Implementation and Evaluation of Consistent Online Backup in Transactional File Systems

Lipika Deka* and Gautam Barua**

* Indian Institute of Technology, Guwahati, India.
Now moved to Loughborough University, United Kingdom, l.deka@lboro.ac.uk, deka.lipika@gmail.com
** Indian Institute of Technology, Guwahati, India, gb@iitg.ernet.in.

A consistent backup which preserves 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 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.

In order to address this issue, we propose a scheme which is referred to as *mutual serializability* assuming that the file system support transactions. *Mutual serializability* captures a consistent backup of an active file system by ensuring that the backup transaction is mutually serializable with every other transaction individually. This mutually serializable relationship is established considering an extended set of conflicting operations which include read-read conflicts. User transactions serialize within themselves using some standard concurrency control protocol such as the Strict 2PL and a set of conflicting operations that only include the traditional read-write, write-write and write-read conflicts.

The proposed scheme has been implemented on top of a transactional file system and workloads exhibiting a wide range of access patterns were used as inputs to conduct experiments in two scenarios, one with the *mutual serializability* protocol enabled (thus capturing a consistent backup) and one without (thus capturing an inconsistent backup). The results obtained from the two scenarios were then compared to determine the overhead incurred while capturing a consistent backup.

The performance evaluation shows that for workloads resembling most present day real workloads exhibiting low intertransactional sharing and actively accessing only a small percentage of the entire file system space, the proposed scheme has very little overhead (a 5.7% increase in backup time and a user transaction throughput reduction of 3.68%). Noticeable performance improvement is recorded when using performance enhancing heuristics which involve diversion of the backup transaction to currently "colder" regions of the file system hierarchy on detecting conflicts with user transactions.

1. Introduction

Backup is the process used to protect primary data stored in file systems from corruption or its total loss as a result of human and machine errors or natural disasters. Historically, backup was taken of an unmounted and nascent file system. But, with current file system sizes touching peta bytes, O(file system size) operations like backup will take hours or even days, and in today's 24X7 business scenario unmounting a file system for even a few seconds may prove to be very costly. Hence, most organizations are resorting to taking the backup of a "live" file system, and such a backup is termed an *online backup*.

An arbitrarily taken online backup may result in an inconsistent backup copy. For example, suppose a program maintains financial information in two files. As part of a transaction, it subtracts an amount X from the first file and then adds X to the second file. It does not use locking because no other process uses these files. The online backup process may capture the new version of the first file and the old version of the second file, leading to an inconsistent state. Further, *Shumway* ([28]) details inconsistencies arising in the backup when file operations like *move, append, truncate*, etc., run concurrent to the backup program. Data is backed up mainly to restore the primary file system copy in case of its accidental loss and restoring from an inconsistent backupcopy may sometimes be more dangerous than data loss, as such errors may go undetected. The broad objective of the present

work ([13], [12]) is to understand the problem and propose 3 a solution for consistent online backup through its theoretical and practical treatment. This paper, after briefly introducing the theoretical basis of the scheme, presents the algorithms, their implementation, and an evaluation.

Current file systems do not provide features such as transactions for users to handle relationships among files and so consistency requirements across multiple files cannot be taken into account by the operating system. Hence, existing online backup solutions ([5], [8], [9], [17], [18], [19], [20], [2], [23]), while promising high system availability cannot do much to ensure the consistency of the data being backed up. This situation is currently found acceptable as user programs do not assume consistency preserving operations in file systems, and if consistency of operations across multiple operations is required, then systems like databases that support consistency are used. However database management systems cannot efficiently handle large units of variable sized data and there are many applications that now need the sharing of such data. The need for transaction support in file systems has been argued by many researchers ([33], [25]). BlogSeer ([22]) is a file system supporting concurrency control to be used with Hadoop. There are proposals too for concurrency control in BigTable implementations ([6]). We feel that support for transactions in general file systems will become common in the near future to meet these kinds of requirements.

We therefore assume that *consistent online backup* is to be provided in a transactional file system, where transactions are used by user programs to specify consistency requirements. The resulting consistent online backup solution is a specialized concurrency control mechanism called *mutual serializability* (MS). 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 will adversely impact system performance and backup time, as these mechanisms may result in the backup transaction getting repeatedly aborted or in user transactions dying one by one as the backup transaction gradually locks the entire file system. While we focus on backup in traditional tree-structured file systems with transaction support, the results can be easily extended to specific systems supporting transactions directly, such as BigTable.

Section 2 presents a brief review of the existing online backup solutions in both the file system and database areas, which forms the background to this paper and highlights why database backup solutions cannot be adopted directly in file systems. A brief conceptual overview of the proposed concurrency control mechanism forms the body of section 3. Section 4, details how the *mutual serializability* protocol is implemented over a transactional file system. In section 5, the implemented system is evaluated using a number of synthetic file access patterns. This section also describes the performance of a heuristic that involves moving the backup process away from areas of contention. We conclude in section 6 by briefly describing the future tasks required to enhance further the proposed consistent online backup technique.

2. Related Work

A popular technique to obtain an online backup is to provide for 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 online backup. A snapshot facility is either provided in a file system ([18], [20], [16]) or in disk storage subsystems ([5], [8]). However, since writes are never in place, snapshots may lead to dispersion of data. 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 online backup ([9], [5], [23]). The challenge here is to identify a consistent point to "split", and to handle the overheads of creating a new mirror. A third approach termed *continuous data protection* maintains a backup of every change made to data ([9], [24]). This suffers from extra writes to logs. Another solution operates by keeping track of open files and maintaining consistency of groups of files that are possibly logically related by capturing a backup copy of all files in a group together. Such groups of files are identified by monitoring modification frequencies across open files ([2]). This technique cannot be generalised and will only work in particular situations. The study presented in [1] and [29] proposed and implemented a system which allows applications to specify consistency requirements. It is much more systematic and efficient to provide a full-fledged transaction facility rather than the ad hoc techniques proposed in the above cited studies. But, current general purpose file system interfaces provide weak semantics (at best per file consistency) to applications for specifying consistency requirements as a result current online backup solutions cannot guarantee a consistent backup copy.

Given a file system with transactions, a solution could borrow online database backup techniques. Hot Backup ([3]) and its ancestor, the *fuzzy dump* ([15]), work by first backing up a "live" database and then running the *redo log* offline on the backup copy to establish its consistency. Using this in file systems will incur high performance cost and will be space inefficient as each entry in the file system log would have to record before and after images of possibly entire files, and at the very least, a block. So, we need to explore different approaches for achieving a consistent online backup copy of a file system.

Pu ([26]) suggested a scheme which reads an entire database in a consistent state in the face of regular transactions active on the database. This can be used to take a backup of the database and so the read can be referred to as a backup transaction. According to the proposed scheme the backup transaction "colours" entities black (originally white) as it reads them. A regular transaction is serialized with the backup transaction if it accesses all black or all white entities. A regular transaction accessing both black and white entities is not serialized and is aborted to ensure that the backup transaction reads a consistent image of the database. The method described in [26] failed to consider dependencies within user transactions and this drawback was rectified by

Ammann et al. ([4]). Our approach is similar in concept to the approach taken by Pu and *Ammann et al*, but novel in many ways. Our approach gives a sounder theoretical basis for identifying conflicts (described briefly in the next section and in more detail in [11] and in [12]). Our approach is independent of any particular concurrency control algorithm. Further, we consider a file system as opposed to a database system, which introduces new issues. For example, file systems have many more operations besides reads, writes, deletes and insertions. There are operations such as rename, move, copy, truncate, etc. Files in hierarchical file systems 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 which complicates online backup as file system namespace is constantly changing even as the backup process is traversing it. Moreover, file system backup does not just involve the backup of data but includes backing up of each file's metadata.

3. Approach

Our approach for capturing a consistent online backup of a file system assumes a transaction as the consistency specifying construct by applications. With the user and backup transactions now accessing the file system through transactions, a consistent online backup can be captured by ensuring overall serializability of the user and backup transactions. The problem has been put into a formal framework and correctness of our proposed mutual serializability technique of capturing a consistent online backup has been proven in [12].

To state the basic protocol for capturing a consistent online backup in a transactional file system, we consider a *flat file system* and a file system interface that provides *transactional* semantics. A file system transaction consists of a sequence called a *schedule* of atomic actions, and each atomic action *reads* or *writes* a file from a set of files in the file system. To ensure file system data integrity in the presence of concurrent accesses, a concurrency control mechanism, such as *strict two phase locking (strict2PL), optimistic concurrency control* etc. is used to serialize user transactions. A concurrency control mechanism establishes the *serializability* of the concurrent schedule where *serializability* is the accepted notion of correctness for concurrent executions.

The *backup transactions* schedule consists of a sequence of read operations to each file in the file system with each file being read at most once. An online backup transaction runs concurrently with the user transactions. For an online backup transaction to read a consistent file system state, it must be serialized with respect to the concurrently executing user transactions. For reasons stated in section 1, we are proposing a backup transaction specific concurrency control mechanism called *mutual serializability* which incurs reduced overhead due to the backup process.

Mutual serializability achieves an overall serializability of backup and user transactions by keeping the backup transaction mutually serializable with every user transaction individually, while the user transactions continue to serialize among themselves using a standard concurrency control protocol, like strict2PL. A pair of transactions in a schedule is said to be *mutually serializable* if on considering the operations of only these two transactions we get a serializable schedule. However, the set of conflicting operations that establishes a *mutually serializable* relationship, differs from the traditional set of conflicting operations as it includes readread conflicts in addition to the existing readwrite, write-read and write-write conflicts. The normal user transactions continue to maintain serializability among themselves by following the traditional definition of conflicting operations which includes only read-write, writeread and write-write conflicts. Consider the following example to illustrate the *mutual serializability* protocol: Let f_1, f_2 and f_3 be files in a file system accessed by transactions t_1, t_2 and the backup transaction t_b in the following order, $r_1(f_3)$, $r_b(f_3)$, $r_1(f_1)$, $w_2(f_3)$, $w_1(f_2)$, $r_b(f_1)$, $r_b(f_2)$, $w_2(f_2)$ where, reading (writing) of a file f_i by transaction t_x is denoted by $r_x(f_i)$ ($w_x(f_i)$). Transactions t_1 and t_2 are serializable among themselves as $r_1(f_3) < w_2(f_3)$, $w_1(f_2) < w_2(f_2)$ ("<" is the "happens before" relationship) and t_2 does not access f_1 at all. So t_1 is before t_2. t_b is mutually serializable with t_1 as $r_1(f_3) < r_b(f_3)$, $w_1(f_2) < r_b(f_2)$, and $r_1(f_1) < r_b(f_1)$. So t_1 is before t_b. It can be similarly seen that t_b is before t_2. It should also be clear that even if we do not consider read-read conflicts, t_1 is before t_b and t_b is before t_2. Therefore the above schedule is equivalent to the serial schedule t_1, t_b, t_2. But the schedule was obtained by only checking for mutual serializability of t_b with each transaction t_1 and t_2 individually. If we had not used mutual serializability, and the other transactions were using strict2PL, we would have had to ensure serializability of t_b with the other transactions by also doing strict two phase locking. But this would have resulted in t_b holding read locks to all files one by one, preventing other transactions from progressing. So, by introducing read-read as a conflict into the definition of mutual serializability, we have found a sufficient condition to ensure that the backup transaction is serializable with other transactions and this condition can be implemented fairly efficiently, as we shall show in the following sections. If we do not consider read-read as a conflict and we check t_b for serializability with each of t_1 and t_2 individually as before, we can get the following schedule: $r_b(f_1)$, $w_1(f_1)$, $r_2(f_1)$, $w_2(f_2)$, $r_b(f_2)$. But, this schedule is not serializable as $r_b(f_1) < r_2(f_1)$, but $w_2(f_2) < r_b(f_2)$, resulting in a cycle. Had we used mutual serializability, the operation $w_2(f_2)$ would not have been allowed to take place as $r_b(f_2)$ has not yet taken place and $r_b(f_1) < r_2(f_1)$ holds.

As read-only user transactions do not make any changes, they do not conflict with the backup transaction. But since read-read conflicts are also taken into account, an exception has to be made for read-only transactions and so by definition the backup transaction and any read-only transaction are mutually consistent.

Multiple file accesses (read or write) are merged into a single access within a transactions schedule in the formal model. It is so because, concurrency control protocols, whether locking or optimistic, ensures the isolation of operations by a transaction on a file. Thus, even though a file may be updated

multiple number of times within a transaction, it is effectively equivalent to a single update. Similarly, as a transaction has an isolated view of a file from the moment a file is "locked" in case of locking protocols or "opened" in case of optimistic protocols, so a schedule lists the order in which operations were successfully "locked" or "opened".

Despite its apparent simplicity, the model and result yields a theory of online consistent backup that can be applied to more complex file system models with transactions consisting of basic file access operations as well as semantically rich operations. This has been proved for semantically rich operations such as link(), rename() etc in hierarchical file systems by mapping into an equivalent sequence of the basic file access operations. For example, the link operation which is used to create a new hard link to a regular file f_i from a directory d_{j2} and with the file originally under directory d_{j1}, inserts the reference (new link) for f_i into d_{j2} and the link count of the file is incremented by one through the update of f_i's inode. Moreover, upon successful completion of a link() operation, the ctime field of the file fi and also, the ctime and mtime fields of the directory d_{j2} are updated. There is no change to directory d_{j1}. Hence, we see that a link operation maps to $(w_x(d_{j2});w_x(f_i))$ and the above stated mutual serializability protocol ensures a consistent backup of all updated files. In [12] it is shown that by adding a new operation, creat node, extending conflicts to include this operation, and by expanding the definition of mutual serializability to take into account this operation, the theory is still applicable. We will not go into details in this paper as it focusses on the implementation aspects.

4. Consistent Online Backup

In section 4.1, we present a conceptual understanding of how the backup utility and the backup specific concurrency control solutions are implemented. Brief design and implementation details are presented in section 4.2 4.1 Conceptual Overview Applications run as a sequence of transactions and under normal circumstances when the backup program is not active, they simply use any standard concurrency control technique such as locking or optimistic protocols to ensure consistent operations. We have used Strict 2PL in the current implementation. Once the backup program is activated, all other transactions are made aware of it by some triggering mechanism (the current implementation sets a global variable) and they now need to serialize themselves with respect to the backup transaction, while continuing to serialize among themselves as before. 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's metadata structure such as an 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 *Pu* ([26])). A *read* bit of value 0 indicates that the file has not yet 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 should be used instead, where the sequence number of the present backup transaction is one greater then its immediate predecessor. As our implementation is at the user level, we cannot make changes to the inode structure and hence we utilised the *sticky bit* bit of an inode as the *read bit*.

The backup transaction traverses the file system namespace reading files on its path to the backup-copy and as it reads each file it sets the *read* bit to 1. The current implementation locks each file before it is read and on successful reading, the file is unlocked before proceeding to read the next file. 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 any file, the user transaction verifies whether it continues to be mutually serializable with respect to the backup transaction by comparing the *read* bit of this file with its own *before-after* bit. The following table enumerates the mutually serializable checking protocol.

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_x 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_x. 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. Cascading roll backs increases the time taken for a backup, thus lengthening the system vulnerability window. Moreover, the notion of a backup is to capture a point-in-time or at-least a near point-

in-time image of the data set and a backup that spreads across a long time-line is not acceptable.

Another method of handling the problem is to abort and restart t_x "hoping" the backup transaction completes reading the "unread" files accessed by t_x. The current solution is applicable only if t_x's *before-after* bit is 0 and it attempts to access a file whose *read bit* is 1. 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_x 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. The current implementation, employs a combination of aborting and pausing the conflicting user transaction, depending on the *before-after* bit status (the transaction is aborted is the *before-after* bit value is 0 and paused if 1).

It must be noted here that the creation of a file by a user transaction is mapped to a write to the parent directory and a *creat node*() operation to the newly created file. Hence, when a file $file_k$ is created by a user transaction t_x, it locks the parent directory d_j under which $file_k$ is created and $file_k$'s *read bit* is set to 0 if the file is created by a "before" transaction (the *before- after bit* of the transaction is 0) and is set to 1 if created by an "after" transaction (the *before- after bit* of the transaction is 1). By doing so we note that if the *read bit* of $file_k$ is 0 (*read bit* of d_j is also 0 as ensured by mutual serializability), t_b will read it and if its *read bit* is 1 (*read bit* of d_j is also 1 as ensured by mutual serializability), the file will not be read by t_b (even though $file_k$ is not actually in the backup). Such a treatment ensures that the transaction which created $file_k$ is *mutually serializable* to t_b.

The implementation needs to ensure that the backup transaction never needs to abort (roll back) due to reasons discussed above. As long as t_b reads committed data and t_b is *mutually serializable* with each concurrent transaction, t_b never *rolls back.* Our implementation ensures that t_b never needs to *roll back* because it ensures that t_b is mutually serializable with each t_x and because user transactions are serialized using the *Strict2PL*protocol (ensuring t_b reads only committed data).

4.2 Implementation Overview

To be able to implement the proposed online backup utility we require an underlying transactional file system. But current transactional file systems either exist as research projects still under development and evaluation ([25], [30]), or closed-source systems limited to a specific file system and operating system ([31]). Hence, we implemented our own basic transactional file system (which we shall refer to as TxnFS) for implementing our proposed consistent on- line backup approach. TxnFS has been implemented in an user level process. The user interface to TxnFS is implemented as a shared library with routines defining the calls to the transactional constructs namely *txn_commit*, *txn_abort*, *txn_begin* and file access namely open(), close(), read(), write() etc. TxnFS guarantees the transactional properties of isolation and consistency through the implemented *strict 2PL concurrency control protocol* and transactional properties of atomicity and durability is ensured through TxnFS's write-ahead logging mechanism. See Figure 1 for a top level view of the implemented TxnFS architecture.

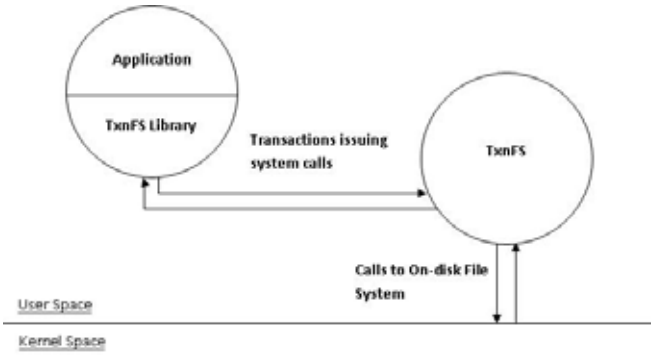

Fig. 1: Top Level TxnFS Architecture

We implement the online backup utility as a service provided by the file system and it is invoked through the call *backup()* provided as part of the TxnFS API. The syntax of the backup call is: *int backup*(*const char *path*). The backup utility runs as a transaction and is thus wrapped between the transactional constructs of *txn begin* and *txn commit*. In the following sections we outline the implementation details of the backup utility and the enforcement of *mutual serializability.*

4.2.1 Data Structure

Adapting the transactional file system to facilitate the backup utility in capturing a consistent file system image, needs just three updates in its data structures. We include a global variable called BACKUP in the TxnFS process. BACKUP is initialized to 0, which indicates that the backup utility is currently inactive. At the beginning of a backup run, the value of BACKUP is changed to 1 indicating a currently active backup process and the "read bit" of every file is set to 0. This bit is implemented through the *sticky bit* present within an ext2fs inode data structure ([7]). The *sticky bit* is a single bit in the st mode field in the ext2 inode data structure which is now rarely used. In our redefined use for the *sticky bit*, a "set" value indicates that the file has been read by the backup utility and is yet to be read other- wise. Thirdly, an integer field representing the "before-after bit" is added to each transaction's housekeeping data structure in the *txn list*. It

is set to 0 when a transaction begins.

4.2.2 The Backup Algorithm

The implemented *mutual serializability* concurrency control protocol will work correctly to facilitate the capture of a consistent file system copy, irrespective of the order in which the file system hierarchy is traversed by the backup module. But we see that files of an application or of a particular user or any logically related groups of files are normally co-located on mutually disjoint subtrees. Thus, there is high probability of a user transaction localizing its access to within a subtree. Considering such access patterns, if the implemented backup module orders its file accesses such that it results in files within subtrees to be accessed temporally apart, such as in the case of breath-first traversal, then on a conflict if the user transaction is made to wait for the backup transaction to read the conflicting file, there may be a long wait. So we have implemented the backup module to traverse the file system hierarchy in a depth-first manner.

The backup algorithm is detailed in Algorithm 1. The algorithm makes use of a *stack* data structure to implement depth-first traversal. The *stack* is referred to simply as "stack" and the variable "top" holds the pointer to the topmost element in the *stack*. Each element in the *stack* is a pointer to a string representing a file name. A call to the function *push* inserts a *file name* into the *stack* and the variable "top" is suitably updated. Similarly a call to the function *pop* returns the *file name* stored at the position pointed at by "top" and later suitably updated.

Algorithm 1 Backup(*root_path*)

```
push(root_path)
repeat
  file ⇐ pop()
  if IS_DIR(file) then
    Lock file in exclusive mode
    Copy file to backup
    Set file's sticky bit
    if file is non empty directory then
      push() each child of file into stack
    end if
    Release lock on file
  else
    if IS FILE(file) then
      Lock file in exclusive mode copy file to backup
      Set file's sticky bit
      Release lock on file
    end if
  end if
until stack is empty
```

The current implementation stores on the *stack* the *file* names (which are absolute path names) of the children of a directory just read. This can result in *file* names stored in the stack to become invalid if a transaction serialized after the backup transaction "moves" an ancestor directory to a new location in the hierarchy. Such an inconsistency would not arise if the inode could be stored in place of the *file* name but this is not possible in our user level implementation. Hence, to prevent such inconsistencies, we used a dedicated thread which is woken up after every successful directory *rename* operation on the underlying on-disk file system. This thread walks through the *stack* searching for descendants of the just *renamed* directory and if present, changes their stored file path name to reflect its new position in the hierarchy.

Recent versions of Linux have added new system calls using which inconsistencies described above can be prevented in a more efficient and clean manner. These new system calls are the *name to handle at*() and *open by handle at*() ([10]).

4.2.3 The Mutual Serializability Algorithm

The algorithm to check and enforce a user transactions *mutually serializable* relation with the backup transaction is described in Algorithm 2. *txnid → bef ore af ter bit* refers to the *before after bit* of transaction *txnid* and *file → read bit* refers to the value of the *sticky bit* in the inode of the file referred to by the string *file*. If the *sticky bit* is set then we consider the *read bit* to have value 1 and a *read bit* value of 0 otherwise. Algorithm 2 returns the value "pass" if the application transaction is *mutually serializable* so far, the value "fail" if the application transaction conflicts with the backup transaction and was so *aborted* and the value "try again" if there was a *mutually serializable* conflict detected and the Transactions progress was "paused" to allow the backup transaction to "catch up".

It must be noted here that the *sticky bit* of a newly created file is set if the *before-after bit* of the user transaction creating the file is 1 and left unset otherwise.

5. Evaluation

The *mutual serializability* protocol ensures a consistent backup copy when captured by an online backup utility, but consistency of the backup copy comes with a cost. In this section we shall determine the overhead of obtaining an online consistent backup using the *mutual serializability* protocol on the performance of user transactions as well as the backup transaction. We shall further propose and evaluate techniques to reduce conflicts of user trans- actions with the backup transaction and thus explore methods to reduce the cost incurred due to the *mutual serializability* protocol. To the best of our knowledge the online concurrency control method proposed in this paper is the first protocol aimed at ensuring backup copy consistency in the file system domain. Thus, the proposed technique could not be compared with other existing techniques as there were none.

Algorithm 1 IS MS(*file,txnid*)

```
if file is the first file to be locked by txnid
then
  txnid → before after bit ( file → read bit
  return (pass)
else
  if (txnid → before_after_bit == file →
  read bit) then
    return (pass)
  else
    if (file → read bit == 0)and(txnid →
    before_after_bit == 1)) then
      Pause txnid for a random period
      return (try again)
    else
      Abort txnid
      return (fail)
    end if
  end if
end if
```

5.1 Workload Models

We ran TxnFS on which the *mutual serializability* protocol is implemented through various types of transactional file system workloads. File system traces can either represent real workload captured from an active file system, or be synthetic workloads, fabricated to recreate the characteristics of real systems. With transactional file systems still at the research stage and not yet used in real environments, there are currently no available real transactional file system traces. Hence, we have evaluated our proposed protocol using synthetic transactional traces.

Synthetically generated traces allow us to isolate different access patterns and test a system on these separate patterns. Moreover through synthetic traces we can model access patterns not yet seen but likely to exist in future, such as an increase in the degree of file sharing among users. The main objective while generating traces was to match it as closely as possible to realistic workloads as described in various file system workload studies ([21], [14], [32], [27]).

Each generated workload is modelled such that transactions comprise of a sequence of file system calls wrapped in transactional constructs of *txn begin* and *txn commit/txn abort*. The file system calls include open(), close(), read(), write() (including appending), lseek(), creat(), unlink(), rename() and stat(). A transaction issues 10 file system calls on an average and unless otherwise stated, a transaction issues any of the listed calls with equal probability. Each workload is generated on a file system image which initially consists of approximately 5000 files. The following sections describe the characteristics of each workload set.

5.1.1 Uniformly Random Pattern (global):

Transactions in this workload set access files randomly and with equal probability from the entire file system hierarchy. Though such an access pattern is unrealistic, it helps us evaluate the implemented system in a "worst case" scenario.

5.1.2 Spatial Locality Access Pattern (local):

File system access patterns exhibit a strong spa- tial locality with logically related files located spatially close to each other in the file system hierarchy, and in all probability under a single subtree. This suite of traces is generated to exhibit such behaviour. Within this category we generate a number of workload sets as described below:

5.1.2.1 Inter-Transactional Sharing:

Studies ([21]) have shown that together with processes exhibiting spatial locality of access, files are infrequently shared among clients and even if they do, sharing is rarely concurrent. But, the future is likely to usher in a higher degree of sharing with transactional file systems providing for more secure and reliable sharing techniques. Hence, we evaluate the system performance on enabling the *mutual serializability* concurrency control protocol with workloads that model access locality along with varying degree of inter-transactional sharing. In this suite we generated four different workload sets, referred to as the **50%share**, **25%share**, **10%share** and **0%share** workloads. Up to 50%, 25%, 10% and 0% of the files are accessed by more then one transaction in workloads **50%share**, **25%share**, **10%share** and **0%share** respectively.

5.1.2.2 Access pattern with high percentage of *stat* calls (stat):

File access pattern studies ([27], [21]) reports a higher percentage of *read* (file attribute or data or both) access as compared to other operations. We model the next set of workloads to reflect such an access pattern. As a higher degree of stat calls is equivalent to higher read access of files in the context of our evaluation, we only model work- loads with higher **stat** calls. The **50%share-stat** and the **0% share-stat** (collectively referred to as the **stat** workloads) sets are generated to consist of transactions making *stat()* calls at most 70% ([27]) of all its file system calls. In the **50%share-stat** trace set transactions share up to 50% and in the **0%share-stat** trace set transactions do not share any files among themselves.

5.1.2.3 Hot and Cold Pattern (hot-cold):

Study in [32] reports that about 10% of files are accessed 90% of the time and the rest 90% remains mostly "cold" by being accessed only 10% of the time. The current workload models this behaviour. Within this category we generated two sets of workloads, the **50%share- hot-cold** and the **0%share-hot-cold** sets (collectively referred to as the **hot-cold** workload) consisting of transactions sharing

up to 50% of the files and not sharing any files among themselves respectively.

5.2 Experimental Setup

We implemented the TxnFS prototype and *mutual serializability* protocol on a laptop with 1.73GHz Pentium CPU and 256MB of RAM. It ran Fedora 14 with Linux kernel version 2.6.35.

TxnFS was built on top of an ext2 file system and the initial file system image is an exact copy of the file system image of one of our personal machines. To ensure consistency and a cold cache, we ran each iteration of a trace set on a newly formatted file system with as few services running as possible. The same initial file system image is copied to the newly formatted file system for each run of the tests and all tests were run at least five times. The log used by TxnFS was maintained on a separate file system on a separate partition and the log partition too is formatted before every test.

Each generated workload set as described in Section 5.1 is replayed such that the degree of concurrency varies randomly with an average of four transactions running concurrently at any moment. The trace replay simulator busy waits for a random period between any two operations within transactions for a realistic representation. Transactions may have been aborted to break a deadlock situation or because it conflicted with the backup transaction. Such aborted transactions are restarted. Hence, all transactions are completed to *commit* unless exclusively *aborted* by users.

5.3 Simulation Results and Performance Analysis

Experiments were run in two scenarios, one with (referred to as **MS enabled**) and the other without (referred to as **MS disabled**) enabling the *mutual serializability* protocol. Comparing the results obtained from the experiments run in the two scenarios allows us to evaluate the overhead of capturing a consistent backup copy over an inconsistent one. The evaluation metrics used are, the percentage of transactions that conflict with the backup transaction (referred to as "conflict percentage"), the time in microseconds taken for the completion of the backup transaction (referred to as "backup time") and the number of user transactions completing to *commit* per microsecond during the duration the backup utility is active (referred to as "throughput").

TABLE 1: Change in Conflicts Between The MS disabled and MS enabled Run.

Metrics / Workload	Increase in Conflict Percentage with MS enabled(%)
global	57
0%share	7.5
10%share	10.6
25% share	13.5
50% share	15
0%share-stat	7
50%share-stat	14
0%share-hot-cold	2.5
50%share-hot-cold	6

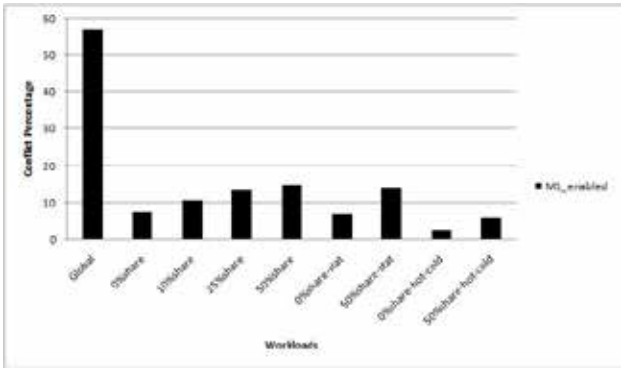

Fig. 2 : Percentage Of Transactions Conflicting In Each Set Of Workload

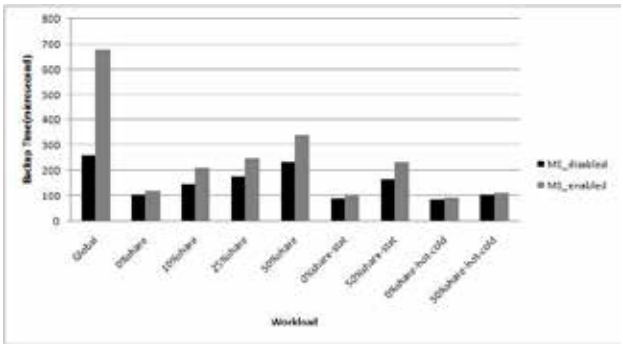

Fig. 3: Duration Of Backup For Each Set Of Workload

Fig. 2 and Table 1 depict the percentage of transactions that conflict with the backup transaction in each of the different workloads. The simulation results illustrating the backup time of the different workloads in the MS enabled and the MS disabled scenarios can be seen in Figure 3 and the values in Table 2 represents the overhead incurred in capturing a consistent online backup in terms of the backup time. Longer the duration needed to perform backup, more skewed the backup copy becomes. A longer backup "window" also increases the vulnerability of the system. Figure 4 describes the throughput of the user transactions for each workload in both the MS enabled and the MS disabled scenarios and Table 3 shows the percentage decrease in throughput in the MS enabled as compared to the MS disabled run.

Table 2 : Change in Backup Time Between The MS disabled and MS enabled Run.

Workload	Percentage increase in Backup time with MS enabled(%)
global	161
0%share	13.8
10%share	39.5
25% share	41.24
50% share	44.5
0%share-stat	12.2
50%share-stat	43
0%share-hot-cold	5.7
50%share-hot-cold	7.6

Table 3: Change in Throughput Between The MS disabled and MS enabled Run.

Workload	Percentage increase in Backup time with MS enabled(%)
global	67.33
0%share	9.85
10%share	27
25% share	27.44
50% share	32.35
0%share-stat	12.2
50%share-stat	33.8
0%share-hot-cold	3.68
50%share-hot-cold	4.37

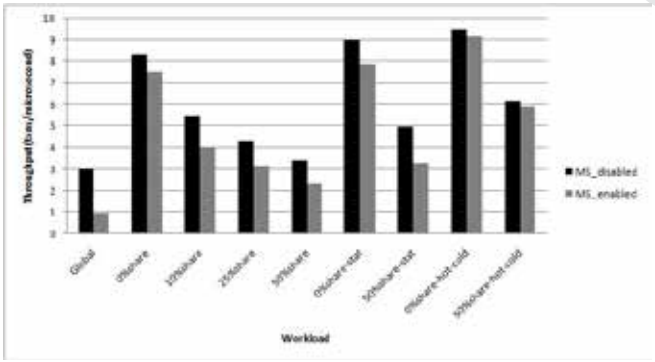

Fig. 4: Throughput Of User Transaction (Measured for the duration the backup was active)

From Fig. 2 and Table 1, we notice that over 50% of transactions conflict with the backup transaction in the global workload. The backup transaction traverses the file system hierarchy in a depth-first manner and reads the file system subtree by subtree (in other words spatial locality by locality). The high percentage of conflicts incurred by the **global** workload is mainly because transactions in this workload do not exhibit locality, thus increasing the probability of conflicts. This deduction is confirmed when we see a steep drop in the percentage of transactions conflicting with the backup transaction in the rest of the workload sets, all modelled to exhibit locality of access. In the local workloads, conflicts only arise if the backup transaction is reading in the locality being accessed by a concurrent transaction. If concurrent user transactions access localities away from the area backup transaction is currently reading, conflicts do not happen. Detection of conflict is followed by measures to resolve these conflicts and such measures lead to further performance degradation. In Figure 3 and Table 2 we see that over 50% conflict percentage in the **global** workload has resulted in 161% increase in backup time in the **MS_enabled** run as compared to the **MS_disabled** run. Similarly, Figure 4 and Table 3 shows a decrease of approximately 67% in throughput for the **global** workload from the **MS disabled** to the **MS enabled** run. Performance degradation incurred while capturing a consistent backup both in terms of backup time as well as throughput, decreases considerably in the **local** workloads as conflict percentage decreases.

In Fig. 2 we see that as the degree of inter-transactional sharing increases, the percentage of conflicts too increases, though in very small amounts. On analysis we see that this is because, higher the degree of sharing, higher the probability of concurrent access to common files and hence if one transaction conflicts with the backup transaction, other transactions accessing the common set of files concurrently also has a high probability of conflicting with the backup transaction. As the degree of inter-transactional sharing increases a slight increase in the backup time is noticed during the **MS_enabled** runs in comparison to the **MS_disabled** runs. For example, **50%share** shows an

increase of 44.5% whereas **25% share** workload shows a 41.24% increase in the **MS_enabled** run as compared to the **MS_disabled** run. This is mainly due to the increase in the conflict percentage as sharing increases. The results obtained from the **0%share** workload is optimistic as it shows only a 13.8% increase in the time for taking a consistent file system backup as opposed to an inconsistent backup. Similarly, we notice a decrease in the throughput of user transactions while capturing a consistent backup as compared to the inconsistent backup with the increase of inter transactional sharing. For example, throughput decreases from 7.5% in the **0%share** to 2.3% in the **50%share** workload under **MS_enabled**. However,the reason behind this is not totally the *mutual serializability* protocol but also due to the increased contention within the user transactions themselves as a result of increased sharing. This argument is re-enforced by the fact that a decrease in throughput is seen with increase in sharing in the corresponding runs under **MS_disabled**.

From the stat workloads simulation results we see that the percentage of conflicts in both the **50%share-stat** and **0%share-stat** is almost equal to the percentage of transactions conflicting in the **50%share** and **0%share** workloads respectively. While establishing the *mutually serializable* relation, *read-read* conflicts are also taken into account and hence even though the **stat** workloads differ from the corresponding **50%share** and **0%share** workloads by issuing a much higher percentage of read accesses, the total percentage of conflicts in the corresponding two workloads is almost the same. So the backup time and throughput value are also similar. But, inter user transactional conflicts in the **stat** workloads are very less (read to the same file by different user transactions is not a conflicting operation). This explains the lesser *backup time* in both the **MS enabled** and **MS disabled** runs of the **stat** workloads when compared to the **50%share** (or **0%share**) work- load. This interesting and encouraging result is echoed by the higher throughput of the **50%share-stat** and **0%share-stat** workloads for both scenarios as compared to the **50%share** and **0%share** workloads respectively.

The most promising result is seen from both the **hot-cold** workloads where a much lesser *conflict percentage* in case of the **50%share- hot-cold** (6%) workload as opposed to the **50%share** workload (15%) is reported. This is because user transactions access only a small part of the file system thus decreasing the probability of the backup transaction and user transactions accessing the same locality concurrently and hence the decrease in *conflict percentage*. Performance improves further if inter transactional file sharing decreases and the **0%share-hot-cold** workload with no inter transactional file sharing shows only about 2.5% of user transactions conflicting with the backup transaction. The simulation results of the **50%share-hot-cold** and the **0%share-hot- cold** workload on metrics *backup time* with **MS enabled** is approximately only 7% and 5.7% respectively, more than that reported from a **MS disabled** run. The positive trend continues as we see throughput results from both the **hot-cold** workloads where the through- put in the **MS enabled** scenario increases by approximately 156% in the **50%share-hot-cold** workload as compared to the **50%share** and by 22% in the **0%share-hot-cold** as compared to the **0%share** workload. As conflicts are rare in the **hot-cold** workloads, more transactions can commit successfully during the backup duration. We also note that overhead in terms of throughput is quite low at 4.37% for **50%share-hot-cold** and 3.68% for **0%share-hot-cold**, making the prospect of capturing a consistent on- line backup attractive.

5.4 Performance Improvement

Consider the **hot-cold** workloads and its associated simulation results reported above. We notice that conflicts among user and backup transactions occur only when the backup transaction is traversing the regions of the file system hierarchy currently being actively accessed by the user transactions and from the work- load model. So we applied the heuristic that, whenever the backup transaction conflicts with a user transaction, the backup transaction is diverted to another part of the file system hierarchy "hoping" that this part of the hierarchy is currently "cold". Transactions exhibit spatial locality of access and so the probability of the region of conflict being "hot" is high. Therefore, diverting the backup transaction from this locality, decreases the probability of further conflicts. Of course, the backup transaction goes back to read the region it was diverted from at a later time "hoping" the region is no longer 'hot". The evaluation results on applying this heuristics are detailed in the following section.

5.4.1 Simulation Result and Performance Analysis

We simulated the enhanced online backup prototype with the **50%share-hot-cold** work- load as well as the **0%share-hot-cold**. For ease of presentation of the results, we shall refer to a **MS enabled** run without application of heuristic as **MS enabled** (no heuristics) and an **MS enabled** run with application of performance enhancing heuristics as **MS enabled** (heuristics).

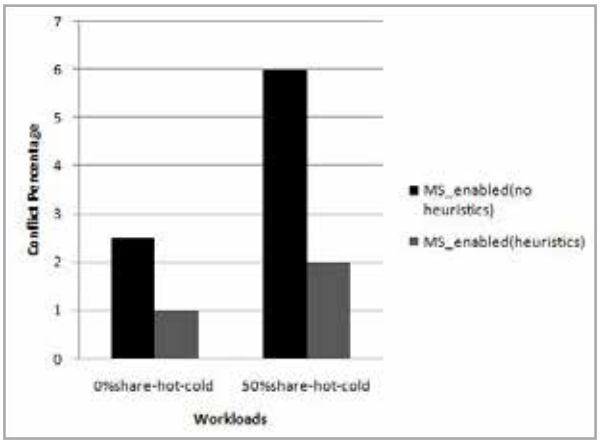

Fig. 5: Conflict Percentage On Applying Heuristics To Improve Performance

See Fig. 5 for a comparison of the **MS enabled** (no heuristics) run and **MS enabled** (heuristics) on *conflict percentage*. We observe a significant drop (by approximately 67% in the **50%share-hot- cold** workload and approximately 60% in the **0%share-hot-cold**) in the percentage of transaction conflicting with the backup in the **MS enabled** (heuristics) in comparison to the **MS_ enabled** (no heuristics) run. This decrease in conflict is as

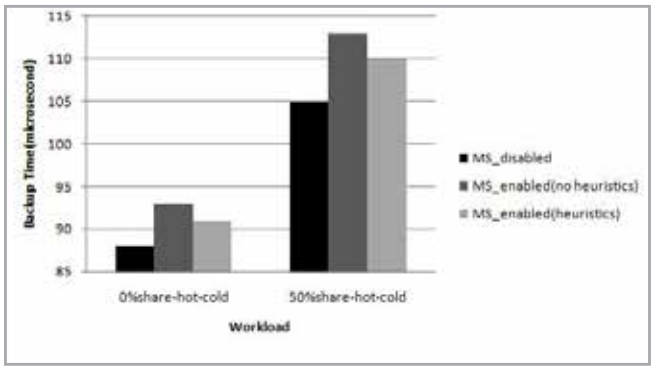

Fig. 6: Backup Time On Applying Heuristics To Improve Performance

a result of prevention of successive conflicts in the same locality by the diversion of the backup transaction from that locality on detection of the first conflict. Now, in the **50%share-hot-cold** under **MS enabled** (no heuristics), the backup transaction is likely to conflict with more then one transaction in the same locality as a result of up to 50% inter-transactional file sharing. Thus, under **MS enabled** (heuristics) on diversion of the backup transaction on detection of a conflict, the backup transaction avoids conflicting with all other concurrent transactions accessing the locality it was diverted from. Whereas, in the **0%share-hot-cold** under **MS enabled** (no heuristics), the backup transaction will conflict with just one transaction in a locality as a result of no inter-transactional file sharing. This explains the higher conflict drop (approximately 67%) in the **50%share-hot-cold** as compared to the drop (approximately 60%) in the **0%share-hot-cold** workload under **MS enabled** (heuristic) from **MS enabled** (no heuristic). The decrease in *conflict percentage* in the **0%share-hot-cold** even though there were no further conflicts with concurrent user transactions in the same locality, is because the backup transaction is diverted to a locality much further away from the current one thus avoiding conflicts that may arise in nearby localities. If the set of "hot" files are not close to each other, the significant drop in *conflict percentage* will not be seen. We note that the conflict that lead to the backup transaction being diverted to another locality, is still a conflict and the corresponding user transaction has to be either aborted or "paused".

See Fig. 6, where we compare the *backup time* for the **MS enabled** (no heuristics) run, **MS disabled** run and **MS enabled** (heuristics). We see a decrease (by 2.6% in the **50%share-hot-cold** workload and 2.1% in the **0%share-hot-cold**) in the *backup time* when we compare the **MS enabled** (heuristics) run in comparison to the **MS enabled** (no heuristics) run. We also see that when compared with the **MS disabled** run the overhead of taking a consistent backup in the **50%share- hot-cold** workload decreases from 7.6% in case of the **MS enabled** (no heuristics) to 4.8% in the **MS enabled** (heuristics). This decrease in *backup time* is attributed to the decrease in the number of conflicts.

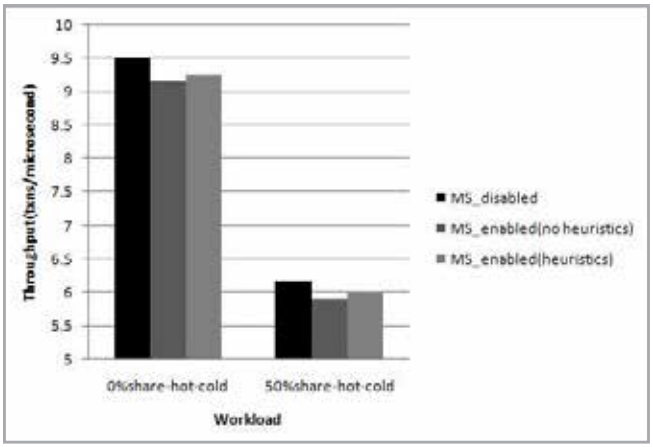

Fig. 7: Throughput On Applying Heuristics To Improve Performance.

Fig. 7 compares the throughput of the user transactions during the backup duration for the **MS disabled**, **MS enabled** (no heuristics), and **MS enabled** (heuristics) run. An increase (1.6% in the **50%share-hot-cold** and 1.1% in the

0%share-hot-cold) in throughput of the user transaction is seen in the case of **MS enabled** (heuristics) as compared to **MS enabled** (no heuristics). Throughput increases as conflicts decrease and hence more transactions can commit. Moreover, the backup time too decreases with a decrease in conflicts, hence in reality throughput has increased more than the percentage value seen. The higher percentage increase in throughput during **MS enabled** (heuristics) with respect to **MS enabled** (no heuristics) in the **50%share-hot-cold** as compared to **0%share-hot-cold** can be attributed to the higher decrease in conflicts in **50%share- hot-cold** as compared to **0%share-hot-cold**

6. Conclusion and Future Work

The process of backup which essentially involves copying and archiving digital data is an important part of data protection against its possible loss or corruption as well as for the purpose of retention of old file versions. To ensure correctness of data upon recovery from a backup copy, the consistency of a backup copy is essential. But, inconsistencies may arise when capturing a backup from an active file system. Assuming a file system that supports transactions, the current study has proposed a backup transaction specific concurrency control protocol for capturing consistent online backup. We refer to this concurrency control protocol as *mutual serializability*. 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. The proposed *mutual serializability* protocol ensures that the backup transaction does not have to be aborted, and delaying a conflicting transaction in most cases resolves conflicts.

In this paper, we have presented the implementation and the evaluation of the *mutual serializability* protocol. To do this we had first designed and developed TxnFS, a prototype transactional file system in user space over an ext2 file system which is used as the actual file store. The implemented backup transaction establishes a *mutually serializable* relationship with every application transaction through a bit (read-bit) in each file's metadata to indicate its backup status and another bit (before-after bit) embedded in each transaction's metadata to indicate its serialization order with respect to the backup. Compromise in the *mutually serializable* relationship is detected when the read bit and before-after bit do not match. A *mutually un-serializable* pair is prevented by either aborting the conflicting user transaction or by "pausing" it and allowing the backup transaction to go ahead till *mutual serializability* can be established.

The implemented system was then simulated through workloads exhibiting a wide range of access patterns and experiments were conducted on each workload in two scenarios, one with the *mutual serializability* protocol enabled (thus capturing a consistent backup) and one without (thus capturing an inconsistent backup) and comparing the results obtained from the two scenarios to determine the overhead incurred while capturing a consistent backup. The performance evaluation shows that for workloads resembling most present day real workloads exhibiting low inter-transactional sharing and actively accessing only a small percentage of the entire file system space, there is very little overhead. This is found to be 2.5% in terms of transactions conflicting with the backup transaction, 5.7% in terms of backup time increase and 3.68% in terms of user transaction throughput reduction during the duration of the backup pro- gram. Noticeable performance improvement is recorded on the use of performance enhancing heuristics which involved diverting the backup program to lesser active regions of the file system on detecting conflicts.

As already seen, the protocol in its present state can benefit tremendously from heuristically driven performance enhancement techniques. With careful analysis of real workload patterns more such heuristics can be applied to improve performance for user applications and as well as the backup utility and such studies are part of our future work. For example, trace history of systems can be used to forecast the movement of "hot" regions and the backup traversal utility can be designed accordingly so that it traverses currently "cold" regions as much as possible. Use of histories to divert the backup utility prevents even the conflict needed to trigger the divert, thus scoring better than the scheme described in this paper. Another method to improve performance is to backup sets of files in parallel, with each parallel run establishing the *mutually serializable* relation with concurrent transactions. Current real workloads share files infrequently and even if they do, sharing is rarely concurrent as reported in [21]. Sets of files which are always accessed in a mutually exclusive manner can be backed up in parallel to decrease the *backup time*. Finally, if transactions can be identified as read-only at the start of transactions, such transactions will not conflict with the backup transaction, and performance will improve as the number of read-only transactions increase. It is expected that in many environments, read-only transactions will be significant in number, and so the results obtained point to even better performance in such environments.

We also plan to implement our backup strategy on existing file systems with no notion of transactions through the use of per system call consistency preserving approaches such as *journalling* and *soft updates* already present in existing file systems.

References

[1] FlashCopy Consistency Group: Creating Consistent PiT Copies. http://www.redbooks.ibm.com/abstracts/ tips0309. html?Open. Published on 15/10/2003. Accessed on 09/04/2012.

[2] Preventing Data Loss During Backups Due to Open Files. St. Bernard Software's White Paper On Open File Manager. November, 2003. http://www.novell.com/ coolsolutions/ netware/assets/ofm white paper.pdf. Ac- cessed on 18/02/2014.

[3] Oracle Database Backup and Recovery Basics (2005). 10g Release 2 (10.2) B14192-03

[4] Ammann, P., Jajodia, S., Mavuluri, P.: On-The-Fly Reading of Entire Databases. In: IEEE Transactions on Knowledge and Data Engineering, vol. 7, pp. 834–838 (1995)

[5] Azagury, A., Factor, M.E., Satran, J., Micka, W.: Point-in-Time Copy: Yesterday, Today and Tomorrow. In: Proceedings IEEE/NASA Conf. Mass Storage Systems, pp. 259-270 (2002)

[6] Baker, J., Bond, C., Corbett, J.C., Furman, J.J., Khorlin, A., Larson, J., Leon, J., Li, Y., Lloyd, A., Yushprakh, V.: Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In: Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (2011)

[7] Bovet, D., Cesati, M.: Understanding the Linux Kernel, 2 edn. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2002)

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

[9] Chervanak, A., Vellanki, V., Kurmas, Z.: Protecting the File System: A Survey of Backup Techniques. In: Proceeding of the Joint NASA and IEEE Mass Storage Conference, pp. 17-32 (1998)

[10] Corbet, J.: Open by handle (2012). http://lwn.net/ Articles/375888/. Accessed on 18/02/2014.

[11] Deka, L.: Consistent Online Backup in Transactional File System. In: PhD Thesis, Indian Institute of Technology Guwahati (2013). http://homepages.lboro.ac. uk/~cgdbd/PhD thesis Lipika Deka.pdf. Accessed on 18/02/2013.

[12] Deka, L., Barua, G.: Consistent Online Backup in Transactional File Systems. In: IEEE Transactions on Knowledge and Data Engineering. (Accepted. DOI (identifier) 10.1109/

TKDE.2014.2302297)
[13] Deka, L., Barua, G.: On-line Consistent Backup in Transactional File Systems. In: Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems (2010)
[14] Gibson, T., Miller, E.L., Long, D.D.E.: Long-Term File Activity and Inter-Reference Patterns. In: Proceedings of the 24th International Computer Measurement Group Conference, pp. 976-987 (1998)
[15] Gray, J.: Notes on Data Base Operating Systems. In: Lecture Notes in Computer Science, vol. 60, pp. 393-481 (1978)
[16] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc. (1992). URL http://portal.acm.org/citation.cfm?id=573304
[17] Green, R.J., Baird, A.C., Davies, J.C.: Designing a Fast, On- line Backup System for a Log-Structured File System. In: Digital Technical Journal of Digital Equipment Corporation, pp. 32-45 (1996)
[18] Hitz, D., M.Malcolm, Lau, J., Rakitzis, B.: Method for Maintaining Consistent States of a File System and for Creating User-Accessible Read-Only Copies of a File System. In: US Patent No. 5,819,292 (1998)
[19] Hutchinson, N.C., Manley, S., Federwisch, M., Harris, G., Hitz, D., Kleiman, S., O'Malley, S.: Logical vs. Physical File System Backup. In: Proceedings of the Symposium on Operating Systems Design and Implementation, pp. 239-249 (1999)
[20] Johnson, J., Laing, W.: Overview of the Spiralog File System. In: Digital Technical Journal, vol. 8, pp. 5-14 (1996)
[21] Leung, A.W., S.Pasupathy, Goodson, G., Miller, E.L.: Measurement and Analysis of Large-Scale Network File System Workloads. In: Proceedings of the USENIX Technical Conference, pp. 213-226 (2008)
[22] Nicolae, B., Moise, D., Antoniu, G., Bouge, L., Dorier. M.: Blobseer: Bringing High Throughput Under Heavy Concurrency to Hadoop Map-Reduce Applications. In: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing, pp. 1–11 (2010)
[23] Patterson, R.H., Manley, S., Federwisch, M., Hitz, D., Kleiman, S., Owara, S.: Snapmirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies (2002)
[24] Peterson, Z., Burns, R.: Ext3cow: a Time-Shifting File System for Regulatory Compliance. In: ACM Transactions on Storage, vol. 1, pp. 190–212 (2005)
[25] [25] Porter, D., Hofmann, O., Rossbach, C., Benn, A., E.Witchel: Operating Systems Transactions. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles, pp. 161-176 (2009)
[26] Pu, C.: On-The-Fly, Incremental, Consistent Reading of Entire Databases. In: Proceedings of the 11th International Conference on Very Large Data Bases, vol. 11, pp. 369–375 (1985)
[27] Roselli, D., Lorch, J.R., Anderson, T.E.: A Comparison of File System Workloads. In: Proceedings of the USENIX Annual Technical Conference, pp. 41–54 (2000)
[28] Shumway, S.: Issues in On-line Backup. In: Proceedings of the 5th Conference on Large Installation Systems Ad- ministration (1991)
[29] S. Marathe, Massiglia, P., Pendharkar, N., Vajgel, P., Kiselev, O.: Using Local Copy Services: How Veritas Stor- age Foundation Snapshot Facilities Protect Data, Reduce Costs, and Enhance the Quality of IT Service. In: Symantec Yellow Books (2006). http://www.symantec. com/en/uk/theme. jsp?themeid=yellowbooks. Accessed on 18/02/2014.
[30] Spillane, R.P., Gaikwad, S., Chinni, M., Zadok, E., Wright, C.P.: Enabling Transactional File Access via Lightweight Kernel Extensions. In: Proceedings of the 7th Conference on File and Storage Technologies, pp. 29–42 (2009)
[31] Verma, S., Miller, T.J., Atkinson, R.G.: Transactional File System. In: US Patent No.6,856,993 (2005). http:// www.google. com/patents/US20050149525. Accessed on 18/02/2014
[32] Wang, J., Hu, Y.: WOLF A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems. In: Proceedings of the 1st Conference on File and Storage Technologies, pp. 47–60 (2002)
[33] Wright, C.P.: Extending ACID Semantics to the File System via ptrace. Ph.D. thesis, Computer Science Department, Stony Brook University (2006). Technical Report FSL-06-04

About the Authors

Lipika Deka received her PhD degree in computer science and engineering from the Indian Institute of Technology, Guwahati, in 2013. Currently,she is working as a post-doctoral researcher in the field of intelligent transport systems and autonomous vehicles at Loughborough University, United Kingdom. Her main research interests include Operating Systems, File Systems,Concurrency Control and VANET. She is a member of ACM and BCS.

Gautam Barua is a professor in the Department of Computer Science and Engineering, Indian Institute of Technology, Guwahati, where he haduntil recently held the position of director. He is currently, also acting as mentor director at Indian Institute of Information Technology, Guwahati. His main research interests include operatingsystems and computer networks. He is a member of the IEEE and ACM.