Proceedings of 2000 USENIX Annual Technical Conference

San Diego, California, USA, June 18–23, 2000

JOURNALING 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, and Christopher A. Stein

Journaling 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[†]
[†]*Harvard University,*[*]*Carnegie Mellon University,* [‡]*Author and Consultant*

1 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: journaling and Soft Updates. Journaling systems use an auxiliary log to record meta-data operations and Soft Updates uses ordered writes to ensure meta-data consistency.

The commercial sector has moved en masse to journaling 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 journaling. In the meantime, Soft Updates holds the promise of providing stronger reliability guarantees than journaling, with faster recovery and superior performance in certain boundary cases.

In this paper, we explore the benefits of Soft Updates and journaling, comparing their behavior on both microbenchmarks and workload-based macrobenchmarks. We find that journaling 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 journaling systems cannot realize their full potential. Only when this synchronicity requirement is relaxed can journaling systems approach the performance of systems like Soft Updates (which also relaxes this requirement). Our asynchronous journaling and Soft Updates systems perform comparably in most cases. While Soft Updates excels in some meta-data intensive microbenchmarks, the macrobenchmark results are more ambiguous. In three cases Soft Updates and journaling are comparable. In a file intensive news workload, journaling prevails, and in a small ISP workload, Soft Updates prevails.

2 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. Many of the obstacles to high performance file service have been solved in the past decade. For example, clustering of sequential reads and writes removes the need for disk seeks between adjacent files [25][26][28]. The Co-locating FFS [13] 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 meta-data problem has been addressed most directly through journaling systems [5][16] and Soft Updates systems[11].

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 journaling 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 journaling file systems, both FFS-compatible (i.e., they have the same on-disk structure); a novel journaling 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 journaling.

The rest of this paper is organized as follows. In Section 3, we discuss a variety of techniques for maintaining meta-data integrity. In Section 4, we describe Soft Updates and in Section 5, we discuss our two journaling implementations. In Section 6, we highlight the semantic differences between journaling and Soft Updates. In Section 7, we describe our benchmarking methodology and framework, and in Section 8, we present our experimental results. In Section 9, we discuss related work, and we conclude in Section 10.

3 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, the data must be written to disk in a fixed order. (E.g., a 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 rest of this section, we discuss some of the most common approaches to solving the *meta-data update* problem, beginning with a brief introduction to Soft Updates [11] and journaling [16], the two techniques under analysis here. Then we discuss Soft Updates in detail in Section 4 and journaling in detail in Section 5, and highlight the differences between the two in Section 6.

3.1 Soft Updates

Soft Updates attacks the meta-data update problem by guaranteeing that blocks are written to disk in their required order without using synchronous disk I/Os. In general, a Soft Updates system must maintain *dependency information*, or detailed information about the relationship between cached pieces of data. For example, when a file is created, the system much ensure that the new inode reaches disk before the directory that references it does. In order to delay writes, Soft Updates must maintain information that indicates that the directory data block is dependent upon the new inode and therefore, the directory data block cannot be written to disk until after the inode has been written to disk. In practice, this dependency information is maintained on a per-pointer basis instead of a per-block basis in order to reduce the number of cyclic dependencies. This is explained more fully in Section 4.

3.2 Journaling Systems

Journaling (or logging) file systems attack the meta-data update problem by maintaining an auxiliary log that records all meta-data operations and ensuring that the log and data buffers are synchronized in such a way to guarantee recoverability. The system enforces *write-ahead logging* [15], which 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. Journaling 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. When the same piece of meta-data is updated frequently (e.g., the directory in which a large number of files are being created and that directory’s inode), journaling systems incur these log writes in exchange for avoiding multiple meta-data writes.

Journaling systems can provide a range of semantics with respect to atomicity and durability. If the log is maintained synchronously (that is, the log is forced to disk after each meta-data operation), then the journaling system provides guarantees identical to FFS. If the log is maintained asynchronously, buffering log writes until entire buffers are full, the semantics are comparable to Soft Updates.

A third configuration that warrants consideration, but is not currently supported by either of the systems described in this paper, is to support group commit. In a group commit system [16], log writes are synchronous with respect to the application issuing the meta-data operation, but such operations block to allow multiple requests to be batched together, providing the potential for improved log throughput. On a highly concurrent system with many simultaneous meta-data operations, group commit can provide a significant performance improvement, but it provides no assistance on single-threaded applications, such as most of the macrobenchmarks described in Section 7.3. The asynchronous implementations described here provide performance superior to a group commit system since they avoid any synchronous writes and never make applications wait.

In the context of building a journaling 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 5, we present two alternative designs for incorporating journaling in FFS, focusing on how each addresses these issues.

3.3 Other Approaches

Some vendors, such as Network Appliance [18], have addressed the meta-data update problem by hardware techniques, most notably non-volatile RAM (NVRAM). Systems equipped with NVRAM not only avoid synchronous meta-data writes, but can cache data indefinitely, safe in the knowledge that 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. Baker and her colleagues quantify the benefits of such systems[1]. Such systems provide performance superior to both Soft Updates and journaling, but with the additional expense of NVRAM.

The Rio system provides a similar solution [3]. 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 with NVRAM, storing meta-data in Rio memory eliminates the need for synchronous writes. The performance trade-off between Rio and Soft Updates or journaling is the cost of protecting and unprotecting the memory region versus maintenance of the dependency or log information.

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 not be possible to write all the related meta-data 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 [27] 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 [28] 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. LFS utilizes a combination of Soft Updates and journaling approaches. Like Soft Updates, it ensures that blocks are written to disk in a particular order, like journaling, it takes advantage of sequential log writes and log-based recovery.

4 Soft Updates

This section provides a brief description of Soft Updates; more detail can be found in other publications [12][14][24].

While conventional FFS uses synchronous writes to ensure proper ordering of meta-data writes, Soft Updates uses *delayed writes* (i.e., write-back caching) for meta-data and maintains dependency information that specifies the order in which data must be written to the disk. Because most meta-data blocks contain many pointers, *cyclic dependencies* occur frequently if dependencies are recorded only at the block level. For example, consider a file creation and file deletion where both file names are in the same directory block and both inodes are in the same inode block. Proper ordering dictates that the newly created inode must be written before the directory block, but the directory block must be written before the deleted inode, thus no single ordering of the two blocks is correct for both cases. In order to eliminate such cyclic dependencies, Soft Updates tracks dependencies on a per-pointer basis instead of a per-block basis. Each block in the system has a list of all the meta-data dependencies that are associated with that block. The system may 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 those parts of the meta-data in the current block are 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. It then restores any rolled back values to their current value so that subsequent accesses to the block will see the up-to-date value. These dependency-required rollbacks allow 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.

Not only does Soft Updates make meta-data write operations asynchronous, it is also able to defer work in some cases. In particular, 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.

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 [24].

As discussed in more detail in Section 6, while Soft Updates preserves the integrity of the file system, it does not guarantee (as FFS does) that all meta-data operations are durable upon completion of the system call.

5 Journaling

In this section, we describe two different implementations of journaling 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 journaling 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). This stand-alone logging service can be used by other clients, such as a database management system [30], as was done in the Quicksilver operating system [17].

5.1 LFFS-file

The LFFS-file architecture is most similar to the commercially available journaling systems. LFFS-file augments FFS with support for write-ahead logging by adding logging calls to meta-data operations. 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.

The fundamental 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 the blocks it is writing, LFFS-file can determine those portions of the log that must first be written. As the log is cached in memory, it must be flushed to disk up to and including the last log entry recorded for the block that is about to be written. In most cases, the relevant log entries will already be on disk, however if they are not, then a synchronous log write is initiated before the block is written. This synchronous flush requires an update of the file system's superblock.

Since the log is implemented as a circular buffer, log space must be reclaimed. LFFS-file uses standard database checkpointing techniques [15]. 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 to which a dirty buffer refers. 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.

LFFS-file maintains its log asynchronously, so like Soft Updates, it maintains file system integrity, but does not guarantee durability.

LFFS-file is a minor modification to FFS. It requires approximately 35 hooks to logging calls and adds a single new source file of approximately 1,700 lines of code to implement these logging calls.

5.2 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 (LSNs), 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 are 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. As is the case for LFFS-file, 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.

By default, LFFS-wafs mounts its log 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. This configuration provides semantics identical to those provided by FFS. 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; instead it provides semantics comparable to LFFS-file and Soft Updates.

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.

Although similar in design to LFFS-file, LFFS-wafs is somewhat more complex. Rather than simply logging to a file, LFFS-wafs implements the infrastructure necessary to support a file system. This results in about three times the number of lines of code (1,700 versus 5,300).

5.3 Recovery

Both journaling 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, thus any log writes that occurred after the last checkpoint are lost. 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 the log is reached.

Once the log has been recovered, recovery of the main file system begins. This process is identical to standard database recovery [15]. First, the log is read from its logical end back to the most recent checkpoint and any aborted operations are undone. LFFS-file uses multiple log records for a single meta-data operation, so it is possible that only a subset of those records have reached the persistent log. While database systems typically use a commit record to identify the end of a transaction, LFFS-file uses uniquely identified record types to indicate the end of a logical operation. When such records do not occur, the preceding operations are treated as aborted operations. Since LFFS-wafs logs at a somewhat higher logical level, 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. In both LFFS-file and LFFS-wafs, aborted operations must be undone 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.

	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 1. File System Configurations.

.

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
Meta-data updates are atomic.	LFFS-file LFFS-wafs-[12]*
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 2. Feature Comparison.

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.

6 System Comparison

When interpreting the performance results of Section 8, it is important to understand the different systems, the guarantees that they make, and how those guarantees affect their performance. Table 1 lists the different file systems that we will be examining and Table 2 summarizes the key differences between them

FFS-async is 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 journaling, 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.

Both journaling and Soft Updates systems ensure the integrity of meta-data operations, but they provide slightly different semantics. The four 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, the guarantees made about the data in files after recovery, and the ability to provide atomicity

The original FFS implemented meta-data operations such as create, delete, and rename synchronously, guaranteeing that when the system call returned, the meta-data changes were persistent. Some FFS variants (e.g., Solaris) made deletes asynchronous and other variants (e.g., SVR4) made create and rename asynchronous. 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 than FFS about when meta-data changes reach disk. Create, delete, and rename operations typically reach disk within 45 seconds of the corresponding system call, but can be delayed up to 90 seconds in certain boundary cases (a newly created file in a hierarchy of newly created directories). 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 report less than the actual 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 [24].

The journaling 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 journaling systems provide guarantees identical to FFS; when the log is written asynchronously, the journaling 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 are on the disk. For example, if block 100 were allocated to file 1, you would want block 100 to be on disk before file 1's inode was written, so that file 1 was 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 7.3.1) to approximately 8% (in the meta-data intensive Netnews benchmark, described in Section 7.3.2). Neither of the journaling file systems provides this stronger guarantee. These differences should be taken into account when comparing performance results.

The final difference between journaling systems and Soft Updates is the ability to provide atomicity of updates. Since a journaling system records a logical operation, such as rename, it will always recover to either the pre-operation or post-operation state. Soft Updates can recover to a state where both old and new names persist after a crash.

7 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.

7.1 The Systems Under Test

We compared the two LFFS implementations to FFS, FFS-async, and Soft Updates. Our test configuration is shown in Table 3.

7.2 The Microbenchmarks

Our microbenchmark suite is reminiscent of any number of the microbenchmark tests that appear in the file system literature [11][27][29]. 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.

	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: 9,088 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 (as of 1/26/00 10:30 PM) config: GENERIC + SOFTUPDATES – bpfilter – unnecessary devices

Table 3. System Configuration.

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 8.1.

7.3 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.

7.3.1 The SSH Benchmark

The most widely used benchmark in the file system literature is the Andrew File System Benchmark [19]. 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 [34]). 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.

7.3.2 Netnews

A second workload that we examine is that of a Netnews server. We use a simplified version of Karl Swartz's Netnews benchmark [31]. 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 journaling 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 [4][7]. Anecdotal evidence suggests that a full news feed today is 15–20 GB per day.

7.3.3 SDET

Our third workload is the deprecated SDET benchmark from SPEC. This benchmark was originally designed to emulate a typical timesharing workload, and was deprecated as the computing landscape shifted from being dominated by timesharing systems to being dominated by networked clients and servers [10]. Nonetheless, as SDET 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), it makes fairly extensive use of the file system. The scripts are generated from a predetermined mix of commands [8][9], and the reported metric is scripts/hour as a function of the script concurrency.

7.3.4 PostMark

The PostMark benchmark was designed by Jeffrey Katcher to model the workload seen by Internet Service Providers under heavy load [21]. 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 file creation or deletion with file read or append. Each pair of transactions is chosen randomly and can be biased via parameter settings. The file creation operation creates a new file. The sizes of thesefilesarechosenatrandomandareuniformlydistributed over the file size range. File deletion removes a file from the active set. File read selects a random file and reads it in its entirety. 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. We initially ran our experiments using the default PostMark configuration of 10,000 files with a size range of 512 bytes to 16 KB. One run of this default configuration performs 20,000 transactions with no bias toward any particular transaction type and with a transaction block size of 512 bytes. However, as this workload is far smaller than the workload observed at any ISP today, we ran a larger benchmark using 150,000 files with the default size range, for a total data size of approximately 1.1 GB. The results presented in Section 8.2.4 show both workloads, and it is important to note that the results change dramatically with the data set size. When we increase the data set by a factor of 15, performance (in transactions per second) dropped by nearly the same factor.

8 Results

8.1 Microbenchmark Results

Our collection of microbenchmarks separates meta-data operations from reading and writing. 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 journaling 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.

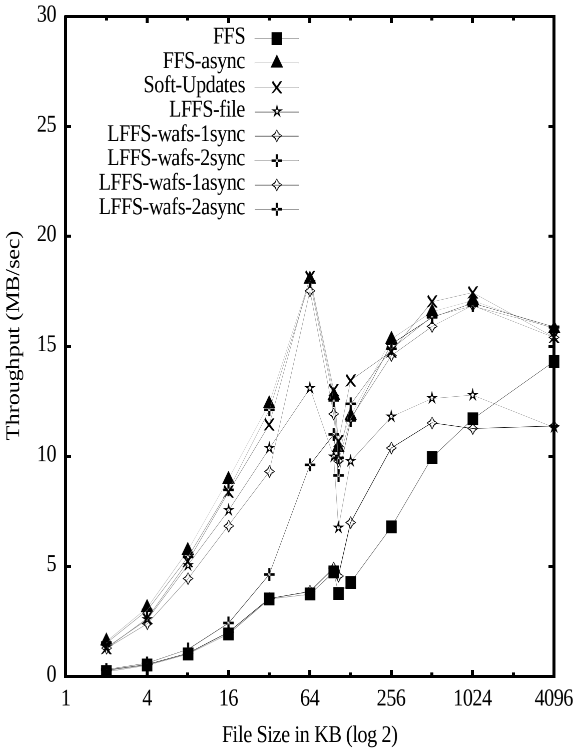

Figure 1. Create Performance as a Function of File Size.

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.

The read and write tests perform comparably, as expected (and are omitted for the sake of space).

Figure 1 shows the results of the create microbenchmark. At the high-performing end of the spectrum, Soft Updates, LFFS-wafs-2async, and FFS-async provide comparable performance. 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 at 104KB. 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 performing costly seeks between the log and data partitions. LFFS-file and LFFS-wafs-1async occupy the region in the middle, reaping the benefits of asynchronous meta-data operations, but paying the penalty of seeks between the data and the log.

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 physically contiguous writes and losing a disk 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 1 MB we see a steady increase back up to the maximum throughput. Between 1 and 4 MB there is a slight decline caused by longer seeks between the first 96 KB of a file and the remainder of the file as the larger files fill cylinder groups more quickly.

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

The delete microbenchmark performance is shown in Figure 2. Note that performance is expressed in files per second. This microbenchmark highlights a feature of Soft Updates that is frequently overlooked. As explained in Section 4, Soft Updates performs deletes in the background. As a result, the apparent time to remove a file is short, leading to the outstanding performance of Soft Updates on the delete microbenchmark. This backgrounding of deletes provides a very real advantage in

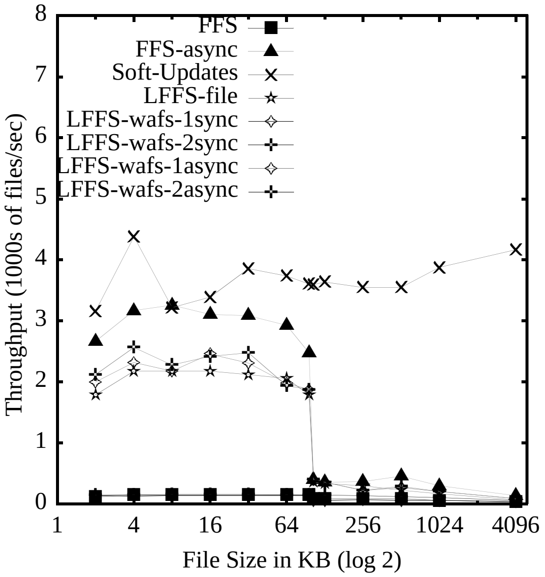

Figure 2. Delete Performance as a Function of File Size.

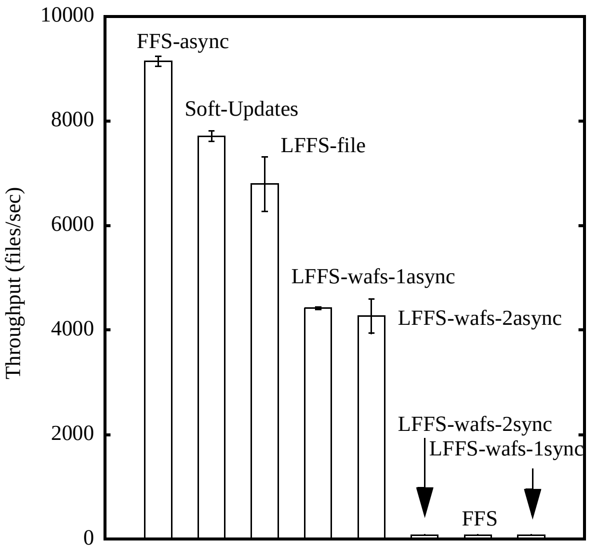

Figure 3. 0-length File Create/Delete Results in Files per Second.

certain workloads (e.g., removing an entire directory tree), while in other cases, it simply defers work (e.g., the Netnews benchmark discussed in Section 7.3).

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 the disk blocks it references. In contrast, by backgrounding the delete, Soft Updates removes this read from the measurement path.

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 journaling 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 benchmark emphasizes the benefits of asynchronous meta-data operations without the interference of data reads or writes. This benchmark also eliminates the overhead of compulsory read misses in the file system cache, as the test repeatedly accesses the same directory and inode data. Figure 3 shows the results for this benchmark. As this benchmark does nothing outside of meta-data operations, the synchronous journaling implementations behave identically to FFS. The one- and two-disk WAFS-based asynchronous journaling implementations perform comparably, achieving less than half the performance of FFS-async. The reason for FFS-async's superiority 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 journaling systems, however, still write log records. LFFS-file outperforms the WAFS-based journaling schemes because it writes log blocks in larger clusters. The WAFS-based systems write 4 KB log blocks while the LFFS-file system write in fully clustered I/Os, typically eight file system blocks, or 64 KB on our system. Soft Updates performs nearly as well as FFS-async since it too removes files from the buffer cache, causing no disk I/O. However, it is computationally more intensive, yielding somewhat poorer performance.

8.2 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. 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.

8.2.1 Ssh

As explained in Section 7.3.1, this benchmark simulates unpacking, configuring and building `ssh` [34].

	Unpack	Config	Build	Total
	Absolute Time (in seconds)			
FFS-async	1.02	10.38	42.60	53.99
	Performance Relative to FFS-async			
FFS	0.14	0.66	0.85	0.73
Soft-Updates	0.99	0.98	1.01	1.01
LFFS-file	0.72	1.08	0.95	0.96
LFFS-wafs-1sync	0.15	1.01	0.88	0.82
LFFS-wafs-1async	0.90	0.94	1.00	0.99
LFFS-wafs-2sync	0.20	0.85	0.93	0.86
LFFS-wafs-2async	0.90	1.05	0.98	0.99

Table 4. Ssh Results. Data gathered are the averages of 5 runs; the total column is the measured end-to-end running time of the benchmark. Since the test is not divided evenly into the three phases, the normalized results of the first three columns do not average to the normalized result of the total column. 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.

Table 4 reports the normalized performance of our systems. While many of the results are as expected, there are several important points to note. The config and build phases are CPU-intensive, while the unpack phase is dominated by disk-intensive meta-data operations. For the CPU-intensive phases, most of the journaling and Soft Updates systems perform almost as well as FFS-async, with the synchronous journaling 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 journaling systems demonstrate only 10 to 20% 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.

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.

8.2.2 Netnews

As described in Section 7.3.2, 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 5. On this benchmark, all of the file systems are completely disk bound.

	Unbatch	Expire	Total
	Absolute Time (in seconds)		
FFS-async	1282	640	1922
	Perf. Relative to FFS-async		
FFS	0.63	0.40	0.53
Soft-Updates	0.86	0.89	0.87
LFFS-file	0.95	0.95	0.95
LFFS-wafs-1sync	0.67	0.48	0.59
LFFS-wafs-1async	0.98	0.92	0.96
LFFS-wafs-2sync	0.91	0.67	0.81
LFFS-wafs-2async	0.97	0.95	0.96

Table 5. 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.

All of the asynchronous journaling systems and the two-disk synchronous system approach the performance of FFS-async (within 5%), but Soft Updates performs at only 87% of FFS-async and the one disk synchronous system performs at less than 60% of FFS-async. The Soft Updates performance is largely due to writes caused by dependency-required rollback. Soft Updates performed 13% more disk writes than FFS. The major cause for these rollbacks is that the data set exceeds the size of the buffer cache. Much of the performance benefits of Soft Updates come from being able to aggregate several meta-data writes into a single write. For example, updating several inodes in a single block at once rather than writing each one individually. To be most effective, it needs to be able to cache blocks for at least 15 and preferably 30 seconds. In the Netnews benchmark, the cache evictions occur much more rapidly, which decreases the aggregation and increases the likelihood of needing to do rollback operations. Recent tuning work (reflected in these results) defers the writing of blocks with rollback dependencies by having them travel around the LRU list twice before they are written. This change eliminated most of the rollbacks associated with directory dependencies. About 65% of the remaining extra I/O operations come from the rollbacks associated with ensuring that inodes not reference data blocks that have not been written (see Section 6 for a discussion of this feature). The other 35% comes from the reduced aggregation caused by the faster buffer flushing and rollbacks associated with directories.

The significant difference between the one and two disk synchronous systems indicates that it is the contention between log I/O and data I/O that hampers performance, not the synchronous writing of the log.

8.2.3 SDET

Figure 4 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 journaling 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. 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.

8.2.4 PostMark

This test, whose results are shown in Figure 5, demonstrates the impact that delayed deletes can have on subsequent file system performance. When we run the test with a small file set (right-hand bars), Soft Updates outperforms the LFFS-wafs systems significantly and outperforms LFFS-file by a small margin. However, with a larger data set (left-hand bars), which takes significantly longer to run (1572 seconds versus 21 seconds for the FFS-async case), the backgrounded deletes interfere with other file system operations and Soft Updates performance is comparable to all the asynchronous journaling systems. When the log writes are synchronous, seeks between the logging and data portions of the disk cause the difference between the 1-disk and 2-disk cases. In the asynchronous case, the ability to write log records lazily removes the disk seeks from the critical path.

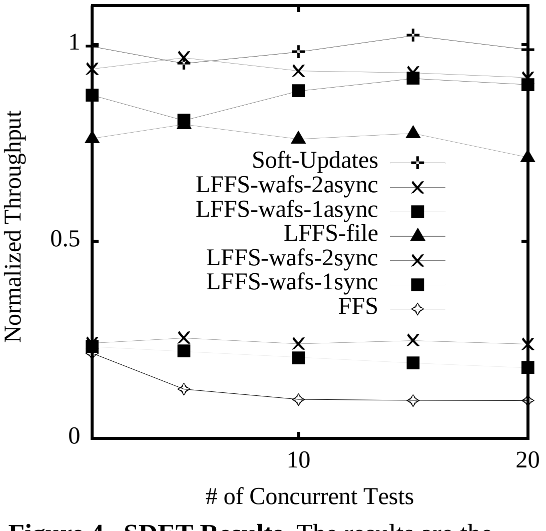

Figure 4. SDET Results. The results are the averages of five runs.

9 Related Work

In Section 3, 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 [27] are one approach to that problem. A second approach is the Virtual Log Disk [32].

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 the journaling 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 journaling, LFS requires a background garbage collector, whose performance has been the object of great speculation and debate [2][22][28][29].

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

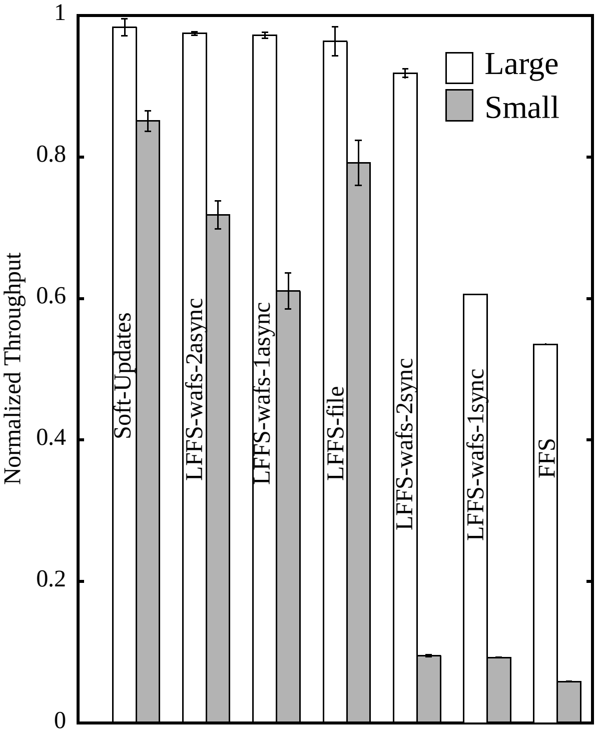

Figure 5. PostMark Results. The results are the averages of five runs; standard deviations are shown with error bars.

of the disk head [32]. The disk must then maintain a mapping of logical block numbers to physical locations. This mapping is maintained in a virtual log that 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 [33]. 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 journaling is the database cache technique [6] and the more recent Disk Caching Disk (DCD) [20]. 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 written lazily. This is essentially a two-disk journaling approach. The difference between the database cache techniques and the journaling 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.

10 Conclusions

We draw several conclusions from our comparisons. At a high level, we have shown that both journaling and Soft Updates succeed at dramatically improving the performance of meta-data operations. While there are minor differences between the two journaling architectures, to a first approximation, they behave comparably. Surprisingly, we see that journaling 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` 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. For the massive data set of the Netnews benchmark, we see that Soft Updates' ordering constraints prevent it from achieving performance comparable to the asynchronous journaling systems, while for the small Postmark dataset, Soft Updates backgrounding of deletes provides superior performance. 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` 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 Netnews and PostMark).

The implications of such results are important as the commercial sector contemplates technology transfer from the research arena. Journaling 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.

11 Acknowledgments

We would like to thank our paper shepherd, Aaron Brown, and the anonymous reviewers for their valuable comments and suggestions. We also thank Timothy Ganger and Teagan Seltzer for their critical commentaries on the final versions of the paper.

The CMU researchers thank the members and companies of the Parallel Data Consortium (including CLARiiON, EMC, HP, Hitachi, Infineon, Intel, LSI Logic, MTI, Novell, PANASAS, Procom, Quantum, Seagate, Sun, Veritas, and 3Com) for their interest, insights, and support. They also thank IBM Corporation and CMU's Data Storage Systems Center for supporting their research efforts. The Harvard researchers thank UUNET Technologies for their support of ongoing file system research at Harvard.

12 References

[1] Baker, M., Asami, S., Deprit, E., Ousterhout, J., Seltzer, M. "Non-Volatile Memory for Fast, Reliable File Systems," *Proceedings of the 5th ASPLOS*, pp. 10–22. Boston, MA, Oct. 1992.

[2] Blackwell, T., Harris, J., Seltzer., M. "Heuristic Cleaning Algorithms in Log-Structured File Systems," *Proceedings of the 1995 USENIX Technical Conference*, pp. 277–288, New Orleans, LA, Jan. 1995.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

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

[10] Gaede, S. "Perspectives on the SPEC SDET Benchmark," `http://www.spec.org/osg/sdm91/sdet/index.html`.

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

[12] 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, MI, Aug. 1995.

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

[14] 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*.

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

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

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

[18] Hitz, D., Lau, J., Malcolm, M., "File System Design for an NFS File Server Appliance," *Proceedings of the 1994 Winter USENIX Conference*, pp. 235–246. San Francisco, CA, Jan. 1994.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

[30] Stein, C. "The Write-Ahead File System: Integrating Kernel and Application Logging," Harvard University Technical Report, TR-02-00, Cambridge, MA, Apr. 2000.

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

[32] 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.

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

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