Consistent Online Backup in Transactional File Systems

Lipika Deka and Gautam Barua, *Member, IEEE*

Abstract—The backup taken of a file system must be consistent, preserving data integrity across files in the file system. With file system sizes getting very large, and with demand for continuous access to data, backup has to be taken when the file system is active (is online). Arbitrarily taken online backup may result in an inconsistent backup copy. We propose a scheme referred to as *mutual serializability* to take a consistent backup of an active file system assuming that the file system supports transactions. The scheme extends the set of conflicting operations to include read-read conflicts, and it is shown that if the backup transaction is mutually serializable with every other transaction individually, a consistent backup copy is obtained. The user transactions continue to serialize within themselves using some standard concurrency control protocol such as Strict 2PL. We put our scheme into a formal framework to prove its correctness, and the formalization as well as the correctness proof are independent of the concurrency control protocol used to serialize user transactions. The scheme has been implemented and experiments show that consistent online backup is possible with reasonable overhead.

Index Terms—Online backup, consistency, concurrency control, file system, transaction

1 Introduction

File system backup, an important but often neglected chapter of data management is facing a paradoxical situation. File system sizes have touched peta bytes and are predicted to grow further, and as a result, operations like backup which need to access all the data, are taking very long to complete [1], [2]. Longer backup time means longer system downtime as traditional backup runs on an unmounted file system to capture a consistent file system image, using utilities like *tar*, *dump* etc. On the other hand, the available "backup window" is shrinking with globalization ushering in a round-the-clock business environment requiring data to be available almost all the time. This has now lead industry and academia to propose a number of *online backup* (backup of an active file system) solutions which perform backup on active file systems.

Arbitrarily taken online backup may destroy data integrity in the backup-copy. Shumway [3] details inconsistencies arising in the backup when file operations like *move*, *append*, *truncate*, etc. run concurrent to the backup program. Compromise on data integrity during online backup is 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 online 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 [4], [5]. But this information on the relationship among the files is not available to the file system and it cannot therefore take into account such consistency requirements. Existing online backup solutions [6], [7], [8], [9], [10], [11], [12], [13], [14], 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 [5], [15]. The advent of distributed storage syetems to handle large amounts of data has brought the issues of replication consistency and transaction support to the forefront. BlobSeer [16] is a file system supporting concurrency control to be used with Hadoop. Similarly, Megastore is a system supporting replication and concurrency control on top of BigTable [17]. Tango is a system to implement metadata structures such as those provided by ZooKeeper [18] with transaction support [19]. Lynx is a distributed store providing transaction support [20]. We feel that support for transactions in general purpose file systems will become common in the near future to meet these kinds of requirements. Our approach to a consistent online backup solution therefore assumes a transactional file system with transactions being utilized as the mode for specifying consistency requirements. The broad objective of the present research is to understand the problem and propose solutions of consistent online backup through its theoretical and practical treatment. 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 Blobseer, Megastore, Tango and Lynx.

- *L. Deka is currently based at Loughborough University, Loughborough LE11 3TU, UK. E-mail: l.deka@lboro.ac.uk.*
- *G. Barua is with Indian Institute of Technology, Guwahati 781039, India. E-mail: gb@iitg.ernet.in.*

Manuscript received 28 Apr. 2013; revised 4 Jan. 2014; accepted 15 Jan. 2014. Date of publication 22 Jan. 2014; date of current version 26 Sept. 2014. Recommended for acceptance by J. Pei.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TKDE.2014.2302297

To the best of our knowledge, no work has been done on the theoretical aspects of consistent online backup of a file system and in this paper we propose a formal framework for modelling and analyzing the consistent online backup problem and its solution. The formalism for a consistent online backup utilizes a flat file system model accessed through a transactional interface with a transaction comprising of a sequence of logically related file *read* and *write* operations. A *backup transaction* comprises of a sequence of *reads* to every file in the file system and it executes concurrently with other user transactions. *Serializability* being the accepted notion of correctness for concurrently executing transactions, a consistent backup copy is read by *serializing* the backup transaction with the user transactions. But, as the backup transaction reads every file in the file system, making use of standard concurrency control methods like two phase locking and optimistic algorithms, proves to be extremely costly. Hence, assuming user transactions serialize among themselves using concurrency control methods like two-phase locking, we propose and prove that a backup transaction reads a consistent file system copy by keeping itself serialized with each concurrently executing transaction separately (thus establishing a pair wise *mutually serializable* relationship). However, the set of conflicting operations that establishes a *mutually serializable* relationship, differs from the traditional set of conflicting operations as it includes read-read conflicts in addition to the existing read-write, write-read and write-write conflicts. We term this specialized concurrency control mechanism as *mutual serializability*.

We then extend the file system model and show that the concurrency control mechanism, *mutual serializability*, proved for a flat file system model with *read* and *write* as the only two operations can be applied to more general purpose file systems like hierarchical file systems with operations beyond the basic *read* and *write*, such as *truncate*, metadata operations such as *chmod* and *chown*, as well as operations that grow and shrink the file system such as *creat()* and *unlink()*. Further, operations on directories, including moves of entire sub-directories, are consistently handled by using *mutual serializability*.

Section 2 presents a review of the existing online backup solutions in both the file system and database areas, providing justification for our adopted transactional system model. Theory of the adopted system model and correctness proof of the proposed concurrency control mechanism forms the body of Section 3. With Section 4 briefly highlighting on the implementation and evaluation of our proposed solution, we conclude in Section 5.

2 Related Work

Current online backup practices for file systems include 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 within some modern file systems [10], [12], [21] or in disk storage subsystems [6], [7]. The windows volume shadow copy service (VSS) [22] is a facility to allow backup programs to create snapshots of a file system. 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 [6], [8], [14]. A third approach termed *continuous data protection* maintains a backup of every change made to the data [8], [23]. Still other existing online 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 [13].

Now, in order for the online 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 [6], [7], [8], [9], [10], [11], [12], [13], [14] only achieving weaker levels of backup-copy consistency, that is, per file consistency or in very specific cases, application level consistency when aided by the applications themselves [24], [25] or through heuristics [13].

Our approach to a consistent online backup solution assumes a transactional file system [5], [26], [27], [28] with transactions being utilized as the mode for specifying consistency requirements.

Given a file system with transactions, the natural next step would be to borrow online database backup solutions. Today's *hot backup* [29] solutions and its ancestor, the *fuzzy dump* [30] 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 a consistent online backup copy of a file system.

Pu [31], suggested a scheme which considers the online 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. According to the proposed scheme the backup transaction "colours" entities black 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 by Pu [31] failed to consider dependencies within user transactions while reading a consistent copy of the entire database by keeping itself serialized with the user transactions. The drawbacks in Pu's approach were identified and modified by Ammann et al. [32].

The scheme proposed by Pu [31] and modified by Ammann et al. [32] lays the foundation of our approach. But their 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. 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 the ext2 file system 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 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 meta-data (such as inodes in ext2).

3 Theoretical Analysis

In Section 3.1 we formally model the problem of capturing a consistent backup by an online backup utility and use this model to put forward the proposed solution and proof of its correctness. In Section 3.2, we formally show that the *mutual serializability* protocol for capturing a consistent online backup can seamlessly be extended to more practical file systems having multiple file types and file system interfaces having operations beyond the basic *read* and *write*.

3.1 The Formal Model

To study the theoretical aspects of consistent online backup of file systems, we consider a *flat file system* and a file system interface that provides *transactional* semantics. The terminology given below is standard and more formal definitions can be found in [33]. A file system *transaction* consists of a sequence of operations (each of which is assumed to be an atomic action), and each operation accesses a file from a set of file system entities called the *access set* of the transaction. This sequence of operations is called a *schedule* of the transaction. The access set that we first consider consists of either a *read* from a file or a *write* to a file.

When more than one transaction is active, we say that these transactions execute concurrently. It is still assumed that only one operation can execute at a time. So the operations of the concurrent transactions get interleaved in the execution sequence. Such an interleaved sequence of operations is also called a *schedule*. Not all schedules ensure correct operations, and so some concurrency control mechanism, such as *strict2PL*, *optimistic concurrency control* etc. is used to control the execution of operations of different transactions.

A concurrency control mechanism establishes the *serializability* of a schedule where *serializability* is the accepted notion of correctness for concurrent executions. A schedule of a set of concurrently executing transactions is serializable if it is equivalent to a serial schedule of the same transactions. A *serial schedule* is one where all the operations of a transaction occur together with no actions of any other transaction executing in between. It is some concatenation of the individual schedules of each of the concurrently executing transactions. Since each transaction is by itself assumed to be correct, a sequence in which transactions execute one after the other will be correct. If the actual schedule is equivalent to one such sequence of executions, then that schedule will also be correct. A transaction is said to be *serialized* with an already serializable schedule if the operations of this transaction are interleaved with the serializable schedule such that the resulting schedule is also serializable. An operation could be similarly serialized with an already serializable schedule. It must be noted that during actual execution, it is not known what operations will occur in future and so the goal of any concurrency control mechanism is to serialize a new operation with the already serializable schedule that has been executed so far. It may not be possible to allow the execution of the new operation immediately, and so it may be delayed, or one or more transactions aborted. Two operations in a *schedule* are said to be *conflicting operations* if they belong to different transactions, act on the same file and both are not the read operation [33] (so the conflicts can be read-write, write-read, or write-write). Given a set of transactions, two schedules of the same set is said to be *equivalent* if in both schedules all pairs of conflicting operations occur in the same order.

An atomic action of a *backup transaction* is a *read* operation to a file and the *schedule* of a backup transaction consists of a sequence of *read* operations to every file in the file system, where a *read* operation to each file appears at most once. An online backup transaction runs concurrently with 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. 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 will lead to frequent conflicts. Resolving these conflicts through a locking protocol may mean either the gradual delay of all concurrently executing user transactions as the backup transaction proceeds to lock files one by one, or otherwise it may result in repeated aborting and restarting of the backup transaction. Similar problems will occur if optimistic concurrency control mechanisms are used. Hence, we are proposing a backup transaction specific concurrency control mechanism called *mutual serializability* that does not affect the overall performance much.

Mutual serializability achieves an overall serializability of backup and user transactions by keeping the backup transaction *mutually serializable* with every concurrent user transaction simultaneously, 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.

The set of conflicting operations that establishes a *mutually serializable* relationship, differs from the traditional set of conflicting operations as it includes read-read conflicts in addition to the existing read-write, 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, write-read and write-write conflicts. 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.

3.1.1 Basic Terminology

A *file system* $\mathbf{F}$ is a set of distinct files, $\{f_1, f_2, \ldots, f_n\}$. Let $\mathbf{T} \bigcup t_b$ be the set of distinct transactions concurrently

accessing **F** where $\mathbf{T} = \{t_1, t_2 \ldots, t_m\}$ is the set of user transactions and t_b is the *backup transaction*. A transaction $t_x, x = 1, 2, \ldots, m$ consists of a sequence of read and write operations on files. A read operation on a file $f_i, i = 1, 2, \ldots, n$ is denoted by $r_x(f_i)$ while a write operation is denoted by $w_x(f_i)$. The *backup transaction*, t_b backs up a file $f_i \in F$ using the read operation denoted by $r_b(f_i)$ and t_b reads every file $f_i \in \mathbf{F}$ at most once.

Let $a_x(f_i)$ be a generic notation denoting either $r_x(f_i)$ or $w_x(f_i)$. A *transaction* is formally defined as a tuple : $t_{id} = \{id, F_{id}, \pi_{id}\}$ where

- *id* is a positive integer and denotes a globally unique transaction identity number.
- F_{id} is the set of files accessed by transaction t_{id} ($F_{id} \in \mathbf{F}$). If $t_{id} = t_b$ then $F_{id} = \mathbf{F}$.
- π_{id} denotes the execution schedule of t_{id}. π_{id} is a *sequence without repetition* of 0 to $|\sum_{id}|$ elements, over the set $\sum_{id} = \{r_{id}(f_i), w_{id}(f_i) | \forall f_i \in F_{id}\}$.

If $t_{id} = t_b$ then a schedule π_b is a *sequence without repetition* of $|\sum_b|$ elements, over the set $\sum_b = \{r_b(f_i) | \forall f_i \in \mathbf{F}\}$.

A transaction t_{id} is defined to be a *read-only transaction* if π_{id} is a *sequence without repetition* of 0 to $|\sum_{id}|$ elements, over the set $\sum_{id} = \{r_{id}(f_i) | \forall f_i \in F_{id}\}$.

Example. $\mathbf{F} = \{f_1, f_2, \ldots f_{11}\}$, $\mathbf{T} = \{t_1, t_2\}$. $t_1 = \{1, F_1, \pi_1\}$, $F_1 = \{f_4, f_8, f_{11}\}$ and $\pi_1 = r_1(f_{11}), r_1(f_4), w_1(f_8)$. $t_2 = \{2, F_2, \pi_2\}$, $F_2 = \{f_1, f_4, f_{11}\}$ and $\pi_2 = r_2(f_1), w_2(f_{11}), w_2(f_4)$. $t_b = \{b, F_b, \pi_b\}$, $F_b = \mathbf{F}$ and $\pi_b = r_b(f_1), r_b(f_2), \ldots, r_b(f_{11})$.

Let, π_C be the execution schedule of the set of concurrently executing transactions, **T**. It must be noted here that π_C is possibly an interleaved schedule of the individual schedules of set **T**. Formally, π_C is a sequence of elements containing every element in the schedules $\pi_1, \pi_2, \ldots, \pi_m$ exactly once and such that:

- *$a_x(f_i) < a_x(f_j)$ in π_C if and only if $a_x(f_i) < a_x(f_j)$ in π_x where, $f_i, f_j \in F_x$ and $t_x \in T$. Symbol " < " is used to denote "precedes" in any schedule (above, $a_x(f_i)$ precedes $a_x(f_j)$).*

Each π_x, $\mathrm{x} = 1, 2 \cdot \cdot \mathrm{m}$ is said to be a participant schedule of π_C.

A pair of access operations $a_x(f_i)$, $a_y(f_j) \in \pi_C$ are said to be conflicting if the following conditions are met:

- $x \neq y$
- $f_i = f_j$
- $<a_x, a_y> \in \{ <r_x, w_y>, <w_x, r_y>, <w_x, w_y> \}$,

A schedule π_C is a serialized (or serializable) schedule if it is *conflict equivalent* to some serial schedule of the set **T**. Formally, a schedule π_C is a serialized schedule, if for every pair of conflicting operations $\{a_x(f_j), a_y(f_j)\}$ in π_C, either

- $\{a_x(f_i) < a_y(f_i) | \forall f_i \in (F_x \cap F_y)\}$

OR

- $\{a_y(f_i) < a_x(f_i) | \forall f_i \in (F_x \cap F_y)\}$.

Let π_{Cb} denote the execution schedule of $\mathbf{T} \bigcup t_b$, which is basically an interleaved schedule of π_b and π_C. Formally, π_{Cb} is a sequence of elements containing every element in π_b and π_C exactly once and such that:

- *$a_x(f_i) < a_y(f_j)$ in π_{Cb} if and only if $a_x(f_i) < a_y(f_j)$ in π_C or π_b where, $f_i, f_j \in \mathbf{F}$ and $t_x, t_y \in \mathbf{T} \bigcup t_b$.*
- *π_C is a serialized schedule.*

3.1.2 Mutual Serializability

Given the formal model, we are primarily interested in an execution schedule π_{Cb} that is *serializable* and a *concurrency control mechanism*, that produces such a schedule. We propose a backup transaction specific concurrency control called *mutual serializability* for serializing t_b with respect to π_C to achieve a serialized π_{Cb}, while user transactions continue to use standard concurrency control protocols to obtain serializable π_C. As discussed earlier it is infeasible for t_b to use such standard concurrency control mechanisms to serialize π_{Cb} because of t_b's uniqueness in that it reads every file in the *file system*. We define the term *mutually serializable*:

Mutually serializable. Let, concurrently executing transactions $t_x, t_y \in (\mathbf{T} \cup t_b)$ access (read or write) the set of files $F_x, F_y \in \mathbf{F}$ respectively. Given, $F_x \cap F_y \neq \emptyset$, t_x and t_y are *mutually serializable* if

- *t_x and t_y are both read-only transactions*

OR

- *for every pair of access operations $a_x(f_i), a_y(f_i)$ where $f_i \in (F_x \cap F_y)$ and*
  $<a_x, a_y> \in \{ <r_x, r_y>, <r_x, w_y>, <w_x, r_y>, <w_x, w_y> \}$, either
  - $\{a_x(f_i) < a_y(f_i) | \forall f_i \in (F_x \cap F_y)\}$
    or
  - $\{a_y(f_i) < a_x(f_i) | \forall f_i \in (F_x \cap F_y)\}$.

We see that while defining the *mutually serializable* relationship between concurrently executing transaction pairs, the traditional set of conflicting operations is extended to include read-read conflicts. By doing so, we observe that, π_{Cb} can be serialized by keeping t_b *mutually serializable* with each $t_x \in \mathbf{T}$ simultaneously, giving rise to the protocol *mutual serializability*. Not considering t_x's read ($r_x(f_i)$) to a file $f_i \in (F_x \cap F_b)$ while resolving conflicts with t_b sometimes leads to the violation of the protocol *mutual serializability*. For example, consider the following schedule, $\pi_{Cb} = r_b(f_1), w_1(f_1), r_2(f_1), w_2(f_2), r_b(f_2)$ of $t_1, t_2 \in \mathbf{T}$ and t_b, where we see that t_b is *mutually serializable* with t_1 and t_2 simultaneously if we do not consider read-read conflicts, but π_{Cb} is not a serialized schedule. But, *mutual serializability* holds when considering both read and write access to a file by t_x to conflict with read access to the file by t_b.

Since the definition of a conflict is different, we denote precedence relationship in which the operations of the backup transaction are involved as $r_b \ll a_x$ or $a_x \ll r_b$ and hence also $t_b \ll t_x$.

We state and prove that we get serializable schedules by using *mutual serializability* in Theorem 1, once we state and show Lemma 1. The given proof uses a serialization testing tool called *serialization graph (SG)* which is derived from the schedule of a set of concurrent transactions. Given π_{Cb} is a schedule over $\mathbf{T} \bigcup t_b$, the serialization graph, denoted by SG(π_{Cb}), is a directed graph whose nodes are the transactions in $\mathbf{T} \bigcup t_b$ and whose edges are all $t_x \rightarrow t_y$ ($x \neq y$ and

$t_x, t_y \in \mathbf{T} \bigcup t_b$) such that one of t_x's operations precedes and conflicts with one of t_y's operations in π_{Cb}. Having defined a serialization graph, a history π_{Cb} is serializable if and only if its serialization graph (SG(π_{Cb})) is acyclic [34]. If there is an edge in SG of the form $t_x \rightarrow t_y$, then t_x is said to *precede* t_y and the relationship is denoted as $t_x < t_y$. In case the backup transaction is involved, the relationship is denoted by $t_b \ll t_x$ or $t_x \ll t_b$.

Lemma 1. *Given that π_{Cb} is a schedule over $\mathbf{T} \bigcup t_b$, where $t_1, t_2, \ldots, t_m$ are transactions in set **T** and t_b the backup transaction, and given that t_b is mutually serializable with every $t_i \in \mathbf{T}$,* **if** *in π_{Cb}, $t_b \ll t_1 < t_2 \cdots < t_m$,* **then** $t_b \ll t_m$.

Proof. Since t_b is mutually serializable with t_m, then either $t_b \ll t_m$ or $t_m \ll t_b$. We need to show that only the former relationship is possible, given the conditions of the lemma. Let us consider the partial relationship $t_b \ll t_1 < t_2$.

Let t_1, t_2 access (read and write) the non-empty set of files F_1 and F_2 respectively. As $t_1 < t_2$, $(F_1 \bigcap F_2) \neq \emptyset$. There must exist at least one file on which the operations of t_1 and t_2 conflict. Without loss of generality, let this be a single file f_1.

So, $a_1(f_1) < a_2(f_1)$ must be in the schedule, where $< a_1, a_2 > \in \{ <r_1, w_2>, <w_1, r_2>, <w_1, w_2> \}$. **(1)**

By definition, t_b reads every file in the file system and hence t_b reads file f_1. Also by *mutual serializability*, a read by t_b conflicts with any access (read or write) by a user transaction to the same file.

Hence, given the relationship $t_b \ll t_1$, $r_b(f_1) \ll a_1(f_1)$ must be in the schedule, where $<r_b, a_1> \in \{ <r_b, w_1>, <r_b, r_1> \}$. **(2)**

From 2 we get $r_b(f_1) \ll a_1(f_1)$. From 1 we get $a_1(f_1) < a_2(f_1)$. Therefore, 1, 2 and the definition of *mutual serializability*, $r_b(f_1) \ll a_2(f_1)$ holds. $a_2(f_1) \ll r_b(f_1)$ cannot hold as t_b reads every file at most once.

Thus, ***if*** $t_b \ll t_1 < t_2$ ***then*** $t_b \ll t_2$. **(3)**

It can be easily seen that the same arguments can be used to prove by induction that the lemma holds. □

Theorem 1. *Given that π_C is a* serialized schedule, *π_{Cb} is* serializable ***if*** *(t_b, t_x is mutually serializable $|\forall t_x \in T$).*

Proof. We use mathematical induction to show that SG(π_{Cb}) is acyclic and hence π_{Cb} is a serialized schedule.

We proceed with the knowledge that π_C is a serialized schedule and hence there cannot exist a cycle in SG(π_{Cb}) involving only transactions from the set **T**.

Base case 1. Let us consider a single transaction say $t_1 \in T$ and the backup transaction t_b. Given that t_b and t_1 is *mutually serializable*, a cycle in SG(π_{Cb}) involving just the two transactions t_b and t_1, is not possible.

Base case 2. Let us now consider any two transactions $\{t_1, t_2\} \in T$ and t_b.

For a cycle to hold, t_b must be part of the cycle. There is no cycle involving t_1 and t_2 as it is assumed that the operations of t_1 and t_2 are interleaved to form a serializable schedule.

Let there be a cycle, $t_b \ll t_1 < t_2 \ll t_b$.

Let t_1, t_2 access (read and write) the non-empty set of files F_1 and F_2 respectively. As $t_1 < t_2$, then $(F_1 \bigcap F_2) \neq \emptyset$. There must exist at least one file on which the operations of t_1 and t_2 conflict. Without loss of generality, let this be a single file f_1. So, $a_1(f_1) < a_2(f_1)$ must be in the schedule, where $<a_1, a_2> \in \{ <r_1, w_2>, <w_1, r_2>, <w_1, w_2> \}$.

Now, by the definition of t_b, t_b reads every file in **F** at most once and it is also given that t_b is *mutually serializable* with all $t_x \in \mathbf{T}$, $F_b = \mathbf{F}$.

Therefore, t_b reads f_1 exactly once and so $r_b(f_1)$ occurs only once in the schedule π_{Cb}. **(4)**

By definition of *mutual serializability*, since $t_b \ll t_1$, t_b reads f_1 before t_1, i.e., $r_b(f_1) \ll a_1(f_1)$ is part of the schedule π_{Cb}.

Again by definition of *mutual serializability*, since $t_2 \ll t_b$, t_b reads f_1 after t_2 and so $a_2(f_1) \ll r_b(f_1)$ is part of the schedule π_{Cb}.

But this implies, $r_b(f_1) \ll a_1(f_1) < a_2(f_1) \ll r_b(f_1)$ is part of the schedule π_{Cb}. But, this is not possible by 4 above.

Therefore, there cannot be any cycle involving any two transactions and t_b .

Induction hypothesis. Let there be m transactions, $t_1, t_2, \ldots, t_m$ and the backup transaction t_b. Let there be no cycles present.

Induction step. Let there be $\mathrm{m}+1$ transactions $t_1, t_2, \ldots, t_m, t_{m+1}$ and the backup transaction t_b. For a cycle to exist, due to the induction hypothesis, all transactions must participate.

So, $t_b \ll t_1 < t_2 \cdots < t_m < t_{m+1} \ll t_b$.

But by Lemma 1, $t_b \ll t_m$ and so the cycle $t_b \ll t_m < t_{m+1} \ll t_b$ must also exist. But a cycle involving two transaction and t_b is not possible as proved in the base case.

Hence, there can be no cycle with m+1transactions. So, $t_b \ll t_1 < t_2 \cdots < t_m < t_{m+1} \ll t_b$ is not possible.

Thus, if (t_b, t_x is mutually serializable $|\forall t_x \in T$), π_{Cb} will be a *serialized schedule*.

It may also be noted if read-read conflicts are not taken into consideration in the *mutual serializability* protocol, then a cycle is possible. Thus, if $a_1 = r_1$, then it is no longer necessary that $r_b(f_1) \ll a_1(f_1)$ and ?? is not violated. However, $t_b \ll t_1 < t_2 \ll t_b$ may hold because there may be some $f_i \in F_1$ such that $w_1(f_i)$ is part of π_1 ($F_1 \bigcap F_2$ is still only f_1) and so $r_b(f_i) \ll w_1(f_i)$ must hold and so $t_b \ll t_1$ will hold. □

The above restriction of extending the traditionally defined set of conflicting operation to include read-read conflicts while establishing the *mutually serializable* relationship, gives us a sufficient condition for serializability of t_b with π_C in π_{Cb}. Not considering read-read conflicts may still result in a serialized schedule as can be seen from the following example: $\{r_b(f_1), r_b(f_2), w_1(f_1), r_2(f_2), w_2(f_3), r_b(f_3)\}$. However, *mutual serializability* gives us a simple way of ensuring serializability, and as we shall see, it enables us to handle other file operations easily.

Imposing read-read conflicts does not lead to any loss of concurrency of running transactions as among these transactions no such restrictions are imposed. They just may have to "slow" down a little to accommodate the backup transaction when it is running. This is a small price to pay to obtain a consistent backup.

3.1.3 A Read and Write Operation

Within a transaction a file can be read or written any number of times. 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. This lead us to merge multiple file accesses (read or write) into a single access within a *schedule* in the formal model.

Similarly, 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*. Hence a *schedule* lists the order in which operations were successfully "locked" or "opened".

3.2 Practical File Systems

Practical file systems are much more complex than the simple flat file systems of our model in Section 3.1, having multiple file types including regular files, directories, special files etc., and file system interfaces have operations beyond the *read* and *write* operations such as, *link*, *unlink* etc. as well as file descriptor operations like updating ownership information, file modification times etc. In addition, practical file systems are not static as assumed in the flat file system model but dynamic, allowing new files to be inserted and existing files to be deleted. Moreover, files in most practical file systems are organized to better reflect the real world and for the ease of searching. This results in inter-file logical relationships, such as the parent-child relationship of hierarchical file systems.

In this section, we formally show that the proposed *mutual serializability* scheme will continue to work when considering dynamic file systems with multiple file types exhibiting inter file logical relationships as seen in a hierarchical file system. We conjecture that the scheme will also work for any file system currently in existence. We use Linux terminology to describe file operations and entities.

3.2.1 Hierarchical File System

To extend the theory of consistent online backup of flat file systems to practical file systems we consider hierarchical file systems, for example, ext2fs. In general, *regular files*, *directories*, *symbolic links*, and *special files* like pipes and sockets make up the different types of files in a hierarchical file system. For the purpose of backup, we restrict ourselves with the backup of regular files and directories, both to be simply addressed as *file* unless otherwise stated. In the case of *symbolic links* there is no guarantee that a link exists or has long been broken, and so we will treat symbolic links as regular files and only store the link in the backup copy. Thus, the hierarchical file system is essentially composed of regular files and directories.

All regular files and directories are logically organized into a multi-level hierarchy called a directory tree, with a top directory called the *root* and denoted by a / (slash). Regular files always occupy leaves of the hierarchy. A directory contains the names and identifiers of a group of files and subdirectories. Every file has a unique identifier in a file system, called an inode number in ext2fs, which allows access to the file. The container directory is termed as the "parent" and the contained regular files and subdirectories are termed as "children" of the parent directory thus resulting in a "$parent \rightarrow child$" relationship between a parent directory and its children files. Just as children are referenced from parent directories through each child's file identifier, parents are referenced from each of their child sub-directories through its unique identifier. This identifier is stored in the ".." entry of a directory in ext2fs. A regular file can be a child to many directories, with a files inode storing the number of parents.

A file in a hierarchical file system is accessed using its *pathname* where a *pathname* is a sequence of names of files where adjacent names are in a $parent \rightarrow child$ relationship, and the last name is the name of the file itself with respect to its parent. Accessing a file essentially involves traversing "$parent \rightarrow child$" relationships as indicated by the pathname to finally locate the required file, by an operation called the *path lookup* operation. In Section 3.2.2 and Section 3.2.3, we assume that file pathnames have been resolved and these resolved filenames are passed as arguments to operations. File pathnames and the effect of *path lookup* operation on the consistency of backup shall be dealt with in more details in Section 3.2.5.

Each file has some data associated with it as well some attributes. These attributes are stored in the inode corresponding to the inode number of the file. Even though the inode of a file and its data are usually stored as separate entitities in a file system, they are not exposed as separate entitities. So all operations on the inode of a file are considered as operations on the file.

A practical hierarchical file system is unlike the file system model detailed in Section 3.1 as such file systems do not consist of a fixed set of files throughout their lifetime. They are dynamic with new files being inserted and existing ones deleted. A hierarchical file system transaction accesses files through a number of operations that keep the file system static for example, *link*, *rename* etc., and those that grow or shrink the file system for example, *creat*, *unlink* etc., each accessing one or more files. We show that *mutual serializability* holds true for hierarchical file systems also, by mapping all file operations of hierarchical file systems to an equivalent sequence of one or more *read* and *write* atomic file access operations. The mappings are detailed in Section 3.2.3.

Coming to the operations that grow/shrink the file system, though the *mutual serializability* protocol extends normally for the file *delete* operations, it does not do so for the operations that *inserts* new files. Extending the file access operations of our model to include a file create operation(denoted by creat_node) apart from the *read* and *write* access operations and also extending the set of conflicting operations to include the conflicts with the creat_node operation, will ensure that the *mutual serializability* protocol remains applicable. This is formally stated and proved in Section 3.2.4.

3.2.2 Hierarchical File System's Formal Model

Formally, a *Hierarchical File System* $\mathbf{F^h}$ is a set of distinct files, $\{f_1, f_2, \ldots, f_n, d_1, d_2, \ldots, d_p\}$ where f_i, $\mathrm{i} = 1, 2, \ldots, \mathrm{n}$ is a regular file and d_j, $\mathrm{j} = 1, 2, \ldots, \mathrm{p}$ is a directory. However, now the number of files, n, and the number of directories, p, can change over time as files and directories may be created

and deleted in the system. Thus, $\mathbf{F^h}$ is the universe of all files that may exist in the file system during its lifetime. Let $file$ be a common notation used for all types of files namely a regular file or a directory. As stated above, in the present model, symbolic links are also treated as regular files. Thus, $file_k \in F^h$, $\mathrm{k} = 1, 2, \ldots, \mathrm{n} + \mathrm{p}$.

It must be noted here that a file system entity $file_k$ is made up of both data and its metadata, that is, the contents of a regular file or directory together with its inode data.

As already stated, files in a hierarchical file system exhibit an inter file hierarchical relationship also termed as a $parent \rightarrow child$ relationship. Let $child(d_j)$ be the set of child files of the directory d_j. Thus, if $file_k \in child(d_j)$ then $file_k$ is a child of the parent directory d_j.

Let $\mathbf{T^h} \bigcup t_b$ be the set of distinct transactions concurrently accessing $\mathbf{F^h}$ where $\mathbf{T^h} = \{t_1, t_2 \ldots, t_m\}$ is the set of user transactions and t_b is the *backup transaction*. A transaction $t_x, x = 1, 2, \ldots, m$ accesses a file $file_k, k = 1, 2, \ldots, n + p$ for reading denoted by $r_x(file_k)$ or writing denoted by $w_x(file_k)$. Both these operations are assumed to be atomic.

We must again remember here that within an atomic *read* or *write*, either the metadata or data or both of $file_k$ may be accessed and access to metadata and data are not considered separate operations.

The *backup transaction*, t_b backs up a file $file_k \in \mathbf{F^h}$ using the read operation denoted by $r_b(file_k)$, where each file $file_k$ is read *at most* once.

As before, let $a_x(file_k)$ be a generic term denoting $r_x(file_k)$ or $w_x(file_k)$, such that $t_x \in \mathbf{T^h}$ and $file_k \in \mathbf{F^h}$.

A hierarchical file system transaction is formally modelled as a tuple : $t_{id} = \{id, F_{id}, \pi_{id}\}$

- *id* is a positive integer and denotes a globally unique transaction identity number.
- F_{id} is the set of files accessed by transaction t_{id} ($F_{id} \in \mathbf{F^h}$). If $t_{id} = t_b$ then $F_{id} = \mathbf{F^h}$.
- π_{id} denotes the execution schedule of t_{id}. If $t_{id} \in \mathbf{T^h}$ then π_{id} is a *sequence without repetition* of 0 to $|\sum_{id}|$ elements, over the set $\sum_{id} = \{r_{id}(f_i), w_{id}(f_i)|\ \forall f_i \in F_{id}\}$.

  If $t_{id} = t_b$ then a schedule π_b is a *sequence without repetition* of $|\sum_b|$ elements, over the set $\sum_b = \{r_b(file_k)|\forall file_k \in \mathbf{F^{hb}}\}$. Where $\mathbf{F^{hb}}$ is a set of distinct files such that $\mathbf{F^{hb}} = \mathbf{F^h} - (F^{delete} \cup F^{inserted})$. Where, $\{F^{delete}$ is the set of files deleted by $t_i|\forall t_i \in T, t_i \ll t_b\}$, and $\{F^{insert}$ is the set of files created by $t_j|\forall t_j \in T, t_b \ll t_j\}$.

  A transaction t_{id} is defined to be a *read-only transaction* if π_{id} is a *sequence without repetition* of 0 to $|\sum_{id}|$ elements, over the set $\sum_{id} = \{r_{id}(f_i)|\ \forall f_i \in F_{id}\}$.

Let, π_{C^h} be the serialized execution schedule of the set of concurrently executing transactions $\mathbf{T^h}$, where π_{C^h} follows the definition of π_C.

Given π_{C^h} and π_{b^h}, $\pi_{C^h b^h}$ denotes the execution schedule of $\mathbf{T^h} \bigcup t_b$, which is basically an interleaved schedule of π_{b^h} and π_{C^h} and follows the given formal definition of π_{Cb}.

Consistent online backup of F^h. As stated by the proposed *mutual serializability* protocol, by including read-read conflicts to the set of traditional conflicting operations which comprises of read-write, write-read and write-write conflicts, $\pi_{C^h b^h}$ can be *serialized* and hence a consistent online backup of F^h be obtained by keeping t_b *mutually serializable* with each $t_x \in \mathbf{T^h}$.

3.2.3 Mapping of Hierarchical File System Operations

Mappings for a few file system calls in Linux to an equivalent sequence of *read* and *write* operations are given below. Other system calls can be similarly mapped and hence have not been explicitly given due to space limitations.

Now, hierarchical file system operations involve directory operations apart from regular file operations. It is easy to see that an access to a regular file in a hierarchical file system corresponds directly to an access of a file in our flat file system model, but correspondence of an access operation to a directory to an access operation to a file of our flat file system model may not be so obvious. Directories are nothing but special files with an internal structure more suitable for insertion, deletion and searching for its child files and subdirectories. Hence, an access operation to a directory maps directly to an access operation to a file and no matter how directory operations conflict within user transactions, with respect to a backup transaction, any access to a directory by a user transaction conflicts with the read access of the directory by the backup transaction.

We broadly divide the file system operations into two categories, those that keep the data set fixed or static and those that grow or shrink the data set. We begin by looking at the *read* operation by the backup transaction.

The backup transaction's read operation. A file is the unit of backup in a file system. A file is made up of its data contents and its associated attributes. The associated attributes of a file is stored in a separate data structure called the inode. The *read* of a file by the backup program involves the *read* of the inode contents by a *stat* system call (which is mapped to a *read* operation) and a *read* of the file's data contents by the *read* system call. The two reads to the same entity is merged into a single read in our theoretical model as reasoned in Section 3.1.3.

Operations maintaining a fixed data set. The read of a regular file and the write to a regular file, f_i can trivially be mapped to the read operation, $r_x(f_i)$ and the write operation, $w_x(f_i)$ respectively of our model. The operation *truncate* of f_i effects only f_i with updates performed in its inode and hence is mapped to a *write* operation, $w_x(f_i)$.

Reading of a directory (*readdir*) also maps directly to a *read* operation to the directory which is nothing but a specially formatted file and hence is equivalent to the *read* of a file of our flat file system model.

The *link* operation is used to create a new hard link to a regular file f_i from a directory d_{j1}. As a result of a *link*, updates are done in the inode and directory entry of d_{j1} and the inode of f_i. Hence, we model the mapping of *link* to $\{w_x(d_{j2}), w_x(f_i)\}$. Directories already containing links to this file are not affected.

For a *rename/move* operation, say a file $file_k$ is "moved" from directory d_{old} to d_{new} by t_x. Of course d_{old} may be the same as d_{new} in cases where the name of the file is changed rather then the entire path to the file. Now, *rename/move* operation is effectively an *unlink* of $file_k$ from d_{old} followed by a *link* of $file_k$ to d_{new}. This is achieved by three write

operations for a directory rename: *write* on d_{old} to delete entry of $file_k$, *write* on d_{new} to create an entry for $file_k$ and a *write* on $file_k$ where the ".." entry is modified (in case of $file_k$ being a directory). For a regular file rename, in many implementations neither the file nor its inode are changed. But if the destination file exists, then the Posix standard states that the old file is deleted. Without a change in access time, a "new" file can become "older" as in some implementations, the st_ctime (creation time) file of the inode's metadata is changed. Further since a file can only be accessed through its name, and its name is getting changed, including a *write* to the file in our mapping will take care of these issues. The *rename/move* operation is hence mapped as $\{(w_x(d_{old}), w_x(file_k), w_x(d_{new})\}$.

Inode operations mostly deal with reading or modifying file attributes like owner, size, count, modification time etc. As already mentioned, *read* and *write* of a file's metadata is a *read* and *write* to the file itself as the file and its metadata are considered as a single entity. Most inode operations can be directly mapped to either a *read* or *write* of the file. For example, **chmod()** is used to set file permissions and the mapping of chmod() acting on a file $file_k$ is simply a *write* to the file i.e., $w_x(file_k)$. Similarly, **stat()** gets a files status and fills a given buffer with the status information. stat() of a file $file_k$ directly corresponds to the *read* of the inode information and hence the file i.e., $r_x(file_k)$.

Operations that grow/shrink the data set. We now map operations that delete existing files and insert new files and look into how the mutual serializability protocol extends to a dynamic file system that grows and shrinks and allows an online backup utility to capture a consistent file system state.

File deletion. Removal of a regular file, f_i, via the $unlink()$ operation from a directory d_j requires the deletion of the file entry in the parent directory, d_j and a decrement of the link count in f_i's inode. The *regular file* deletion operation maps as $\{w_x(d_j), w_x(file_k)\}$. Similarly, removing a directory d_{j2} results in updating the parent directory d_{j1} by removing the reference to d_{j2} and then removing d_{j2}. Hence, mapping of the *directory* removal operation, is $\{w_x(d_{j1}), w_x(d_{j2})\}$. rmdir() removes a directory only if it is an empty directory and this fact is significant while considering path lookups, as discussed below. If any file system allows removal of a non-empty directory, it has to specify that it will be done atomically as otherwise our mapping will fail. From a user's perspective too, this is reasonable, as a non-atomic directory removal will make it difficult to handle sharing of files.

Let the deleted file (regular or directory) be referred to as $file_k$. Now, whether t_x is serialized "before" t_b and $file_k$ is never copied to backup or t_x is serialized "after" t_b and $file_k$ has been copied to backup, the *mutual serializability* protocol will work correctly since $file_k$ is not accessed by any other user transaction after its deletion by t_x.

File creation. Creation of a file, $file_k$ ($creat(file_k, d_j)$ for a regular file and $mkdir(file_k, d_j)$ for a directory) under a directory d_j requires an entry of the file name and its unique identifier (inode number in this case) in the directory d_j and allocation of an inode and its initialization for the created file $file_k$. Let the corresponding mapping of *file* creation be $\{w_x(d_j), w_x(file_k)\}$.

Now, does *mutual serializability* extend naturally to support creation of files and hence a growing file system? The answer is no as shown by the following example: As before, symbol " $<$" is used to denote "precedes" in any schedule for user transaction operations, and when the backup transaction is involved the symbol is "$\ll$". Say, $t_x \ll t_b$. So t_b has read d_j before $file_k$ was created. Now, let there be another transaction t_y in $\pi_{C^h b^h}$ which accesses the newly created $file_k$ and another file $file_l$, where $file_l$ has not been read by t_b. Since t_y accesses $file_k$ which was created by t_x, $t_x < t_y$ holds. As both $file_k$ and $file_l$ have not been read by t_b, mutual serializability between t_b and t_y should be $t_y \ll t_b$. But, $t_y \ll t_b$ cannot hold because t_b will never read $file_k$, as it has been created "after" t_b.

The following section considers the presence of file *create* operations. We revisit the *mutual serializability* protocol and modify it to handle file *creates*.

3.2.4 Enhanced Mutual Serializability

To extend the *mutual serializability* backup specific concurrency control protocol so that it efficiently handles insertion of new files into a file system, we extend the set of atomic operations accessing files in a file system to include an operation for creating a file, termed as $creat\_node$. $creat\_node$ is different from the file system call *creat* in that a *creat* includes an update of the parent directory (d_j here) and the creation of a file $file_k$. Thus, *creat()* is essentially $\{w_x(d_j), creat\_node_x(file_k)\}$.

Formally, now $a_x(file_k)$ is a generic term denoting $r_x(file_k)$,$w_x(file_k)$ or $creat\_node_x(file_k)$, such that $t_x \in \mathbf{T^h}$ and $file_k \in \mathbf{F^h}$.

With the introduction of a new operation, the set of conflicting operations must be extended while continuing to adhere with the definition of a conflict which states that two operations conflict if they belong to different transactions, act on the same file and both are not *read* operations [33]. Thus, the set of conflicting operations based on which user transactions are serialized, now also includes, *creat_node-read*, *read-creat_node*, *creat_node-write* and *write-creat_node* conflicts. A *read* by t_b conflicts with a *creat_node* operation of a user transaction on the same file.

The *mutual serializability* definition now also needs modification. Again, we begin by redefining *mutually serializable* with the only difference from its original definition being the set of access operations which now includes *creat_node*.

Mutually Serializable. Let concurrently executing transactions $t_x, t_y \in (\mathbf{T} \cup t_b)$ access (read, write or creat_node) the set of files $F_x, F_y \in \mathbf{F}$ respectively. Given, $F_x \cap F_y \neq \emptyset$, t_x and t_y are *mutually serializable* if

- t_x and t_y are both *read-only* transactions

OR

- for every pair of access operations $a_x(file_j), a_y(file_j)$ where $file_j \in (F_x \cap F_y)$ and $<a_x, a_y> \in \{<r_x, r_y>, <r_x, w_y>, <w_x, r_y>, <w_x, w_y>, <creat\_node_x, r_y>, <r_x, creat\_node_y>, <creat\_node_x, w_y>, <w_x, creat\_node_y>\}$, either
  - $\{a_x(file_i) < a_y(file_i) | \forall f_i \in (F_x \cap F_y)$,
    or
  - $\{a_y(file_i) < a_x(file_i) | \forall f_i \in (F_x \cap F_y)$.

We see that while defining the modified *mutually serializable* relationship between concurrently executing transaction pairs t_x and t_y, if $t_x \ll t_y$ then the following sequence of operations cannot occur: $w_x(file_i) \ll creat\_node_y(file_i)$ and $r_x(file_i) \ll creat\_node_y(file_i)$. Similarly if $t_y \ll t_x$, the following sequence of operations cannot occur: $w_y(file_i) \ll creat\_node_x(file_i)$ and $r_y(file_i) \ll creat\_node_x(file_i)$. It is simply because, a file cannot be read or written into before it is created. Given these basic definitions we observe that, $\pi_{C^hb^h}$ can be serialized by keeping t_b *mutually serializable* with each $t_x \in \mathbf{T^h}$ simultaneously, provided there is no $t_y \in \mathbf{T^h}$ such that $t_b \ll t_y$ and $creat\_node_y(file_k)$ in π_y **if** there is a $t_x \in \mathbf{T^h}$ such that t_x reads or writes into $file_k$ and $t_x \ll t_b$. Thus, this gives rise to a modified version of the protocol *mutual serializability*. We state and prove the modified *mutual serializability* protocol in Theorem 2.

Theorem 2. *Given π_{C^h} is a serializable schedule, $\pi_{C^hb^h}$ is serializable* ***if*** *$\forall t_x \in \mathbf{T}$ the following hold:*

1. *t_b, t_x are* mutually serializable *and*
2. *if $t_x \ll t_b$, then there is no $t_y \in \mathbf{T}$ such that*
   a. *$t_b \ll t_y$, and*
   b. *there is a $creat\_node_y(file_k)$ in π_y, and $creat\_node_y(file_k) < a_x(file_k)$ for some $a_x(file_k)$ in π_x where $a_x \in \{r, w\}$.*

Proof. The proof of Theorem 2 is similar to the proof of Theorem 1, once we show that Lemma 1 continues to hold with the set of file access operations now including *creat_node* and the set of conflicting operations extended to include the conflicts with the *creat_node* access operation. Hence, we only show here that the following statement of Lemma 1 holds:

A: t_b is mutually serializable with every $t_i \in \mathbf{T^h}$, ***if*** $t_b \ll t_1 < t_2 < \cdots < t_m$, ***then*** $t_b \ll t_m$.

Since t_b is mutually serializable with t_m, then either $t_b \ll t_m$ or $t_m \ll t_b$. We need to show that only the former relationship is possible, given **A**. Let us consider the partial relationship $t_b \ll t_1 < t_2$.

Let t_1 and t_2 access (read,write or creat_node) the nonempty set of files F_1 and F_2 respectively. As $t_1 < t_2$, $(F_1 \bigcap F_2) \neq \emptyset$. There must exist at least one file on which the operations of t_1 and t_2 conflict. Without loss of generality, let this be a single file f_1.

So, $a_1(f_1) < a_2(f_1)$ must be in the schedule, where $< a_1, a_2 > \in \{ < r_1, w_2 >, < w_1, r_2 >, < w_1, w_2 >, < creat\_node_1, r_2 >, < creat\_node_1, w_2 > \}$ **(5)**

We need to consider two separate cases:

Case I: $< a_1, a_2 > \in \{ < r_1, w_2 >, < w_1, r_2 >, < w_1, w_2 > \}$.

By definition, t_b reads every file in the file system that has not been created during backup (and so f_1 also since it has not been created by t_1 or t_2; the case of another transaction t_i such that $t_b \ll t_z < t_1$ and t_z creates f_1, is covered by Case II below). Hence t_b reads file f_1. Also by *mutual serializability*, a read by t_b conflicts with any access (read or write) by a user transaction to the same file.

Hence, given the relationship $t_b \ll t_1$, $r_b(f_1) \ll a_1(f_1)$ must be in the schedule, where $< r_b, a_1 > \in \{ < r_b, w_1 >, < r_b, r_1 > \}$. **(6)**

From (6) we get $r_b(f_1) \ll a_1(f_1)$. From (5) we get $a_1(f_1) < a_2(f_1)$. Therefore, by (5), (6) and the definition of *mutual serializability*, $r_b(f_1) \ll a_2(f_1)$ holds. $a_2(f_1) \ll r_b(f_1)$ cannot hold as t_b reads every file at most once.

Thus,

if $t_b \ll t_1 < t_2$ ***then*** $t_b \ll t_2$. **(7)**

Case II: $< a_1, a_2 > \in \{ < creat\_node_1, r_2 >, < creat\_node_1, w_2 > \}$.

In this case, $r_b(f_1)$ will not exist in the schedule as $t_b \ll t_1$ and file f_1 has been created by t_1 which is after t_b. Now, if t_2 only accesses f_1, then F_b intersect F_2 is null and so by definition of *mutually serializable* t_b is *mutually serializable* with t_2 as there is no dependency between t_b and t_2. So $t_b \ll t_2$ holds.

If F_b intersect F_2 is not null, and is say f_2, then depending upon when t_b reads f_2, $t_b \ll t_2$ or $t_2 \ll t_b$ will hold. But if $t_2 \ll t_b$ holds, then it violates the second condition of the theorem where $t_x = t_2$, and $t_y = t_1$. Therefore, even in this case, $t_b \ll t_2$ must hold.

It is to be noted that the arguments will continue to hold if F_1 intersect F_2 consists of more than one file and the conflicting operations are from any of the pairs of conflicting operations. Either Case I or Case II or both will have to hold, and the same arguments will apply.

It can be easily seen that the same arguments can be used to prove by induction that **A** continues to hold.

Having shown this, the proof of Theorem 2 is similar to the proof of Theorem 1 and is omitted here. □

3.2.5 Path Lookups

One of the issues that need to be considered in a hierarchical file system is that, when a user transaction reads a file, the file's name can be a pathname, and so to access the file, the directories in the path have to be read. Are these reads to be considered as "reads" by the user transaction? If so, it can create difficulties, as the backup transaction has to either read all these directories before the user transaction or has to read them all after the user transaction. This can severely restrict the degree of concurrency. But user transactions do not read directory contents one by one to resolve a pathname. Pathname traversal is done inside the kernel within a system call. Such a traversal may not be able to proceed if a directory is locked by another user transaction, but this is not visible to the user transactions or the backup transaction. So these directory reads by the kernel do not need to be considered to be reads by the user transaction. Now t_b, the backup transaction captures the backup of the entire file system and not of any particular file. So neither is it given any file pathname to backup, nor does it do any file name lookups during the backup of the file system. It does have to traverse the directory tree, but only to go from one node to another. Which node it next goes to does not matter, and it will go to a node only if it is accessible from the current node. The pathname of a file that is backed up in the backed-up copy will depend on the order in which the backup is being done and on what operations user transactions have executed. Suppose the pathname in the filesystem of the file "b" at the moment it is being backed up is /usr/a/b. If the backup programme has already backed up directories "usr" and "a", then the path name in the backed-

up file system will also be /usr/a/b. Any operations on "usr" or "a" by any user transaction will be after t_b and so it will not show up in the backed up file system. If, on the other hand directories "usr" and "a" have not yet been backed up, and a subsequent transaction t_x moves directory "a" from "usr" to "etc", then with reference to directory "usr" t_x will be before t_b and so the pathname in the backed-up copy will be /etc/a/b.

3.2.6 Realizable Schedule

Our model has assumed that t_b reads every file at most once, because this is what a backup means. In an actual implementation, transactions including t_b may have to abort (roll back). A transaction aborts either on its own account, to resolve conflicts or deadlocks, or as a result of reading dirty data written by a transaction that later aborts.

We define a *realizable* schedule as one in which t_b will never have to *roll back*. To understand why this restriction is required, consider the following example. Let the following be part of some schedule: $\{\ldots, w_1(f_1), r_b(f_1), w_2(f_1), r_b(f_2), w_2(f_2), w_2(f_3), \ldots\}$. As can be seen, t_1 precedes t_b in serialization order. Suppose after t_b has read f_1, t_1 aborts due to some reason. This renders t_b's read of f_1 inconsistent. t_b only *reads* every file in the file system exactly once and hence t_b's consistency can be re-established by a partial *roll back* of abandoning t_b's read of f_1 and reading it again. But this will make $t_b \gg t_2$ on the basis of f_1, while $t_b \ll t_2$ due to f_2. So the read by t_b of f_2 will also have to be rolled back. In the worst case scenario t_b has to *roll back* all its reads and restart from the beginning. Such *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. Hence, for practical purposes we need to consider only *realizable schedules*.

We assume that the backup transaction t_b does not *roll back* on its own account. Hence, t_b would require to *roll back* if another transaction it reads from goes on to abort (t_b becomes a victim of the "cascading abort" phenomenon) or it is forcefully *rolled back* to resolve a conflict or deadlock.

Standard serializability theory deals with cascading aborts by allowing transactions to read only those values that are written by a committed transaction or itself. The resulting serializable schedule is called a *cascadeless schedule*. Let the last operation of every transaction t_x be $commit_x$. This identifies the commit point of the transaction. Then, a schedule is said to be *cascadeless* if, whenever t_x reads $file_k$ from t_y, $(x \neq y)$, $commit_y < r_x(file_k)$. A transaction t_x is said to read data item $file_k$ ***from*** t_y, if t_y was the transaction that had last written into $file_k$ before t_x read $file_k$.

Roll back of transaction t_b as a result of a user transaction t_x aborting can be avoided by applying similar restrictions on the schedule $\pi_{C^h b^h}$. Thus, if t_b reads data written by a user transaction t_x, it does so only after t_x commits.

A need for a transaction to abort or roll back (in case of t_b) also arises when it is chosen as a "victim" to resolve a conflict or a deadlock scenario. Thus, if t_b is never chosen as a "victim" whenever t_b conflicts with a $t_x \in \mathbf{T^h}$ then t_b will not need to *roll back*.

We state and prove these results in Theorem 3.

Theorem 3. *When a backup transaction t_b operates concurrently with a set of transactions T^h that are serialized among themselves, and where t_b is mutually serializable with t_x $(\forall t_x \in T^h)$, t_b will never have to* roll back *any of its read operations if the following conditions hold:*

1. *whenever t_b reads $file_k \in \mathbf{F^h}$ **from** $t_x \in \mathbf{T^h}$, $commit_x \ll r_b(file_k)$ holds*
2. *if t_b and t_x conflict, it is always t_x which resolves the conflict and never t_b.*

Proof. By the definition of a cascadeless schedule, condition 1 ensures that t_b will not roll back even if any transaction $t_x \in \mathbf{T^h}$ aborts.

Condition 2 ensures that t_b is never chosen as a victim when there is a conflict with any $t_x \in \mathbf{T^h}$.

Thus, t_b never needs to *roll back* any of its operations. □

One way of enforcing condition 1 of the above theorem is to ensure that the concurrency control protocol which is used to serialize the transactions in $\mathbf{T^h}$ does not expose uncommitted writes or creates to other transactions, including the backup transaction. Strict two phase locking or strict timestamp based concurrency control protocols fulfill this criterion.

4 Implementation

In order to implement and evaluate the proposed consistent online backup facility, we first implemented our own basic transactional file system, referred to as TxnFS. A need to develop TxnFS arose because current transactional file systems either exist as research projects still under development and evaluation [5], [28] or are closed-source systems limited to a specific file system and operating system [27]. TxnFS has been built as a user level file system using an underlying ext2 Linux file system to actually store information. A backup utility which traverses the file system hierarchy in a depth-first manner *copying* every file on its path to a backup medium has been developed over TxnFS. To serialize the backup utility with concurrently executing application transactions, the *mutual serializability* concurrency control protocol has been implemented.

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 implementations 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. *Mutual serializability* is realized using two bits, one reserved in a file's metadata called the *read* bit (we utilized the sticky bit of an inode to implement the *read* bit) and another bit called the *before-after* bit stored in a transaction's housekeeping data structure. As the backup utility reads files, it marks a backed up file by changing its *read* bit to 1. A transaction's *before-after* bit is initialized to the value of the *read* bit of the first file it accesses. Thereafter, a

transaction is detected to violate *mutual serializability* if it attempts to access a file whose *read* bit value is not equal to the value of its *before-after* bit. Our implementation maintains mutual serializability by either aborting the user transaction or pausing it for a random period (depending on whether the user transaction is initially serialized before or after the backup transaction respectively) and allowing the backup utility to go ahead and read the files in conflict.

We evaluated the effect of obtaining an online consistent backup using the *mutual serializability* protocol on the performance of user transactions as well as the backup transaction by running a number of simulated transactional file system workloads on the implemented system. As transactional file systems are still at the research stage and not yet widely used in real environments, we could not find any real transactional file system trace to use as input. We therefore simulated workloads of different categories. Synthetically generated traces allow us to isolate interaction patterns and to 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 [35], [36], [37], [38]. The different simulated workloads used for our experiment were:

- the **global** workload which accessed the entire file system hierarchy with equal probability (a worst case scenario).
- the **local** workloads where transactions exhibited spatial locality of access. The set of files were spread over a number of subtrees, and each transaction accessed files of a particular subtree only, each with equal probability, and with equal probability of an access being a read or a write. Within this larger group the following workloads were modeled
  - **50 percent share**, **25 percent share**, **10 percent share** and **0 percent share** workloads: in **50 percent share** for example, 50 percent of the files were shared. This was implemented by dividing the set of files in each subtree into a set of shared and a set of non-shared files. A transaction would access each file in a subtree with equal probability, except that a non-shared file was constrained to be accessed by only one transaction (to study the effect of differing level of inter-transactional sharing on the backup process).
  - the **stat** workload modeled higher percentage of read access (70 percent of *stat* calls in our implementation; this was to model the effect of a higher percentage of reads over writes).
  - the **hot-cold** workload where about 10 percent of files are accessed 90 percent of the time and the rest 90 percent remain mostly "cold" by being accessed only 10 percent of the time [37]. Within this group the **50 percent share-hot-cold** and the **0 percent share-hot-cold** sets were generated consisting of transactions sharing up to 50 percent of the files and not sharing any files among themselves respectively.

TABLE 1
Overheads Due to Online Backup

Workload \ Metrics	% increase in conflicts	% increase in backup time	% decrease in through-put
global	57	161	67.33
0%share	7.5	13.8	9.85
10%share	10.6	39.5	27
25%share	13.5	41.24	27.44
50%share	15	44.5	32.25
0%share-stat	7	12.2	12.2
50%share-stat	14	43	33.8
0%share-hot-cold	2.5	5.7	3.68
50%share-hot-cold	6	7.6	4.37
50%share-hot-cold (heuristic)	2	5	3.4

The metrics used for evaluation were the percentage of transactions conflicting with the backup transaction, time taken for the backup to complete, and the number of transactions completing to *commit* per microsecond during the duration the backup utility was active (the throughput of user transactions). Experiments were run with (referred to as **MS_enabled**) and without (referred to as **MS_disabled**) enabling the *mutual serializability* protocol to evaluate the overhead of capturing a consistent backup copy over an inconsistent one. 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.

Table I shows the results of the experiments as percentage change in the number of user transactions conflicting with the backup transaction, percentage change in the backup time and percentage change in the throughput of the **MS_enabled** as compared to the **MS_disabled** run. It can be seen that the **global** workload gives poor performance with 54 percent increase in the percentage of conflicts, 161 percent increase in backup time and 67 percent decrease in throughput during the **MS_enabled** run as opposed to its **MS_disabled** run). However, this is not a realistic load and only serves as a worst case. As expected, incurred overhead decreases sharply in workloads exhibiting spatial locality of access, within which overheads decrease as the degree of inter-transactional sharing decreases. With **hot-cold** workloads, the performance improves further. A **hot-cold** workload with **50 percent-share** could be considered to be a realistic case as seen from previous file access pattern studies [37]. Here the overheads are very reasonable: about 6 percent increase in conflict percentage, 7.6 percent increase in backup time and 4.37 percent decrease in user transaction throughput. It is to be

noted that an increase in the percentage of reads over writes (by including many more stat calls in the workloads) did not reduce the overheads of backup. This is because the mutual serializability algorithm has to consider read-read conflicts also.

To improve performance further, performance enhancing heuristics which involved diverting the backup program to lesser active regions of the file system on detecting conflicts were implemented and evaluated. We applied these heuristics techniques on the **hot-cold** workload with **50percent share**, as it resembles most closely existing workloads [35]. As can be seen from the table, there was further improvements: only 2 percent increase in conflict percentage, 4 percent increase in backup time and 3.4 percent decrease in user transaction throughput.

5 Conclusion

Data backup is extremely important for the protection of data against its possible loss or corruption, as well for the purpose of retention of old file versions. A backup is consistent when taken of an unmounted and inactive file system. But, a backup may be inconsistent when taken arbitrarily while a file system is up and running. The current study has considered the issue of a consistent online backup in a file system supporting transactions. 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. In this paper we have presented a theoretical framework for the study of online file system backup and have formally shown that extending the set of conflicting operations to include read-read conflicts, 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 user transactions continue to serialize within themselves using some standard concurrency control protocol. The backup transaction does not have to be aborted, and delaying a conflicting transaction for a short while in most cases resolves conflicts. We term this concurrency control protocol as *mutual serializability*. A prototype implementation has shown that the protocol can be implemented efficiently. As transaction support becomes part of file systems, the protocol will allow consistent online backup of large file systems.

Acknowledgments

This study was conducted during L. Deka's tenure at the Indian Institute of Technology Guwahati.

References

[1] V. Henson, A. Gud, A. van de Ven, and Z. Brown, "Chunkfs: Using Divide-and-Conquer to Improve File System Reliability and Repair," *Proc. Second Conf. Hot Topics in System Dependability*, 2006.

[2] J. Gray, "Long Term Storage Trends and You," Talk to Microsoft Storage Gurus in September 2006, http://research.microsoft.com/en-us/um/people/gray/talks/io_talk_2006.ppt, Apr. 2012.

[3] S. Shumway, "Issues in On-line Backup," *Proc. Fifth Conf. Large Installation Systems Administration*, Sept. 1991.

[4] L. Deka and G. Barua, "On-Line Consistent Backup in Transactional File Systems," *Proc. First ACM Asia-Pacific Workshop Systems*, pp. 37-42, Aug. 2010.

[5] D. Porter, O. Hofmann, C. Rossbach, A. Benn, and E. Witchel, "Operating Systems Transactions," *Proc. 22nd ACM Symp. Operating Systems Principles*, pp. 161-176, Oct. 2009.

[6] A. Azagury, M.E. Factor, J. Satran, and W. Micka, "Point-in-Time Copy: Yesterday, Today and Tomorrow," *Proc. IEEE/NASA Conf. Mass Storage Systems*, pp. 259-270, 2002.

[7] C. Chang, Y. Chu, and R. Taylor, "Performance Analysis of Two Frozen Image Based Backup/Restore Methods," *Proc. IEEE Int'l Conf. Electro Information Technology*, May 2005.

[8] A. Chervanak, V. Vellanki, and Z. Kurmas, "Protecting the File System: A Survey of Backup Techniques," *Proc. Joint NASA and IEEE Mass Storage Conf.*, pp. 17-32, Mar. 1998.

[9] R.J. Green, A.C. Baird, and J.C. Davies, "Designing a Fast, On-Line Backup System for a Log-Structured File System," *Digital Technical J. Digital Equipment Corporation*, vol. 8, no. 2, pp. 32-45, Oct. 1996.

[10] D. Hitz, M. Malcolm, J. Lau, and B. Rakitzis, *Method for Maintaining Consistent States of a File System and for Creating User-Accessible Read-Only Copies of a File System*, US Patent No. 5,819,292, 1998.

[11] N.C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley, "Logical vs. Physical File System Backup," *Proc. Symp. Operating Systems Design and Implementation*, pp. 239-249, 1999.

[12] J. Johnson and W. Laing, "Overview of the Spiralog File System," *Digital Technical J.*, vol. 8, pp. 5-14, 1996.

[13] "Preventing Data Loss During Backups due to Open Files" St. Bernard Software's White Paper On Open File Manager, http://www.novell.com/coolsolutions/netware/assets/ofm_white_paper.pdf, Nov. 2003.

[14] R.H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara, "Snapmirror: File-System-Based Asynchronous Mirroring for Disaster Recovery," *Proc. First USENIX Conf. File and Storage Technologies*, 2002.

[15] C.P. Wright, "Extending ACID Semantics to the File System via ptrace," PhD dissertation, Computer Science Dept., Stony Brook Univ., May 2006.

[16] B. Nicolae, D. Moise, G. Antoniu, L. Bouge, and M. Dorier, "Blobseer: Bringing High Throughput under Heavy Concurrency to Hadoop Map-Reduce Applications," *Proc. IEEE Int'l Symp. Parallel and Distributed Processing*, pp. 1-11, 2010.

[17] J. Baker, C. Bond, J.C. Corbett, J.J. Furman, A. Khorlin, J. Larson, J.L. Eon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," *Proc. Fifth Biennial Conf. Innovative Data Systems Research*, 2011.

[18] "Zookeeper: Because Coordinating Distributed Systems is a Zoo" http://zookeeper.apache.org/doc/trunk/, Nov. 2013.

[19] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. Davis, S. Rao, T. Zou, and A. Zuck, "Tango: Distributed Data Structures over a Shared Log," *Proc. 24th ACM Symp. Operating Systems Principles*, pp. 325-340, 2013.

[20] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M.K. Aguilera, and J. Li, "Transaction Chains: Achieving Serializability with Low Latency in Geo-Distributed Storage Systems," *Proc. 24th ACM Symp. Operating Systems Principles*, pp. 276-291, 2013.

[21] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., http://portal.acm.org/citation.cfm?id=573304, 1992.

[22] A. Sankara, K. Guinn, and D. Nguyen, "Volume Shadow Copy Service," Mar. 2004.

[23] Z. Peterson and R. Burns, "Ext3cow: A Time-Shifting File System for Regulatory Compliance," *ACM Trans. Storage*, vol. 1, pp. 190-212, May 2005.

[24] "FlashCopy Consistency Group: Creating Consistent PiT Copies", http://www.redbooks.ibm.com/abstracts/tips0309.html?Open. Published on 15/10/2003, Apr. 2012.

[25] S. Marathe, P. Massiglia, N. Pendharkar, P. Vajgel, and O. Kiselev, "Using Local Copy Services: How Veritas Storage Foundation Snapshot Facilities Protect Data, Reduce Costs, and Enhance the Quality of IT Service," *Symantec Yellow Books*, http://www.symantec.com/en/uk/theme.jsp?themeid=yellowbooks, 2006.

[26] B. Liskov and R. Rodrigues, "Transactional File Systems Can Be Fast," *Proc. 11th Workshop ACM SIGOPS European Workshop*, Sept. 2004.
[27] S. Verma, T.J. Miller, and R.G. Atkinson, "Transactional File System," US Patent No.6,856,993, http://www.google.com/patents/US20050149525, 2005.
[28] R.P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C.P. Wright, "Enabling Transactional File Access via Lightweight Kernel Extensions," *Proc. Seventh Conf. File and Storage Technologies*, pp. 29-42, 2009.
[29] "Oracle Database Backup and Recovery Basics," 10g Release 2 (10.2) B14192-03, Nov. 2005.
[30] J. Gray, "Notes on Data Base Operating Systems," *Proc. Operating Systems, an Advanced Course*, vol. 60, pp. 393-481, 1978.
[31] C. Pu, "On-the-Fly, Incremental, Consistent Reading of Entire Databases," *Proc. 11th Int'l Conf. Very Large Data Bases*, vol. 11, pp. 369-375, 1985.
[32] P. Ammann, S. Jajodia, and P. Mavuluri, "On-the-Fly Reading of Entire Databases," *IEEE Trans. Knowledge and Data Eng.*, vol. 7, no. 5, pp. 834-838, Oct. 1995.
[33] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman, 1986.
[34] K. Vidyasankar, "Serializable Graphs," *Proc. 14th Int'l Workshop Graph-Theoretic Concepts in Computer Science*, pp. 107-121, 1989.
[35] A.W. Leung, S. Pasupathy, G. Goodson, and E.L. Miller, "Measurement and Analysis of Large-Scale Network File System Workloads," *Proc. USENIX Technical Conf.*, pp. 213-226, June 2008.
[36] T. Gibson, E.L. Miller, and D.D.E. Long, "Long-Term File Activity and Inter-Reference Patterns," *Proc. 24th Int'l Computer Measurement Group Conf.*, pp. 976-987, 1998.
[37] J. Wang and Y. Hu, "WOLF A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems," *Proc. First Conf. File and Storage Technologies*, pp. 47-60, 2002.
[38] D. Roselli, J.R. Lorch, and T.E. Anderson, "A Comparison of File System Workloads," *Proc. USENIX Ann. Technical Conf.*, pp. 41-54, 2000.

Lipika Deka received the 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.

Gautam Barua is a professor in the Department of Computer Science and Engineering, Indian Institute of Technology, Guwahati, where he had until 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 operating systems and computer networks. He is a member of the IEEE and ACM.