Structures for Efficient File System-Scale Partial Persistence

Dan R. K. Ports and Austin T. Clements
{drkp,amdragon}@mit.edu

May 12, 2005

Abstract

A persistent file system *stores every previous state of each file, allowing convenient access to the full state of the file system as it appeared at any point in the past. Achieving this convenient feature presents a challenging data structural problem because the amount of data involved is so large: it must use as little space as possible, and provide efficient operations for modifying the current state and accessing both current and past states. We formalize persistent file systems as a problem in data structures, and analyze it in the context of the external memory model. We begin by considering the design of our initial solution to this problem from the PersiFS$_1$ file system, which is based on a log of metadata changes and an indirection layer for storing file data. These "systems" data structures support the desired operations, but are not asymptotically efficient. Applying more advanced data structures, we refine the design into the next version, PersiFS$_2$. We use B^+-trees for file content indexing in order to improve the space efficiency of the system, and we present a novel partially-persistent B^+-tree design, which can be used to track changes to files with logarithmic modification and query cost. PersiFS$_2$ has been implemented as a working file system with these data structures, and our measurements indicate that the new file system data structure provides dramatically improved access time for previous revisions with a small increase in cost for modifications.*

1 Overview

The data-structural notion of *persistence* — the ability to access previous states — is powerful in many situations. We consider persistence in the context of file systems, where it provides access to previous versions of data that may have been modified or deleted. This capability has great practical utility, but finding an appropriate representation for the state of the file system presents several challenges. The volume of data involved necessitates a compact data structure that supports efficient queries and modifications.

In order to address these challenges, we present a novel set of data structures optimized for the requirements of a persistent file system: applying modifications to the content of files and reading the contents of any revision. We develop a partially-persistent B$^+$-tree for storing the history of the metadata state of the file system, and a block store that uses cryptographic fingerprints and content-sensitive chunking to leverage file similarity for compactness.

The outline of this report is as follows: we begin by motivating the need for a persistent file system data structure in Section 2, and describe the high-level design of our application, the PersiFS persistent network file system. In Section 3, we discuss the requirements and challenges for file system data structures. We first present an initial design in Section 4, based on simple data structures that provides the necessary functionality but limited efficiency. We then discuss in Section 5 how to refine the representation using more advanced data-structural techniques, including a new persistent variant of B^+-trees. Finally, Section 6 discusses the implementation of these data structures and provides measurements comparing the two designs, and Section 7 provides concluding remarks.

2 Background

The data structures described in this paper were created as a part of our implementation of a persistent (i.e. version-controlled) network file system called PersiFS [4]. In this section, we examine the features, challenges, and high-level design of PersiFS in order to understand the context for our data structures, and how they interact with the other components of PersiFS.

2.1 Motivation

Users frequently wish to undo changes they have made, whether intentionally or inadvertently, to the file system — for example, inadvertent deletions, restoring files corrupted by application bugs, or simply reverting to an earlier revision of a document. Regular file systems do not support this operation natively, and periodic backups, "trash can" interfaces, and "undelete" tools provide only a partial, inadequate solution. By allowing access to any previous state of the file system, PersiFS makes these tools unnecessary, since files are never truly deleted from the file system. Fears of accidental overwrite, rename or deletion are unnecessary.

Backup systems that archive snapshots of the state of the filesystem are a common solution. Along the same lines, there exist *snapshot-based* version-controlled file systems [11, 7] that are the natural extension of this idea, essentially making automated backups that are readily available (often with some space optimizations). These systems, however, cannot track changes that occur in the interval between snapshots. PersiFS improves on these systems by using *continuous versioning*: each change to the file system is stored as a new revision, which solves this problem. However, this makes an efficient representation all the more critical.

2.2 PersiFS Design

PersiFS is a networked file system that provides access to previous revisions through a novel, convenient file system interface. A client system can mount a PersiFS volume via a standard NFS interface, say as `/persifs`, and then the current version of the file system is available for read/write access at `/persifs/now`. Moreover, previous versions of the file system can be accessed simply by specifying any time stamp instead. For example, `/persifs/2005-05-12-12-00-00` is a read-only snapshot of the entire file system as it appeared at noon on May 12$^{\text{th}}$, 2005. The interface is entirely through the file system: unlike some other version-controlled file systems, no special tools are required to access previous revisions.

The implementation of PersiFS is based on a centralized user-mode NFS server. A translation layer converts NFS requests to operations on the *file system data structure*, described below, that is the core of the system.

3 File System Data Structures

In order to analyze and develop data structures for PersiFS, we will formalize the requirements in the form of a *file system data structure*.

We view our file system as a collection of files, each of which can be represented simply as a string of data. In reality, files are made up of both data and metadata such as last-modified time, and as a practical matter these are often stored separately, but from an abstract data type perspective this distinction can be ignored. Each file is uniquely identified by a numeric identifier; in keeping with standard Unix terminology, we refer to this identifier as the file's *inumber*.

The file system data structure supports two operations. Given a file ID, it can *modify* the current version of the file's contents — inserting, removing, or modifying a substring at any point in the file. It can also *read* a substring from a given file either in the current version or in a previous revision identified by a time stamp.

The amount of data typically stored in a file system is very large, and we can anticipate that a persistent file system will be even larger, since it must store not only the current state but all previous states. This requires that our data structure be *space-efficient*: the amount of space required to store a file modification should be proportional to amount of unique data introduced, not to the total size of the file (or, worse, the total size of the filesystem). It must also be able to efficiently support the READ and MODIFY operations. The entire data structure is likely to be far too large to be stored in memory, so our analysis of the efficiency of these operations must be in the *external-memory model* [1]. Note that cache-obliviousness is not required, since the file system implementation can be made aware of the characteristics of its disk and cache and can explicitly perform memory management accordingly.

4 An Initial Representation

The first implementation of PersiFS, PersiFS$_1$, was designed from a systems mindset, where simple, though slow data structures trump advanced, fast data structures. This section gives an overview of the design and structures used in PersiFS$_1$ because PersiFS$_2$ retains the same overall architecture and to serve as a straw-man implementation for comparison.

4.1 Metadata Log

The data structure at the core of the persistence of PersiFS$_1$ is the *metadata log*. This simply contains a set of file metadata modification records with monotonically increasing time stamps. Each record is a 3-tuple of $\langle$*time stamp*, *inumber*, *metadata*$\rangle$, where *time stamp* indicates the time of the modification, *inumber* indicates the file being modified, and *metadata* contains the new metadata for that file. For our purposes, we can treat the metadata as simply a representation of the file's contents, using a layer of indirection that will be discussed in Section 4.2, so that a change to the contents of a file results in the metadata log recording a metadata change for that file. In the actual implementation, files have distinct metadata such as their size and last modification time, and this is stored directly in the log for more convenient access, but this is simply an optimization.

In order to lookup the metadata for an inumber at a given time stamp, the log is replayed until either the end is reached or the replay encounters a modification entry for the inumber containing a time stamp greater than the requested time stamp. Because the metadata log requires monotonic ordering of time stamps, such an entry guarantees that the last encountered entry for the requested inumber must contain the requested metadata. In order to modify metadata, the metadata log needs only append the modification to the log with the current time stamp.

Assuming a relatively constant rate of change in the file system, queries take $O(t)$ memory transfers, where t is the amount of time that has passed between the beginning of the file system (and thus the beginning of the metadata log) and the requested time stamp. Writes take $O(1)$ memory transfers and consume $O(1)$ additional space. The implementation optimizes the common case of queries to the latest revision by maintaining a separate map that contains only the latest revision.

In order to make query time reasonable, the metadata log introduces *snapshots*. After every C modification entries the metadata log contains a snapshot of the entire mapping of inumbers to metadata at the current time. This allows log replays to begin at the latest snapshot preceding the requested time stamp, thus bounding the time span the replay must account for. Of course, this comes at the cost of writing the snapshot, which is of size $O(n)$, where n is the number of files in the file system. With snapshots, replays take $O(n + C)$ memory transfers (we assume the locations of each snapshot are small enough to fit in cache, which is reasonable for a real file system), and writes take $O(n/C)$ memory transfers, amortized. In the actual system, C is constant and snapshots are taken in the background, so writes are always $O(1)$ as long as the file system is not heavily loaded and replays have a minimal constant factor.

4.2 Superblob

The metadata log only stores metadata, so PersiFS_1 also needs somewhere to store actual file contents. This is the purpose of the *superblob*, which is simply an append-only vector of strings. The contents of a file are broken into chunks whose concatenation forms the original contents of the file (Section 4.3 discusses how this chunking is done). These chunks are then appended to the superblob. The locations of the chunks in the superblob and the order in which they constitute the file are recorded in the file metadata so the file can be reconstructed later.

Assuming chunks are approximately constant size, reads and writes to the superblob both take $O(1)$ memory transfers. Writes consume $O(1)$ additional space.

Using chunks instead of storing the entire file as one string allows modification to some substring of a file to update only the affected chunks. Because of this, writes to a file consume additional superblob space linearly proportionate to the size of the write instead of the length of the modified file. No additional memory transfers are required to perform a write because it just appends to the superblob. Likewise, reads from the superblob require a number of memory transfers proportionate to the size of the read.

4.3 Content-Sensitive Chunking

When dividing the contents of a file into chunks for insertion into the superblob, one possibility is to use fixed chunk boundaries at regular intervals. However, while this is optimal for modifications in which a substring s at position n is overwritten by a new substring of length $|s|$, modifications in which a substring is overwritten by a substring of different length (for example, inserting), leads to a framing change for the remainder of the file, which requires medications linear in the length of the file. Optimally, the number of chunk modifications required should be linear in the length of the new substring.

To efficiently support this operation, chunk boundaries are not placed at regular intervals, but instead use *content-sensitive chunking*. This technique places chunk boundaries based on file contents such that local modifications to file contents, including insertions and deletions, only affect chunks in that region of the file. We use the same Rabin fingerprint-based algorithm as LBFS [10] for content-sensitive chunking.

This algorithm slides a Rabin fingerprint window of fixed size W across the string being chunked. Let $f(A)$ denote the value of this fingerprint at position A. Whenever $f(A) \equiv x \pmod K$, for some arbitrary but fixed x, a chunk boundary is placed at position A. Thus, the expected length of a chunk is K. A modification to the string can only affect the values of $f(A)$ for which the modification lies between A and $A + W$. Any chunk boundaries, and thus any chunks, before position A or after position $A+W$ cannot be affected. Careful implementation of this algorithm requires expected chunk modifications proportional to the length of the change.

5 Improving the Representation

The data structure described above provides the necessary functionality to implement a persistent file system, and indeed we were able to use it to build our initial system, PersiFS_1. However, it leaves much to be desired. Though the use of the superblob with content-sensitive chunking allows the space for storing file data to grow proportionally to the size of the modifications, this is still not optimal. Adding a bit more cleverness can reduce space usage. The metadata log's query time bounds are also troublesome, since performing a query requires replaying the full length of the log from the last snapshot, and snapshots are relatively expensive to perform in terms of both storage and write time.

We developed the next version of our system, PersiFS$_2$, using more advanced data structure techniques to improve these aspects of the performance of the file system. In doing so, since we are performing searches in the external-memory model, we start with the canonical structure for external-memory searching, the B-tree. Recall the standard definition and properties of a B-tree:

Definition 1. *A* B-tree *of order T is a multi-way search tree where each leaf is at the same level, and each non-leaf node (except possibly the root) contains between $T-1$ and $2T-1$ key/value pairs, and an internal node with k keys has $k+1$ children.*

Theorem 2. *If $T = \Theta(B)$, a* B-tree *of order T with N elements supports* Insert*,* Delete*, and* Search *operations in $O\left(\lg_{B+1} N\right)$ memory transfers.*

Proof by indirection. Folklore. See, e.g., CLRS [5] or Knuth [8] for details. □

We will in fact concentrate on B$^+$-trees, which are identical to B-trees except that they store values only in the leaves, which allows the branching factor T to be increased because more keys can be fit into each block. The bounds of Theorem 2 of course still apply.

5.1 Chunk Fusion

As described in Section 4.2, the superblob structure is simply a sequence of data chunks that are pointed to by file metadata entries. Appending a new chunk to the superblob requires constant time, and retrieving data also requires constant time because it is pointed to by an offset into the structure. The size of the superblob is precisely the amount of data inserted, which with content-sensitive chunking is proportional to the amount of data changed with each revision. This is not optimal, because multiple chunks may have the same content — we can achieve size closer to the total amount of *distinct* content in the file system.

This is accomplished using a technique we call *chunk fusion*: when a new chunk is added to the superblob, rather than immediately appending it to the disk, we first check whether another chunk with the same content already exists. If so, we *fuse* the new chunk with the existing chunk, storing only a single copy in the structure, and pointing to it multiple times. We do this by identifying chunks in the superblob with a cryptographic hash (in our implementation, a SHA-1 fingerprint), and storing a mapping of fingerprints to existing blocks with that hash.

The mapping of fingerprints to superblob addresses is accomplished using a standard (non-persistent) B$^+$-tree to index the superblob. Since insertion into the superblob now requires a B$^+$-tree Search in order to check whether the chunk already exists in the system, and a Insert into the blob index if not, insertion now has $O\left(\log_{B+1} N\right)$ cost, rather than $O(1)$. The cost for retrieving data from the superblob remains constant, since chunks can also be identified by their offset in the superblob.

Identifying chunks by their cryptographic hash does present the possibility of error if two distinct chunks should happen to have the same hash value. This probability is, of course, small because the fingerprint is a 160-bit SHA-1, so in practice it is reasonable to disregard it. This is a standard systems technique employed by systems such as the Venti [12] and LBFS [10] storage systems and the Arpeggio [3] content distribution system. For additional reliability, it is possible to retrieve the chunk identified from the blob index and verify that it has exactly the same content before performing chunk fusion. Note that this would require the blob index B$^+$-tree to be able to support multiple values for the same key

— though if it does not, the system will still operate correctly but with decreased efficiency since chunk fusion can not be performed for certain chunks.

5.2 Persistent B$^+$-trees

We now turn to the problem of how to represent the evolution of the file system more efficiently than the metadata log described in Section 4.1. Recall that the metadata log meets our requirement of being able to retrieve the content of a file (modulo the indirection through the superblob) as it appeared at any previous time, but is not efficient: it can perform modifications in constant time, but reads require potentially linear time because they must replay the log. We will now proceed to the design of a B$^+$-tree with partially-persistent operation, which achieves logarithmic memory transfers for both modifications and queries. Of course, in addition to providing an asymptotically more efficient replacement for the metadata log, this generic data structure is of independent interest as it has many other applications.

5.2.1 Interface

Our partially-persistent B$^+$-tree supports the standard modification operations of Insert and Delete, which only modify the current state of the tree. The Search operation takes as parameters both the key to search for and a version number; the corresponding state of the tree is searched.

In addition, our B$^+$-tree also supports a Commit operation, which is used to group together logically related modifications to the tree into a single revision. Performing a sequence of Insert and Delete operations changes the current state of the tree, but does not increment the version number until a Commit is performed. Thus, when performing a search, we can only access the state of the tree at Commit-points.

Essentially, the Commit operation is discarding some intermediate states. This provides two advantages. The first is efficiency: performing one Insert operation, for example, may result in many changes to the tree as nodes are split, and these intermediate states are *redundant*, as querying them gives no useful information. The second is atomicity: our application may perform two related changes — for example, moving a file from one directory to another by adding it to the new directory and then removing it from the old — and accessing the version in between would give an *inconsistent* state in which the file existed in both directories. Performing a Commit only after both changes are made prevents the inconsistent state from being stored.

5.2.2 Design

To implement a partially-persistent B$^+$-tree, we use a variant of the generic technique for adding persistence to data structures introduced by Driscoll, Sarnak, Sleator, and Tarjan [6]. Their technique adds a *modification box* to each node, which stores a single change to the data or pointers in that node, along with a time stamp indicating when the change took effect. To make a change to a node, their scheme requires first attempting to place the change in the modification box of that node; if it is not available, they instead make a duplicate copy of the node, apply the change to it (leaving the modification box empty), and recursively change the parent to point to the new copy instead of the old. Performing a search simply requires checking the time stamp at each node traversed in order to decide whether to use or ignore the change stored in the modification box.

We modify this technique slightly such that the modification box holds an entire copy of the changed node rather than simply a single modification. This allows for versions that have more than one change, which is necessary in order to implement our desired version-on-Commit semantics and achieve a space-efficient representation.

Our data structure can therefore be implemented using the standard Insert and Delete

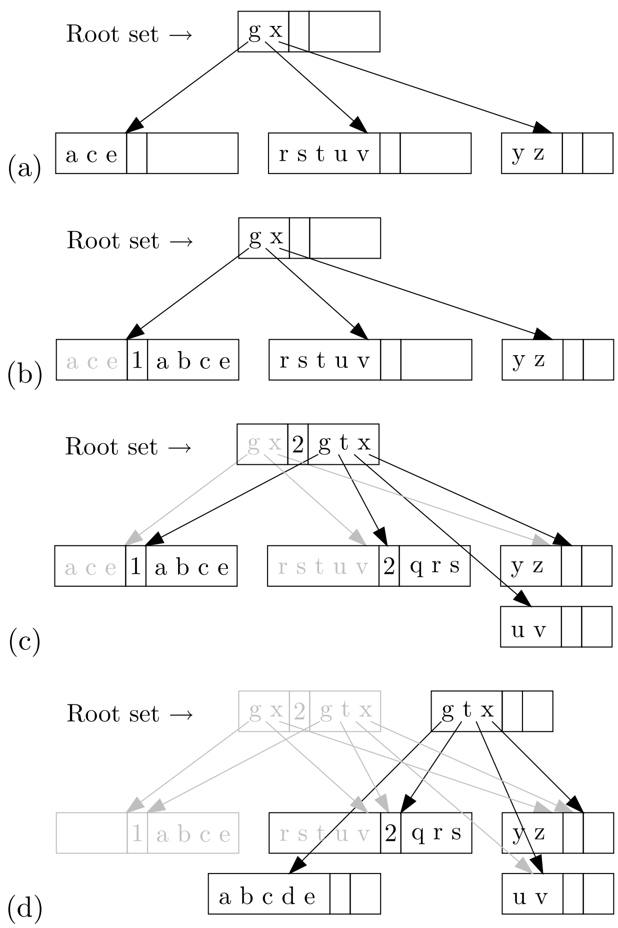

Figure 1: Four modifications to a persistent B$^+$-tree. Data in the latest revision is shown in black. (a) The initial tree at revision 0. (b) Inserting b changes a modification box and creates revision 1. (c) Inserting q requires a split and creates revision 2. (d) Inserting c requires creating a new node for revision 3 because the modification box is already full.

algorithms for a non-persistent B$^+$-tree, augmented with the following persistence procedure. The tree maintains a logical "current time stamp" that is incremented by each Commit operation. Whenever a node is to be modified, if the node's time stamp is equal to the current time stamp, the modification can be performed directly — this change should be merged together with any other changes that occurred since the previous Commit.

Otherwise, a new version of the node must be created. If the node's modification box is unused, we copy the contents of the node into the modification box, set the time stamp of the modification box to the current time stamp, and apply the change there, as in Figures 1(b) and 1(c). If the modification box is already in use, we create a new node containing a copy of the existing contents of the node with the change applied and an empty modification box, and update the parent's child pointers to point to this node, as in Figure 1(d). Modifying the parent requires recursively applying this procedure, which may need to copy it and recurse again. If this recursion reaches the root node, copying will create a new root; we maintain a non-persistent B$^+$-tree that stores pointers to all the roots of the persistent B$^+$-tree indexed by the time stamp at which they became valid.

Related work Other designs have been proposed for adding persistence to B$^+$-trees. Lanka and Mays presented a design for a *fully*-persistent B$^+$-tree [9], though our application does not require full persistence, and so a simpler design is more appropriate. Arge, Danner, and Teh created a partially-persistent B$^+$-tree [2] that avoids modification boxes by applying a time stamp to each node and enforcing invariants about how many dead and live children a node can have. This design achieves the same asymptotic bounds as our design with better constant factors; we opted to trade off constant factors in space and time efficiency for simplicity and ease of implementation. In particular, our implementation was greatly simplified by the fact that persistence is performed purely at the *node* level: our implementation of the standard B$^+$-tree insertion and modification algorithms remained unchanged, layered atop a new abstraction that selected the appropriate copy of a given node and handled modification boxes and path copying when appropriate.

5.2.3 Analysis

The persistent B$^+$-tree data structure described above provides asymptotically efficient operations, as the following theorems show. In each, we assume that the branching factor T is of the same order of the block size B.

Theorem 3. *A* SEARCH *operation on a persistent B$^+$-tree containing t versions, at a time stamp where the size of the tree was N elements, requires* $O\left(\log_{B+1} N + \log_{B+1} t\right)$ *memory transfers.*

Proof. Searching a persistent B$^+$-tree requires initially searching the B$^+$-tree of root positions for the predecessor of the desired time stamp; by Theorem 2, this requires $O\left(\log_{B+1} t\right)$ memory transfers. Once this is completed, performing the search can proceed in the same way as a standard B$^+$-tree search. We need only check the time stamp of the internal nodes at each level in order to decide whether to use the version in the modification box or not; this adds only a small constant factor overhead, so performing this part of the search requires $O\left(\log_{B+1} N\right)$ memory transfers. □

Note that if the access sequence consists solely or primarily of INSERT operations rather than DELETES, as in our application, then $t \leq n$, and so this is a $O\left(\log_{B+1} N\right)$ bound.

Lemma 4. *Performing a modification to a node in a persistent B$^+$-tree requires amortized constant additional cost due to persistence in both space and time.*

Proof sketch. The proof is in Driscoll et al. [6], and is based on a charging argument using the number of nodes in the current version of the tree with their modification box full as the potential function. □

Theorem 5. INSERT *and* DELETE *operations on a persistent B$^+$-tree containing N elements in the current version can be performed using* $O\left(\log_{B+1} N\right)$ *memory transfers.*

Proof. Immediate from Theorem 2 and Lemma 4. Note also that because versions are only created by a COMMIT operation, it is not always necessary to perform the persistence procedure before modifying a node, since a copy of that node at the current time stamp might have been created by a previous operation. □

Theorem 6. *Each modification to a persistent* $B^+ - tree$ *requires* $O\left(1\right)$ *amortized space.*

Proof. We consider only INSERT operations here; the argument for DELETE operations is analogous. A result of Theorem 2 is that $O\left(1\right)$ nodes are split in performing each insertion, amortized. Since new nodes are only created by splits (and once per INSERT for the new leaf), this means the size of the tree grows by a constant amount, and by Lemma 4, the persistence procedure only adds another constant factor. □

5.2.4 Application

We now show how to use the generic mechanisms of persistent and ephemeral B$^+$-trees to address our original problem of building a file system data structure, replacing the inefficient metadata log with an *arborescent metadata map.* Not surprisingly, the core of this structure is a persistent B$^+$-tree which maps file IDs to file metadata (and hence to file contents, via the superblob indirection). The only source of additional complexity is the need for a mapping between the logical time stamps (i.e. revision numbers) used by the persistent B$^+$-tree and clock time. A standard B$^+$-tree serves this purpose neatly, as shown in Figure 2.

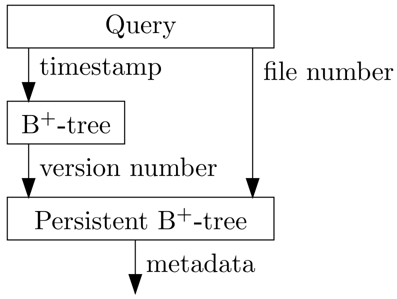

Figure 2: Replacing the metadata log with B+-trees

6 Implementation and Evaluation

We built the second version of PersiFS, $\mathrm{PersiFS}_2$, using the data structures described in Section 5. The implementation consists of slightly over 10,000 lines of C++ source code[1] written atop the `libasync` toolkit, including 2,000 lines directly related to generic implementations of persistent and ephemeral B+-trees. The data structure implementations are fully templatized and include a marshaling framework so as to be suitable for other applications.

In order to compare the actual performance of $\mathrm{PersiFS}_1$ and $\mathrm{PersiFS}_2$, we ran two benchmarks. In order to emphasize the performance properties of the arborescent metadata map, we reduced T from 1024 to 8 for these benchmarks. Also, in order to capture the full cost of writes to the metadata log, we included the cost of periodic snapshots in the cost of the write operation.

The first test, whose results are illustrated in Figure 3, created 5000 files, recording the number of memory transfers performed during the creation of each file. The metadata log of $\mathrm{PersiFS}_1$ performed as expected, generally requiring only one or two memory transfers. Write operations that trigger snapshots are expensive, requiring memory transfers linear in the number of files. Writes to the arborescent metadata map of $\mathrm{PersiFS}_2$ were generally somewhat more expensive than writes to the log, but scaled gracefully with the number of files. With $T = 1024$, the average number of memory transfers was merely 5 with 5000 files.

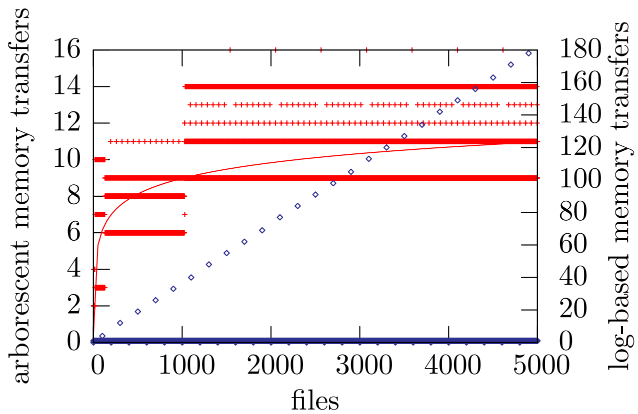

Figure 3: Inserting 5000 files into the file system scales logarithmically with the arborescent metadata map. Doing the same with the metadata log results in constant memory transfers for most write operations, with large, linear time required for periodic snapshots.

The second test, illustrated in Figure 4, retrieved past versions of files from file systems of differing sizes. Here the arborescent metadata map truly shines, requiring only a logarithmic number of memory transfers with a very small constant, while the metadata log scales linearly with a large constant.

The benchmark results reflect the designs of the underlying systems. The structures in $\mathrm{PersiFS}_1$ are heavily optimized for reads and writes to the current version, while they greatly sacrifice the performance of reading from past versions. $\mathrm{PersiFS}_2$, on the other hand, strikes a better balance between the features of a persistent file system, providing somewhat slower write performance, but dramatically improved access to past versions.

[1]The development source code is currently available from `http://www.ambulatoryclam.net/svn/classes/6.824/project/src-v2/`

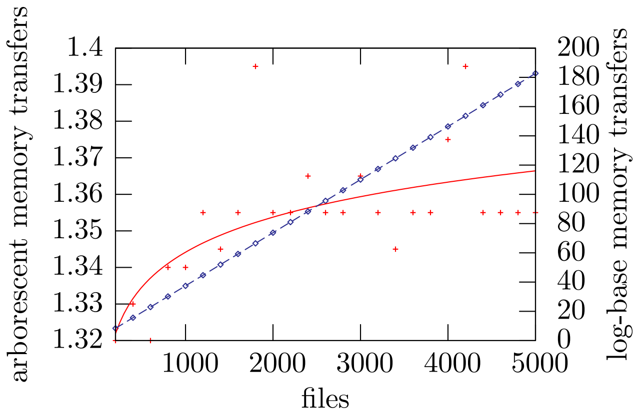

Figure 4: Recalling past file metadata from file systems of different sizes scales logarithmically with a small constant for the arborescent metadata map. For the metadata log, this operation is expensive, requiring linear time with a large constant.

7 Conclusion

Continuously versioned file systems like PersiFS have the potential to change the way users view and interact with their files. By adding a time axis to the file system and giving it a very tangible archeology, users gain an extra dimension of expressive power over their manipulations of the file system. They obtain peace of mind and ease of use by eliminating the need to worry about the integrity of their files.

This power can only be achieved through a correspondingly powerful data structure underlying the file system. Time and space efficiency are crucial. Our persistent B^+-tree-based data structure meets these requirements, not only in the theoretical, asymptotic sense, but also in practice.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[2] L. Arge, A. Danner, and S.-M. Teh. I/O-efficient point location using persistent B-trees. *J. Exp. Algorithmics*, 8:1.2, 2003.

[3] A. T. Clements, D. R. K. Ports, and D. R. Karger. Arpeggio: Metadata searching and content sharing with Chord. In *Proc. IPTPS '05*, Ithaca, NY, Feb 2005.

[4] A. T. Clements, D. R. K. Ports, B. A. Schmeckpeper, and H. Yuen. PersiFS: A continuously versioned network file system. 6.824 project report, MIT, May 2005. Available from `http://www.ambulatoryclam.net/svn/classes/6.824/project/report/-report.pdf`.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.

[6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[7] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 235–246, San Fransisco, CA, USA, 17–21 1994.

[8] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Boston, MA, second edition, 1998.

[9] S. Lanka and E. Mays. Fully persistent B+-trees. *Proc. ACM SIGMOD '91*, pages 426–435, 1991.

[10] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Symposium on Operating Systems Principles*, pages 174–187, 2001.

[11] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.

[12] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey, CA, 2002.