# On-line Consistent Backup in Transactional File Systems

Lipika Deka and Gautam Barua
Department of Computer Science and Engineering
Indian Institute of Technology Guwahati
Guwahati 781039,India
l.deka@iitg.ernet.in, gb@iitg.ernet.in

## ABSTRACT

A consistent backup, preserving data integrity across files in a file system, is of utmost importance for the purpose of correctness and minimizing system downtime during the process of data recovery. With the present day demand for continuous access to data, backup has to be taken of an active file system, putting the consistency of the backup copy at risk.

We propose a scheme to take a consistent backup of an active file system assuming that the file system supports transactions. We take into consideration that file operations include, besides reading and writing of files, directory operations, file descriptor operations and operations such as append, truncate, etc. We are putting our scheme into a formal framework to prove its correctness and we have devised algorithms for implementing the scheme. We intend to simulate the algorithms and run experiments to help tune the algorithms.

## Categories and Subject Descriptors

E.5 [**Data**]: Files—*Backup/recovery*

## General Terms

Algorithm, Theory

## Keywords

File Systems, Online Backup, Consistency, Transactions

## 1. INTRODUCTION

*Backup* is the process of creating a copy of the file system data called the *backup-copy* to ensure against its accidental loss. Traditional backup approaches like *tar*([22]) and *dump*([21]) perform on a unmounted, quiescent file system. But, with file system sizes touching peta bytes, use of traditional backup mechanisms means hours or even days of system downtime. A study in the year 2000 projected the cost of an hour's computer downtime at $2.8 million for just the energy industry sector, proving traditional backup techniques to be infeasible in today's 24/7 computing scenario([24]). Such realizations lead industry and academia to bring forth a host of *on-line backup* solutions which performs backup on active file system.

Although existing on-line backup solutions promises high system availability but not much has been mentioned on preserving backup-copy data integrity . Arbitrarily taken on-line backup may destroy data integrity in the backup-copy as detailed in [18] and highlighted by the following example: local user and group accounts on Linux are stored across three files that need to be mutually consistent: /etc/passwd, /etc/shadow, and /etc/group. Arbitrary interleaving of the on-line backup program with the related updates to the three files may lead the backup-copy to have an updated version of /etc/group and versions of /etc/passwd and /etc/shadow before they were updated([16]). Recovery from an inconsistent backup-copy may sometime be more dangerous than data loss, as such errors may go undetected. We focus on on-line backup that preserves file system data integrity and term it *on-line consistent backup*.

Our broad objective is to understand the problem and solutions of *on-line consistent backup* through its theoretical and practical treatment. The results are obtained assuming a transactional file system as we need a mode for specifying consistency information.

In this paper, we informally show that a consistent on-line backup-copy can be created by employing a specialized concurrency control mechanism called *mutual serializability*(MS). In this mechanism, a consistent backup copy is obtained by ensuring that the backup program is separately serializable with each user transaction simultaneously. In the presence of many tried and tested protocols, the need for a specialized concurrency control protocol arises because the backup transaction reads the entire file system. The use of standard concurrency control mechanisms may result in the backup transaction getting aborted and this will adversely impact performance and backup time.

The remainder of the paper is structured as follows: Section 2 presents a review of the existing on-line backup solutions in both the file system and database area, building up the motivation behind our adopted system model. Section 3 presents our *on-line consistent backup* protocol. We then introduce on-line backup algorithms and illustrate the implementation issues and solutions arising from active modification of the file system name space while the backup copy is being generated. We conclude in section 4 with a future


*APSys 2010,* August 30, 2010, New Delhi, India.

road map by discussing the analytical, simulation and experimental approaches.

## 2. RELATED WORK

In order for the on-line backup process to read a consistent state of the file system, there must be a way to specify the consistency requirements of the applications running on the file system. Current general purpose file system interfaces provide weak semantics for specifying consistency requirements. Hence, we see existing solutions([2, 3, 4, 8, 9, 10, 11, 14]) only achieving weaker levels of backup-copy consistency i.e. per file consistency or in very specific cases, application level consistency.

For example, many modern day file systems([8, 9, 11]) and disk storage subsystems([2, 3]) come with a copy-on-write based snapshot facility that creates a point-in-time, read-only copy of the entire file system, which then acts as the source for on-line backup([2, 4, 3, 8, 11]). Another approach called the "split mirror" technique maintains a "mirror" of the primary file system, which is periodically "split" to act as the source for backup([2, 4, 14]). *Snapshots* are created and *mirrors* are "split" after buffers are flushed and a *file system consistency check* is conducted. But, file system consistency check is not capable of ensuring consistency across files.

Application level consistency has been achieved using *snapshot*([9]) and *mirroring*([14]) solutions by including the application to be a part of the *snapshot copy* or *mirror* "splitting" initiation process. Such methods are only possible for advanced applications like databases, having well developed consistency specifying mechanisms like transactions. A third approach termed *continuous data protection* maintains a backup of every change made to the data([4, 15]), but again this only manages to achieve per file consistency.

Still other existing on-line file system backup solutions operate by keeping track of open files and maintain consistency of groups of files that are possibly logically related by backing them up together. Such groups of files are identified by monitoring modification frequencies across open files([23]). Resorting to heuristics and ad hoc concurrency control mechanisms may result in a consistent backup copy only by accident.

Recent times have seen a phenomenal growth of *unstructured and semi-structured* data i.e. data stored in files, with files spread across machines, disks and directories and accessed concurrently by myriad of applications. With vital data stored in files, among many other management issues, there is an urgent need for efficient ways to specify and maintain the consistency of the data in the face of concurrent accesses. Recognizing this need, file systems supporting transactions have been proposed by a number of researchers([13, 16, 19, 20]), and it is very likely that *Transactional File Systems* will become the default in systems. We have therefore assumed that file systems support transactions as a mechanism to ensure consistency of concurrent operations.

Now, given a file system with transactions, the natural next step would be to borrow on-line database backup solutions. Today's *Hot Backup*([25]) solution and its ancestor, the *fuzzy dump*([6]) facility, work by first backing up a "live" database and then running the *redo log* offline on the backup copy to establish its consistency. Similar approaches for file systems will incur high performance cost(write latency) and is space inefficient as each entry in the file system log would have to record before and after images of possibly entire files. Hence, we need to explore different approaches for achieving an on-line consistent backup copy of a file system.

[17] suggested a scheme which considers the on-line backup of a database by a global read operation (we refer to it as a backup transaction) in the face of regular transactions active on the database. This scheme ensures that the backup transaction does not abort, and active transactions abort if they do not follow certain consistency rules. The scheme suggested was shown to be in error in [1] and the authors suggested a modified form of the algorithm to ensure a consistent backup. Every entity is coloured white to begin with. When the backup transaction reads an entity, its colour turns black. A normal transaction can write into entities that are all white or all black. A transaction that has written into an entity that is of one colour and then attempts to write into another entity of a different colour, has to abort. Transactions writing into white entities are ahead of the backup transaction in the equivalent serial schedule, while the transactions writing into black entities are after the backup transaction in the equivalent schedule. Each entity is also given a shade which changes from pure to off when a black transaction reads or writes the entity (whichever is allowed on it). Restrictions are placed on reading entities whose shades are off, by white transactions.

The scheme in [17] lays the foundation of our approach. But, the scheme was designed for a database and it is assumed that two phase locking is the concurrency control algorithm. Our approach described in succeeding sections gives a sounder theoretical basis for identifying conflicts and this makes our approach independent of any particular concurrency control algorithm. Due to space restriction this paper only provides a conceptual explanation of the theoretical basis of our approach, and formal correctness proofs have been omitted. Further, we consider a file system as opposed to a database system, which brings forth issues unique to a file system. For example, file systems cater to many more operations besides reads, writes, deletes and insertions of entities, such as rename, move, copy etc. Files in most popular file systems like ext2 are not accessed directly, but are accessed after performing a "path lookup" operation. File system backup involves backing up of the entire namespace information (directory contents) along with the data. This complicates online backup as a file system's namespace is constantly changing even as the backup process is traversing it. Moreover, a file system backup does not involve just the backup of data but also includes the backing up of each file's meta-data (e,g inodes of ext2).

## 3. SYSTEM MODEL

Assuming a file system that provides a common transactional interface to all applications running on it, the problem of creating a consistent backup copy of an active file system now fundamentally reduces to a concurrency control problem. But, the problem proves to be challenging as the backup program reads the entire file system, and this is further aggravated by the fact that the file system namespace may be actively changing even as the backup program is traversing it, thus posing the danger of complete omission of sections of the file system hierarchy from the backup copy([18]). This section outlines our concurrency control mechanism and discusses implementation issues and possible solutions for creating a consistent backup copy in the face of a constantly changing file system.

## 3.1 Consistent Backup Protocol

A file system transaction is defined to consist of one or more accesses to a file or set of files, with an *access* being either a *write* to or a *read* from a file([13]). To ensure file system data integrity in the presence of concurrent accesses, a concurrency control mechanism, say, strict two-phase locking([7]) is employed. It is well established that concurrency control mechanisms produce serializable *schedules* of concurrent transactions where *serializability* is the accepted notion of correctness for concurrent executions([7]).

The *backup transaction* consists of a sequence of file *read* operations. For an on-line backup transaction to read a consistent backup copy, it must be *serialized* with the concurrently executing user transactions. Now, the backup transaction is rather unique as it reads every file in the file system and treating it like any other transaction for the purpose of concurrency control leads to frequent inter transactional conflicts. Resolving these conflicts may mean either the gradual death of all concurrently executing user transactions as the backup transaction proceeds to lock all files in accordance with locking protocols or otherwise result in repeated abortion and restarting of the backup transaction. Hence, we are proposing a backup transaction specific concurrency control mechanism called *mutual serializability*(MS) that does not affect the overall performance efficiency. We first revisit two basic concepts in serializability theory before formally defining the core concept behind our proposed protocol. Two *schedules* are *equivalent*([7]) if the order of execution of the *conflicting operations* in the two schedules are identical. A pair of operations are said to be *conflicting operations*([7]) if they belong to different transactions, access the same file and one is an update operation.

***Mutually Serializable:*** A pair of transactions is said to be *mutually serializable* if and only if their concurrent execution *schedule* is equivalent to some *serial* schedule. In other words, the *backup transaction* is mutually serializable with a *user transaction* if and only if their schedule is *equivalent* to some *serial schedule* of the pair.

Given the definition of mutually serializable, we now describe the *mutual serializability* protocol. We assert that if all *access* operations of user transactions are considered to be update operations and all operations of the backup transaction are *read* operations, keeping the backup transaction mutually serializable with every concurrent user transaction will ensure that the backup transaction will be part of a serializable schedule. The serializable schedule involves user transactions which continue to use some concurrency control protocol and which differentiates between read and write *accesses* while resolving conflicts among themselves. The protocol can be formally stated as:

***Mutual Serializability***: Given a serializable schedule of user transactions and considering all file operations of user transaction to be update operations with respect to the backup transaction, the concurrent execution of the backup transaction and user transactions is serializable **if and only if** the backup transaction is mutually serializable with each concurrent user transaction.

The essence of the stated protocol is illustrated through the following simple examples. Consider the following interleaved execution sequence: $read_b(C)$, $read_1(A)$, $write_2(C)$, $write_1(B)$, $read_b(A)$, $read_b(B)$, $write_2(B)$

Where, subscripts 1, 2 and b stand for transaction 1, transaction 2 and the backup transaction respectively and A, B, C are files. We can see that the backup transaction is part of a serializable schedule and is mutually serializable to both transaction 1 and transaction 2 simultaneously.

Now consider the schedule: $read_b(A)$, $write_1(A)$, $read_2(A)$, $write_2(B)$, $read_b(B)$

We see that the backup transaction is mutually non serializable with transaction 2 resulting in a schedule which is not serializable. Notice that in both the schedules the user transactions are serialized among themselves.

Considering all file accesses by user transactions to be update operations while resolving conflicts with the backup transaction, does not lead to any loss of concurrency of running transactions as among these transactions no such restrictions are imposed. What this implies is that when the backup transaction is running, other, active transactions will have to “slow" down a little to accommodate the backup transaction. This is a small price to pay to obtain a consistent backup.

Moreover, as opposed to [12] which builds up directly on [17] and [1]to serialize read only transactions, we observe that read-only transactions do not effect the consistency of the backup-copy, hence in our approach they are identified at their initiation and do not require to enter into contention with the backup transaction at all. As read-only transactions consist of the bulk of operations in a file system([13]), this implies further that considering user operations to be update operations with respect to the backup transaction does not effect efficiency.

## 3.2 Implementation Issues

This section sketches our approach for implementing the concurrency control protocol proposed in the previous section. Without loss of generality, we assume that the user transactions use strict two-phase locking to serialize among themselves.

The basic idea is as follows: applications run as a sequence of atomic transactions and under normal circumstances when the backup program is not active, they simply use strict two-phase locking to ensure consistent operations. Now, once the backup program is activated, all other transactions are made aware of it by some triggering mechanism and they now need to serialize themselves with respect to the backup transaction, while continuing to *serialize* among themselves as before. We distinguish transactions as read-only and update transactions. Now, read-only transactions inherently do not conflict with the backup transaction. Hence, read-only transactions are identified at their initiation and do not require to use the concurrency control mechanism needed for serializing with the backup transaction.

Our approach reserves a bit called the *read* bit in each file descriptor (in the inode) to indicate whether the concerned file has been read or not by the backup program (this is the same as colouring the file, as proposed by [17]). If this bit is 0 it indicates that the file has as yet not been read by the backup transaction and 1 indicates that it has. This bit of all files is initialized to 0 before a backup program starts. But, initializing the *read bit* of the entire file system before every backup may not be very efficient and so a sequence number of the backup transaction may be used instead, where the sequence number of the present backup transaction is one greater then its immediate predecessor. For ease of discussion, we continue using the *read* bit. The

backup transaction traverses the file system namespace using traversal algorithms such as depth first traversal, reading files on its path to the backup-copy and as it reads each file it sets the *read* bit to 1.

A user transaction *serializes* with respect to the backup transaction by establishing with it a mutually serializable relationship using a bit called the *before-after* bit, reserved in each "live" user transaction's house keeping data structure. When a user transaction succeeds to lock its first file for access, it initializes the *before-after bit* to 1 if the file's *read bit* is 1 and to 0 otherwise. A 0 stored in the *before-after* bit means that the transaction's mutually serializable order is *before* the backup transaction and a 1 indicates that it is ordered *after* the backup transaction. On subsequent successful locking of a file, the user transaction verifies whether it continues to be mutually serializable with respect to the backup transaction by comparing the *read* bit of the file with its own *before-after* bit. The following table enumerates mutually serializable checking .

| *before-after bit* | *read bit* | mutually serializable |
|---|---|---|
| 1 | 1 | yes |
| 1 | 0 | no |
| 0 | 1 | no |
| 0 | 0 | yes |

If a mutually non serializable operation is detected then the conflict must be resolved before the execution of the user transaction can proceed. Now, mutually non serializable transactions have accessed or tried to access both *read*(1) and *unread*(0) files. Let $T_{mc}$ be the user transaction mutually non serializable with the backup transaction. One way of resolving the conflict is to *roll back* the backup transaction's "read" of the files marked 1 and presently accessed by $T_{mc}$. Rolling back the "reads" of files essentially means to mark them 0(*unread*) and to remove them from the backup-copy. The backup transaction will re-read them later. Unfortunately, although this solution does not hamper the execution of user transactions , it may lead to a cascading roll back of backup "reads" as roll backs may render previously consistent transactions to now be mutually non serializable. In the worst case scenario, the backup transaction has to be restarted.

Another method of handling the problem is to abort and restart $T_{mc}$ "hoping" the backup transaction completes reading the "unread" files accessed by $T_{mc}$. The solution seems quite attractive in a scenario where user transactions do not have a hard time constraint. But, if concurrency control mechanisms like a simple two-phase locking is used by user transactions, an abort may lead to cascading aborts.

A third solution for resolving conflicts is to pause $T_{mc}$ when possible. If the transaction has accessed a file already read by the backup transaction and it then needs to access another file which the backup transaction has not yet read, then the transaction could be made to wait and the backup transaction signalled to read this file immediately.

Thus, one of the conflict resolving techniques or a combination of techniques discussed above have to be employed depending on issues like the underlying concurrency control mechanism for user transactions and a transaction's *before-after bit* status, to ensure mutual serializability with the backup transaction.

There are many other files operations such as append, truncate, create, unlink, and operations that operate on the file's descriptor, such as ownership, access permissions etc. Further, there are a number of file types such as regular files, directories, special files, etc. (we are using Linux terminology and not defining these items,to conserve space). For most of them, we have shown that they can be treated as equivalent to an update of an ordinary file. Thus, for example, creating a file is equivalent to writing to the directory under which the file is to be created.

However, special treatment is required when dealing with directories. Consider two directories d1 and d2 with d2 having a child directory d3. Suppose the backup transaction (BT) reads d1, and then reads d2. Before it can read d3, another transaction TA moves d3 from under d2 to under d1. Transaction TA is mutually serializable with respect to BT with order as "after" and so it is allowed to proceed. But then d3 will not be in the backup as BT has already traversed its part of the file system namespace, and there will be a dangling reference to it in the backed up version of d2. One way of ensuring correctness is to ensure that all reads are done "bottom up" by BT. In this example, d3 will have to be read before d2 and the move of d3 from d2 to d1 by TA will be allowed only after BT has read d2 also. But, we need to specify conditions that are not dependent on an implementation and this is work that is in progress.

Overall performance during an on-line backup using our approach can be considerably improved if conflicts between user transactions and the backup transaction can be kept low. Most applications running on a system develop a pattern of file accesses. Storing the history of file accesses and then using these patterns to direct the backup program away from active regions effectively reduces contention. File system traces collected over a period of time is also likely to show groups of files that are accessed together and hence are possibly related. Backup of such groups may be taken together to improve performance.

Studies show that only about 1% of all files are used daily([5]). Thus even without using conflict reducing heuristics as discussed above, conflicts will be rare.

# 4. CONCLUSION AND FUTURE WORK

This paper argues that backups of file systems must ensure consistency and for this, a transactional file system is required. Current techniques of *hot backups* do not ensure consistent backups. Using standard concurrency control techniques for backup in transactional file systems can be inefficient as the backup process, considered as a transaction, has to access every file in the system. Aborting this transaction will be expensive, and other transactions may experience frequent aborts because of this transaction. We informally show that a simple restriction of considering every operation of normal transactions (be it a read or a write) to conflict with a read of the same file by the backup transaction results in a scheme in which ensuring that the backup transaction is mutually serializable with every other transaction results in a consistent backup copy. The backup transaction does not have to be aborted, and delaying a conflicting transaction in most cases resolves conflicts. An outline of an implementation has also been presented.

We intend to formalize our scheme and prove our assertions. We then intend to design an algorithm to implement the scheme. We will formally consider other operations on files such as truncate, delete, create, move, link, unlink etc. We will also handle operations on directories, including mov-

ing and renaming of directories. We will then analyze the performance of our algorithm by simulation with various access patterns of files and we will suggest and evaluate heuristics to improve performance.

# 5. REFERENCES


[1] P. Ammann, S. Jajodia, P. Mavuluri, *On-The-Fly Reading of Entire Databases*, In IEEE Transactions on Knowledge And Data Engineering,Vol. 7, No. 5, October 1995.

[2] A. Azagury, M.E. Factor, J. Satran, W. Micka, *Point-in-Time Copy: Yesterday, Today and Tomorrow*, In Proceeding of the IEEE/NASA Tenth Goddard Conference on Mass Storage Systems and Technologies, April 2002.

[3] C.Y. Chang, Y. Chu, R. Taylor, *Performance Analysis of Two Frozen Image Based Backup/Restore Methods*, In Proceedings of IEEE International Conference on Electro Information Technology, May 2005.

[4] A.L. Chervanak, V. Vellanki, Z. Kurmas, *Protecting the File System: A Survey of Backup Techniques*, In Proceeding of the Joint NASA and IEEE Mass Storage Conference, March 1998.

[5] T. Gibson, E.L. Miller, D.D.E. Long, *Long-term File Activity and Inter- Reference Patterns*, In CMG98 Proceedings. Computer Measurement Group, December 1998.

[6] J. Gray, *Notes on Data Base Operating System*, In Lecture Notes In Computer Science;Vol.60, Operating System,An Advanced Course, 1978.

[7] J. N. Gray, A. Reuter, *Transaction processing: Concepts and Techniques*, Morgan Kaufmann Publishers Inc., 1993

[8] R.J. Green, A.C. Baird, J.C. Davies, *Designing a fast, On-Line Backup System for a Log-Structured File System*, In Digital Technical Journal, *Vol. 8 No. 2* 1996.

[9] D. Hitz, J. Lau, M. Malcolm, *File System Design for a File Server Appliance*, In proceedings of the USENIX Technical Conference, San Francisco, CA, January 1994.

[10] N.C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, S. O'Malley, *Lovical vs. Physical File System Backup*, In Proceedings of the USENIX Technical Conference, San Francisco, CA, January 1994.

[11] J.E. Johnson, W.A. Laing, *Overview of the Spiralog File System*, Digital Technical Journal, Vol.8, No.2, 1996

[12] K. Lam, V.C.S. Lee, *On Consistent Reading of Entire Databases*, In IEEE Transaction on Knowledge and Data Engineering, Vol.18, No.4, April 2006

[13] B. Liskov, R. Rodrigues, *Transactional File Systems Can Be Fast*, In Proceedings of the ACM SIGOPS European Workshop, 2004.

[14] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, S. Owara, *SnapMirror: File System Based Asynchronous Mirroring for Disaster Recovery*, In Proceeding of the Conference on File and Storage Technologies,Monterey, California, January 2002.

[15] Z. Peterson, R. Burns, *Ext3cow: A Time-Shifting File System for Regulatory Compliance*, in ACM Transactions on Storage, May 2005.

[16] D.E.Porter, O.S. Hofmann, C.J. Rossbach, A. Benn, E. Witchel *Operating Systems Transactions*, in Proceedings of 22nd ACM Symposium on Operating Systems Principles, October 2009.

[17] C. Pu, *On-the-fly, incremental, consistent reading of entire databases*, Algorithimica 1, 3,271-287. 1986

[18] S. Shumway, *Issues in On-line Backup*, In USENIX Proceedings of the 5th Conference on Large Installation Systems Administration, September 1991.

[19] R.P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, C. P. Wright, *Enabling Transactional File Access via Lightweight Kernel Extensions*, In Proceedings of the 7th USENIX Conference on File and Storage Technologies, February 2009.

[20] S. Verma, *Transactional NTFS(TxF)*, http://msdn.microsoft.com/en-us/library/aa365456(VS.85).aspx, 2006.

[21] *Dump*. Unix System Man Pages

[22] *Tar* Unix System Man Pages.

[23] *Open File Manager: Preventing Data Loss During Backups Due to Open Files*, as White Paper, St. Bernard Software. November 2003.

[24] *Quantifying Performance Loss: IT Performance Eng. and Measurement Strategies*, as Report, META Group. 2000.

[25] *Oracle Backup and Recovery*, *http* : *//www.oracle.com/technology/deploy/availability/htdocs/BR*$_O$*verview.htm.*