CPU utilization running on FFS). As a result, we expect to see the least performance difference here.

We run the three phases of the benchmark consecutively, so the config and build phases run with the file system cache warmed by the previous phases.

4.3.2 Net News

A second workload that we examine is that of a Netnews server. We use a simplified version of Karl Swartz's Netnews benchmark [Swartz96]. It simulates the work associated with unbatching incoming news articles and expiring old articles by replaying traces from a live news server. The benchmark runs on a file system that is initialized to contain 2 GB of simulated news data. This data is broken into approximately 520,000 files spread over almost 7,000 directories. The benchmark itself consists of two phases:

- *Unbatch* This phase creates 78,000 new files containing 270 MB of total data.
- *Expire* This phase removes 91,000 files, containing a total of 250 MB of data.

In addition to the sheer volume of file system traffic that this benchmark generates, this workload has two other characteristics that effect the file system. First, successive create and delete operations seldom occur in the same directory. Because FFS places different directories in different regions of the disk, this results in little locality of reference between successive (synchronous) meta-data operations, causing a large number of disk seeks.

The second characteristic of interest is that due to the large data set that the benchmark uses, it is difficult for the file system to maintain all of the meta-data in its buffer cache. As a result, even the Soft Updates and logging file systems that we are studying may incur many seeks, since the meta-data on which they need to operate may not be in cache. It is important to note that our benchmark is actually quite small compared to current netnews loads. Two years ago, a full news feed could exceed 2.5 GB of data, or 750,000 articles per day [Christenson97, Fritchie97]. Anecdotal evidence suggests that a full news feed today is 15 – 20 GB per day.

4.3.3 SDET

Our third workload is 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 [Gaede81, Gaede82]. The reported metric is scripts/hour as a function of the script concurrency.

4.3.4 Postmark

The PostMark benchmark was designed by Jeffrey Katcher to model the workload seen by Internet Service Providers under heavy load [Katcher97]. Specifically, the workload is meant to model a combination of electronic mail, netnews, and web-based commerce transactions. To accomplish this, PostMark creates a large set of files with random sizes within a set range. The files are then subjected to a number of transactions. These transactions consist of a pairing of create or delete a file, and read or append to a file. Each pair of transactions is chosen randomly, and can be biased with modifiable parameters. File creation involves creating a file of a random size within the set range. File deletion involves unlinking a file from the active set. File read involves choosing a random file and reading the entire file in transaction block size chunks. File append opens a random file, seeks to the end of the file, and writes a random amount of data, not exceeding the maximum file size. Our experiments use the default PostMark configuration of 10,000 files with a size range of 512 bytes to 16 KB. One run of this default performs 20,000 transactions with no bias toward any particular transaction type and with a transaction block size of 512 bytes.

Feature	File Systems
Meta-data updates are synchronous	FFS, LFFS-wafs-[12]sync
Meta-data updates are asynchronous	Soft-Updates LFFS-file LFFS-wafs-[12]async
File data blocks are freed in background	Soft-Updates
New data blocks are written before inodes	Soft Updates
Recovery requires full file system scan	FFS
Recovery requires log replay	LFFS-*
Recovery is non-deterministic and may be impossible	FFS-async

Table 3. Feature comparison.

5 Results

Before examining the results in detail, it is important to understand the different systems under test, the guarantees that they make, and how those guarantees affect their performance. These differences are summarized in Table 3.

Both logging and Soft Updates systems ensure the integrity of meta-data operations, but they provide slightly different semantics. The three areas of difference are the durability of meta-data operations such as create and delete, the status of the file system after a reboot and recovery, and the guarantees made about the data in files after recovery.

The original FFS implemented meta-data operations such as create, delete, and rename synchronously, guaranteeing that when the system call returns, the meta-data changes are persistent. Some FFS variants (e.g., Solaris) have made deletes asynchronous and other variants (e.g., SVR4) did not make create and rename synchronous. However, on FreeBSD, FFS does guarantee that create, delete, and rename operations are synchronous.

FFS-async makes no such guarantees, and furthermore does not guarantee that the resulting file system can be recovered (via fsck) to a consistent state after failure. Thus, instead of being a viable candidate for a production file system, FFS-async provides an upper bound on the performance one can expect to achieve with the FFS derivatives.

Soft updates provides looser guarantees about when meta-data changes reach disk. Create, delete, and rename operations reach disk within 35 seconds of the corresponding system call. Soft updates also guarantees that the file system can be restarted without any file system recovery. At such a time, file system integrity is assured, but freed blocks and inodes may not yet be marked as free and, as such, the file system may under report the amount of free space. A background process, similar to `fsck`, restores the file system to an accurate state with respect to free blocks and inodes [McKusick99].

The logging file systems provide a spectrum of points between the synchronous guarantees of FFS and the relaxed guarantees of Soft Updates. When the log is maintained synchronously, the logging systems provide guarantees identical to FFS; when the log is run asynchronously, the logging systems provide guarantees identical to Soft Updates, except that they require a short recovery phase after system restart to make sure that all operations in the log have been applied to the file system.

The third area of different semantics is in the guarantees made about the status of data in recently created or written to files. In an ideal system, one would never allow meta-data to be written to disk before the data referenced by that meta-data is on the disk. For example, if block 100 is allocated to file 1, you would want block 100 to be on disk before file 1's inode is written, so that file 1 is not left containing bad (or highly sensitive) data. FFS has never made such guarantees. However, Soft Updates uses its dependency information to roll back any meta-data operations for which the corresponding data blocks have not yet been written to disk. This guarantees that no meta-data ever points to bad data. In our tests, the penalty for enforcing this ranges from 0 (in the less meta-data intensive `ssh` benchmark described in Section 4.3.1) to approximately 8% (in the meta-data intensive news benchmark, described in Section 4.3.2). Neither of the logging file systems provides this stronger guarantee. These differences should be taken into account when comparing performance results.

5.1 Microbenchmark Results

This collection of microbenchmarks separates meta-data operations from conventional read and write tests. As the systems under test all use the same algorithms and underlying disk representation, we expect to see no significant performance difference for read and write tests. For the create and delete tests, we expect both Soft Updates and the logging systems to provide significantly improved performance over FFS. The important question is how close these systems come to approaching the performance of FFS-async, which

might be viewed as the best performance possible under any FFS-based system.

All of the microbenchmarks represent the average of at least five runs; standard deviations were less than 1% of the average. The benchmarks were run with a cold file system cache.

Figure 1 and Figure 2 show the read and write performance of the various systems. As expected, all file system configurations perform comparably.

Figure 3 shows the results of the create microbenchmark. The most surprising result is that Soft Updates outperforms FFS-async for small file sizes. We believe that this counter-intuitive behavior is a result of an interaction between our benchmarking methodology (unmounting the file system before each run) and the fact that Soft Updates prereads all of its cylinder groups at mount time. *For the final paper we will either modify our benchmarking methodology or provide a more complete explanation of this phenomenon.*

The next significant observation is the shape of the curves with the various drops observed in nearly all the systems. These are idiosyncrasies of the FFS disk layout and writing behavior. In particular, on our configuration, I/O is clustered into 64 KB units before being written to disk. This means that at 64KB, many of the asynchronous systems achieve nearly the maximum throughput possible. At 96 KB, we see a drop because we are doing two writes and losing a rotation between them. At 104 KB we see an additional drop due to the first indirect block, which ultimately causes an additional I/O. From 104 KB to our maximum file size of 4 MB we see a steady increase back up to the maximum throughput.

At the low performance end of the spectrum, we see that FFS and LFFS-wafs-1sync perform comparably until the introduction of the indirect block. This introduces a synchronous write on FFS which is asynchronous on LFFS-wafs-1sync, so LFFS-wafs-1sync takes a lead. As file size grows, the two systems converge until FFS ultimately overtakes LFFS-wafs-1sync, because it is not per-

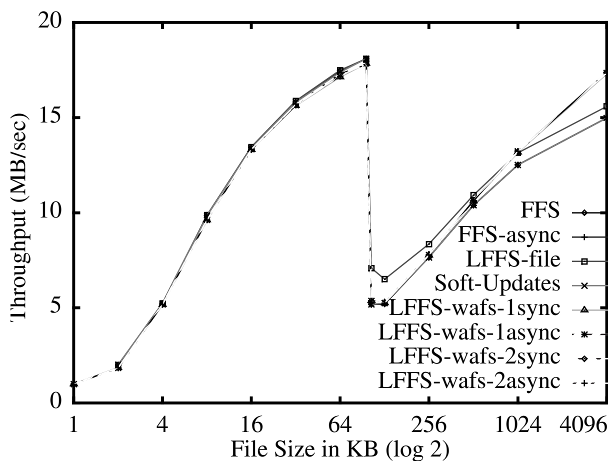

Figure 1. Read Performance as a function of file size. As expected, the systems show no significant difference in read performance.

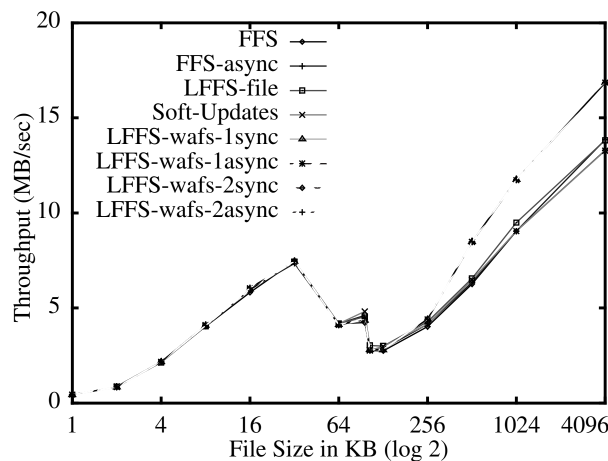

Figure 2. Write Performance as a function of file size. Like the read case, the systems show no significant difference in write performance.

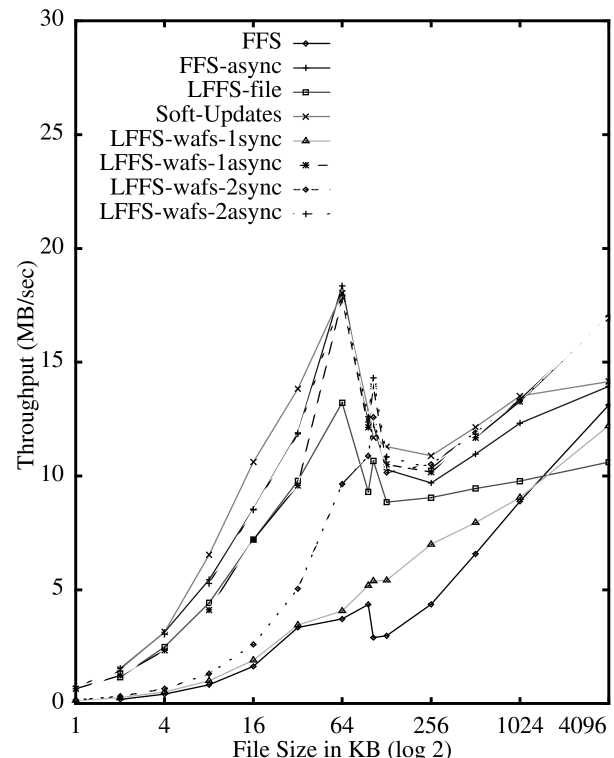

Figure 3. Create Performance as a function of file size.

forming costly seeks between the log and data partitions.

For small file sizes, where meta-data operations dominate, LFFS-wafs-2async offers a significant improvement over LFFS-wafs-2sync. As file size grows such that the benchmark time is dominated by data transfer time, the two systems converge.

The delete microbenchmark performance is shown in Figure 4. Note that performance is expressed in files per second. This microbenchmark highlights a feature of Soft Updates that is frequently overlooked. Not only does Soft Updates make meta-data write operations asynchronous, it also performs deletes in the background. That is, when a delete is issued, Soft Updates removes the file's name from the directory hierarchy and creates a remove dependency associated with the buffer holding the corresponding directory data. When that buffer is written, all the delete dependencies associated with the buffer are passed to a separate background syncer task, which does the work of walking the inode and indirect blocks freeing the associated file data blocks. This background deletion typically occurs 30 to 60 seconds after the system call that triggered the file deletion. Thus the apparent time to remove a file is short, leading to the outstanding performance of Soft Updates on the delete microbenchmark.

As soon as the file size surpasses 96 KB, all of the systems without Soft Updates suffer a significant performance penalty, because they are forced to read the first indirect block in order to reclaim disk blocks. In contrast, by backgrounding the delete, Soft Updates removes this read from the measurement path.

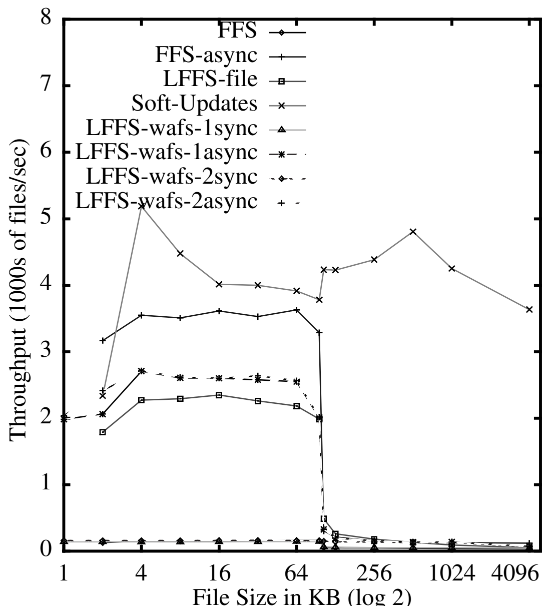

Figure 4. Delete Performance as a function of file size.

In the region up to and including 96 KB, Soft Updates still enjoys increased performance because it performs deletes in the background, but the effect is not as noticeable. The logging systems write a log message per freed block, so they suffer from a slight decrease in performance as the number of blocks in the file increases.

Our final microbenchmark is the 0-length file create/delete benchmark. This emphasizes the benefits of asynchronous meta-data operations outside the presence of any data to allocate/free, thus removing the effects of the ratio of data writes to meta-data writes and the backgrounding of frees in Soft Updates. This benchmark also eliminates the overheads of compulsory read misses in the file system cache, as the test repeatedly accesses the same directory and inode data. Table 4 shows the results for this benchmark. As this benchmark does nothing outside of meta-data operations, the synchronous logging implementations behave identically to FFS. The asynchronous logging implementations and Soft Updates perform comparably, achieving less than half the performance of FFS-async. The reason for this is that when the system is running completely asynchronously, the files are created and deleted entirely within the buffer cache and no disk I/O is needed. The logging systems, however, still write log records to record this activity, thus requiring disk I/O. LFFS-file outperforms the WAFS-based logging schemes because it writes log blocks in larger clusters. The disappointing performance for Soft Updates is due to an implementation problem that causes many non-persistent files to be deleted as though they

FFS	80.4	(0.0)
FFS-async	10782.8	(47.8)
Soft-Updates	4962.0	17.7)
LFFS-file	7695.7	(182.6)
LFFS-wafs-1sync	80.3	(0.0)
LFFS-wafs-1async	4846.4	(74.7)
LFFS-wafs-2sync	80.5	(0.0)
LFFS-wafs-2async	4692.7	(194.0)

Table 4. 0-length File Create/Delete in Files per Second. Standard deviations are shown in parentheses.

were already persistent, which results in added disk I/O and inode utilization.

This benchmark also shows another advantage of the asynchronous approaches. Not only do they improve file system performance, they also allow performance to scale with processor speed, rather than disk speed.

5.2 Macrobenchmark Results

In this section, we present all results relative to the performance of FFS-async, since that is, in general, the best performance we can hope to achieve.[1] For throughput results (where larger numbers are better), we normalize performance by dividing the measured result by that of FFS-async. For elapsed time results (where smaller numbers are better), we normalize by taking FFS-async and dividing by each measured result. Therefore, regardless of the measurement metric, in all results presented, numbers greater than one indicate performance superior to FFS-async and numbers less than one indicate performance inferior to FFS-async. As a result, the performance of FFS-async in each test is always 1.0, and therefore is not shown.

5.2.1 Ssh Build

As explained in Section 4.2.1, this benchmark simulates unpacking, configuring and building `ssh` [Ylonen96]. Table 5 reports the normalized performance of our systems. While many of the results are as expected, there are several important points to note. For the CPU-intensive config and build phases, most of the logging and Soft Updates systems perform almost as well as FFS-async, with the synchronous logging systems exhibiting somewhat reduced throughput, due to the few synchronous file creations that must happen.

During the unpack phase, Soft Updates is the only system able to achieve performance comparable to FFS-async. The synchronous logging systems demonstrate little improvement over FFS, indicating that the ratio of meta-data operations to data operations is significant and that the meta-data operations account for nearly all the time during this phase. Both the LFFS-wafs-async systems approach 90% of the performance of FFS-async. On LFFS-wafs-1async, we attribute the 10% degradation to the seeks required between the log and the data.

The LFFS-file system has slower file create performance on files larger than 64KB, and the build benchmark contains a sufficient number of these to explain its reduced performance on the unpack phase.

	Unpack	Config	Build	Total
FFS	0.13	0.73	0.83	0.74
Soft-Updates	0.98	0.99	1.01	1.01
LFFS-file	0.79	0.98	0.99	1.01
LFFS-wafs-1sync	0.17	0.92	0.92	0.84
LFFS-wafs-1async	0.91	0.98	0.99	0.99
LFFS-wafs-2sync	0.20	0.93	0.94	0.87
LFFS-wafs-2async	0.91	0.99	1.00	1.00

Table 5. Ssh Results Normalized to FFS-async. Data gathered are the averages of 5 runs. All standard deviations were small relative to the averages. As the config and build phases are the most CPU-intensive, they show the smallest difference in execution time for all systems. Unpack, the most meta-data intensive, demonstrates the most significant differences.

5.2.2 Netnews

As described in Section 4.3.2, the Netnews benchmark places a tremendous load on the file system, both in terms of the number of meta-data operations it performs, and the amount of data on which it operates. The impact of these stresses is apparent in the benchmark results shown in Table 6. On this benchmark, all of the file systems are completely disk bound.

All of the asynchronous logging systems approach the performance of FFS-async (within 5%), but neither Soft Updates nor the synchronous systems come close. The Soft Updates performance is largely due to writes caused by dependency-required rollback. Soft updates performed

	Unbatch	Expire	Total
FFS	0.57	0.37	0.48
Soft-Updates	0.86	0.76	0.82
LFFS-file	0.95	0.95	0.95
LFFS-wafs-1sync	0.67	0.46	0.58
LFFS-wafs-1async	0.98	0.94	0.97
LFFS-wafs-2sync	0.88	0.58	0.96
LFFS-wafs-2async	0.99	0.96	0.98

Table 6. Netnews Results Normalized to FFS-async. These results are based on a single run, but we observed little variation between multiple runs of any configuration.

1. We have two different test machines whose code bases have diverged enough to give us results that differ by approximately 10% between the two systems. After submission, we will merge code bases and insure that the numbers from both systems are comparable.

13% more writes than FFS. The major cause for these rollbacks is because the data set exceeds the size of the buffer cache. Frequent cache evictions force Soft Updates to flush blocks to disk in the order indicated by the cache's LRU list, rather than in its preferred order, based on the dependency data.

The relatively poor performance of LFFS-wafs-2sync indicates that there are sufficiently many meta-data operations to cause the synchronous write of the log to become a bottleneck. However, as LFFS-wafs-1sync performs even worse, we see that seeks between the log and the file system also account for a significant performance penalty. When the log writes are asynchronous, these seeks incur a much smaller penalty, evidenced by the closeness in performance of LFFS-wafs-1async and LFFS-wafs-2async.

5.2.3 Sdet

Figure 5 shows the results for the SDET test. Once again we see the systems diverge into the largely asynchronous ones (Soft Updates, LFFS-file, LFFS-wafs-[12]async) and the synchronous ones (FFS, LFFS-wafs-[12]sync), with the synchronous logging systems providing minimal improvement over FFS. As expected, the synchronous schemes drop in performance as script concurrency increases, because the scripts compete for the disk. Soft updates outperforms the other schemes because of its backgrounding of file deletion. We believe that LFFS-file suffers the same performance problem here that we observed in the `ssh` unpack test, namely that it creates files larger than 64 KB more slowly than the other systems.

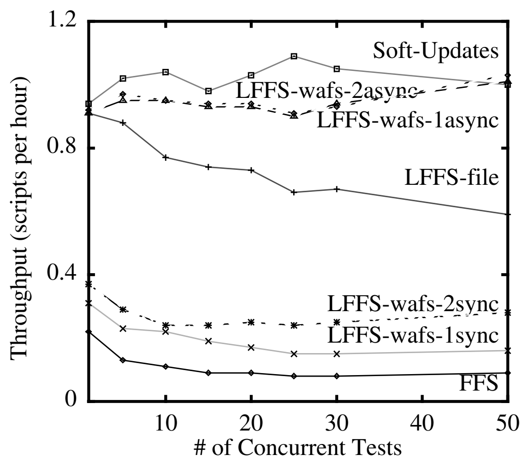

Figure 5. Sdet Results Normalized to FFS-async. The results were averaged from five runs with small standard deviations.

5.2.4 Postmark

This test demonstrates the same phenomenon that we saw in the create microbenchmark, that Soft Updates can actually outperform FFS-async. This is a case that demonstrates the effectiveness of Soft Updates' delayed deletions. As 25% of the operations are file deletions, the lazy deletion of Soft Updates demonstrates performance superior to that of FFS-async.

The asynchronous logging systems approach the performance of FFS-async, with the two-disk approach yielding only marginal benefit (4%) over the one-disk approach. In the synchronous case, LFFS-wafs-1sync provides only marginal improvement over FFS, but the addition of the second disk (LFFS-wafs-2sync) provides a significant benefit. When the log writes are synchronous, this causes the difference between the 1-disk and 2-disk cases. However, in the asynchronous case, the ability to write log records lazily removes the disk seeks from the critical path.

6 Related Work

In Section 2, we discussed much of the work that has been done to avoid synchronous writes in FFS. As mentioned in the introduction, small writes are another performance bottleneck in FFS. Log-structured file systems [Rosenblum92] are one approach to that problem. A second approach is the Virtual Log Disk [Wang99].

Log-structured file systems (LFS) solve both the synchronous meta-data update problem and the small-write problem. Data in an LFS are coalesced and written sequentially to a segmented log. In this way, LFS avoids the seeks that a conventional file system pays in writing data back to its original location. Using this log-structured technique, LFS also solves the meta-data consistency problem by carefully ordering blocks within its segments. Like

	Normalized Performance
FFS	0.24
Soft-Updates	1.15
LFFS-file	0.95
LFFS-wafs-1sync	0.37
LFFS-wafs-1async	0.95
LFFS-wafs-2sync	0.61
LFFS-wafs-2async	0.99

Table 7. Postmark Results Normalized to FFS-async. The pre-normalized results were the averages of 5 runs; standard deviations were all under 2%.

the logging systems, LFS requires a database-like recovery phase after system crash and like Soft Updates, data are written in an order that guarantees the file system integrity. Unlike either Soft Updates or logging, LFS requires a background garbage collector, whose performance has been the object of great speculation and debate [Seltzer93, Blackwell95, Seltzer95, Matthews97]

Building on the idea of log-structured file systems, Wang and his colleagues propose an intelligent disk that performs writes at near maximum disk speed by selecting the destination of the write based upon the position of the disk head [Wang99]. The disk must then maintain a mapping of logical block number to physical location. This mapping is maintained in a virtual log which is written adjacent to the actual data being written. The proposed system exists only in simulation, but seems to offer the promise of LFS-like performance for small writes, with much of the complexity hidden behind the disk interface, as is done in the AutoRaid storage system [Wilkes95]. While such an approach can solve the small-write problem, it does not solve the meta-data update problem, where the file system requires that multiple related structures be consistent on disk. It does however improve the situation by allowing the synchronous writes used by FFS to occur at near maximum disk speed.

Another approach to solving the small-write problem that bears a strong resemblance to logging is the database cache technique [Elkhardt84] and the more recent Disk Caching Disk (DCD) [Hu96]. In both of these approaches, writes are written to a separate logging device, instead of being written back to the actual file system. Then, at some later point when the file system disk is not busy, the blocks are transferred back. This is essentially a two-disk logging approach. The difference between the database cache techniques and the logging file system technique is that the database cache tries to improve the performance of data writes as well as meta-data writes and does nothing to make meta-data operations asynchronous; instead, it makes them synchronous but with a much lower latency. In contrast, DCD places an NVRAM cache in front of the logging disk, making all small writes, including meta-data writes, asynchronous.

7 Conclusions

We draw several conclusions from our comparisons. At a high level, we have shown that both logging and Soft Updates succeed at dramatically improving the performance of meta-data operations. While there are minor differences between the two logging architectures, to a first approximation, they behave comparably. Surprisingly, we see that logging alone is not sufficient to solve the meta-data update problem. If application and system semantics require the synchronicity of such operations, there remains a significant performance penalty, as much as 90% in some cases. In most cases, even with two disks, the penalty is substantial, unless the test was CPU-bound (e.g., the config and build phases of the ssh-build benchmark).

Soft Updates exhibits some side-effects that improve performance, in some cases significantly. Its ability to delay deletes is evidenced most clearly in the microbenchmark results. It is also true that Soft Updates caches inode blocks preferentially, so on workloads where the meta-data is larger than the meta-data cache, we have observed a performance boost as a result. However, this does not seem to manifest itself in any of our macrobenchmarks. For the massive data set of the news benchmark, we see that Soft Updates' ordering constraints prevent it from achieving performance comparable to the asynchronous logging systems. The race between increasing memory sizes and increasing data sets will determine which of these effects is most significant.

If our workloads are indicative of a wide range of workloads (as we hope they are), we see that meta-data operations are significant, even in CPU-dominated tasks such as the ssh-build benchmark where FFS suffers a 25% performance degradation from FFS-async. In our other test cases, the impact is even more significant (e.g., 50% for news and 75% for Postmark).

The implications of such results are important as the commercial sector contemplates technology transfer from the research arena. Logging file systems have been in widespread use in the commercial sector for many years (Veritas, IBM's JFS, Compaq's AdvFS, HP's HPFS10, Irix's XFS), while Soft Updates systems are only beginning to make an appearance. If vendors are to make informed decisions concerning the future of their file systems, analyses such as those presented here are crucial to provide the data from which to make such decisions.

8 References

[Blackwell95] Blackwell, T., Harris, J., Seltzer., M. "Heuristic Cleaning Algorithms in Log-Structured File Systems,"

Proceedings of the 1995 USENIX Technical Conference, 277–288, New Orleans, LA, January 1995.

[Chen96] Chen, P., Ng, W., Chandra, S., Aycock, C., Rajamani, G., Lowell, D., "The Rio File Cache: Surviving Operating System Crashes," *Proceedings of the 7th ASPLOS,* pp. 74–83. Cambridge, MA, Oct. 1996.

[Christenson97] Christenson, N. Beckemeyer, D., Baker, T. "A Scalable News Architecture on a Single Spool," *;login:*, 22(5), pp. 41-45. Jun. 1997.

[Chutani92] Chutani, S., Anderson, O., Kazer, M., Leverett, B., Mason, W.A., Sidebotham, R. "The Episode File System," *Proceedings of the 1992 Winter USENIX Technical Conference,* pp. 43–60. San Francisco, CA, Jan. 1992.

[Elkhardt84] Elkhardt, K., Bayer, R. "A Database Cache for High Performance and Fast Restart in Database Systems," *ACM Transactions on Database Systems,* 9(4), pp. 503–525. Dec. 1984.

[Fritchie97] Fritchie, S. "The Cyclic News Filesystem: Getting INN To Do More With Less," *Proceedings of the 1997 LISA Conference,* pp. 99–111. San Diego, CA, Oct. 1997.

[Gaede81] Gaede, S., "Tools for Research in Computer Workload Characterization," *Experimental Computer Performance and Evaluation*, 1981, ed Ferrari and Spadoni.

[Gaede82] Gaede, S., "A Scaling Technique for Comparing Interactive System Capacities," *Proceedings of the 13th International Conference on Management and Performance Evaluation of Computer Systems,*1982 62–67.

[Ganger94] Ganger, G., Patt, Y. "Metadata Update Performance in File Systems," *Proceedings of the First OSDI,* pp. 49–60. Monterey, CA, Nov. 1994.

[Ganger95] Ganger, G., Patt Y., "Soft Updates: A Solution to the Metadata Update Problem in File Systems," Report CSE-TR-254-95, University of Michigan, Ann Arbor, August 1995.

[Ganger97] Ganger, G., Kaashoek, M.F. "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small File," *Proceedings of the 1997 USENIX Technical Conference,* pp. 1–17. Anaheim, CA, Jan. 1997.

[Ganger00] Ganger, G., McKusick, M.K., Soules, C., Patt, Y., "Soft Updates: A Solution to the Metadata Update Problem in File Systems," to appear in *ACM Transactions on Computer Systems*.

[Gray93] Gray, J., Reuter, A. *Transaction Processing: Concepts and Techniques.* San Mateo, CA: Morgan Kaufmann, 1993.

[Hagmann87] Hagmann, R., "Reimplementing the Cedar File System Using Logging and Group Commit," *Proceedings of the 11th SOSP,* pp. 155–162. Austin, TX, Nov. 1987.

[Haskin88] Haskin, R., Malachi, Y., Sawdon, W., Chan, G. "Recovery Management in QuickSilver," *ACM Transactions on Computer Systems,* 6(1), pp. 82–108, Feb. 1988.

[Howard88] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., West, M. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, 6(1), pp. 51–81, Feb. 1988.

[Hu96] Hu, Y., Yang, Q. "DCD—disk caching disk: A new approach for boosting I/O performance," *Proceedings of the 23rd ISCA*, pp. 169–178. Philadelphia, PA, May 1996.

[Katcher97] Katcher, J., "PostMark: A New File System Benchmark," Technical Report TR3022, Network Appliance, October 1997.

[Matthews97] Matthews, J., Roselli, D., Costello, A., Wang, R., Anderson, T. "Improving the Performance of Log-Structured File Systems with Adaptive Methods," *Proceedings of the 16th SOSP*, pp. 238–251. Saint-Malo, France, Oct. 1997.

[McKusick84] McKusick, M.K., Joy, W., Leffler, S., Fabry, R. "A Fast File System for UNIX," *ACM Transactions on Computer Systems* 2(3), pp 181–197, Aug. 1984.

[McKusick99] McKusick, M.K., "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Proceedings of the 1999 Freenix track of the USENIX Technical Conference*, June 1999, 1–17.

[McVoy91] McVoy, L., Kleiman, S., "Extent-like Performance From a UNIX File System," *Proceedings of the 1991 Winter USENIX Technical Conference*, pp. 33–44. Dallas, TX, Jan. 1991.

[Peacock88] Peacock, J.K. "The Counterpoint fast file system," *Proceedings of the 1988 Winter USENIX Technical Conference*, pp. 243–249. Dallas, TX, Feb. 1988.

[Rosenblum92] Rosenblum, M., Ousterhout, J. "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, 10(1), pp. 26–52. Feb. 1992.

[Seltzer93] Seltzer, M., Bostic, K., McKusick, M.K., Staelin, C. "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the 1993 USENIX Winter Technical Conference*, pp. 307–326. San Diego, CA, Jan. 1993.

[Seltzer95] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V. "File System Logging versus Clustering: A Performance Comparison," *Proceedings of the 1995 USENIX Technical Conference*, pp. 249–264. New Orleans, LA, Jan. 1995.

[Swartz96] Swartz, K. "The Brave Little Toaster Meets Usenet," *LISA '96*, pp. 161–170. Chicago, IL, Oct. 1996.

[Vahalia95] Vahalia, U., Gray, C., Ting, D. "Metadata Logging in an NFS Server," *Proceedings of the 1995 USENIX Technical Conference*, pp. 265–276. New Orleans, LA, Jan. 1995.

[Wang99] Wang, R., Anderson, T., Patterson, D., "Virtual Log Based File Systems for a Programmable Disk," *Proceedings of the 3rd OSDI*, pp. 29–44. New Orleans, LA, Feb. 1999.

[Wilkes95] Wilkes, J., Golding, R., Staelin, C., Sullivan, T. "The HP AutoRAID hierarchical storage system," *15th SOSP*, pp. 96–108. Copper Mountain, CO, Dec. 1995.

[Ylonen96] Ylonen, T. "SSH—Secure Login Connections Over the Internet," *6th USENIX Security Symposium*, pp. 37–42. San Jose, CA, Jul. 1996.