Online backup and versioning in log-structured file systems

Ravi Tandon*
Student Author*
Depatment of Computer Science and Engineering
Indian Institute of Technology Guwahati
Email: r.tandon@alumni.iitg.ernet.in

***Abstract*—With the advent of large and fast storage devices (eg. SSDs, Flash Disks) file system sizes have grown beyond bounds. Large volume of dynamically changing data makes it difficult to store data on the fly. Consistent backup ensuring integrity and correctness of data is of utmost important. Offline backup schemes result in heavy losses due to large down time. Online backup schemes offer weak consistency guarantees. Transactional file systems offer strong guarantees; however, they hinder the normal functioning of user applications. This results in transaction aborts and suboptimal performance. This work proposes an online backup approach that circumvents user transaction aborts due to the backup process in a transactional file system. Conventional serialization approaches ensure consistency with transaction aborts. This work defines a concept "Conflict Dependence" that helps identify conflicts within backup and user transactions. An efficient implementation within a log-structured file system (LFS) is proposed. It does not require aborts because of the backup process. The backup scheme extends naturally to a versioning in a LFS.**

***Index Terms*—Online backup, Log-Strucuted File Systems, Consistency**

I. Introduction

Software bugs, system crashes, user application failures, etc. are some of the reasons due to which file systems may lose essential data and become inconsistent. Therefore, it becomes imperative for large organizations to keep a keep a reliable image of file system data. The process of storing dependable versions of file system data is referred to as "backup". Moreover, system administrators require systems to be up all the time. As disk sizes continue to grow at rates of 50% per year for the next decade file system sizes are expected to grow to multiple terabytes [1]. Disk and tape read speeds grow only at 20%. Hence, offline backups will become even more expensive. Therefore, active consistent backup becomes a primary requirement for large businesses.

Hitherto existing techniques such as dump [2], tar [3], cpio [4] etc. build a consistent image of file system only in an offline mode. These techniques employ a two phase backup approach. These are *scan* and *dump*. These approaches do not guarantee reliability of data backup (in online mode) primarily because the backup process does not serialize with other processes. Movement of files, deletion of directories, etc. are some of the common pitfalls of an online backup scheme. If any of the files is moved during the backup process, then depending on the state of the file system, the backup process may not encounter the moved file or may read that file more than once resulting in an inconsistent image of the file system. For eg. Consider two directories A and B and a file C. The file is moved from directory B to A. If the backup process first copies the directory A and then B then it might not copy file C. This can lead to inconsistency in the data that has been backed up. Backup programs need to be careful of special signals such as SIGSEGV, when files get truncated, else they might backup stale data which may even cause security breach [5]. Other file modifications such as file creation, deletion, compression etc. may cause inconsistencies.

This work proposes an online versioning file system design for log-structured file systems. Log-structured file systems [6] store data in the form of a continuous log within the file system. Copy-on-write semantics of a log-structured file system extends naturally to versioning in file systems [7]. This work, OBVLFS (Online backup and versioning in log- structured file systems), proposes a technique that ensures efficient serialization of an active backup process with user transactions. It assumes the existence of a transactional file system. The backup algorithm ensures that user transactions do not abort due to serialization with the backup process through the identification of *conflict dependent transaction.* Transactions which are inconsistent with the current backup are identified by the transactional semantics and metadata bookkeeping. Any changes made by them are not copied to the backup storage. Backup and user transactions can operate concurrently on a file. Each file inode stores various versions in the form of snapshots. Backup and user transaction operate on different copies and hence remain isolated even for a single file. Such a scheme, therefore, avoids the overhead of restarting either the backup process or the user transaction. This results in performance improvement over traditional backup approaches.

The rest of the work is organized as follows: Section II focuses on traditional backup schemes, categorization of backup processes and their drawbacks. Section III describes the system model, requirements of a backup scheme, implementation details, proposed backup protocol and contributions of this work. Section IV concludes the work with ideas on future implementation and scope of the project.

II. Related Work

Multi-process applications require a large number of concurrent transactions to proceed in a consistent manner. Supporting transactions ubiquitously thus becomes an inherent requirement of modern day systems [8]. Traditional backup schemes such as point-in-time copy [9] and Dell/EMC's SnapView [10] have focused predominantly on the reduction of time required for the backup to complete. Consistency is guaranteed only when the backup process runs in an offline mode. This incurs losses due to system down time. Hence, online backup schemes were developed to improve system performance.

Online backup schemes, predominantly, use three approaches to maintain consistency and integrity of data viz. *locking, detection of file movement* and *copy-on-write* [5]. Locking prevents modification of files, directories during the period they are being backed up. However, a strict locking scheme reduces concurrency and thus degrades system performance. Detection of file movement is based on the comparison of file modification time before and after backup. However, it requires post backup consistency checks. These checks require overhead of maintaining a database of files, and may not be very accurate. Copy-on-write is a scheme that makes shadow copies of data modified. The previous version remains available for backup while the current version of data is being modified. Point-in-time copying [9], [10] provides copy of large amounts of data in a very small period of time. *Split-mirror, changed-block* and *concurrent backup* are three different classes of implementation of point-in-time copy of file system data. Split-mirror copies the data to be backed up before; changed-block is similar to copy-on-write, where blocks are written to a different location when they are modified. Concurrent backup schemes monitor data that has been and physically copies the data in background. Split-mirror requires planning in advance, duplication of data; changed-block and concurrent backup do not provide enough guarantees of consistency across different files. File system backup techniques include read-only backup of the current file system data. This data is copied to a backup tape. Andrew file system [11], Petal [12], Spiralog [13] etc. snapshot file system state before copying the data. Spiralog [13] is a file based, incremental backup scheme. Data is stored in the form of structures called savesets on the disk. Savesets are then physically copied to backup tapes in the form of savesnaps. Lack of inter-volume data consistency and the absence of support for consistent backup in the event of collision of applications with the backup are some of the drawbacks of Spiralog LFS. Any application that has to write data to the backup has to commit data separately to the snapshot. This requires backup specific implementation within each application. Dell's PowerEdge servers use Volume Shadow Copy Service (VSS) [14] which reduces time and complexity of backup through a snapshot framework. Another approach of data backup is disk shadowing [15], which saves data simultaneously to two disks. Extra hardware cost is a major drawback of such a scheme.

Other schemes employ continuous backup such as in unitree. Unitree [16] employs a continuous backup scheme, where files are backed up within a threshold time after they have been modified. The transactional semantics are weakened by regular purging of files from caches to disk. Legato Networker [17] employs application level consistency semantics. These consistency semantics are not generic enough for extensive for different applications. Other file systems such as IBM's ADSM [18] provide different levels of consistency during backup. It provides four modes viz. static, shared static, shared dynamic, dynamic backup mode. The static mode allows backup of only files to which no modification is taking place whereas the dynamic mode cannot guarantee consistency.

Calton Pu [19] discusses an online backup approach that guarantees complete consistency for database entities. Each database entity is marked either white or black depending on whether the backup process has read it or not. Any transaction that writes on entities of two different colors (called a "gray" transaction) is aborted as it cannot be serialized with the backup process. The scheme is limited only to databases and restart of "gray" transactions consumes system resources and is inefficient. Lipika Deka et.al. [20] propose a concurrency control protocol *mutual serializability* for backup transaction. A backup is consistent as long as the backup transaction is mutually serializable with all other transactions. One of the major drawbacks of this approach is that transactions which read from already backed up files and subsequently write to non-backed up files are aborted for consistency. This work proposes an approach that avoids restart of any transaction by identifying a conflict dependent set (refer to subsection III-B) of transactions and by storing the previous versions of file's data in the log.

III. System Model

This work considers a transactional model of file system. Each transaction consists of a set of actions. Each transaction starts with txn_beg() call and ends with a txn_end() system call. File system transactions consist of read and write accesses to files. The backup process reads data from the file system in cycles. Each cycle forms a new version of the file system data. Each cycle constitutes a single transaction. The backup process performs only read operations on all the objects of the file system. Traditional online backup schemes abort user transactions on a conflict with the backup transaction [20], when the user transaction fails to serialize with the backup transaction. User transaction aborts result in poor performance of applications. This scheme avoids these aborts by making use of copy-on-write nature of a log-structured file system. A log-structured file system has a shadow copy of each page being modified. This shadow copy is used by the backup process during the backup thus avoiding transaction aborts and making the system much more performant.

A. Backup Requirements

The requirements of a backup process are as follows:

1) The backup process must cause the least amount of interference possible to user applications.
2) The backup process must use a small fraction of system's resources.
3) The backup copy of file system must be consistent.

B. Terms and Definitions

This subsection defines terms and definitions which are essential to understand the backup protocol.

- *Post Backup Data* - Data written to a file which has already been backed up in the current cycle is called post backup data. This data will be backed up in the next cycle after the cycle in which the transaction that has written/modified it finishes.
- *Conflict Dependency* - Conflict dependency is a relationship between two transactions (T_A, T_B), where T_A writes post backup data to a file F and T_B reads it. A transaction (T_C) that writes post backup data to a file has a reflexive relationship (i.e. it is conflict dependent on itself).
- *Conflict Depender Transaction* - A transaction (T_B) that reads post backup data is a conflict depender transaction. It depends on the transaction which has written it.
- *Conflict Dependee Transaction* - A transaction (T_C) that has written/modified post backup data is called a conflict dependee transaction. The transaction is a also a conflict depender, where the conflict dependency relationship is reflexive.
- *Conflict Dependent Transaction Set* - The set of all transactions which are either conflict dependees or conflict dependers constitute the conflict dependent transaction set. These transactions are also called conflict dependent transactions.
- *Primary Conflict Dependent Transaction* - A transaction (T_A) that writes/modifies post backup data is called a primary conflict dependent transaction.

Eg. 1 Suppose, $R_1(A)$ represents an action, where R denotes a read operation, A is a file system object and 1 denotes transaction identifier. Transaction identifier B denotes a backup transaction. Consider a schedule as follows: $R_B(A)$, $W_1(A)$, $R_2(A)$. Transaction 2 is a conflict depender transaction as it has read data written by transaction 1 which will not be backed up in the current backup cycle. Transaction 1 is a primary conflict dependee transaction.

C. Implementation Details

This subsection outlines the implementation details of the backup protocol along with the necessary data strucutres. Each file's inode contains a version number that identifies its backup version. The backup version of a file is the version of the last backup cycle that copied it. The file system maintains a global structure that identifies the version of the current backup cycle. These two structues are used by a transaction to judge whether it is a conflict dependent transaction. A file whose backup version identifier is lesser than the current backup cycle's version has not been backed up in the current cycle. Whenever a transaction modifies a file that has already been backed up, it identifies itself as a primary conflict dependent transaction. The file system maintains a global list of all the transactions. Each transaction structure stores the following information:

- transaction identifier - It uniquely identifies each transaction.
- status bit - It identifies whether a transaction is active or finished.
- conflict status bit - It identifies whether a transaction is conflict dependent or not.
- list of modified file structures - Each transaction stores a list of all the files that it has modified. Along with the file inode number, the backup version of the file at the time of modification is also stored.

After the completion of a transaction if the transaction remains a non-conflicting transaction then it updates the inodes of all the files that it had modified. The file system maintains two maps. They are also called dependency maps. They are:

1) Depender-dependee map - It maps the set of dependee transactions that each depender transaction depends upon. This map is used to identify whether a transaction is conflict dependent or not. Every transaction whose dependee set is empty becomes a non-conflicting transaction.
2) Dependee-depender map - It maps every dependee transaction to the set of transactions that depend upon it. Whenever a transaction becomes non-conflicting, its entry is removed from this map. The transaction is also removed from the dependee set of each of its depender transactions.

When the backup process completes, all the transactions that have completed become non-conflicting transactions (refer to subsection III-E). The entry corresponding to such transactions is removed from the dependee - depender map. Each depender transaction removes the dependee from its record in the depender-dependee map. After, this cleanup process, if the dependee list for a depender transaction is empty, it becomes a non-conflicting transaction. This gives rise to a cascade effect.

In traditional log-structured file systems each inode has a tnode tree that maps logical pages to physical on-disk pages. This is stored either on disk [6] or built on the fly (for Flash File Systems [21]). Our design considers the latter approach. In this design, each node within this tree has a list of pointers to on-disk pages. Our design provides a configurable option of allowing as many versions of the file system as required. This is done by providing a list of pointers (backup list) to backup copies on disk. Each backup pointer corresponds to a particular backup version and provides different snapshots of file over time. This model provides the flexibility to view changes over time for a particular page. This feature is termed as time travel [22] for file systems. Each entry in the tnode tree has at least three different pointers in our scheme. They are:

1) Pointer to current copy - This is the latest copy of a particular page. It may not be consistent with the current backup cycle.
2) Pointer to current consistent copy - This the latest copy of a particular page. It is consistent with the current backup cycle. The backup transaction copies this image to the backup disk.
3) List of pointers to backup copies - This is a list that stores pointers to previous copies of a page. This represents snapshots of the page at for various backup cycles.

Besides, a snapshot map is maintained. It essentially stores for each snapshot version, a list of pointers to the on-disk inode structures.

D. Backup Protocol

1) Backup Protocol Setup: The file system is transactional in nature [20]. Each transaction is allocated a unique identifier by the file system. They are incremental in nature. Transactions have to be serialized with the backup process for a consistent backup copy. User transactions implement strict two phase locking [23]. Implementing strict two phase locking for backup process would imply stopping all the user transactions for a certain period of time. Deka et.al. [20] propose a novel concurrency control protocol, *Mutual Serializability (MS)*, that serializes the backup process in a consistent manner. However, it requires user transaction aborts. Our protocol improves MS and allows concurrent backup of a file as it is being modified. The hypothesis is that backing up writes of any conflict dependent transaction leads to inconsistency in the backup copy. In the preceding example (example 1, refer subsection III-B), any writes of transaction 1 or 2 should not be backed up because of the following reasons:

1) Transaction 1 has written post backup data to file A. This data will be backed up in the next backup cycle. Hence, transaction A is a primary conflict dependent transaction and any modifications made by it are should not be saved to the backup disk.
2) Transaction 2 has read post backup data. Any modifications made by transaction 2 should not be copied back to the backup disk, as they might depend on post backup data.

The backup process runs as a separate process in the background. Modifications made by the conflicting set of transactions are not copied to the backup disk. The backup process reads every file and backs up those copies of pages of a file that have been written to the disk by non-conflicting transactions. This makes identification of conflicting set of transactions imperative. Before reading a file, the backup process acquires a backup lock. Only those transactions that have a backup lock can modify the pointers to the current consistent copies of pages of a file.

E. Identification of Conflict Dependent Transaction Set

The backup protocol identifies the set of conflict dependent transactions. The set of conflicting transactions includes:

- All those transactions that have not finished yet belong to the set of conflicting transactions. When a transaction starts, its conflict-status bit is set to 1 (denoting it is conflict dependent). The rationale behind this is that this transaction might write post backup data subsequently. The data will not be backed up leading to inconsistency.
- Any transaction that writes to a file, checks the backup version of the file and compares it with the current backup cycle's version number. If they are equal, the file is backed up already. The transaction marks itself primary conflict dependent transaction and necessary updates are made to the dependency maps.
- Any transaction that reads from a file identifies the transaction that had previously modified that part of the file. In a log-strucutred file system each page has metadata information about the transaction that has written it to the disk. If a transaction reads post backup data, it identifies itself as a depender transaction and the transaction that had modified the page as a dependee transaction. It also updates the dependency maps.

When a transaction modifies a file, it updates the pointer to the current copy of each individual tnode of file's tnode tree. It also stores the file's inode number and current backup version to the list of modified file structures. When a transaction finishes, it may abort or commit. On an abort all its entries from conflict dependence maps are removed. If the transaction commits, then

- if the transaction has read post back up data, then it remains a conflict dependent transaction.
- if the transaction has not read any post back up, it checks whether it has written any post backup data. The transaction acquires a backup lock over each file that it has modified. A backup lock is an exclusive lock that does not allow a file to be backed up at the same time. If none of the files have been backed up since the time the transaction modified them, the transaction becomes a non-conflicting and

updates the tnode pointers to consistent copies of each file within their respective inodes. The transaction removes its entries from each of the dependency maps.

F. Transaction Collision Prevention

Due to copy-on-write property of log-structured file systems, for each page that is modified a new copy is made to the disk. Each page contains metadata information about the transaction which modified it. The tnode tree (a tree that maps each logical page to physical page on disk) within the inode stores information about the last consistent page and the new page on disk. The backup process thus reads from the last consistent page on disk. The modifications are made to the new page on disk. Hence, the protocol prevents a collision between backup process and the current transaction ensuring consistency, efficiency (by avoiding transaction restart) and a higher level of concurrency to the file system.

G. Contribution

Subsection III-A lays down three basic requirements of a backup process. This subsection justifies how our protocol satisfies these three requirements.

1) In our scheme, user applications and the backup process can operate on a file simultaneously. Copy-on-write and support for multiple versions of pointers to a page within the tnode tree improve the concurrency level of file system operations.
2) The backup process, makes copies of the file's inodes. Only the metadata needs to be copied. Snapshot based copy scheme requires a small fraction of disk bandwidth for backup. System resources are thud highly available leading to optimal performance. Avoidance of user transaction aborts improves throughput and system usage.
3) Identification of conflicting set of transactions (refer subsection III-E) ensures serialization of backup process and user transaction. This guarantees a consistent backup image.

IV. Conclusion

This work presents an online backup scheme which avoids any transaction aborts. Consistency of backup copy of file system is maintained by deploying a transactional file system, semantically grouping transactions into a set of conflict dependent and non-conflicting transactions. Traditional online backup schemes either guarantee weak consistency or require user transaction aborts leading to poor performance of user applications. This work makes use of copy-on-write nature of a log-structured file system and proposes an online backup scheme that allows backup and user transaction to proceed concurrently and at the same time ensures consistency. This work provides a framework for constructing snapshots and storing them as versions of the file system.

The work will be formalized to prove consistency requirements of file system. A backup algorithm will be implemented to measure its performance with the existing schemes. Metadata compression will be looked into while storing backup versions of file system. Incremental versioning will be incorporated to the backup algorithm to eliminate storage of redundant data for successive versions of file system.

References

[1] A. Chervenak, V. Vellanki, and Z. Kurmas, "Protecting file systems: A survey of backup techniques," in *Proceedings Joint NASA and IEEE Mass Storage Conference*, vol. 3, 1998.

[2] *dump*, linux man page.

[3] *tar*, linux man page.

[4] *cpio*, linux man page.

[5] S. Shumway, "Issues in on-line backup," in *Proceedings of the Fifth Large Installation Systems Administration Conference*, 1991, pp. 81–87.

[6] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 26–52, Feb. 1992. [Online]. Available: http://doi.acm.org/10.1145/146941.146943

[7] J. Ousterhout and F. Douglis, "Beating the I/O bottleneck: a case for log-structured file systems," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 1, pp. 11–28, 1989.

[8] D. Porter, I. Roy, A. Matsuoka, and E. Witchel, *Operating system transactions.* Computer Science Department, University of Texas at Austin, 2008.

[9] A. Azagury, M. Factor, J. Satran, and W. Micka, "Point-in-time copy: Yesterday, today and tomorrow," in *NASA CONFERENCE PUBLICATION.* NASA; 1998, 2002, pp. 259–270.

[10] R. Hou, S. Feibus, and P. Young, "Data Replication and Recovery with Dell/EMC SnapView 2.0 and MirrorView," 2003.

[11] J. Howard and C.-M. U. I. T. Center, *An overview of the andrew file system.* Carnegie Mellon University, Information Technology Center, 1988.

[12] E. Lee and C. Thekkath, "Petal: Distributed virtual disks," *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5, pp. 84–92, 1996.

[13] R. Green, A. Baird, and J. Davies, "Designing a fast, on-line backup system for a log-structured file system," *Digital Technical Journal*, vol. 8, pp. 32–45, 1996.

[14] A. Sankaran, K. Guinn, and D. Nguyen, "Volume shadow copy service," *Power Solutions, March*, 2004.

[15] J. Gray and D. Bitton, "Disk shadowing," in *VLDB*, vol. 88, 1988, pp. 331–338.

[16] "UniTree Mass Storage System." [Online]. Available: http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/UniTree/

[17] "EMC NetWorker Unified Backup and Recovery Software." [Online]. Available: http://www.emc.com/domains/legato/index.htm

[18] R. Heyt, M. Landzettel, R. Leins, F. Ramozzi, M. Standau, and D. Talbot, *Tivoli Storage Manager Version 3.7: Technical Guide.* IBM Corporation, 1999.

[19] C. Pu, "On-the-fly, incremental, consistent reading of entire databases," *Algorithmica*, vol. 1, no. 1, pp. 271–287, 1986.

[20] L. Deka and G. Barua, "On-line consistent backup in transactional file systems," in *Proceedings of the first ACM asia-pacific workshop on Workshop on systems.* ACM, 2010, pp. 37–42.

[21] A. One, "YAFFS: Yet another Flash file system," 2002. [Online]. Available: http://www.dubeiko.com/development/FileSystems/YAFFS/HowYaffsWorks.pdf

[22] M. Rosenblum, "The Design and Implementation of a Log-structured File System," Ph.D. dissertation, EECS Department, University of California, Berkeley, Jun 1992. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1992/6267.html

[23] J. Gray and A. Reuter, *Transaction processing: concepts and techniques.* Morgan Kaufmann, 1993.