Efficient Metadata Maintenance in Versioning File Systems

A. Rama Kotaiah
D.E. Shaw Inc,
Bangalore, India
Email: Rama.Kotaiah.Annadanam@deshaw.com

Gautam Barua
Dept. of CSE, IIT Guwahati
Guwahati 781039, India
Email: gb@iitg.ernet.in

***Abstract*— A number of multi-version file systems have been proposed and implemented in recent years. Besides presenting an option for the user to recover from mistakes, versioning is also useful for crash recovery. This paper presents a multi-version file system. The design is based on providing versioning support for selected files in a system, and in an environment where such multi-version files are long lasting and change relatively slowly. As with most multi-version systems, the aim is to ensure that access to the current version of a file are not affected by the presence of multiple versions. Writes are made ”in-place” with the older data being shifted to another location. The main situation identified is an office environment where multiple versions of documents for rules, regulations, etc., are common. Separate strategies are proposed for maintaining version information of attributes and of the addresses of blocks of different versions. An analysis of the performance of the system using trace data from University of California, Berkeley, shows that the degradation in performance of accessing the current version in the presence of multiple versions of a file, is small and acceptable. The schemes presented allow older versions to be accessed with an overhead that is less as compared to other multi-version systems whose information is available in the literature.**
KEYWORDS:file systems, versioning, document management

I. Introduction

Keeping old data that has been overwritten by new data is to maintain multiple versions of data. There are a variety of reasons for doing this and there are a variety of ways of doing this. One way of maintaining versions is to use some version control system that is implemented on top of a traditional file system. Such systems have been developed mainly for enabling the sharing of code or data by multiple users working independently on the data. The goals of such systems are different from the goals being addressed in this paper. Multiple versions are also kept in some database systems as a means to implement concurrency control. Interest in multi-version file systems have arisen from the need to handle the following requirements: recovery from user mistakes (to implement an *undo* facility), recovery from system corruption (there may be a need to *rollback* changes to some globally consistent state), or to analyse historical data for specific purposes (one such is in implementing *self securing* systems where analysing changes to detect intrusion is a requirement ([1]).

In business and in official work, multiple versions of documents is a common occurrence. Be it the Income Tax Rules of a Government, the leave rules of an organisation, or the Latex manual: all have versions. There is a need to access earlier versions of a document. A typical example is when the pay of some employees in an office are to be changed with retrospective effect based on the pay scales prevalent at an earlier date (this is a common occurrence in Govt. organisations in India). The version of the document containing the pay scales and rules on that date has to be accessed. This is an error-prone task under current methods of keeping versions. A variety of ad hoc methods are currently employed to maintain multiple versions of these documents. While higher level software can be used to meet such requirements (for example version control systems such as CVS ([2]) and specific systems such as reported in [3]), a file system that automatically maintains versions is an attractive alternative. Of course, higher level software will be required to avoid sequential rewrite of a file on every change as is done by most word processors. But we do not consider these issues in this work. While this kind of application is the main motivation behind the design presented in this paper, other, more general requirements have also been kept in mind.

The following are the major characteristics of this kind of requirement that have been used to design the system:

- files with multiple versions need to co-exist with files with only a single version.
- the number of versions of a file are typically not very large and the changes are not particularly frequent.
- there is no need to create a separate version for every write to a file: some form of *pruning* of versions can be done. In fact we assume that new versions are created only on every close of an open file that has been written into.
- there should be no performance penalty in accessing the current version as this is the most frequently accessed version.

A. Versioning File Systems

Journaling File Systems have been developed and two systems are available as part of Linux distributions, JFS from IBM ([4]), and Ext3 ([5]). Journaling systems make an entry of a change in a log or a journal first, and then the change at the actual place is made. The main motivation is to allow recovery from crashes, and to undo operations. Not much attention is therefore paid to the issue of storage overheads (journal entries get purged periodically), or of access overheads. As can be seen, a journal keeps multiple versions of a file. Log Structured File Systems ([6]) are another type of file systems that keep multiple versions of files as a side effect. Instead of having separate areas for a log and for the actual file, all information is kept in the form of logs. So when a file is written into, new versions of the changed blocks are created at the end of the log. The main motivation is to improve write performance and to allow better recovery. Space overheads are controlled by purging older versions of blocks. Here too, the system is not efficient for a versioning system with data being kept for long periods and where performance of access to older versions should be kept in mind.

Comprehensive Versioning File System(CVFS)([1]) is a system catering to the need of self-securing storage. So versions of all files will have to be kept, and some files may have frequent updates. The system uses two versioning techniques- Journal based metadata maintenance and B-Tree based metadata ([7]) for versioning metadata of files and directories respectively. In the journal based metadata storage, a journal stores the differences between the data block pointers and file attributes, and to access the older versions from the current version, the differences need to be roll-backed until the metadata of the required version is obtained. A B-Tree based metadata structure is similar to the single version B-Tree with a slight modification. Along with an entry of the form <*key,data*>, there is an additional attribute called timestamp defined for each entry. This makes the entry unique in the whole tree as no two entries have the same key and timestamp values.

The Elephant file system ([8]) works on the basis of retention policies for each file. A file can follow any one of the four retention policies - keep one, keep all, keep safe and keep landmarks. Keep one policy stores only one copy of the file similar to normal file systems whereas keep all policy stores all the versions of a file. Keep safe and keep landmark policies store a subset of the versions. In contrast to traditional file systems, the inode numbers denote an inode log instead of denoting the actual inode. The inode log contains the actual inode addresses of all the versions of a file.

Versionfs ([9]) is a Stackable File System providing the user with three space saving policies - normal, compression and sparse. In normal operation, the version information is stored in its entirety. In compression policy, the version information is compact and in sparse versioning, a sparse file stores the changes among the versions, containing holes of zeros where there are no changes.

A data block is the lowest level of granularity of access in file systems. Every modification results in the allocation of a new block even if only a few bytes in the block were changed. Byte range differencing ([11]) stores the bytes that have changed instead of storing the whole block.

CVFS meets the current version requirements well but suffers in the case of older version accesses. Versioning file systems have a disadvantage that they fragment the file system. To illustrate this, consider the FFS file system ([12]) and a file with five data blocks. Initially with the block allocation policy, the five blocks are stored most probably stored contiguously and no disk seeks are required to access all the data blocks. If the second and fifth blocks are modified and the new blocks are allocated in another block group, then in the view of the current version, the data blocks are scattered over two block groups and a disk seek is required to access all the blocks. Elephant integrates versioning at the block level, but then suffers from fragmentation. The design presented here works at the block level and tries to reduce fragmentation.

A recent work extends the ext3 file system ([10]) to provide a time-shifting snapshot of a file system. The main motivation of the design is to provide versioning so as to comply with various electronic record retention legislations that have come recently. While some of the implementation details are similar to the work presented here, the system does not provide for versions at logically significant points and so is not suitable for the environment envisaged in this work.

II. Metadata Versioning

The versioning of metadata includes versioning of both the attributes and block pointers. Attribute versioning is simpler as compared to block versioning as most attributes remain the same across versions. Different methods are proposed to handle attribute versions and block versions. A new version of attributes is created when the first write is issued by a process after the opening of a file. For every block, a copy of the block is created on the first write to the block. Further writes to a block till the file is closed results in no new versions. Using the terminology prevalent in the literature, in this system, versioned are being *pruned* to the close of a file. The user is provided with options to have no versioning on a file, standard versioning as described above, or to force creation of versions by a system call.

A. Attribute versioning

Attribute versioning is achieved by maintaining a *version tree* that stores attribute differences among various versions in a binary tree, excluding the current version. Information on the current version is kept with the inode so that there is no penalty in accessing the current version. Each node in the tree represents a version and it stores the difference between the attributes of the version it represents, and the version its parent represents. The root of a *version tree* stores all the attributes of the version it represents. The nodes are in inorder with respect to the version number. An example of a *version tree* with n versions is shown in Fig. 1. There are four attributes, ¡a,b,c,d¿. The attribute vector $(a1, b6, c8, d1)$ denotes that the attribute *a* has the value 1, attribute *b* the value 6, etc. Thus version n/2 stores values for all the attributes, being the root. It right child, represents version 3n/4 and attribute *b* is different with value 2 and attribute *d* is different with value 2 with respect to version n/2, and so only these two attribute values are stored. The attribute vector of version 3n/4 is $(a1, b2, c1, d2)$, obtained by merging the information in node 3n/4 and node n/2. In general, the information in all nodes from the node of interest to the root node has to be merged to get the attribute vector.

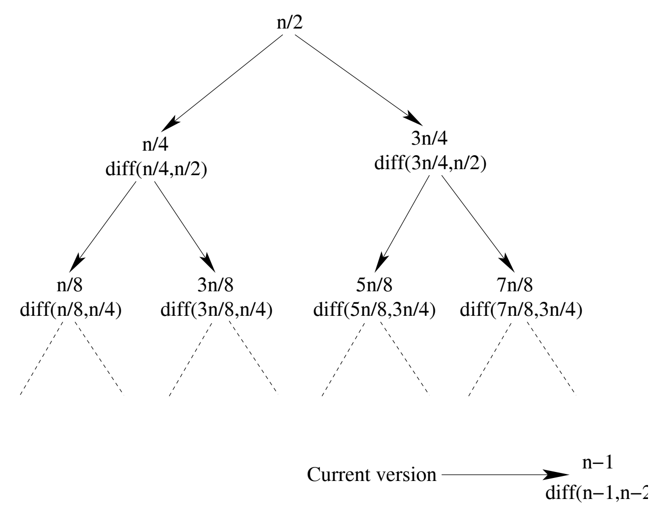

Fig. 1. A *version tree* with n-1 versions

So, in the worst case, x nodes have to be traversed, where x is the depth of the tree, to get the attributes for a version.

The inode of a file has pointers to the root of its *version tree*, and to the node representing the last version in the tree which is always the rightmost leaf node of the tree (this is the node representing version $n - 1$ in Fig 1). Version creation always results in a node attached as a right child to the right most node of tree. Keeping the tree balanced makes accesses of older versions faster as the depth of the tree is reduced, but balancing the tree after the creation of every version increases the time of writes to files. A compromise has been made and the tree is balanced only after every N_{th} new version. N_{th} is a configuration parameter.

Attributes store significant information and they always need to be maintained in a consistent state. A user may have access to the current version but may be denied access to older versions. This information can only be obtained from the access privileges of the older version metadata. Stable storage is used to maintain the consistency of attribute information.

Whenever a version is created, the only operation on the *version tree* is the addition of a node to the rightmost node of the tree. This can be performed in a single I/O for each version as the pointer to the node representing the last version is present in the inode. However, every N_{th} version the tree needs to be balanced. After every N_{th} versions, the tree comprises of a balanced tree with n nodes and a skewed tree with N_{th} nodes. The only nodes that need to be traversed and balanced are those that are in the right most branch of the tree, and their number is the depth of the tree which is given by

$$\log n + N_{th} \quad (1)$$

where
n is the number of versions present prior to the last balance of the *version tree*
N_{th} is the number of versions created after the last balance of the *version tree*.

TABLE I
PERFORMANCE OF *version tree*

No. of versions	Depth of *version tree*	No. of blocks required	Maximum I/Os to access a version
8	3	1	1
16	4	1	1
24	4	2	2
32	5	2	2
64	6	4	4

The attributes of older versions can be obtained by performing a binary search on the *version tree* starting from the root of the tree until the desired version is found, performing *roll backs* of the attributes at the intermediate nodes as explained above. In Fig. 1, to obtain the attributes of version 3n/8 we start from version n/2 obtaining the values $(a1, b1, c1, d1)$, through n/4 obtaining $(a2, b1, c2, d1)$, and reaching 3n/8 with $(a2, b3, c2, d3)$ as the attribute vector for version 3n/8. The maximum number of nodes that need to be traversed is the depth of the tree, given in Equation 1.

The number of nodes of the tree that can be fitted into a single block is a crucial factor as it decides the space required to store the entire *version tree*. Consider an inode of the ext2 file system of size 128 bytes (as per kernel 2.4.20) in which attributes take up 68 bytes. Those attributes that generally change for different versions include file size, access time, modification time, access modes etc. The space for these add upto 30 bytes. So each node in the tree, except for the root, will store about 30 bytes of attribute data, 3 node pointers (2 child pointers and a parent pointer), a balance factor and a *version-id* for a size of 47 bytes. Assuming a block size of 1KB, the number of nodes that can be stored in a block are 21 (1024/47). The Table I shows some of the performance measurements of a *version tree* on the basis of the number of versions stored with a block size of 1KB and with N_{th} equal to 8.

B. Block versioning

The block is the lowest level entity that can be accessed by a file system and so whenever a write requires a new version, a new block is allocated. But instead of keeping the new data in the new block, the older data can be placed in the newly allocated block and the new data placed in the old block. This avoids fragmentation of a file in its current version and improves read performance of the current version considerably. On the other hand, it increases the cost of every write as two writes have to be done now. Delayed writing helps mitigate this overhead. Every logical data block is associated with a *block list*, with each entry in the *block list* being of the form <*version_id, address*>. Consider the list for a block *b* with entries of the form $< V_1, A_1 >, < V_2, A_2 >$,......, $< V_n, A_n >$. In this list, the address of block *b* is A_1 for all the versions between V_1 and V_2, A_2 for all the versions between V_2 and V_3, and A_n for all the versions beyond V_n.

The list associated with each data block is stored at the last level of indirection of block pointers. So at the last level of

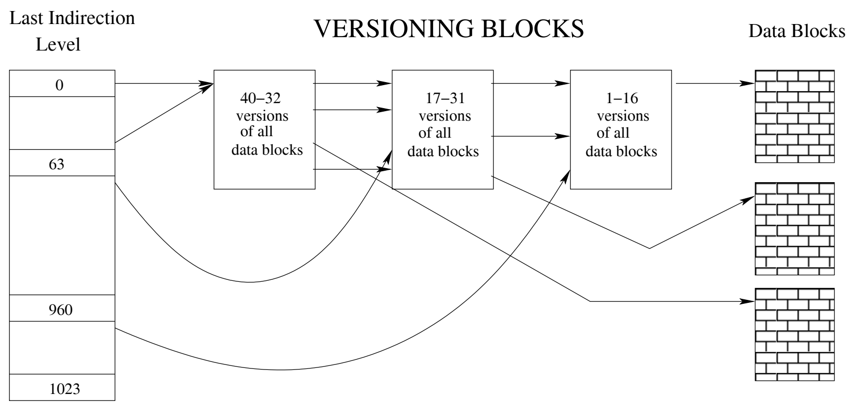
Last Indirection Level
VERSIONING BLOCKS
Data Blocks
0
63
960
1023
40–32 versions of all data blocks
17–31 versions of all data blocks
1–16 versions of all data blocks

Fig. 2. *Versioning blocks* with *block versioning factor* 64 and current version 40

indirection, the entries do not point to the actual data blocks but they point to another level of indirection blocks called *versioning blocks*. The *versioning blocks* are located here for two reasons, the first being faster access of the data block irrespective of the version it belongs to and the second is that the space required for the *versioning block* can be allocated dynamically. Initially when the file is created, the maximum number of versions that can be created for a file is assumed to be a base number of versions (which should be chosen such that a *block list* should fit in one disk block) and when the number of versions grow beyond this base, another *versioning block* is created. These two blocks are linked together with the first block containing information of recent versions and the second of the older versions. So suppose the base number of versions is 16. When a 17th version is created, a new block is allocated and this contains the <*version_id, address*> for the 17 th block and a pointer to the old block which contains information from versions 1 to 16. So now there will be one extra I/O for accessing (subject to a cache miss of course) versions 1 to 16, while version 17 and newer versions upto version 32 will need just one extra I/O. The number of block lists that can be stored in a single block is called the *block versioning factor* and is defined as follows

$$BVF = \frac{BS}{BV \times sizeof(Addr)} \tag{2}$$

where
BVF is the *block versioning factor*
BS is the block size in bytes
BV is the base number of versions initially assumed
Addr is the physical address of a data block
sizeof() returns the size of requested entity in bytes
Fig. 2 shows an example with the current version being number 40 and the base number of versions being 16.

C. Time Analysis

In versioning file systems, the accesses are of two categories - older version and current version. The Table I describes the I/Os required to access attributes of an older version from the *version tree*. To create an entry in the tree a single I/O is required. The data block *reads* of the current version include the access of a single *versioning block* apart from the inode and indirection blocks if required. But in case of older version accesses, there are two cases

1) ***Number of versions are below the assumed base versions*** A single *versioning block* is present and so one extra I/O is required.
2) ***Number of versions are beyond the assumed base versions*** A chain of *versioning blocks* is present in this case. Suppose v is the required version, cv is the current version, bv is the base versions initially assumed to be maximum.
   Then the number of *versioning blocks* required to be accessed are

$$\left\lceil \frac{cv - bv}{bv - 1} \right\rceil + 1 \tag{3}$$

when $v \leq bv$ and

$$\left\lceil \frac{cv - bv}{bv - 1} \right\rceil + 1 - \left\lceil \frac{v - bv}{bv - 1} \right\rceil \tag{4}$$

when $v > bv$.

D. Space Analysis

The extra space required for the designed versioning file system includes *versioning blocks* and the *version tree* for each file. The Table I shows the space required for the *version tree* on the basis of the number of versions present. It is clear that the extra space required for storing the version tree of a file is negligible. The space required for *versioning blocks* is defined by the *block versioning factor* given in Equation 2. As the *block versioning factor* increases, the space required for *versioning blocks* decrease, when the number of versions are below the base number of versions. Figure 3 shows the relation between file size and the space for *versioning blocks*. The overhead is more for small files, and it increases with block size. But as the file size increases, the extra space required is less as a percentage of the file size and the block size has a negligible effect. Even with an increase in the base versions stored, the same pattern is followed.

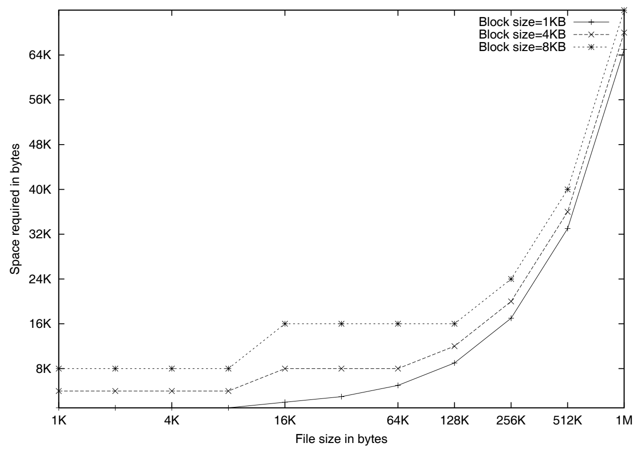

Fig. 3. Space required for *versioning blocks* to store a maximum of 16 versions and base versions as 16

III. Trace studies

The traces from the University of California, Berkeley ([13]) was used to evaluate the performance of the designed file system. The aim was to study the extra I/Os that would accrue as a result of the presence of multiple versions of a file. As far as the data blocks are considered, there will be no difference in the number of I/Os due to reads. For writes, every actual write to a data block that results in a new version (one write per open-close interval of a file) will result in one extra write. While it will be instructive to assess what the impact of this would be on actual data, the nature of the traces available prevented us from conducting this study. In the worst case, write overhead will double, but given the nature of actual accesses to files, we suspect the overhead will be significantly lower. So the analysis was restricted to the study of the overhead due to the I/Os incurred on the metadata. A metadata buffer cache that holds the inode, indirect blocks and the *versioning blocks*, was simulated and run on the traces. The following sections explain the environment of the traces and the results of the trace runs on the file system.

A. Trace environment

The trace files used represent three kinds of environments - Instructional(INS), Research(RES) and Web(WEB) workloads. All the environments used Hewlett-Packard series 700 workstations running HP-UX 9.05 with a 64 MB memory. The INS environment contains 20 machines located in laboratories of undergraduate classes. The RES workloads are obtained from a group of 13 machines used by the faculty, graduate students and the staff of a research group project. The WEB workloads are taken from the web server of an online library project at the University of California. Even though the traces do not represent the main application we have in mind, the experiments have given a level of confidence on the soundness of the proposal and it has also indicated that the scheme can be used for more general environments. The trace contained certain records that had inconsistent information (and these records resulted in the creation of a very large number of versions). While it was not clear what was the reason behind these inconsistent records, it was decided to ignore such records in the interest of obtaining meaningful input data.

B. Results of Trace runs

The LRU replacement strategy is used for the buffer cache along with a deferred write. The cache misses result in disk I/Os and they decrease as the buffer sizes increases. The Figures 4, 5 and 6 show the performance of the caches for current version accesses of INS, RES and WEB workloads respectively with a block size of 4KB and a delay threshold (for deferred writes) of 5 seconds.

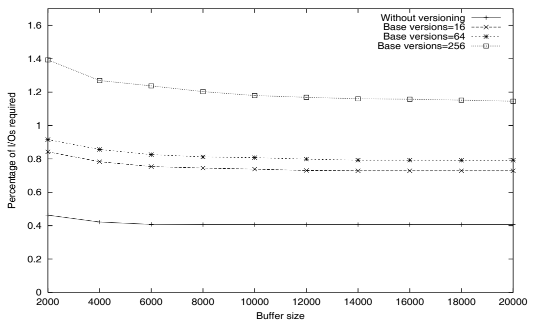

Fig. 4. Performance of buffer cache on INS workload traces

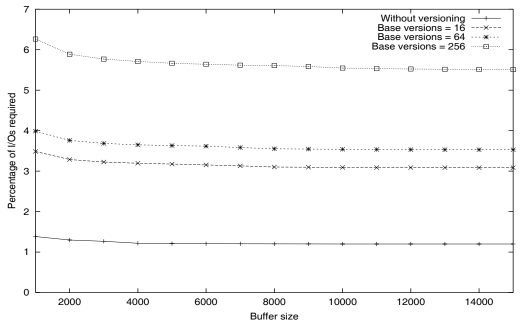

Fig. 5. Performance of buffer cache on RES workload traces

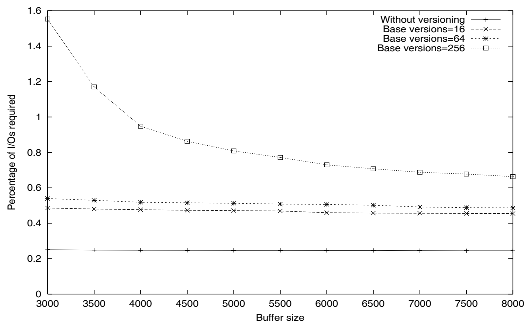

Fig. 6. Performance of buffer cache on WEB workload traces

The graphs show the percentage of reads and writes to metadata that actually result in I/O operations, for a normal file, and for a multi version file. New versions are created as per the conditions given earlier, based on the trace data. For the first workload (the instructional workload), when no versions are present, the percentage of actual I/Os vary from 0.37% to 0.31%, depending on the size of the buffer cache. The "knee" of the curve is at a buffer cache size of 6000 block (24 MB). Similarly, when versioning is present, the percentage of actual I/O varies between 0.48% and 0.38% with a "knee" at 40 MB of buffer size. These results, and the results of the other workloads clearly show that the overhead in terms of I/O, is negligible due to the presence of multiple versions of a file. This is a strong indication that the schemes proposed in this paper are sound.

IV. Conclusion

The design of a versioning file system has been presented. The emphasis has been on designing a system that handles multiple versions of documents in an office environment. However, the design can be used in a general purpose file system too. Simulation of the performance of the scheme on trace data that is from a development environment (and so in some sense a "general" environment' while at the same time sharing the characteristic of reads predominating over writes, which is likely in an office environment), shows that the overheads in terms of extra I/Os due to the presence of multiple versions of a file, is very low and is acceptable. The feasibility of the scheme has therefore been demonstrated.

One of the key features of the design is the "in-place" writing of the new version with the older version copied to another location. While increasing the write overhead, the strategy preserves the contiguity of data of the current version resulting in efficient read performance. Different techniques have been used for the handling of the attributes of a file and of the pointers to data. A balanced binary tree scheme is proposed for the former, while for the latter, an added level of indirection in address translation is introduced. The space and time overheads due to this grows gracefully with the increase in the number of versions, with no penalty for accessing the current version as the number of versions are increased.

The file system needs to be implemented and experience gained in order to fully understand the system. If there is a need to support multiple versions of documents while only storing the changes incrementally, changes in the way space is allocated to files will have to be made. In a traditional file system using a standard word processor, any insertion of material into a file containing a document will result in a re-write of the file in a new area. Multiple versions will then mean multiple versions of the entire file. A scheme with a level of indirection in addressing the contents of files may be required. There will be a logical address space which will not be linear and into which new address sub-spaces can be added dynamically. The ideas developed in ([3]) needs to be further explored and incorporated into file systems to realise the efficient implementation of multi-version file systems. Another direction that is being explored is to try and predict that a new version is created when a new file is created and the older one deleted. Rather than keep two copies of the file (the deleted and the newly created), the attempt will be to recognise that these two files are different versions of the same file and to then keep multiple versions of only the portions that are different. This way, versioning support can be provided without requiring to change the way existing software such as word processors handle new versions of documents.

References

[1] Craig A.N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of 2nd USENIX Conference on File and Storage Technologies*, pages 43–58, March 2003.

[2] D. Grune, B. Berliner, and J. Polk. Concurrent versions system. http://www.cvshome.org/.

[3] Smriti Sinha and Gautam Barua. Page Cube: A Model for Storage and Retrieval of Documents Relevant to a Document Production Workflow in an Office. *Proceedings of the DASFAA 2001*, Hongkong, April 2001.

[4] Steve Best, Dave Kleikamp and Dave Blaschke. IBM's Journaled File System. http://oss.software.ibm.com/developerworks/opensource/jfs.

[5] Stephen Tweedie. Ext3 Journaling Filesystem, ftp://frp.kernel.org/pub/linux/kernel/people/sct/ext3.

[6] M. Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System *ACM Transactions on Computer Systems*, 10(1):26-52, February 1992.

[7] B. Becker, S. Gschwind, P. Widmayer T. Ohler, and B. Seeger. An asymptotically optimal multiversion b-tree. *Very Large Data bases*, 5(4):264–275, 1996.

[8] D. S. Santry, M. J. Feeley, N. C. Hutchinson, R. W. Carton, J. Ofir, and A. C. Veitch. Deciding When to Forget in the Elephant File System. *ACM Symposium on Operating System Principles. Published as Operating Systems Review*, 33(5):110–123, 1999.

[9] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of 3rd USENIX Conference on File and Storage Technologies*, pages 115–128, March 2004.

[10] Zachary Peterson, R. Burns. Ext3cow: A Time-Shifting File System for Regulatory Compliance. In *ACM Transactions on Storage*, 1(2):190-212, May 2005.

[11] J. MacDonald, P. N. Hilfinger, and L. Semenzato. The Project Revision Control System. In *Proceedings of European Conference on Object-Oriented Programming, Springer-Verlag*, pages 33–45, 1998.

[12] Marshall K. Mckusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for Unix. *ACM Transactions on Computer Systems*, 2(3):181-197, August 1984.

[13] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of 2000 USENIX Annual Technical Conference*, June 2000.