

Meta-data Snapshotting: A Simple Mechanism for File System Consistency

Livio B Soares[§], Orran Y Krieger[†], and Dilma Da Silva[†]

[§]Department of Computer Science, Universidade de São Paulo, Brazil

[†]IBM T.J. Watson Research Center, USA

Abstract—File system consistency frequently involves a choice between raw performance and integrity guarantees. A few software-based solutions for this problem have appeared and are currently being used on some commercial operating systems; these include log-structured file systems, journaling file systems, and soft updates. In this paper, we propose *meta-data snapshotting* as a low-cost, scalable, and simple mechanism that provides file system integrity. It allows the safe use of write-back caching by making successive snapshots of the meta-data using copy-on-write, and atomically committing the snapshot to stable storage without interrupting file system availability. In the presence of system failures, no file system checker or any other operation is necessary to mount the file system, therefore it greatly improves system availability. This paper describes meta-data snapshotting, and its incorporation into a file system available for the Linux and K42 operating systems. We show that meta-data snapshotting has low overhead: for a microbenchmark, and two macrobenchmarks, the measured overhead is of at most 4%, when compared to a completely asynchronous file system, with no consistency guarantees. Our experiments also show that it induces less overhead than a write-ahead journaling file system, and it scales much better when the number of clients and file system operations grows.

Furthermore, this new technique can be easily extended to provide file system snapshotting (versioning) and transaction support for a collection of selected files or directories.

Index Terms—Operating Systems, File Systems, Consistency, Availability

I. INTRODUCTION

FILE system consistency, in the presence of system crashes, has always been a strong concern in the operating system community. Availability, integrity, and performance are commonly the main requirements associated with file system consistency. Consistency in file systems is achieved by performing changes to the on-disk version of meta-data in a consistent manner. That is, when a system crash occurs, the on-disk version of the meta-data should contain enough information to permit the production of a coherent state of the file system.

In recent years, several novel software-based approaches for solving the meta-data update problem have been studied and implemented. One of these approaches in particular, journaling [1], has been implemented and is in use in a wide range of server platforms today. Other approaches, such as log-structured file systems [2] and soft updates [3] have had less

adoption by commercial operating systems. The soft updates technique has been incorporated into the 4.4BSD fast file system [4], and we have no knowledge of the incorporation of log-structured file systems on any commercial operating system. The implementation complexity imposed by these mechanisms is a disadvantage that may inhibit their adoption and inclusion in commercial file systems. Even though there is evidence [5] that the soft updates approach can achieve better performance and has stronger integrity guarantees than journaling file systems, the industry so far has adopted the latter.

In this paper, we present **meta-data snapshotting**, a new, efficient, scalable and simple solution to the meta-data consistency problem. This technique basically consists of maintaining a “snapshot” of a consistent state of the file system’s meta-data. The snapshot is kept intact during subsequent file system operations by writing meta-data to new locations on stable storage, using copy-on-write. The new data comprises a new generation of the file system. When all changed meta-data has been propagated to storage, the new generation becomes the current consistent snapshot of the system. This approach was conceived and prototyped in the Hurricane [6] File System [7], [8], [9] to provide recoverability in a new operating system environment where failures were very frequent. Our current research extends this initial work to take into consideration performance and it generalizes the basic approach to achieve file system versioning and transaction support (for example, by allowing several generations to be simultaneously active).

When the system needs or desires to make the current file system state persistent, it forces a consistent collection of dirty meta-data buffers out (to new locations), finally writing back a new file system superblock for this new snapshot. If the system crashes before the superblock is committed to stable storage, the current on-disk version of the meta-data is still consistent, as the “snapshotted” version has not been altered. In fact, meta-data snapshotting is similar to the *shadow-paging* [10], [11] technique designed for recovery in database systems.

Meta-data snapshotting allows the use of write-back caching, guaranteeing that there is always a consistent version on stable storage, so that in the presence of a system failure, the file system can become available instantaneously. There is no need for a file system checker, or any other pre-processing before recovery. Our experiments indicate that availability with meta-data snapshotting can be achieved at little cost, at most 4% in meta-data update intensive workloads, when

compared to a completely asynchronous file system, with no consistency guarantees. Additionally, in the same workloads, meta-data snapshotting can improve performance up to 109% when compared to a write-ahead journaling file system.

Meta-data snapshotting is being designed for use in the K42 File System (KFS) [12], which has been strongly influenced by the Hurricane File System [9]. KFS is available for the K42¹ [13] and Linux [14] operating systems. KFS, as other components of K42, uses a building-block approach [15], [9], which permits a high degree of customization for applications.

Several service implementations (alternative structuring and policies) are available in KFS. Each element (file, directory, or open file instance) in the system can be serviced by the implementation that best fits its requirements; if the requirements change, the component representing the element in KFS can be replaced accordingly. Parallel applications can achieve better performance by using the services that match their access patterns and synchronization requirements.

The remainder of this paper is organized as follows. Section II further describes the meta-data consistency problem and discusses previous solutions. Section III describes the meta-data snapshotting technique, and the implementation details of a prototype incorporated into KFS. An experimental evaluation is presented in Section IV. A couple of extensions to meta-data snapshotting are discussed in Section V. Finally, we conclude summarizing the article's contribution in Section VI.

II. THE META-DATA CONSISTENCY PROBLEM

File system operations, such as file creation, file truncation, file renaming, etc., frequently manipulate and alter different meta-data. These meta-data can be, and usually are, located on distinct blocks of the stable storage. For this reason, file system operations which alter more than one block of meta-data are non-atomic operations. Furthermore, for performance reasons, file systems commonly manipulate their meta-data on fast and volatile storage (memory) and the propagation of these changes to the stable storage is done asynchronously (i.e., write-through caching).

The lack of atomicity in updating a determined set of meta-data is what leads to file system corruption or inconsistency. For example, when renaming a file, the file system must perform two sub-operations: remove the directory entry pointing to the inode, and create a new directory entry that points to the same inode. If the system crashes while updating the blocks which have been altered in these two sub-operations, independently of the order in which they are performed, the file system will be inconsistent. If the directory entry removal is done first, then the file system will have lost the inode, because there are no entries pointing to that inode anymore. On the other hand, if the new directory entry creation is performed first, it will end up with two entries pointing to the same inode. While the latter case is certainly preferred against the first, as it does not incur in losing information, it is still not the result expected from the operation.

Software based solutions that have been previously proposed for achieving file system consistency in the presence of crashes fit into two strategies. The first strategy is to try to guarantee atomicity by performing write-ahead logging. If the system crashes, it has enough information on the stable storage to recover the meta-data into a consistent state. Journaling and log-structured file systems are based on this approach and they are described on sections II-A and II-B.

The second strategy is to determine a strict ordering of updates to the stable storage, so that, even though the image on the stable storage is not correct at all times, it is consistent and recoverable. Historically, systems such as FSF[4] have met this requirement by synchronously writing each block of meta-data, thereby hindering file system performance. The *soft updates* approach (described below in Section II-C) guarantees that blocks are written to disk in their required order without using synchronous I/O.

A. Journaling File Systems

Many modern file systems use the *journaling* (also called *logging*) approach to keep meta-data consistent (e.g., CedarFS[1], Episode[16], JFS[17], XFS[18], reiserFS[19], ext3[20], VERITAS[21], NTFS[22], BFS[23]). Some systems perform data and meta-data logging, but in general the idea is to keep an auxiliary log that records all meta-data operations. The log is written sequentially, in large chunks at a time, in such a way to guarantee that the log is written to disk before any page containing modified data, i.e., the file system must never perform an in-place update until the meta-data is written out to the log. If the system crashes, the file system recovers by replaying the log, using its information to update the disk.

The on-disk structure of the file system can be left undisturbed. The log is accessed read-only during crash recovery and sometimes for log space reclamation; operations that update the meta-data use the log in append-only mode. The meta-data log typically records changes to inodes, directory blocks, and indirect blocks. Information regarding the superblock and disk allocation maps can be either stored in the log or reconstructed during crash recovery. The log entries can either store both the old and the value of the meta-data (allowing redo-undo operations) or only the new value (redo-only). Changes made by operations that modify multiple meta-data objects (e.g., rename of files, creation or removal of directories) are collected in a single log entry.

The log may reside either inside the file system itself or externally as an independent object. Seltzer et al.[5] discuss implementation issues and performance tradeoffs between the two approaches.

Recovery activity requires identifying the beginning and end of the log, since it wraps around continuously. The recovery time is proportional to the active size of the log at the time of the crash.

Vahalia et al.[24] and Seltzer et al.[5] provide insights into the performance issues involved in journaling file systems. Journaling systems always perform additional I/O to write the log, but it is also meant to reduce the number of in-place meta-data writes by deferring them. In addition, the log itself may become a performance bottleneck.

¹K42 is a research operating system being developed for 64-bit cache-coherent multiprocessors, designed to scale to hundreds of processors.

B. Log-structured File Systems

Log-structured file systems (LFS) explore the journaling approach further by making the log itself the file system. It uses a sequential, append-only log as its only on-disk structure containing both data and meta-data. The fundamental idea is to improve write performance by buffering a sequence of file system changes in the file cache and writing all the changes to disk sequentially in a single large disk write operation (typically a full disk track), eliminating the need for rotational interleaving. When writing each segment of the log, blocks are carefully ordered. Since the log is append-only, each write operation flushes all the dirty data from the cache, which means the log contains all the information required for a complete recovery. Implementations of this approach are Sprite LFS [2] (the original implementation, developed for the Sprite network operating system) and BSD-LFS[25] .

To service file system read requirements, an efficient way of retrieving information from the log is necessary. Traditional file system implementations associate each inode with a fixed location on disk. In the LFS approach the inodes are written to disk as part of the log, therefore they do not reside in fixed positions. A data structure called *inode map* is used to keep track of the current location of each inode. This mapping is kept at memory and written to the log at periodic checkpoints.

A high degree of reliability is achieved with LFS because all components of an operation (data blocks, attributes, index blocks, directories) are propagated to stable storage through a single atomic write. If it is not possible to write all the related meta-data in a single disk transfer, it is necessary to LFS to maintain log segment usage information in such a manner to guarantee that it can recover the file system to a consistent state. Sprite LFS applied a logging approach to the problem by adding small log entries to the beginning of a log segment and BSD LFS used a transaction-like interface.

Crash recovery locates the latest checkpoint, reinitializes the in-memory inode map and segment usage table. Then it replays the part of the log following the checkpoint. The work is proportional to the amount of file system activity since the last checkpoint.

The most difficult design issue for log-structured file systems is the management of free space in the log. The goal is to maintain large free extents for writing the new data. This requires a garbage collection scheme to collect data from one segment and move it to a new location, making the original segment reusable.

Rosenblum et al reports in [2] that LFS can use 70% of the disk bandwidth for writing, whereas Unix file systems typically can use only 5-10%. Seltzer et al. report in [25], [26] that BSD-FFS is clearly superior than the traditional FFS implementation or Sun-FFS in meta-data-intensive tests, but Sun-FFS was faster in most I/O-intensive benchmarks.

C. Soft Updates

Soft Updates[27], [28] uses write-back caching for meta-data and maintains explicit dependency information that specifies the order in which data must be written to the disk. When the system selects a block B to be written, it allows the Soft

Updates code to review the list of dependencies. If there are any blocks that have to be written before B , then the parts of B relating to these blocks are replaced with an earlier version (where no dependency exists). B , which has been *rolled back* to a state that does not depend on any cached meta-data not propagated to storage, is then written to disk. After the write has completed, the system updates the dependency information, and it restores any rolled back values to their current value. Applications always see the most recent copies of the meta-data blocks and the disk always see copies that are consistent with its other contents.

The dependency information covers the main changes that require meta-data update ordering: block allocation and deallocation, link addition and removal. To maintain the dependency information, a natural solution would be to keep a dynamically managed graph of dependency at the block level. But meta-data blocks usually contain many pointers (e.g., block pointers and directory entries), leading to many cyclic dependencies between blocks. Also, blocks could consistently have dependencies and never be written to storage. Like false sharing in multiprocessor caches, the problem is related to the granularity of the dependency information. With Soft Updates, dependency information is maintained per field or pointer. “Before” and “after” versions are kept for each individual update together with a list of updates on which it depends.

Soft Updates rollback operations may cause more writes than would be minimally required if integrity were ignored, because blocks with dependencies become dirty again immediately after writing (when rollbacks are undone). If no other changes are made to the block before it is again written to the disk, then there is an extra write operation. To minimize the frequency of such extra writes, the cache reclamation algorithms and the syncer task attempt to write dirty blocks in an order that minimizes the number of rollbacks.

If a Soft Updates system crashes, the only inconsistencies that can appear on the disk are unused blocks/inodes that may not appear in the free space data structures and inode link counts that may exceed the actual number of associated directory entries. Both situations can be easily fixed by running a file system check utility on the background, so the file system can be immediately reusable after a crash.

Ganger et al. report in [27] encouraging performance numbers. For workloads that frequently perform updates on meta-data (such as creating and deleting files), Soft Updates improves performance by more than a factor of 2 and up to a factor of 20 when compared to the conventional synchronous write approach and by 4-19% when compared to an aggressive write-ahead logging approach. In addition, Soft Updates can improve recovery time in more than two orders of magnitude. Seltzer et al.[5] compares the behavior of Soft Updates to journaling. Their asynchronous journaling and Soft Updates systems perform comparably in most cases. While Soft Updates excels in some meta-data intensive microbenchmarks, for three macrobenchmarks Soft Updates and journaling are comparable. In a file intensive news workload, journaling prevails, and in a small ISP workload, Soft Updates prevails.

III. META-DATA SNAPSHOTTING

This section describes the meta-data snapshotting mechanism. It gives an overview description of the technique, while also describing relevant implementation issues of meta-data snapshotting in KFS. Then, it goes on to discuss some relevant design issues, and the solutions we have adopted in our implementation.

A. Mechanism Overview

Meta-data snapshotting is a mechanism that efficiently maintains the on-disk version of meta-data consistent **at all times**, while allowing the use of write-back caching. It does not impose any ordering on the propagation of changed metadata to the server or any restriction on cached meta-data availability to applications.

The main problem in maintaining consistent meta-data state is to commit completed operations to stable storage atomically. To achieve this, the snapshotting mechanism, instead of altering current meta-data on the stable storage, copies new and altered meta-data to new locations on the stable storage, thus preserving the old meta-data intact. This new version of the meta-data is called a *generation* of the meta-data. The new generation is committed to disk by updating the superblock at the correct moment. Section III-E further describes the process of committing a generation to stable storage and how it is done atomically.

In essence, a snapshot of the on-disk version of the file system meta-data is created whenever the user, operating system or file system wants or needs to commit the file system state. For subsequent file system operations which alter meta-data, a copy-on-write technique is used. The blocks pertaining to the saved snapshot will not be overwritten until the next snapshot is committed. An example of the execution of meta-data snapshotting when a file system operation alters an inode is illustrated in Figure 1.

Observe, however, that for this mechanism to work using copy-on-write, a level of indirection is required to access physical meta-data block numbers; the only blocks with fixed locations are the ones which belong to the superblock. While it is common in various file systems to contain that level of indirection inside inodes, for example, several file systems maintain fixed positions for inode and free-block bitmaps (such as the BSD Fast File System [4], and Linux Second Extended File System [29]). This level of indirection already exists in KFS to allow each file to be composed of different sets of building-blocks. Furthermore, changing the physical layout of a file system to remove static positions of physical structures to dynamic assignments is not a daunting task.

With meta-data snapshotting, to commit a set of dirty meta-data blocks, the file system must perform two steps:

- 1) To prevent the file system from being unavailable while a snapshot is being flushed to stable storage, a new *generation* of in-memory meta-data is created (i.e., prohibiting changes to both the currently committed version of on-disk meta-data and blocks from current generations being flushed). Changes to meta-data blocks belonging to an older generation are done by creating

a copy of that block, and working with the new copy. If some meta-data point to the altered block, they also need to be altered to reflect that the block has now a new location, so a new copy of that meta-data is also made. This should happen iteratively until the superblock itself is updated (if the newest version of the superblock does not belong to the new generation, a new copy of the superblock is also created and its generation version updated).

- 2) When all meta-data blocks in the old generation have been written out to disk, the generation's superblock is finally written out. Writing out the superblock is what actually commits (persists) the generation; overwriting the previous superblock makes previous generations inaccessible.

When performing copy-on-write of meta-data blocks, it is important to free previous resources from the inode maps and free/allocated block bitmaps. For example, if a meta-data block located at a certain position p needs to be copied for alteration, a new position q is allocated, and the previous position p is freed. This prevents the leakage of resources which are not in use in the current in-core generation. However, this introduces a special case in which a generation may interfere with the data of another generation. It occurs when reutilizing meta-data blocks which were freed in an in-memory generation. Meta-data blocks freed in a previous generation can not be reused until that generation's superblock is written out (i.e., all the files that referred to those blocks are now gone from the disk image). If freed meta-data blocks are used before the generation that freed them has been committed (and therefore written out to stable storage), meta-data which is still valid on the committed version of the stable storage may be overwritten.

A solution to this problem is to have the superblock only make freed meta-data locations available for allocation, after the generation in which the freed occurred is committed to disk. This guarantees that future generations will not try to overwrite previous generation's valuable meta-data.

In KFS, we currently adopt another solution to this problem. We refrain the file system from submitting dirty buffers to I/O from newer generations until the previous generation is completely committed. With this policy, no valuable meta-data from an older generation will be overwritten, because writes from newer, in-memory generations will wait until the older generation is completely committed to stable storage. Furthermore, not submitting dirty buffers to I/O from newer generations ensures that the operating system, when reclaiming memory, will choose to write out buffers from the oldest generation, speeding up that generation's commit process, and shortening the overall meta-data vulnerability window.

In a file system with meta-data snapshotting, if a crash occurs at any time, the file system will have a consistent image:

- If the up-to-date version of the superblock has not been written to disk, we lose the very recent modifications to the file system, **but** the file system is consistent. This is because all of the "new" generations (which were on volatile storage, and therefore lost in the crash), had written altered meta-data blocks on other locations,

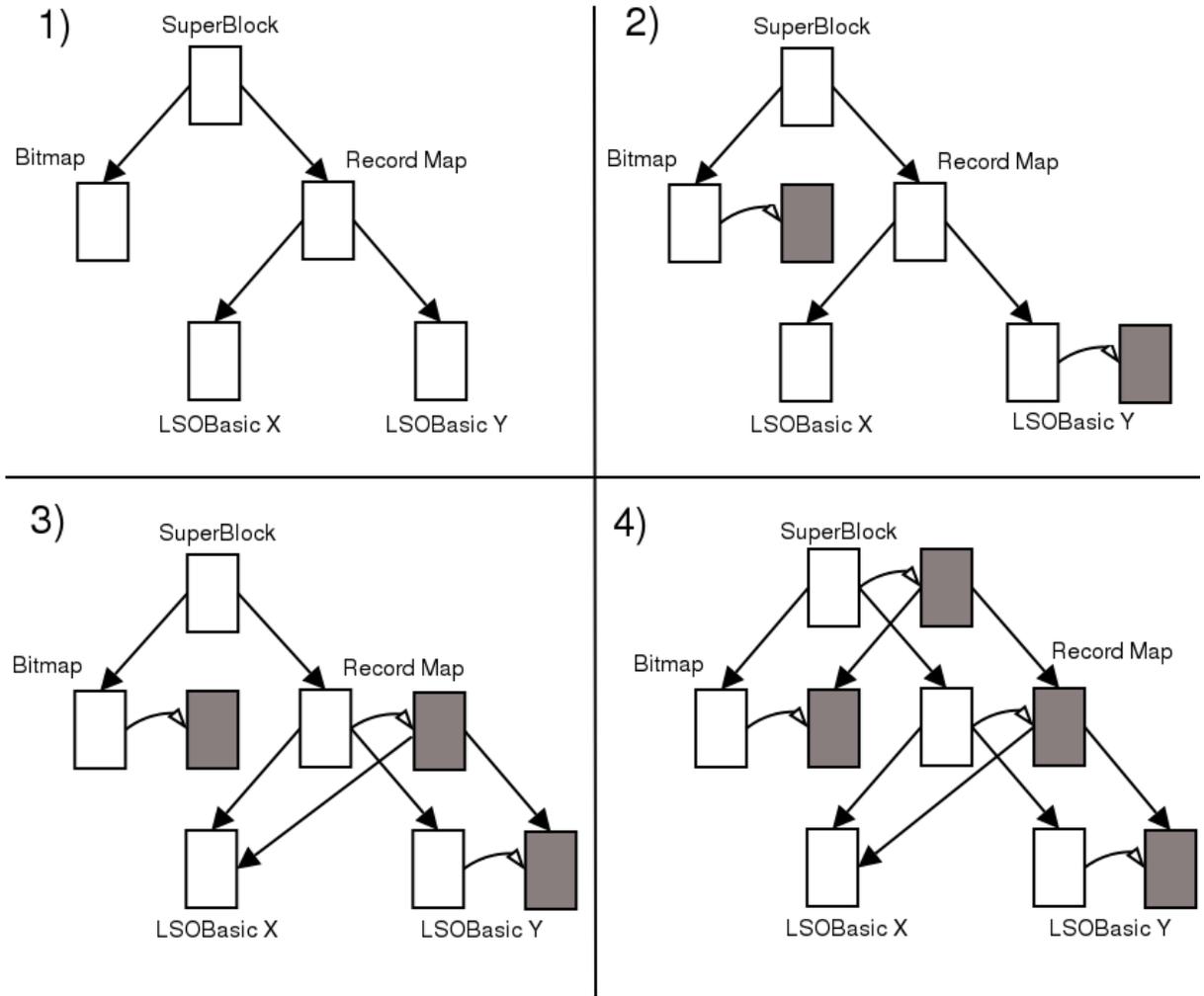


Fig. 1. Basic snapshotting example: white rectangles represent a clean meta-data object (i.e., residing on a clean buffer page), while gray rectangles represent dirty objects, which need to be committed to a *new location* on stable storage when propagated to storage. The *RecordMap* object is responsible for inode mapping. LSOBasic X and LSOBasic Y are objects representing meta-data for inodes X and Y, respectively. Diagram (1) illustrates an initial setup for the file system. (2) shows the result of a file system operation on an inode Y that requires allocation or release of blocks, such as *append* or *truncate*. Note that besides the change to LSOBasic Y, the global BitMap is also altered. (3) shows the propagation of meta-data snapshotting: a new location is allocated for the LSOBasic Y, so the Record Map has to change its internal data to reflect this change. Finally, (4) shows the end result of the changes, when the SuperBlock *forks* to point to the altered versions of the BitMap and the Record Map. Note that the original file system tree (in white) is **unchanged** at all times, as the gray tree (comprising only modified meta-data) is being formed to represent a new generation. Also, it is possible to have several generations active at the same time, each with its partial set of copy-on-write blocks being propagated to the disk.

not on the current ones. Since the old version of the superblock is pointing to the old locations, nothing got corrupted/inconsistent.

- If the up-to-date version of the superblock did get out to disk, then all other blocks pertaining to that generation had already gone to disk, and the file system is certainly consistent, and contains all recent modifications.

B. Spawning generations

The meta-data snapshotting mechanism revolves around the concept of generations. A generation is, in fact, a collection of file system operations (or transactions), each of which needs to be committed atomically. In theory, at any given moment in time, the file system could have any number of in-memory generations; it would be possible to adjust the file system with meta-data snapshotting to an extreme scenario where

each generation is representing only an individual file system operation.

A central point of meta-data snapshotting is deciding when to spawn a new generation. On the one hand, delaying the creation of a new generation is good for performance; there is no need to create new copies of altered meta-data blocks, therefore no extra memory pressure is made on the system. But on the other hand, delaying the creation of new generations has a bad side-effect, namely, the window of vulnerability for new operations increases.

Given this dichotomy, the following events should trigger the spawning of a new generation:

- The number of dirty meta-data buffers hits a certain limit. Fixing a maximum number of dirty meta-data buffers prevents starvation for the superblock, and consequently, for the generation per se, when a large number of

operations are issued in a short period of time. If the file system is receiving a higher rate of operations which affect meta-data than the stable storage can write out, the number of dirty meta-data buffers will only increase. In effect, the superblock would have to wait indefinitely, and would not be able to commit itself to the stable storage.

- The superblock has not been written out for some time (a few minutes, for example). Creating a new generation after a fixed amount of time, in a scenario where a small amount of operations is being generated, prevents the window of vulnerability for new data from becoming greater than desired. Additionally, it provides an *upper-bound* on the amount of time a certain generation can live without being committed to stable storage. Depending on the consistency guarantees that are expected or demanded from the file systems, the upper-bound can be adjusted accordingly.

Note, however, that since meta-data snapshotting does not issue dirty meta-data buffers to stable storage synchronously, the upper-bound is not only this predefined fixed amount of time; it is augmented by the time it takes for the operating system to actually write out old dirty buffers. In the Linux kernel, for example, a kernel thread (*kupdate* in the 2.4 version) wakes up every 30 seconds, and starts sweeping the queue for dirty buffers, sending the buffers to I/O. The time actually necessary for a particular buffer to go to disk, depends on the length of the queue, and the disk speed.

- A `sync()` system call is issued to the file system. The reason for creating a new generation for the `sync` system call is straightforward; the system must commit its current in-memory version to disk. Therefore, a new generation is created, and the previous is synchronously committed to disk. The call can only return to the user after the snapshot is completely on the stable storage.
- A `fsync()` system call is issued. The `fsync()` system call should return to the user only after file data and meta-data have been propagated to storage. A new generation is necessary for `fsync()` because there is no way of committing only a particular file, without having to commit all of the transactions of the same generation. This is true due to the fact that specific data about the transactions (which files, directories, and blocks participated in each transaction) are not kept. A problem with this approach is that it might be unreasonable to make an application wait for the whole generation to be committed. An alternative solution is to extend the snapshot mechanism to use *logging* specifically for this case.

One important issue with spawning generations is that of guaranteeing that operations begin and finish in the same generation. Suppose, for instance, that a file creation starts in a generation and ends in the next generation, due to the spawning of a new generation while the operation was proceeding. The file creation operation typically involves a few other sub-operations such as: inode allocation in the inode map and its initialization, and directory entry creation in the directory. If the inode allocation is performed in one generation

and the directory entry creation in the next, the file system will possibly end up with an inaccessible inode (lost resource), if the next generation does not have a chance to commit to stable storage. On the other hand, if the directory entry is created first, the file system might contain a directory entry pointing to an invalid inode.

One way of guaranteeing that operations are completed inside a single operation is to spawn new generations when the file systems reaches a quiescent state. But, as previously discussed, this could cause the generation to starve. A synchronization barrier issued when spawning a new generation could enforce the quiescing of the file system. This barrier would wait for all ongoing operations to finish, and block new operations until the superblock *fork* is completed. This solution, however, would render meta-data snapshotting impractical. Its performance would be closely based on the speed of the stable storage, and would, in practice, perform as slow as synchronous, write-through file systems.

The solution adopted in KFS is to make every file system operation receive a reference to the current context, which, among other things, includes a reference to the superblock. The context is then passed on to every sub-operation so that the entire operation belongs to the same generation and operates using the correct version of the superblock, inode maps, free block bitmaps, etc.

In summary, the following steps are executed, in this order, when spawning a generation:

- 1) The current superblock is *forked*, and a new version is created. The new *generation version* of the superblock is incremented, and the global context is updated to point to this new version. This is done at the very beginning of the spawning process, so that subsequent file system operations are serviced by the new superblock, and the old generation has a chance to be committed.
- 2) Transfer any in-memory meta-data to their respective I/O buffers, and dirty those buffers, so that the operating system starts writing them out to stable storage. Operating systems usually write out dirty buffers for two reasons: a) in order to reclaim system memory: when it is low, buffers are sent to I/O so that their pages can be reused; and b) if a predefined amount of time period has passed, a thread sends aged dirty buffers to be written out to stable storage.
- 3) Due to the possible latency caused by step (2), the superblock must wait until every dirty buffer from this generation has been written to stable storage. This is done by placing a reference to the dirty buffer on a hash on the superblock. A callback is registered in the I/O layer of the operating system so that whenever the write for a particular buffer is finished, the superblock removes the reference from its hash table.
- 4) When the last buffer is removed, the entire generation has been written to stable storage. The superblock per se is now committed to stable storage too, and its memory representation can be safely deleted.
- 5) The new superblock can now safely make the freed meta-data blocks (which were freed due to copy-on-write during the previous generation) available for al-

location again. The previous generation has been successfully committed to stable storage, so it is now safe to reuse the meta-data blocks belonging to the previous snapshot and modified during the generation that has been committed.

In KFS, the function of determining when to generate a new snapshot, and effectively issuing its creation, is delegated to a single kernel thread. The great advantage for using an external thread is that no user file system operation will have to be delayed, or need to wait for a snapshot to be committed; the thread alone sleeps and waits for the superblock to hit stable storage.

C. Short life span for files

According to [30], around 80% of all new files are deleted or overwritten within about 3 minutes of creation. In asynchronous file systems, with no consistency guarantees, depending on the memory size and usage, these files with very small life span are never written to disk at all.

Meta-data snapshotting, or any write-back consistency mechanism for that matter, has to efficiently deal with this class of files in order to keep its overhead low. With meta-data snapshotting, some files are naturally aged in the interim of the generation in which they are created. Nevertheless it is also true that file creation can commence at any given time, therefore, a file can be created a very short time before the spawning of a generation. This will lead to the inclusion of altered meta-data blocks, representing such files, in a snapshot. The snapshot will be scheduled to be written to stable storage while containing blocks of meta-data of newly created files. Given that the great majority of newly created files are deleted or overwritten, the snapshot will induce overhead by issuing writes on meta-data that could have been saved if the system permitted the file to “age”.

To overcome this possible source of overhead, meta-data snapshotting should be implemented so that when a file is deleted, the file system also deletes that file from previous, uncommitted generations. File deletion is propagated to older superblocks, and the altered meta-data buffers are *cleaned*. If the file was created only a short time previous to the deletion, the probability that its creation has not been committed to disk, and is still in one of the in-memory generations, is high. If that happens, the file, and its meta-data, will not be written to disk, saving valuable disk I/O.

D. Directory Data

It is common in various file systems, to represent directory entries as the directory’s data, in a similar way to file data blocks. For file system consistency, it is necessary to also ensure that directories entries are consistent with file system meta-data; they organize the file system namespace, and grant accessibility to underlying inodes. It would be inconsistent, for example, for a directory entry to point to an uninitialized inode or one that has not been allocated in the inode map yet.

For this reason, directory data is also subject to meta-data snapshotting. In KFS, it is particularly easy to implement this. The same object responsible for performing copy-on-write on

meta-data is used for directory data. When a change occurs on the directory data (due to the addition or removal of an entry), this object prevents the data from overwriting the current block on stable storage: it allocates a new location, and writes the data block to this new location.

E. Superblock atomicity

A minor implementation issue we must also raise is that, unlike previously suggested, writing out the superblock is *not* an atomic operation. Usually superblocks can have different sizes, depending on the file system, typically greater than or equal to 4KiB. It is also true that sector sizes for hard-disk devices can be smaller than the superblock size, typically of only 512 bytes.

Committing the superblock atomically, however, is *essential* for meta-data snapshotting to work. If only some parts of the superblock are updated, it will contain information from both the current and previous generations. In this scenario, meta-data snapshotting would not guarantee consistency in the presence of system failures.

The solution we have adopted for this problem is to have two superblocks, instead of only one. Adding the generation version number to a field on the last word on *each* sector of the superblock, and alternating between the two locations when writing out the superblock, will guarantee that on any given moment, the on-disk superblock with the greatest generation version is the one consistent with the on-disk image of the file system. When the system is mounted, the two superblocks are read and the version number of all sectors in a superblock are verified to determine that they are all the same, (i.e., the superblock was completely written to stable storage), and finally the two superblocks are compared to determine the correct (most up-to-date) superblock.

IV. PERFORMANCE EVALUATION

In this section, we analyze the performance behavior of our implementation of meta-data snapshotting. We show that a file system with meta-data snapshotting can achieve meta-data integrity and consistency at a small cost in performance. Additionally, we also compare the performance of meta-data snapshotting with a write-ahead journaling file system, and show that meta-data snapshotting induces a smaller overhead, and scales better with increased I/O workloads.

The results of one microbenchmark and two macrobenchmarks are presented. The microbenchmark shows the performance impact on a simple workload focusing on file system operations that involve only meta-data. The macrobenchmarks investigate a real use scenario for file systems and how concurrency of requests affects the performance.

A. Experimental Setup

As previously stated, KFS is being implemented on the Linux and K42 operating systems. In order to produce comparable results, we have chosen to perform all experiments on the Linux operating system. The kernel version in use was version 2.4.22. All of the experiments performed compare the

performance of four different file systems: **ext2fs**, **ext3fs**, **KFS** and **KFS+snap**.

The *ext2fs* [29] was chosen because, besides being the default file system in many Linux distributions, it is the only available file system which has a journalled version, namely, *ext3fs* [20]. The ext2 and ext3 file systems are compatible file systems, and share the same on-disk structures. This compatibility suggests that differences in performance are likely to be related to the journaling capability.

We refer to *KFS* in this article as being the K42 File System without any consistency guarantees, or data ordering, and *KFS+snap* as the version of *KFS* with meta-data snapshotting capability. In *KFS*, meta-data buffers are written out to stable storage in a completely asynchronous manner, and therefore its performance can be considered an upper-bound for *KFS+snap*.

In order to better compare *KFS*-based file systems with ext2-based ones, we have taken extra care with *KFS*'s on-disk file system structure. While it is true that *KFS* is not compatible with the ext2 on-disk structure, we have made available in *KFS* specific inode and directory structure implementations that match ext2's. Implementing these specific structures are specially easy to do in *KFS*, due to its component-based architecture. While some structures are inherently different between these two file systems, such as block bitmaps and the superblock, we believe that having the same on-disk structure for inodes and directories allows for the performance comparisons we make in this article.

The ext3 file system has three journaling modes: *journal*, *ordered* and *writeback*. In the *journal* mode, full data and meta-data journaling is provided. Both *ordered* and *writeback* modes do not perform any data journaling, but only meta-data journaling. The difference between them is that the *ordered* mode guarantees that file system data is written to stable storage before their respective meta-data, so that all on-disk meta-data point to valid data. This guarantee does not exist in the *writeback* mode, where meta-data can be written out before that actual data. The *ordered* mode is the default mode used by the ext3 formatting tools, and is the mode used on all experiments in this article.

All experiments were performed on a commodity *PC* system containing a 2.00 GHz Pentium IV processor, 628 MiB of main memory, and two disk drives. One of them contained the Linux installation and other system files, while the other was used to run the experiments. The disk drive used in the experiments is a 60GiB IBM drive (ATA/100 EIDE, 7200RPM, with 9ms of average read seek time). While running the experiments, the system was not running any other activity, besides the essential system daemons.

Recall from Section III-A that a separate thread issues the creation of new snapshots if one has not been created in a configurable amount of time after the previous generation has been committed. The *time-out* used for all experiments in the *KFS* thread for generating new snapshots is of 10 seconds.

B. Microbenchmark Performance

In this section we present the results of a microbenchmark for the four file system implementations. This microbenchmark

Operation	ext2	ext3	KFS	KFS+snap
<i>create</i>	369	192	626	629
<i>delete</i>	366	190	621	623
<i>read</i>	368	191	623	626
<i>append</i>	367	190	622	624

TABLE I

POSTMARK RESULTS, IN TRANSACTIONS PER SECOND, FOR THE FOUR FILE SYSTEM IMPLEMENTATIONS. EACH LINE COMPARES A DIFFERENT, SPECIFIC FILE SYSTEM OPERATION. THE VALUES REPRESENT THE AVERAGE OF 5 CONSECUTIVE RUNS.

focuses on operations which *only* manipulate meta-data (and no file data): creation of empty files and their deletion. The experiment consists in repeating the same workload 3 times in a row: the creation of 100,000 *empty* files, distributed amongst 1,000 directories, and their deletion.

Since the files are empty, and no data is being transferred, with the total amount of memory available on our test machine, all the buffers and other pages this experiment needs to run fit entirely in memory. The result is that this benchmark was completely CPU-bound, and all four file system implementations obtained very similar results. Therefore we have limited the operating system memory to only 64MiB, in order to make this benchmark more I/O intensive.

Figure 2 presents the results for this benchmark, comparing the total execution time of the four file system implementations. The results are very encouraging, showing that meta-data snapshotting adds only 3.6% in the execution time when compared to its upper-bound. Journaling, on the other hand, provokes a bit more degradation in performance: ext3 takes 9% more time to finish this benchmark than ext2.

C. Macrobenchmark Performance

To demonstrate the performance of meta-data snapshotting under “real world” scenarios, we have benchmarked the four mentioned file systems against two benchmarks, and present the results here. The benchmarks used are the *PostMark* and *Dbench* benchmarks. They are both synthetic benchmarks that try to reproduce the load observed in Internet servers (e.g., file, mail, newsgroup, etc.).

D. Postmark

Figure 3 shows measurements (overall transactions per second) of the four file systems running *PostMark* [31]. The *PostMark* benchmark was designed to simulate the workload seen by Internet Service Providers under heavy load. The load comprises the manipulation of a large amount of very small files, with short lifetimes. It first creates a pool of randomly sized files, and then measures the time required to perform a series of transactions. Each transaction is chosen at random, and consists of a pair of file system operations:

- Create a randomly lengthened file, or delete a file.
- Completely read a file, or append data to a file.

We have used the following settings in this experiment: 100,000 transactions with equal bias for read/append and

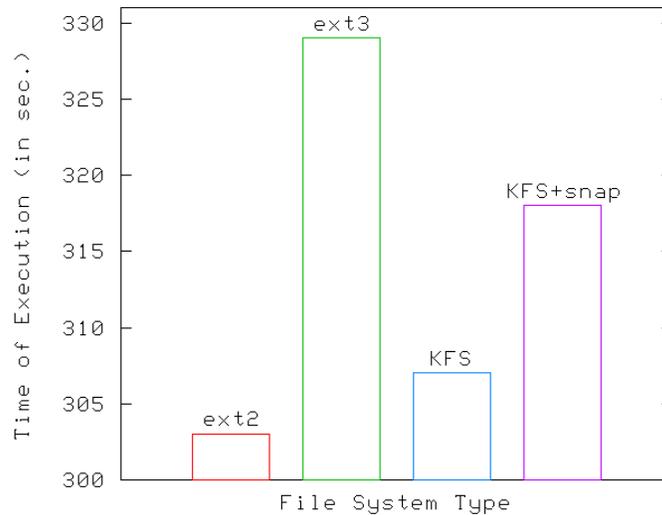


Fig. 2. Microbenchmark results, representing the total time of execution, in seconds, for each of the four file system implementations. The values represent the average of 3 consecutive runs.

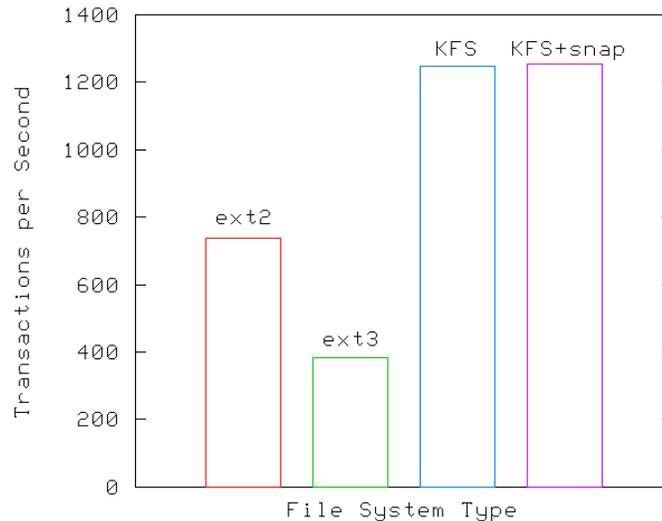


Fig. 3. PostMark results, in total transactions per second, for each of the four file system implementations. The values represent the average of 5 consecutive runs.

create/delete, 20,000 initial files, with file sizes ranging from 500 bytes to 16KiB, distributed in 400 directories, so that directory lookup times are not a source of major overhead.

At the end of execution, PostMark reports the overall transactions per second measure, and in addition, transactions per second for each type of file system operation performed. Each test was executed 5 consecutive times, and the average of the numbers is reported. The same file system partition was used in all runs to avoid seek time discrepancies. Figure 3 shows the results for overall transaction measurements, and the results per specific operation is listed on Table I.

The overall results indicate that, for this workload, journaling in ext2 causes a big degradation in performance: ext3 takes 90% more time to finish the benchmark than ext2 does. With meta-data snapshotting no degradation is observed, and

in fact, a very small improvement is noticed (less than 0.5%).

E. Dbench

Figure 4 compares the four file system implementations under the Dbench [32] benchmark. The Dbench benchmark is an open-source benchmark, very much used in the Linux development community. Its creation was inspired by the Netbench [33] benchmark. Netbench is a bench created to measure the performance of Windows file-servers implementations, such as WindowsNT [22] and Samba [34]. The problem with this benchmark is that it requires a large number of clients to produce a significant load on the tested server. Dbench, therefore, is a synthetic benchmark which tries to emulate NetBench, and the load imposed of such a file-server. Specifically, dbench tries to produce only the *file system* load,

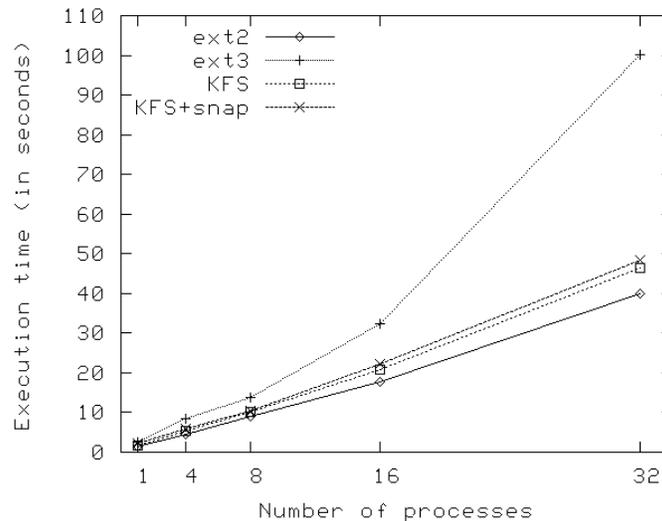


Fig. 4. Dbench results. The plotted values represent the execution time necessary to complete a dbench run with the indicated number of clients, for each of the four file system implementations. The values represent the average of 5 consecutive runs.

Number of clients	ext2	ext3	KFS	KFS+snap
1	323.362	251.179	291.108	279.665
4	328.636	186.646	288.199	277.702
8	330.204	204.118	290.809	278.369
16	323.293	178.313	271.658	261.600
32	282.465	111.326	240.511	232.517

TABLE II

DBENCH RESULTS, IN MEGABYTES PER SECOND, FOR EACH OF THE FILE SYSTEM IMPLEMENTATIONS. THE VALUES REPRESENT THE AVERAGE OF 5 CONSECUTIVE RUNS.

and another benchmark, the tbench, is used to produce the TCP and process load.

Dbench loads a file describing the operations it must perform, parses the file, and executes the operations. Among the operations performed are: file creation, deletion, and renaming; reading, writing and flushing out data; directory creation and removal; and retrieval of information about files and directories (through the `stat` system call). The file used in our experiments is the default `client_plain.txt`, which was obtained by a network sniffer dump of a real Netbench run. It consists of more than 120 thousand operations, and manipulates files with size ranging from a few kilobytes up to more than 20 megabytes.

Additionally, the dbench benchmark allows one to specify the number of concurrent clients to run. A dbench process creates all the client processes, and through a simple barrier, waits for the creation of all child processes to finish. After that, the barrier is released, and each client creates a private directory, and executes all operations. In the end, a single throughput value (MiB/second) is reported.

On these tests, we have also performed 5 consecutive runs,

on the same partition (reformatting in between tests), and the average was taken. To better illustrate the differences between the tested file system implementations, we have plotted the **time** of execution of each run in Figure 4, and the throughput results, as reported by dbench, are listed on Table II.

The results of this test indicate the performance impact of meta-data snapshotting on a file-server workload. It also illustrates how the overhead behaves in a multiprogramming environment, as the number of clients and requisitions grows. As expected, the overhead caused by meta-data snapshotting is small; the highest overhead was measured with only 1 process, and was of about 4%. Interestingly, as the number of client processes grows, and the benchmark becomes more I/O intensive, and less CPU bound, due to shortage of memory, meta-data snapshotting overhead slightly decreases to almost 3%.

In addition, also for this workload, journaling in ext2 causes a big degradation in performance. But besides the degradation, it is interesting to notice how ext3 scales as the number of clients grows. With only one process, ext3 takes 60% more time to finish than ext2; and as Figure 4 suggests this overhead grows exponentially. With 32 concurrent client processes, ext3 takes 152% more time to finish than ext2.

V. EXTENSIONS

Meta-data snapshotting, unlike soft updates, journaling and log-structured mechanisms, does not require specific actions for each type of file system operation. The only guarantee needed for meta-data snapshotting to work is that all blocks pertaining to the same snapshot be coherent with respect to each other. The use of copy-on-write to guarantee this coherency, and the lack of specific actions for different operations is what makes meta-data snapshotting so simple.

In addition, due to its characteristics, meta-data snapshotting can be easily extended for regular file data. In fact, snapshotting of data is already done in regular meta-data snapshotting

of directories, as described in Section III-D. In this section we discuss two extensions for meta-data snapshotting: file system transaction support; and versioning of files and directories.

A. Transactions

In KFS's snapshot mechanism, it is possible, for example, to support transaction-like capabilities (with all-or-nothing semantics) for a series of `write()` operations on any specific file data, or even a series of operations on directories (creation, deletion, rename, etc.). To do this, it is necessary to include all dirty file and meta-data pages of a transaction in the same generation. If the generation containing the transaction's dirty buffers is committed to stable storage, then the transaction is completely committed. However, if there is a system failure which prevents the generation from being committed, the **entire** transaction is prevented from altering the file system. With this mechanism, you have a guarantee that not only *part* of the modifications of the transaction are made: either all of the modifications are made (generation is committed), or none of them (generation is not committed).

B. Versioning

If the meta-data technique is changed so that blocks for older generations are not freed, it is possible to have various snapshots (or versions) of a selected set of files or directories, or even the entire file system, similar to snapshot capability available in logical volume managers (LVMs).

This kind of versioning snapshot is also expected to be created efficiently. All that is needed is to mark the selected blocks belonging to files and directories, and their respective meta-data, as immutable; no data or meta-data has to be copied or duplicated for copy-on-write versioning. Further modifications will be made to new locations of the stable storage, similar to the description of snapshotting directory data in Section III-D.

A key difference between meta-data snapshotting and versioning snapshots is that while the first operates completely in memory, the latter has to persist versioning information to be able to determine (across reboots, for example) which are immutable and which are mutable data and meta-data blocks. So, in order to manage various versions, and keep track of immutable blocks, additional features need to be incorporated to directory and file implementations. Besides persisting information related to immutable data and meta-data blocks, a user interface needs to be made available so that users can access different versions of the file or directory.

VI. CONCLUSIONS

In this paper, we have presented **meta-data snapshotting** as a low-cost, scalable and simple technique for file system consistency. One novelty which meta-data snapshotting brings in comparison to other techniques aimed at file system consistency, is that meta-data snapshotting permits the on disk image of the file system to be consistent at all times. Our experiments indicate that meta-data snapshotting provides this kind of availability at very little cost. In meta-data update

intensive workloads, it performs with at most 4% performance degradation when compared to its upper-bound, and runs 109% faster when compared to a write-ahead journaling file system.

To summarize some of the advantages of meta-data snapshotting presented:

- 1) It reduces the number of writes issued by the file system, if compared to journaling, log-structured or soft update mechanisms. Journaling and log-structured use write-ahead logging, and therefore have to write the same information twice, while the soft updates mechanism has to possibly write the same buffer more than one time in order to respect the buffer's pending dependencies.
- 2) It is a much simpler technique than soft updates, and also arguably simpler than logging-based solutions.
- 3) The meta-data snapshotting scheme is highly concurrent, and results show that it scales at the same rate as an asynchronous file system, with no consistency guarantees. Journaling file systems have a central point of contention: the transaction journal. It is unclear how journaling file systems scale as the size of the file system grows. (Bryant et al.[35] describe results of performance and scalability for four Linux journaling file systems). As disk seek times have not improved at the same rate as CPU and memory performance and disk transfer rates, keeping the journal up-to-date could become a performance bottleneck.
- 4) It can be easily extended to support transaction and versioning capabilities for both meta-data and data.
- 5) The on-disk version of the file system is consistent, at all times. With meta-data snapshotting, as with other file system consistency techniques, applications always see the most current version of meta-data, while the disk always contains a consistent copy. But an advantage with meta-data snapshotting is that the file system is completely consistent upon a system crash, whereas soft update based file systems have to run file system checkers to recollect unused resources [36] and logging schemes have to process the log. Even though the file checker has been implemented to run as a background task, to not hinder availability, as disk sizes grow, so should the time needed for the file checker to complete scanning the file system.

REFERENCES

- [1] R. Hagmann, "Reimplementing the Cedar File System using Logging and Group Commit," in *Proceedings of the 11th SOSF*, Austin, TX, Nov. 1987, pp. 155–162.
- [2] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992. [Online]. Available: <http://citeseer.nj.nec.com/rosenblum91design.html>
- [3] G. R. Ganger and Y. N. Patt, "Metadata Update Performance in File Systems," in *Proceedings of the 1st OSDI*, Monterey, CA, USA, Nov. 1994, pp. 49–60. [Online]. Available: <http://citeseer.nj.nec.com/ganger94metadata.html>
- [4] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, vol. 2, no. 3, pp. 181–197, 1984. [Online]. Available: <http://citeseer.nj.nec.com/article/mckusick84fast.html>

- [5] M. Seltzer, G. Ganger, M. K. McKusick, K. Smith, C. Soules, and C. Stein, "Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems," in *USENIX Annual Technical Conference*, June 2000, pp. 18–23. [Online]. Available: <http://citeseer.nj.nec.com/seltzer00journaling.html>
- [6] R. C. Unrau, O. Krieger, B. Gamsa, and M. Stumm, "Hierarchical clustering: A structure for scalable multiprocessor operating system design," *The Journal of Supercomputing*, vol. 9, no. 1–2, pp. 105–134, 1995. [Online]. Available: citeseer.nj.nec.com/unrau93hierarchical.html
- [7] O. Krieger and M. Stumm, "HFS: A flexible file system for large-scale multiprocessors," in *Proceedings of the DAGS/PC Symposium (The Second Annual Dartmouth Institute on Advanced Graduate Studies in Parallel Computation)*, 1993.
- [8] O. Krieger, "HFS: A flexible file system for shared-memory multiprocessors," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Toronto, 1994.
- [9] O. Krieger and M. Stumm, "HFS: A performance-oriented flexible filesystem based on build-block compositions," *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 286–321, 1997.
- [10] R. A. Lorie, "Physical Integrity in a Large Segmented Database," *ACM Transactions on Database Systems*, vol. 2, no. 1, pp. 91–104, 1977.
- [11] J. Gray, P. R. McJones, M. W. Blasgen, B. G. Lindsay, R. A. Lorie, T. G. Price, G. R. Putzolu, and I. L. Traiger, "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys*, vol. 13, no. 2, pp. 223–243, 1981.
- [12] "K42 File System - KFS," <http://lcpd.ime.usp.br/twiki/bin/view/Main/KfsGroup>.
- [13] "The K42 Project," <http://www.research.ibm.com/K42/>.
- [14] "The Linux kernel," <http://www.kernel.org/>.
- [15] M. Auslander, H. Franke, B. Gamsa, O. Krieger, and M. Stumm, "Customization lite," in *Hot Topics in Operating Systems*. IEEE, 1997, pp. 43–48.
- [16] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham, "The Episode file system," in *Proceedings of the USENIX Winter 1992 Technical Conference*, San Francisco, CA, USA, 1992, pp. 43–60. [Online]. Available: citeseer.nj.nec.com/chutani92episode.html
- [17] "JFS for Linux," <http://oss.software.ibm.com/jfs>.
- [18] "XFS: A high performance journaling file system," <http://oss.sgi.com/projects/xfs>.
- [19] "Reiserfs," <http://www.namesys.com>.
- [20] S. Tweedie, "Journaling the Linux ext2fs Filesystem," in *LinuxExpo '98*, 1998. [Online]. Available: citeseer.nj.nec.com/288237.html
- [21] "VERITAS file system," http://www.sun.com/storage/software/storage_mgmt/filesystem/.
- [22] H. Custer, *Inside the Windows NT File System*. Microsoft Press, 1994.
- [23] D. Giampaolo, *Inside BeOS: Modern File System Design*. Morgan Kaufmann Publishers, 1998.
- [24] U. Vahalia, C. Gray, and D. Ting, "Metadata logging in an NFS server," in *Proceedings of the Winter 1995 USENIX Technical Conference*, 1995, pp. 265–276.
- [25] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," in *USENIX Winter*, 1993, pp. 307–326. [Online]. Available: <http://citeseer.nj.nec.com/seltzer93implementation.html>
- [26] M. I. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. N. Padmanabhan, "File system logging versus clustering: A performance comparison," in *USENIX Winter*, 1995, pp. 249–264. [Online]. Available: citeseer.nj.nec.com/seltzer95file.html
- [27] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt, "Soft updates: a solution to the metadata update problem in file systems," *ACM Transactions on Computer Systems*, vol. 18, no. 2, pp. 127–153, 2000. [Online]. Available: citeseer.nj.nec.com/ganger00soft.html
- [28] M. K. McKusick and G. R. Ganger, "Soft updates: A technique for eliminating most synchronous writes in the fast filesystem," in *Proceedings of the FREENIX track: 1999 USENIX Annual Technical Conference*, 1999, pp. 1–17. [Online]. Available: citeseer.nj.nec.com/mckusick99soft.html
- [29] R. Card, T. Ts'o, and S. Tweedie, "Design and Implementation of the Second Extended Filesystem," in *Proceedings of the First Dutch International Symposium on Linux*, 1994. [Online]. Available: <http://e2fsprogs.sourceforge.net/ext2intro.html>
- [30] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson, "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," in *Proceeding of the 10th SOSF*, Orcas Island, Washington, Dec. 1985, pp. 15–24.
- [31] J. Katcher, "Postmark: A new file system benchmark," Network Appliance, Tech. Rep. TR-3022, October 1997.
- [32] A. Tridgell, "The dbench benchmark." [Online]. Available: <http://samba.org/ftp/tridge/dbench/>
- [33] K. L. Swartz, "Adding response time measurement of CIFS file server performance to NetBench," in *Proceedings of the USENIX Windows NT Workshop*, 1997, pp. 87–94. [Online]. Available: citeseer.nj.nec.com/154290.html
- [34] J. D. Blair, *Samba: Integrating UNIX and Windows*. Specialized Systems Consultants, Inc., 1998. [Online]. Available: <http://www.ssc.com/ssc/samba/>
- [35] R. Bryant, R. Forester, and J. Hawkes, "Filesystem performance and scalability in linux 2.4.17," in *Proceedings of the Freenix Track of 2002 USENIX Annual Technical Conference*, 2002.
- [36] M. K. McKusick, "Running 'fsck' in the Background," in *BSDCon 2002*, 2002. [Online]. Available: <http://www.usenix.org/publications-library/proceedings/bsdcon02/mckusick.html>