Consistent Online Backup in Transactional File Systems

Thesis submitted

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

by

Lipika Deka

Under the guidance of

Prof. Gautam Barua

Department of Computer Science and Engineering

Indian Institute of Technology Guwahati

Guwahati - 781 039, INDIA

April 2012

Certificate

This is to certify that the thesis entitled **"Consistent Online Backup in Transactional File Systems"**, submitted by **Lipika Deka**, a research student in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati, for the award of the Degree of **Doctor of Philosophy**, is a record of an original research work carried out by her under my supervision and guidance. The thesis has fulfilled all requirements as per the regulations of the Institute and in my opinion has reached the standard needed for submission. The results embodied in this thesis have not been submitted to any other University or Institute for the award of any degree or diploma.

Date:
Guwahati

Gautam Barua,
Professor,
Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati,
Guwahati- 781 039, India.

Acknowledgements

At the onset I would like to express my sincere gratitude to Prof. Gautam Barua for always doing what is best for me. From the very beginning I knew I had the best supervisor, and I may have doubted many other things at one time or another, but I never doubted that. His analytical and thinking clarity amidst his busy schedule shall always inspire me. Without him I would not have been able to complete this research.

My dissertation committee members, Dr. G. Sajith, Prof. D. Goswami, and Dr. H. Kapoor, were generous of their time and provided helpful input and comments on the work. The Departments as well as the Institutes, technical and secretarial staff provided valuable administrative support. Here, I would like to specially mention Amaya Phukan, Bhriguraj Borah, Nanu Alan Kachari, Nava Kumar Boro, Raktajit Pathak, Souvik Chowdhury, Naba Kumar Malakar and Prabin Bharali.

My fellow research scholars have throughout provided me inspiration as well as healthy competition. Mamata taught me patients and perseverance, Amrita showed me the strength of will power and Nitya Sir taught me the importance of sincerity. Maushumi has been a constant companion and a sounding box. She showed me that work and play can go hand in hand. I would also like to mention Rajendra, Bidyut, Alka, Ferdous, Kamakhya, Mohanty, Pallav and my childhood friend Aurelia.

This work would not have been possible without the loving support of my family. My parents to whom I dedicate this thesis, have unselfishly loved and supported me through thick and thin. Without Diganta's patience, kindness, and perseverance, I never would have been able to accomplish this much. I sincerely acknowledge the support and understanding I received from my parent-in-laws. Last but not the least, that final push, motivation, inspiration or whatever you call it was given by my "Chunglung", our son Anuron who came to us within this thesis work and filled our world with sweetness.

Abstract

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

We propose a scheme 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*. Starting with considering only reads an writes, we extend the scheme to include file operations such as directory operations, file descriptor operations and operations such as append, truncate, rename, etc., as well as operations that insert and delete files. We put our scheme into a formal framework to prove its correctness, and the formalization as well as the correctness proof is independent of the concurrency control protocol used to serialize the user transactions.

The formally proven results are then realized by a practical implementation and evaluation of the proposed scheme. In the practical implementation, 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 timestamp based protocols (Strict 2PL in the current implementation) to ensure consistent operations. Now, once the backup program is activated, all other transactions are made aware of it by some triggering mechanism and they now need to serialize themselves with respect to the backup transaction also. If at any moment a conflict arises while establishing the pairwise *mutually serializable* relationship, the conflicting user transaction is either *aborted* or *paused* to resolve the conflict. We ensure that the backup transaction completes without ever having to *rollback* by always ensuring that it reads only from committed transactions and never choosing it as the victim for resolving a conflict.

To be able to simulate the proposed technique, we designed and implemented a user space transactional file system prototype that exposes ACID semantics to all applications. We simulated the algorithms devised to realize the proposed technique and ran experiments to help tune the algorithms. The system was 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 online backup) and one without (thus capturing an online inconsistent backup) and comparing the results obtained from the two scenarios to calculate 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, has very little overheads (2.5% in terms of transactions conflicting with 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 program). Noticeable performance improvement is recorded on use of performance enhancing heuristics which involved diverting the backup program to lesser active regions of the file system on detecting conflicts.

Contents

1 Introduction **1**

1.1 Approach taken in the thesis 2

1.2 Contributions of the thesis 4

1.3 Organization of the thesis 5

2 Literature Survey of Data Backup Techniques **7**

2.1 The Backup Process 8

2.1.1 Backup Storage Media 8

2.1.2 Data Repository Models And Management 8

2.1.3 Logical and Physical Backup Strategies 9

2.1.4 Backup Dataset Optimization and Security 10

2.1.5 Backup Storage Format 10

2.2 Types of Data Consistency 10

2.3 Traditional Backup Techniques 12

2.3.1 Tar 12

2.3.2 Dump 12

2.3.3 dd 12

2.4 Issues in Using Traditional Backup Techniques for Online Backup 12

2.5 Online Backup 15

2.5.1 Online File System Backup 16

2.5.2 Online Database Backup 22

3 Flat File System Formal Model **25**

3.1 Flat File System Model 25

3.2 Basic Terminology 27

3.3 Mutual Serializability 28

3.3.1 A *read* and *write* Operation 32

4 Hierarchical File System Formal Model **33**
4.1 Hierarchical File System . . . 33
4.2 The Formal Model . . . 36
4.3 Mapping of Hierarchical File System Operations . . . 37
4.3.1 The Backup Transaction's *read* Operation . . . 37
4.3.2 Operations Maintaining a Fixed Data Set . . . 37
4.3.3 Operations that Grow/Shrink the Data Set . . . 39
4.4 Enhanced *Mutual Serializability* . . . 40
4.5 Path Lookups . . . 45
4.6 Realizable Schedule . . . 45

5 Implementation **49**
5.1 Design and Implementation of TxnFS . . . 50
5.1.1 Transactional Model . . . 51
5.1.2 Module Architecture and Implementation . . . 53
5.1.3 Transactional Handling of File Operations . . . 58
5.1.4 Application Programmers Interface to TxnFS . . . 60
5.1.5 Contributions and Limitations of TxnFS . . . 60
5.2 Implementation of Consistent Online Backup . . . 61
5.2.1 Conceptual Overview . . . 61
5.2.2 Implementation Details . . . 63

6 Evaluation **73**
6.1 Workload Models . . . 74
6.1.1 Uniformly Random Pattern(global) . . . 74
6.1.2 Spatial Locality Access Pattern(local) . . . 75
6.2 Experimental Setup . . . 77
6.3 Simulation Results and Performance Analysis . . . 77
6.3.1 *Conflict Percentage* under different workloads . . . 78
6.3.2 *Backup Time* . . . 81
6.3.3 Overall Throughput . . . 83
6.4 Performance Improvement . . . 85
6.4.1 Algorithm and Implementation . . . 85
6.4.2 Simulation Result and Performance Analysis . . . 86
6.5 Discussion . . . 89

7 Conclusion **91**

7.1 Future Work 93

7.1.1 Performance improvement 93

7.1.2 Mutual Serializability vis-a-vis Non Transactional File Systems 93

7.1.3 Improving TxnFS 94

List of Figures

2.1 File System State 14
2.2 Inconsistent File System State In Backup 14

5.1 Top Level TxnFS Architecture 51
5.2 Modular Architecture of TxnFS 54

6.1 Percentage Of Transactions Conflicting In Each Set Of Workload 80
6.2 Duration Of Backup For Each Set Of Workload 82
6.3 Throughput Of User Transaction(Measured for the duration the backup was active) 83
6.4 Conflict Percentage On Applying Heuristics To Improve Performance 87
6.5 Backup Time On Applying Heuristics To Improve Performance 88
6.6 Throughput On Applying Heuristics To Improve Performance 88

List of Tables

5.1 Establishing Mutual Serializability. 62

6.1 Change in Conflicts Between The MS_disabled and MS_enabled Run. 79
6.2 Change in Backup Time Between The MS_disabled and MS_enabled Run. . . 81
6.3 Change in Throughput Between The MS_disabled and MS_enabled Run. . . . 83

Chapter 1

Introduction

Small businesses to multimillion dollar businesses store data in digitalized format. Digital data and the programs that mine and manipulate the data are the very essence of these organizations existence. Moreover, the past two decades have seen computers becoming more and more affordable leading to an individual's personal data such as photographs, music, videos, etc., too going the digital way.

Data stored digitally may become the victim of possible disasters like errors due to users, hardware and software failures, natural calamities like earthquake, flood etc. Digital history is full of horror stories involving loss of data due to corruption, theft or natural calamity. Hence, the requirement to protect digital data and programs (which we shall collectively refer to as data) from loss as well as corruption is and should be the utmost priority of any organization as well as individuals.

Data is protected from loss and corruption by making copies of the data at regular intervals and storing these copies elsewhere. This method of data protection is termed as *backup*. The term *backup* is used as a *verb*, a *noun*, and an *adjective*. The backup copy of a set of data is usually stored in a geographically remote location so that events that destroy the primary copy of the data do not destroy the backup copy too. Apart from the primary purpose of recovering data after its loss or corruption, backups are also performed by organizations to meet legal and business data archival requirements.

Much of this data reside in files and will continue to do so, ever increasing in volume. The present study addresses the protection of data residing in files and hence backup of *file systems*. Traditionally, a backup utility runs on an unmounted file system to capture a consistent file system image, using utilities like *tar* [9], *dump* [3] etc. But, file system backup, an extremely important but often neglected chapter of data management, faces a paradoxical situation today. File system sizes have touched peta bytes and are predicted to grow further, and as a result, "file system size operations like backup will take much

longer. Longer backup time means longer system downtime. On the other hand, the backup "window" is shrinking with globalization ushering in a 24X7 business environment, which means that even a few minutes of system downtime can prove to be extremely costly. A study in the year 2000 projected the cost of an hours computer downtime at $2.8 million for just the energy industry sector, proving traditional backup techniques to be infeasible in todays 24/7 computing scenario [11]. Hence, system administrators are left with the only option of taking a backup of a mounted file system, even as the underlying file system is being constantly modified. Such a backup is termed as an *online backup*.

Arbitrarily taken online backup using the traditional techniques may destroy data integrity in the backup copy. For 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 updates to these 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 [51, 28]. Possible compromise in data integrity as a result of backup being taken of an active file system is illustrated in detail in Section 2.4.

Data is backed up mainly to restore the primary file system copy in case of its accidental loss and restoring from an inconsistent backup-copy may sometime be more dangerous than data loss, as such errors may go undetected and render the restored file system unusable.

This thesis focuses on an online backup utility, termed *consistent online backup* which preserves data integrity within and across logically related files. The primary objective of this thesis is to examine the problem and to devise a solution for online backup that captures a consistent image of the file system being backed up. While doing so we shall study the adverse performance impact of the online backup scheme on the system and suggest methods to reduce such impacts.

The remaining sections of this chapter are organized as follows: Section 1.1 introduces the approach taken in this study for analyzing and solving the problem of consistent online backup with respect to existing approaches. Primary contributions made in this thesis are highlighted in Section 1.2 and Section 1.3 presents the layout of the overall thesis structure.

1.1 Approach taken in the thesis

Considering the importance of taking a backup and faced with the self contradictory scenario depicted above, industry and academia have brought forth a host of online backup solutions which perform backup on active file systems. Prominent among these are the *Snapshot* [47, 35, 56, 34, 36, 23, 75, 37, 21, 10, 44, 17, 5] and the *Mirroring* [38, 14, 17, 49, 21, 65] techniques. Snapshots are created and mirrors are "split" by momentarily pausing the sys-

tem, completing the execution of in-flight operations and flushing the buffers, and sometimes after conducting a file system consistency check. Techniques such as the versioning file systems facilitate online backup through the process of *continuous protection of data* [27, 50]. Still other existing online file system backup solutions operate by using heuristics to group related files and perform backup on them together [8].

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 only achieving weaker levels of backup-copy consistency i.e. per file consistency or in very specific cases when supported by the applications themselves, application level consistency is achieved.

This lead us to assume a transactional file system as a mode for specifying consistency requirements while tackling the issue of an consistent online backup. With transactions, the natural next step is to consider the use of techniques in transaction based database management systems. But consistent online backup solutions practised in the database area cannot be exported to the file system area as backup of an active file system brings forth issues unique to file systems. For example, file systems cater to many more operations besides reads, writes, deletes and insertions of entities, such as rename, move, copy etc. Files in most popular file systems like ext2 are not accessed directly, but are accessed after performing a "path lookup" operation. File system backup involves backing up of the entire namespace information (directory contents) along with the data. This complicates online backup as a file systems namespace is constantly changing even as the backup process is traversing it. Moreover, a file system backup does not involve just the backup of data but also includes the backing up of each files meta-data (example, inodes of ext2).

Assuming a transactional file system, this thesis shows that a consistent online backup-copy can be created by employing a specialized concurrency control mechanism called *mutual serializability*. In this mechanism, a consistent backup copy is obtained by ensuring that the backup program is separately serializable (*mutually serializable*) with each user transaction simultaneously. The *mutually serializable* relationships are established by extending the traditional set of conflicting operations to include the *read-read* conflict. In the presence of many tried and tested protocols, the need for a specialized concurrency control protocol arises because the backup transaction reads the entire file system. The use of standard concurrency control mechanisms may result in the backup transaction getting aborted or the gradual death of user transactions as backup gradually locks the entire file system and this will adversely impact performance and backup time. The proposed consistent online backup method is then made concrete through theoretical analysis and a prototype implementation. With the help of numerous workloads, the cost incurred on deploying the *mutual serializ-*

ability protocol, in terms of the number of conflicts between user and backup transaction and the time taken for the backup operation to complete, is then evaluated and methods suggested to reduce this cost.

The approach taken in this thesis to backup a consistent file system is similar in concept to the approach taken by *Pu. et.al* [53, 54] to read an entire database. The method described by *Pu. et.al* [53, 54] 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 the approach taken by *Pu. et.al* [53, 54] were identified and modified by *Ammann et. al.* [16]. Our approach, though similar to the cited studies, is novel in many ways. First of all, we consider a file system as opposed to a database system, which brings forth issues unique to a file system. Secondly, we present a theoretical basis for understanding the problem whereas the earlier works concentrated on implementation, in a scheme with two phase locking as the concurrency control algorithm of the user transactions. Our formulation is therefore independent of any particular concurrency control algorithm. Finally, we have implemented a backup scheme in a transactional file system and shown that an efficient solution is feasible.

1.2 Contributions of the thesis

One major contribution of this work is that we have showed that it is possible to capture a file system backup image preserving integrity across logically related files even as the underlying file system is constantly changing. We identified the need for transaction like constructs for specifying logically related files in order for an online backup utility to be able to capture a consistent backup copy.

Our conceptual and technical contributions include the following:

- The primary conceptual contribution of this study is the protocol termed *mutual serializability* (MS) put forward to facilitate an online backup utility to capture a consistent image of a file system. Moreover, 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 thesis we put forward a formal framework for modelling and analyzing the consistent online backup problem and its solutions.

- The main technical contribution include the algorithms and implementation of the MS concurrency control protocol.

- Additional technical contributions include the methods proposed to improve performance during an online backup through knowledge acquired from studying access

patterns and impromptu diversions of the backup transactions to currently "colder" regions of the file system hierarchy.

- While laying the foundation for implementing and simulating the proposed MS concurrency control protocol, the present study has contributed an alternative transactional file system implementation (TxnFS) which without much complexity can easily be added over existing non-transactional file systems and provide transactional security to application programs.

1.3 Organization of the thesis

Rest of the thesis is organized as follows:

Chapter 2 This chapter provides a general discussion on the different tracks backup has developed over the years, defining backup related terminologies and administrative policies. The chapter then does a survey of existing approaches and their effectiveness in capturing a consistent online backup.

Chapter 3 The formal framework for modeling and analyzing the consistent online backup problem and its solutions is put forward in Chapter 3. The formal theory for a consistent online backup presented here, utilizes a flat file system model accessed through a transactional interface that allows file system access through only *read* and *write* calls.

Chapter 4 We then extend the file system model in Chapter 4 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 system with operations including *rename*, *truncate*, directory and metadata operations.

Chapter 5 This chapter, details how the *mutual serializability* protocol is implemented over a transactional file system. Due to the non-availability of source code of a transactional file system, we designed and implemented a transactional file system prototype in user space.

Chapter 6 In this chapter, the implemented system is evaluated and the cost incurred due to the use of *mutual serializability* protocol, is reduced with the help of a number of workload sets with varying characters.

Chapter 7 This chapter summarizes the overall contribution of this thesis and identifies some future research directions.

Chapter 2

Literature Survey of Data Backup Techniques

Protection of digital data from loss and corruption is necessary and hence data backup strategies existed from the time a computer came into being and data took a digital form. Recent federal and legal data retention directives of several countries have reinforced the development of backup techniques [6, 48]. During the early days data was manually copied to storage media, for example, punch cards. As computers and associated software matured, the manual backup of data to storage devices was taken over by specialized backup programs. Initially these backup programs were relatively simple and essentially involved copying all files to a storage device, which would then be stacked away to be used when needed. As computers penetrated into all realms, the amount of data to be protected increased enormously and as technology evolved the threats against which data needed to be protected also increased. This lead backup procedures to become increasingly complex. As scenarios changed, backup schemes had to meet with changing limitations or challenges. At present the unprecedented increase in digital data and the requirement of this data to be online at all times, has ushered in the necessity of online backup schemes, more specifically a online backup scheme that preserves the integrity of a file system.

The current chapter discusses in brief the broad topic of backup and then discusses the available online backup schemes in both the file system and the database areas.

A backup procedure is a complex mix of a number of techniques to be determined under a disaster recovery and backup management plan. A backup involves determining the storage media to store the backed-up data, the program to use for reading and copying data to storage media, fixing a backup strategy of when to backup, what to backup, deciding on storage optimization techniques like compression and de-duplication, and backup copy security techniques like encryption. Hence, in literature and in practice, the development of

backup techniques can be tracked in all these directions apart from the concerned topic of this thesis. Delving deeper into these issues is beyond the scope of our present work and we shall briefly survey them in Section 2.1.

2.1 The Backup Process

As already mentioned, the entire backup process involves a multitude of tasks. We briefly review some of the important tasks in this section.

2.1.1 Backup Storage Media

Data to be backed up must be stored on some storage medium. This backup media should be affordable, not too bulky to facilitate transport and storage, contain the necessary capacity so that too many media swaps are not needed during the backup process and the storage media must be durable and reliable as it may be stacked for a long period.

Punch cards can symbolically be considered as the first backup storage media for digital data. Thereafter, magnetic tapes have been used as a secondary storage medium for decades owing to its high capacity, affordability and high recording speed as it is a sequential access medium. Advances in technology has ensured that tapes remain the most prevalent medium for backups. Recent times have seen hard disks giving some competition to magnetic tapes [33] where storage medium is needed in bulk due to its improving capacity/price ratio, added to its low access time, high capacity and ease of use. Apart from these two media, optical disks, floppy disks, flash storage and more recently Blu-ray disks are used as backup storage media but mostly for low capacity needs.

2.1.2 Data Repository Models And Management

Regardless of the storage medium, it is required to decide how often backup should be taken, to select data to be backed up each time and to organize the backed up data. We briefly discuss some of the data repository models in use.

A repository to which the entire file system is copied each time a backup is performed is called a *full backup*. Such archives provide easy and fast recovery, but scores low on speed and space utilization. The *incremental backup* model captures data created or modified from the last backup, which though fast and space efficient makes restoration slow and complicated. Thus, most backup schemes usually take a hybrid path of occasional full backups and frequent incremental backups [22]

Many systems provide different levels of incremental backups [3] called *multilevel backup* techniques where level 0 is a full backup and a level n backup is a backup of files new or

modified since the most recent level n-1 backup [3, 39]. The *rsync* [68] utility is used to select file portions to transmit to backup storage media and to make an earlier version of a backed up file up-to-date. Another model called the *reverse delta backup* scheme stores a "mirror" of the primary copy, occasionally synchronizing it with the primary copy and also storing the differences between the mirror and current primary copy to reconstruct older versions when necessary. *Continuous data protection* scheme [50] is another repository model where every new and modified data is backed up. This scheme records byte-level, block-level or file-level modifications.

These backup repositories are catalogued simply on a sheet of paper with list of backup media and dates of backup or with a more sophisticated computerized index or with a relational database.

2.1.3 Logical and Physical Backup Strategies

Files consist of logical blocks spread over physical disk blocks that may not be contiguous. Consequently, logical and physical file system backup are two common approaches for performing data backup [39]. Logical or file-based backup systems backup files and directories via the underlying file system and writes contiguously to the backup medium thus facilitating fast individual recovery. Reading individual files one at a time has the disadvantage of increased seeks as files may be fragmented on disk. Backup utilities like tar [9] and cpio [1] are examples of file-based backup systems native to UNIX.

Device-based or physical backup systems traverse contiguous physical blocks bypassing the file system and copying each low level physical block to the backup medium. Physical backup schemes achieve higher throughput, better performance and scalability [39], but is not portable as restore from it needs the knowledge of the original file system. The dd program of UNIX [2] is a common device-based backup utility. Dump [3], even though it reads data at the disk level, is considered to be a hybrid approach as it gathers files and directories to backup with the knowledge of the file system structure. The scheme described in [34] incorporates the advantage of the two approaches to perform backup.

The objective in our present study is to capture a consistent image of the file system even as it is changing constantly. Knowledge regarding file system data integrity is most likely to be available either implicitly or explicitly at the logical file system level. To interpret the integrity requirements, map it to the physical block layout of files on disk and then to take a device-based backup would be a tedious task. Hence, the backup schemes we propose in this thesis are logical backup schemes.

2.1.4 Backup Dataset Optimization and Security

As file system sizes touch peta bytes, storage space as well as time taken by the backup process is increasing. This necessitates the optimization of media usage, backup speed and bandwidth usage. Backup storage space is reduced considerably in systems through compression using well established utilities like *gzip* and through deduplication [74]. Deduplication or single-instance storage techniques identify similar data portions within and across files and stores only one instance of such portions. Backup speed and bandwidth optimization is achieved through the use of algorithms like *rsync* [68] for data transfer and by using the excess space on cooperating peers having common data [26] as backup storage. Backup speed can also be increased by employing "disk staging" where backup is first done on to a disk as it is faster and for long term storage the contents of the secondary disk is transferred to tape and the disk space reutilized. Backup storage always has the risk of being stolen or getting lost. Data encryption techniques are used to protect sensitive backup data [4].

2.1.5 Backup Storage Format

Efficient storage media utilization, recovery speed and data portability depends a lot on the format of the stored backup data. The file-based backup programs such as tar and dump, define their own formats for the backed up data, which are well documented and architecture neutral. These formats have been modified [35] [legato, IBM] and imbibed into a number of backup schemes for platforms ranging from Solaris to Linux. The dump [3] format begins with two bit-maps describing the system at the time of backup, including the inodes of directories and files that have been backed up. The bitmaps are followed by directories which are then followed by the files, both arranged in ascending order of the inode numbers. Each file and directory is preceded by its metadata. A directory simply contains file names and their inode numbers. The tar archives file format [9] is a concatenation of one or more files that are backed up and each file is preceded with a header of file metadata and header block checksum. To restore, the tar utility traverses the tar file sequentially extracting one file after another. The tar file format is used in conjunction with compression utilities like gzip, bzip2 etc. Physical backup programs like dd [2] simply copy raw data from one device to another and do not have any specialized format. Knowledge of the original file system is hence necessary to restore from a physical backup.

2.2 Types of Data Consistency

Data recovered from a backup copy is as correct and useable as the backup process that captured the backup copy. Hence, a backup technique must ensure the correctness, integrity,

and the consistency of the captured backup copy. Various kinds of backup copy consistency has been identified, that include *file system level consistency*, *point-in-time consistency*, *application level consistency* and *transaction level consistency*.

Backup techniques preserving *file system level consistency* in the backup copy ensures the integrity of file system structures including metadata, directory entries and file data. Such techniques ensures consistency at the granularity of a single system call. Many file systems [47, 35, 56, 40] have been designed to inherently maintain a *file system consistent state* on-disk to avoid tedious file system consistency checks after a system failure. Modified backup utilities [34, 36, 49] take advantage of the consistency maintaining features of such file systems to capture a *file system level consistent* backup copy.

A *point-in-time consistent* backup image, represents the state of a file system at a single instant in time [21, 17]. A spontaneously taken *point-in-time* image cuts through single system call updates as well as transactions which comprise of more than one logically related system call and so may result in a corrupt backup copy. When supported by the underlying file system, a *point-in-time* backup utility is capable of capturing a *file system level consistent* backup [35, 40, 34]. When supported by suitable applications like databases, it is possible to capture an application-level *point-in-time* consistent image.

With file system transactions defined as comprising of one or more logically related system calls to one or more files, a backup copy that preserves *transactional consistency* must reflect modifications brought about by transactions that have completed (committed) and not any modifications done by half completed transactions.

Application consistency in a backup copy stands for the consistency of data as defined by an application which may include support for the notion of transactions at the application level. A backup program must be helped by the application to capture such an image by deferring further writes and flushing its caches, thus bringing on-disk application data to a consistent state before the backup utility reads the data.

The file system, including all applications on it are in a consistent state at transaction boundaries. In the current work presented in this thesis, we aim to study online backup techniques capable of capturing a *file system wide transactionally consistent* backup copy (reference to consistent backup shall mean file system wide transactional consistency if not otherwise specified).

2.3 Traditional Backup Techniques

2.3.1 Tar

tar is a logical backup utility that traverses the file system in a depth first order, scanning and copying files that meet a specified criteria in a single pass of the file system. Files backed up by the *tar* utility are stored in the tar archive format as detailed in Section 2.1.5.

2.3.2 Dump

Unix's *dump* operates partly at the file system level and partly at the raw disk level. It "understands" the structure of the file system being backed up and uses this knowledge to scan the file system, listing files and directories to be backed up. Files are listed and backed up according to a given incremental backup level and time of the previous backup. *dump* finally backs up data by accessing data through the raw device driver. *dump* completes a backup in four passes of the file system. In pass I, it builds up the list of files to be backed up and in pass II directories to be backed up are listed. In pass III and IV, the listed directories and listed files are respectively backed up. Each directory and file are preceded by a header which includes the inode in the backup copy.

2.3.3 dd

Unix's *dd* [2] backup utility copies data from one medium to another by reading the raw disk block by block.

2.4 Issues in Using Traditional Backup Techniques for Online Backup

The traditional backup utilities just described were designed to capture the file system state of an un-mounted, inactive file system when the file system state is guaranteed to be consistent. Backup when taken of a mounted active file system using these traditional utilities, may result in an unpredictable backup copy state that includes "torn pages", missing or duplicated files, file system metadata corruption, or inconsistent referential integrity across multiple files. The type and level of corruption may differ in different scenarios. The extent of corruption increases with multi-pass backup programs such as the *dump* utility as the time taken during and between passes provide an increased window for backup inconsistency. Some issues encountered when taking a spontaneous online backup are listed below and detailed by *Preston* [52], *Shumway* [64] and *Zwicky* [76, 77].

- **File Deletion:** The backup program, *dump*, writes out the directories and file contents separately in two different passes. If a file that was listed for backup was deleted after the directory contents were copied, dump would still copy the blocks belonging to the file irrespective of the fact that the concerned file does not exist anymore and the blocks may have since been reused. This may result in the copy of garbage values or contents of a different file or even files. The consequence may be specially hazardous as pointed out by *Zwicky* [76] if the deleted file inode becomes a directory or a deleted directory becomes a file between directory copy and its child file's copy, as the inode is of the original type and contents are new.

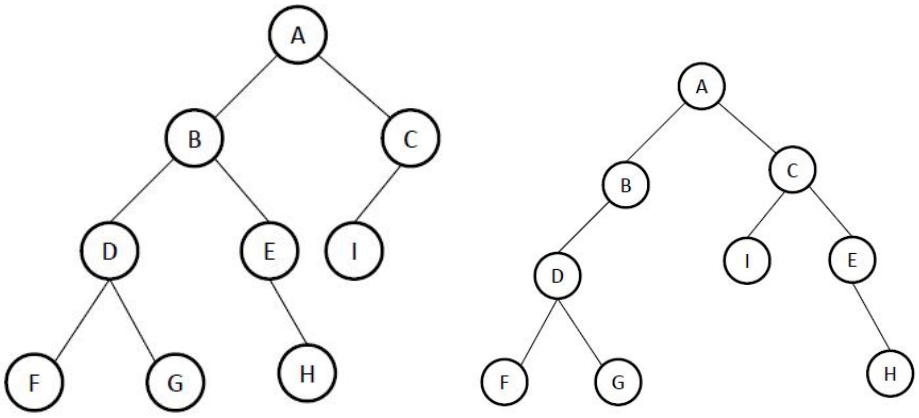
  Single pass utilities and those accessing through the file system such as *tar*, will receive an error when trying to access the file if it was deleted after its parent directory was read and may read the wrong file if the inode has since been reused.

- **File Modification:** Another very common scenario occurs when a file's contents are being updated while it is simultaneously being backed up. The corrupted file in the backup copy will contain portions of modified and portions of unmodified data. Such a file is also referred to as a "torn page".

- **File Truncate and Append:** Archive formats of utilities such as tar write a header containing the file's metadata before each file's contents. In scenarios where a file shrinks or grows even as it is being backed up an incorrect file length may be written in the header of the file. A future reading of individual files from the archive will result in unfortunate results as a file's length in the header does not match the actual length of the archived file. A similar scenario with *dump* is not as serious as it will copy garbage values if a file was shrunk and truncate the file if the file expanded even as it was being backed up.

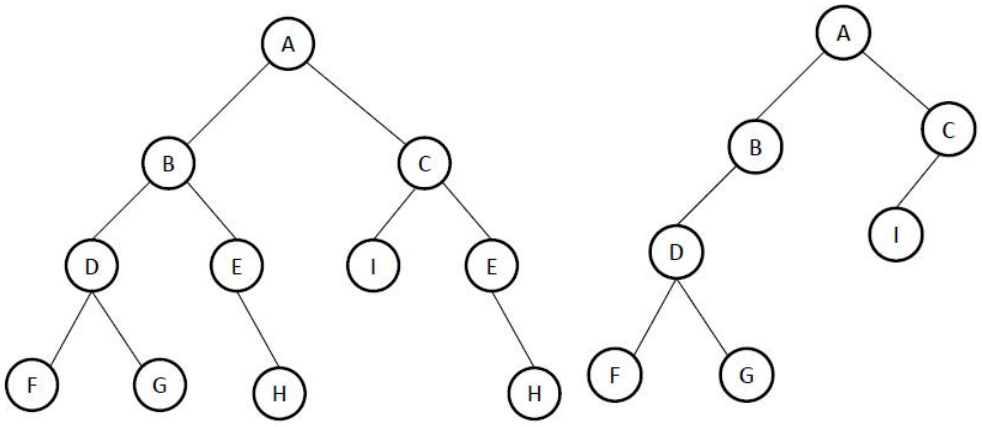
- **File Movement:** *Shumway* [64] identifies file movement operations being executed simultaneously with backup to be one of the most common and serious threats to backup reliability as it may result in an inconsistent file system hierarchy in the backup copy. Consider the file system tree in Figure 2.1a. Suppose a move command $rename(/A/B/E, /A/C/E)$ is being executed concurrently with a backup program. The file system state after the "move" is as depicted in Figure 2.1b. Now, an integrity preserving backup must either copy the file system state of Figure 2.1a or the file system system of Figure 2.1b. But, file system based single pass utilities such as *tar*, which traverses the file system tree in a recursive depth first order, may access directory C before the file/directory E was moved to it and directory B after E has been removed from it. This results in a missing file E or as in the present case a missing subtree

rooted at E in the backup copy as depicted in Figure 2.2b. Again, E may appear twice in the backup copy if C is read after the move and B before the move of E from B to C as shown in Figure 2.2a.



(a) Before rename (b) After rename

Figure 2.1: File System State



(a) Case 1 (b) Case 2

Figure 2.2: Inconsistent File System State In Backup

Now, if the move occurs during the scan phase of a multipass backup program such as dump, then file D may appear to exist in two places or not-at-all in the file system. And if the move occurs during the actual reading phase, then D will actually appear twice or not-at-all in the backup copy.

File movement during backup may result in more fatal consequences depending on the backup program implementation. For example, if a file system based backup utility traversing the file system tree in depth-first order, is designed to return to the parent through the ".." link, the backup program may lose its position in the hierarchy after

the execution of a rename operation.

- **Multi-file Integrity Problem:** Time taken to backup an entire file system is directly proportional to the size of the file system and may take much longer when backup is taken of a mounted active fie system due to resource contention. Because of this, different files maybe backed up at considerably different times. If the files are unrelated and the online backup issues listed previous to the current issue are not encountered then the backup copy is consistent. Now, if the files are inter-related and are undergoing modifications in such a way that if one changes, a related change effects one or more other files, and if this takes place even as the backup utility is backing them up then the versions of the related files copied to the backup may not be consistent with respect to each other. Such scenarios are common in database applications where files need to maintain referential integrity. But, it is not uncommon in many other applications scenarios. We illustrate a simple but common scenario where more than one file is logically related and a backup copy must preserve data integrity across these files. Multi modular software development involves related updates in multiple modules and hence in multiple files. For example suppose a function call is made in module A of file X and the function is declared and defined in module B of file Y. Now if an online backup utility captures file X after it is updated and file Y before the function referred from file X is defined and declared, the resulting backup is an inconsistent backup.

- **Issues arising in purely physical level online backup:** Backup programs such as UNIX's dd which operate at the physical level and read the raw disk block by block, causes even more dramatic results when run on an active file system. This is so because the logical view of a file system as seen by applications and its corresponding physical layout on disk are very different. A file on disk may be spread over non-consecutive disk blocks, raising the risk of issues like "torn pages" in the backup copy. Similar issues arise as inode and corresponding file blocks, directory blocks and blocks belonging to its children and other such logically related components may be heavily fragmented and spread across a raw disk.

2.5 Online Backup

Proposals on file system online backup solutions as reviewed in the current section, have either overlooked the necessity of file system wide consistent backup or have tried to achieve it through adhoc heuristics [8]. This is primarily because most current file system interfaces provide weak semantics for specifying consistency requirements of applications running on the file system. Hence, we see existing solutions [34, 36, 21, 17, 49, 22, 39, 40] only achieving

weaker levels of backup-copy consistency, that is, per file consistency or in very specific cases, application level consistency.

Recent times have seen a phenomenal growth of *unstructured and semi-structured* data i.e. data stored in files, with files spread across machines, disks and directories and accessed concurrently by a myriad of applications. With vital data stored in files, among many other management issues, there is an urgent need for efficient ways to specify and maintain the consistency of the data in the face of concurrent accesses. Recognizing this need, file systems supporting transactions have been proposed by a number of researchers [51, 45, 67, 69], and it is very likely that *Transactional File Systems* will become the default in systems. We have therefore assumed that file systems support transactions as a mechanism to ensure consistency of concurrent operations. Now, given a file system with transactions, the natural next step would be to borrow online database backup solutions and we study the usability of database backup solutions in Section 2.5.2.

2.5.1 Online File System Backup

2.5.1.1 Snapshots

A snapshot is a read-only copy of a file system that represents the state of the system at a particular instant. At the instant of snapshot creation, it resides on the same media and shares almost all data blocks with the "live" file system with the exception of the data structures needed to maintain the snapshot. Almost all snapshot implementations then use copy-on-write techniques to maintain the snapshot, while the "live" file system is actively modified. Once captured, a snapshot acts as a source for online backup. Snapshots are implemented either at the file system level [47, 35, 56, 34, 36, 23, 75, 37, 21, 10] or at the volume manager level [44, 17, 5]. All snapshot implementations attempt to capture a consistent file system image by temporarily deferring writes, completing in-flight system calls and purging caches before capturing the snapshot, with subtle conceptual and implementation differences. We explain a few snapshot based online backup implementations in the following paragraphs.

A. File System Based Snapshots: Snapshot capturing capability is at the core of a WAFL file system [35, 36]. WAFL file system is a hierarchical block based network file system with inodes used to describe files and directories. The WAFL file system stores its inodes too on a file called the inode file and the file system is a tree of blocks rooted at the inode of the inode file. All dirty data in a NVRAM cache is marked as IN_SNAPSHOT. When a snapshot process is initiated, new operations are stalled, in-flight system calls are completed, and data marked as IN_SNAPSHOT are flushed to disk to create a consistent snapshot.

When the cache is being flushed all NFS read requests are serviced, but write requests to IN_SNAPSHOT data are deferred. The bit map indicating blocks occupied by both the snapshot and the primary file system is then updated and the root inode is duplicated marking the end of snapshot capture. All block modifications thereafter are done after the original block is copied to the snapshot tree. An online logical backup [39] is performed of the snapshot using the *dump* utility in-built into the micro kernel of the system. A modified *dd* utility is used for online physical backup [39] after the utility gathers information regarding the raw blocks belonging to the snapshot. WAFL snapshot also facilitates incremental physical dumps [39].

Soft updates, a mechanism for enforcing metadata update dependencies to ensure file system consistency has been incorporated into the 4.4BSD fast file system [47]. Snapshot feature in the fast file system is implemented with support from the soft update mechanism which ensures consistency of the snapshot. On initiation of snapshot creation, file system operations are briefly suspended, in-flight system calls are allowed to complete, the file system is synchronized to disk and a snapshot file is created. The snapshot file basically contains a point-in-time image of the entire file system. Initially only a few blocks of the "live" file system, such as those holding copies of the superblock and cylinder group maps are copied to the snapshot file. As file system operations resume, blocks are copied to the snapshot file before write-in-place is done. The "live" *dump* utility dumps the snapshot, even as modifications continue to take place in the active file system. *dump* reads the blocks that have undergone copy-on-write from the snapshot file and the rest from its original locations.

Inherent features of log structured file systems [56] such as its no-overwrite feature and temporal ordering of on-disk data structure facilitates the capture of a consistent online backup. The Spiral log file system of the OpenVMS Alpha Operating system [34] is such a log structured file system where the "log" is a sequence of segments with monotonously increasing identity numbers. Write operations are gathered into segments and written on disk. At regular intervals a checkpoint, representing the position of the last consistent update to the file system, is written at a known location on disk. At any point in time an entire volume can be described in the checkpoint position and a segment list. Snapshots are taken and preserved by maintaining an old checkpoint and disabling the reuse of snapshot disk space. Once a snapshot is taken, a physical backup utility copies the snapshot to a secondary storage media by reading the segments belonging to the snapshot, even as operations continue on the "live" file system. Incremental physical backup of the file system is facilitated by the temporal ordering of data on disk.

B. Volume Manager Based Snapshots: Many volume managers too implement snapshots using copy-on-write based techniques, examples are Linux's native LVM [44], Flash-

Copy facility of IBM's Enterprise Server Storage(ESS) [17, 5] and VxVM version 4.0 [21]. Linux's Logical Volume Manager(LVM) [44] captures a snapshot by first creating a snapshot volume in empty disk space and then conveying to the file system that a snapshot has been initiated. The file system brings the data in a volume to a consistent state through the VFS-lock patch for LVM1 or automatically by file systems in the 2.6 kernel, in much the same way as in file system based snapshot mechanisms. Once the data is consistent, an "exception table" to keep track of modified blocks is created. As blocks modify, the original is first copied to the snapshot volume and marked as "copied" in the exception table, thus maintaining a point-in-time image. The snapshot volume is then backed up by mounting the snapshot volume.

ESS's FlashCopy facility creates a point-in-time consistent copy of data across multiple volumes using the *consistency group* construct [5]. Consistency group is a container of mappings consisting of predefined data dependencies across volumes, as specified by an application. To capture a volume snapshot, I/O activity to the volume is held off for the time period required to create the snapshot and a bitmap indicating occupied blocks is set up. Dependent writes to other volumes as indicated by the consistency group, are also held off for this period, thus preserving data integrity across multiple volumes. Thereafter, blocks are copied to a snapshot volume before any block is overwritten and this information is updated in the bitmap[17].

Online file system backup from a copy-on-write based snapshot whether captured at the volume or file system level, requires very little system downtime. Creation of a snapshot needs negligible additional space as it shares almost all blocks with the "live" file system. But, write performance overhead increases and storage requirement due to copy-on-write, increases over time, especially for systems with high write traffic. Almost all copy-on-write based snapshots are captured after blocking operations, completing on-going system calls and flushing caches. Such consistency preserving methods can only manage to establish data integrity within a file but is not capable of ensuring consistency across files.

Some implementations though are capable of ensuring application level or transactional level consistency when aided by suitably qualified applications such as databases [5, 65].

2.5.1.2 Mirroring

Mirroring is a volume-level, online backup technique provided by many volume managers such as Veritas's volume manager(VxVM), NetApps filer and EMC arrays. When a volume needs to be backed up, a mirror volume is setup and synchronized with the primary volume to hold an exact copy of the data in the primary volume. Once synchronized, all writes are done on the primary and the mirror simultaneously, thus keeping the two copies in

sync. A point-in-time backup is captured by separating or "splitting" the mirror nearly instantaneously and the "mirror" is an exact copy of the state of a "live" volume at the time of a "split" action. The mirror can then be used directly as a backup copy or as a source for backup. Many systems improve fault tolerance by keeping a third simultaneously modified copy of the volume called a business continuance volume(BCV).

A volume mirror represents the data that exists in a volume and the volume managers controlling the mirroring and ultimate "splitting" have no knowledge as such of the data cached by overlying file systems or files opened by applications in these file systems. Thus, data integrity of the backup copy taken through the "split mirror" technique is ensured by file systems or applications like database systems which are customized to support "split-mirror" capability of the volume manager. To preserve consistency, VxVM snap mirror technique [38], defers writes during a "mirror splitting" event, then runs *file system consistency check* on the mirror volume and if found inconsistent, a cleanup operation is performed. In other implementations, applications like databases, switch to "hot mode" during a "split" and the "separated" mirror is brought to a consistent state by running archive logs kept elaborately during the "splitting" process [14]. EMCs Time Finder and Hitachis ShadowImage [17] relies on the overlying filesystem to ensure that the data is consistent in the mirror when it is "split".

The SnapMirror [49] technique implemented in NetApps filers obtains a data integrity preserving mirror copy by keeping mirrors asynchronously, and periodically synchronizing it with the primary volume by transferring only the data blocks that have been modified from the previous "split" action. At the time of synchronizing the mirror, NetApps WAFL file system [35] captures a data integrity preserving file system snapshot and the synchronizing mechanism identifies the blocks to transfer to the mirror copy with the help of WAFL and snapshot data structures. Apart from ensuring a consistent point-in-time copy in the "split" mirror, consumption of network bandwidth and performance degradation due to write latency reduces considerably. Similarly, the FastResync feature of Veritas Volume Manager(VxVM) volume snaphot [21, 65] also speeds up the initial synchronization phase considerably and captures a consistent point-in-time copy. In this technique a feature called Data Change Object(DCO) is implemented and it is a disk-based persistent data structure that includes a bitmap called a Data Change Log (DCL). Each bit in the DCL represents consecutive blocks on the disk. It is setup when the mirror is split in both the original and the mirror. After a mirror is taken, a block change in the original sets the corresponding bit in the DCL. For re-synchronization changed blocks as indicated by the DCL is copied to the snapshot.

"Split-mirror" based online backup has the advantage of negligible recovery time, but, similar to other snapshot based backup techniques, data integrity across files which were

being modified by logically related operations at the time of the "split" event cannot be guaranteed. The existing consistency preserving methods of "split-mirror" based backup ensures consistency at the level of a single file and in some cases when leveraged by applications, it can guarantee application level consistency.

2.5.1.3 Locking

Proposed solutions by *Shumway* [64] requires the entire file system to be locked against modifications during the time a backup is taking place as a defensive mechanism against inconsistencies arising from *file movement* and file transformation like compression, deletion, creation etc.

Alternatively another solution relies on the method of detecting *file movement* and *file transformations* like compression, deletion, creation etc by comparing file modification times before and after a backup and if inconsistencies are detected then the concerned file/files are backed up again. The prevention and detection solutions given by *Shumway* [64] reduces file activity and consumes time apart from providing a solution at the granularity of a file operation whereas we aim for backup copy preserving file system integrity across multiple related operations addressing one or more files.

2.5.1.4 Continuous Data Protection

Continuous data protection(CDP) [27], also known as real-time backup, is yet another method of online backup which essentially performs a backup of every change that is made to the data. Continuous data protection saves every block level, byte level or file level changes. On the conservative side, the changed data must be stored at a physically separate media but many CDP techniques such as versioning file systems stores each data change on the same media as the primary data. A significant advantage of the CDP technique is that it can be used to restore data to any point-in-time and hence acts more like a journal holding the journey of every block or file. CDP is ideal for providing an "auditable trail of changes" [50] of digitally stored records, required by many government regulations for digital storage. Moreover, CDP eliminates the backup window totally and can provide fine granular restore. CDP solutions are inherent to certain file system designs [50], while other solutions incorporate CDP technology into their storage systems [73]. We now study a few of these solutions here.

Continuous data protection through versioning file system have long existed. The Cedar file system [31] was one of the earliest file systems providing per file versioning by writing a different version of a file on an update and suffixing the files name with the verion number. The Elephant file system [58], Comprehensive versioning file system [66], WayBack

file system [25] are further examples of versioning file systems. Log structured file systems [56, 63, 40] are also file system implementations providing fine granularity versioning. Snapshot file systems such as NetApp's WAFL [35, 36] also provide coarse grained versioning but they are temporal rather then continuous. Ext3COW(Ext3 Copy-On-Write) [50] built on the stable ext3 file system, provides *continuous data protection* through copy-on-write based occasional file system wide snapshot and per file versioning at every write between snapshots. Ext3COW is unique in that it provides a user interface for version access. The snapshot and versioning features are implemented by adapting existing ext3 data structures like superblock, directories and inode. On-disk as well as in-memory file system super-block is modified to include a system-wide epoch counter to act as a reference to mark versions of a file. Each entry in a directory now includes a birth epoch number and a death epoch number. Unused fields of on-disk and in-memory inodes have been utilized to include an inode epoch number, a copy-on-write bitmap and a field pointing to the next inode in the version chain in-order to support snapshot and copy-on-write. Each file may have more then one inode structure linked together through the i_nextinode field and each version of the same inode points to blocks of different versions of the same file. The snapshots are used for online backup using traditional utilities like *tar* and *dump*.

Advances in disk technology is facilitating continuous capture of updates as it happens. For example, the *Peabody disk* maintains a log of all writes to a disk block and by traversing the log in reverse order, a past state of the block and hence the disk can be recovered. Similarly, *Laden et al.* [41] puts forward four architectures that integrates CDP on the storage controller and uses a form of logging to store every data update at the block level. *Continuous data protection* consumes a lot of space in storing every version of a block. Hence, recent research has seen proposals for making CDP space efficient. *Xiao et al.* [73] puts forward a time and space efficient block level CDP technique implemented in a disk array architecture that provides "Timely Recovery to Any Point-in-time(TRAP)". Storage space utilization is reduced by storing the "XORed" value of the original and modified block. *Flouris et al.* [29] puts forward a proposal for block level *near continuous data protection* which saves storage space by binary differential compression and stores only the delta or differences in the data between two consecutive versions.

Continuous Data Protection comes with the advantage of a zero backup window and the capability to restore to any point-in-time. But, such capabilities are limited to single files and cannot provide a consistent restore across multiple files which are logically related, unless transactions are supported. These techniques need more and more space as time elapses as old versions have to be kept if there is no other backup being taken. It may not be acceptable to remove old versions as auditing requirements may need to view files that have been deleted. Any effort to prune the versions will result in the need to backup data,

which is begging the question. Thus, CDP is not really a backup solution.

2.5.1.5 Transactional Backup Integrity Using Heuristics

Many backup solutions have attempted to achieve integrity of backed up data that is logically related but physically spread across files. The solution proposed in St. Bernard Software's White Paper On Open File Manager [8] illustrates a few such solutions that is application specific or uses third party software. In some third party software, the administrators are required to manually identify files that are logically related and provide the information to the backup software. Such solutions maintain per file integrity and also backup, related files together to maintain inter file transactional integrity. St. Bernard Software's White Paper On Open File Manager [8] then proposes an *Open File Manager* to coherently backup files that are open or in-use during backup and also maintain inter file integrity. Once the first file is read by the backup application the *Open File Manager* begins its task of maintaining a consistent image of all "open" files by allocating a "pre-write cache" where a block of data is copied to before it is overwritten by applications. When the backup application reads the "open" file, the *Open File Manager* redirects reads of blocks modified since backup began to the "pre-write cache" area, thus succeeding in capturing a consistent file image. To capture a backup image preserving transactional integrity across multiple files, *Open File Manager* defines a *Write Inactive Period*(WIP), and files modified more frequently than WIP are considered to be related. *Open File Manager* defers access of these related files by the backup application. *Open File Manager* facilitates the backup of these related files together. Modifications that happen at a intervals greater than WIP are considered to be unrelated and *Open File Manager* allows backup to read those files.

2.5.2 Online Database Backup

A database is basically an organized collection of data, with its logical structures such as tablespaces laid out either on top of a file system or on top of the raw disk itself. Data on a database are queried and manipulated though operations encapsulated in atomic transactions. Transactions are a collection of logically related operations and a database management system ensures that a transaction is executed either completely or not at all. Its complete execution guarantees to leave a database in a consistent state.

Backup of a database can either be performed at the logical level or physical level. Logical backup backs up logical constructs i.e. schema objects such as tables and stored procedures into binary files. Physical backup of a database is the backup of the physical database files such as data files and control files on which the logical schema objects are laid. Database backup strategies are similar to file system backup strategies as described earlier, but are

supported by transaction constructs to obtain a consistent backup image during an online backup. We walk through some of the online database backup strategies in the following paragraphs.

One of the earliest online databases backup strategies is the *fuzzy dump* [32] strategy which involves a dump of the "live" database, following which the log generated during the dump is merged with the "fuzzy dump" to obtain a "sharp dump" or a consistent backup of the database. Rooted in the "fuzzy dump" strategy is the current "hot backup" strategies [13] such as Oracle's Recovery Manager(RMAN) [13] and PostgreSQL's Point-in-time recovery [7], where the data files are backed up even as they are being modified continuously. "Hot Backup" is performed once the database is switched to the "hot backup mode" which triggers extensive recordings in the *redo logs*, which are also backed up. The backup copy captured in this manner may have "fractured blocks", that is blocks that were being modified during the time of backup and hence has both unmodified as well as modified data. These backup copies are brought to a consistent state with the help of the simultaneously backed up *redo logs* either during recovery or before. Similarly, online backup strategies based on *snapshots* and *mirrors* rely on replaying the *write ahead log* or *redo logs* to reach a consistent point-in-time snapshot [7].

Still other online database techniques include the SQL dump, in which a text file of SQL commands executed during a "live" dump of the database is generated. To restore a consistent database, the SQL commands are replayed in-order to recreate the database [7].

Another consistent online backup strategy is through the use of rsync [7]. In this scheme, rsync is first run while the database server is running and then rsync is run again after shutting down the server. The second rsync copies only the difference in data from the first rsync and hence it is much faster. As a database is consistent when closed, the backup copy is a consistent copy.

Now, a single file system write results in a number of writes. For example a file create results in the file block write, file inode write, parent directory block write and parent directories inode write. Moreover, studies [15, 43] show that file systems are becoming more write oriented with read to write byte ratios slowly decreasing, added by gradual increase in file sizes, thus requiring frequent writes of large volume of data per write into *redo* and *undo* logs. This means that transactional file system backup approaches similar to online database backup [13] will incur high performance cost (write latency) and will be space inefficient. The ext4 file system [46] provides a journaling facility (which is essentially a log) but when data writes are "journaled", performance deteriorates. As a result, only metadata journaling is usually done. This only ensures that the file system meta-data remains consistent in face of failures. Similarly, the Episode file system [23] mentions that it implements atomic transactions. However, this is restricted to metadata logging only and it is targeted towards

faster recovery from crashes. Its techniques cannot be used for achieving a consistent online backup. Hence, we need to explore different approaches for achieving an online consistent backup copy of a file system.

Pu et al. [53, 54] 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. Serializing the backup transaction with respect to the regular transactions is at the core of the said scheme. This scheme ensures that the backup transaction does not abort, and active transactions abort if they do not follow certain consistency rules. The scheme suggested was shown to be in error by *Ammann et al.* [16] and the authors suggested a modified form of the algorithm to ensure a consistent backup. Every entity is coloured white to begin with. When the backup transaction reads an entity, its colour turns black. A normal transaction can write into entities that are all white or all black. A transaction that has written into an entity that is of one colour and then attempts to write into another entity of a different colour, has to abort. Transactions writing into white entities are ahead of the backup transaction in the equivalent serial schedule, while the transactions writing into black entities are after the backup transaction in the equivalent schedule. Each entity is also given a shade which changes from pure to off when a black transaction reads or writes the entity (whichever is allowed on it). Restrictions are placed on reading or writing on entities whose shades are off, by white transactions. By doing so, the backup transaction performs a consistent read of the entire database. *Bhalla et al.* [19] extends the scheme by removing the restrictions imposed on on-going transactions processing activity and uses a log to bring a global read to a consistent state. *Lam et al.* [42] includes read-only transactions into the scheme described by *Pu et al.* [53, 54] and *Ammann et al.* [16].

The scheme proposed by *Pu et al.* [53, 54] and *Ammann et al.* [16] lays the foundation of our approach. But, the scheme was designed for a database and it is assumed that two phase locking is the concurrency control algorithm. Our approach described in succeeding chapters 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 ext4 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. Moreover, a file system backup does not involve just the backup of data but also includes the backing up of each file's meta-data (e.g inodes of ext4).

Chapter 3

Flat File System Formal Model

The correctness of any protocol can be judged properly only through its formal analysis and proof. In this chapter 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.

The rest of the chapter is organized as follows: Section 3.1 informally lays the foundation of the model and defines the *mutual serializability* protocol. Section 3.2 formalizes the model, defining basic terminologies used. Finally, Section 3.3 begins by defining pair-wise *mutually serializable* relationship followed by the formal proof of correctness for the *mutual serializability* protocol.

3.1 Flat File System Model

To study the theoretical aspects of *consistent online backup* of the file system, in our formal model we consider a *flat file system* and a file system interface that provides *transactional* semantics. A file system *transaction* consists of a sequence called *schedule* of atomic actions, and each atomic action accesses a file from a set of file system entities called the *access set* of the transaction.

Each access operation of a *user transaction* either *reads* data from a file or writes data to a file. To ensure file system data integrity in the presence of concurrent accesses, a concurrency control mechanism, such as *strict2PL*, *optimistic concurrency control* etc. is used to serialize the user transactions.

A concurrency control mechanism establishes the *serializability* of the concurrent 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. Such a schedule is also referred to as a *serialized schedule*. A *serial schedule* is a sequential schedule of a set of transactions with no overlap of

the individual schedules in time. Now, given a set of transactions, two schedules of the same set is said to be *equivalent* if both schedules have the same sets of respective chronologically-ordered pairs of conflicting operations (same precedence relations of respective conflicting operations). Two operations in a *schedule* are said to be *conflicting operations* if they belong to different transactions, act on the same file and one is a write operation.

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 atmost 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. Now, the backup transaction is rather unique as it reads every file in the file system and treating it like any other transaction for the purpose of concurrency control leads to frequent inter transactional conflicts. Resolving these conflicts may mean either the gradual death of all concurrently executing user transactions as the backup transaction proceeds to lock all files in accordance with locking protocols or otherwise result in repeated abortion and restarting of the backup transaction. Hence, we are proposing a backup transaction specific concurrency control mechanism called *mutual serializability* that does not appreciably affect the overall performance of a system during a backup.

Mutual serializability achieves an overall serializability of backup and user transactions by keeping the backup transaction *mutually serializable* with every concurrent application 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 when we consider only the operations of these two transactions and when we assume even a read to conflict with another read, we get a serializable schedule. A formal definition is given below.

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. Despite its apparent simplicity, the model and result yields a theory of consistent online backup that can be applied verbatim to more complex file system models with transactions consisting of the basic file access operations to semantically rich operations.

3.2 Basic Terminology

A *File System* **F** is a set of distinct files, $\{f_1, f_2, ..., f_n\}$. Let $\mathbf{T} \bigcup t_b$ be the set of distinct transactions concurrently accessing **F** where $\mathbf{T} = \{t_1, t_2..., t_m\}$ is the set of user transactions and t_b is the *backup transaction*. A transaction $t_x, x = 1, 2...m$ accessing a file $f_i, i = 1, 2, ..n$ for reading denoted by the operation $r_x(f_i)$ while for writing it is denoted by the operation $w_x(f_i)$. 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 (the reason we state at most once instead of exactly once is clarified in the next chapter).

Let $a_x(f_i)$ be an element of the set $\{r_x(f_i), w_x(f_i) | \forall t_x \in \mathbf{T}, \forall f_i \in \mathbf{F}\}$. A *transaction* is formally defined 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} ($F_{id} \in \mathbf{F}$). 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 $\sum_{id} = \{r_{id}(f_i), w_{id}(fi) | \forall f_i \in F_{id}\}$.

  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: Let $\mathbf{F} = \{f_1, f_2, ...f_{11}\}$, $\mathbf{T} = \{t_1, t_2\}$ and t_b is the *backup transaction* where $t_1 = \{1, F_1, \pi_1\}$, $F_1 = \{f_4, f_8, f_{11}\}$ and $\pi_1 = a_1(f_{11}), a_1(f_4), a_1(f_8)$. $t_2 = \{2, F_2, \pi_2\}$, $F_2 = \{f_1, f_4, f_{11}\}$ and $\pi_2 = a_1(f_1), a_2(f_{11}), a_1(f_4)$. $t_b = \{b, F_b, \pi_b\}$, $F_b = F$ and $\pi_b = r_b(f_1), r_b(f_2), ..., r_b(f_{11})$.

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 string of access operations from the expanded set $\sum_C = \{\pi_1, \pi_2, ...\pi_m\}$ such that:

1. every symbol from $\sum_C$ appears exactly once in π_C;and
2. $a_x(f_i) < a_x(f_j)$ 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.

An individual schedule of the set $\sum_C$ which is part of the schedule π_C is said to be a *participant* schedule of the schedule π_C. Thus, each π_x, $x = 1, 2..m$ is 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 they adhere to the following properties

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

A schedule π_C is a serialized schedule if it is conflict equivalent to some serial schedule of the set $\mathbf{T}$. Formally, a schedule π_C is a serialized schedule, if every pair of conflicting operations $\{a_x(f_i), a_y(f_i)\}$ in π_C is ordered as

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

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 string of access operations from the expanded set $\sum_{Cb} = \{\pi_C, \pi_b\}$ such that:

1. every symbol from $\sum_{Cb}$ appears exactly once in π_{Cb};
2. The partial order of operation within π_C is preserved within π_{Cb}; and
3. π_C is a *serialized schedule.*

3.3 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 serializes π_{Cb}. User transactions use standard concurrency control protocols like strict 2PL, optimistic protocols to serialize π_C. But, 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. Hence 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 protocols to serialize π_C. 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* transaction

OR

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

  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. 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 can lead to schedules that are not serializable as can be seen from 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, 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 the precedence relationship in which the operations of the backup transaction are involved as $r_b << a_x$ or $a_x << r_b$ and so also $t_b << tx$.

We state and prove that we get serializable schedules by using *mutual serializability* in Theorem 3.3.1, once we state and show Lemma 3.3.1. The given proof uses a serialization testing tool called *serialization graph* 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(SG), 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 [70]. 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 then the relationship is denoted by $t_b << t_x$ or $t_x << t_b$.

Lemma 3.3.1. *Given that π_{Cb} is a schedule over $\mathbf{T} \bigcup t_b$, where $t_1, t_2, ..t_m$ are transactions in set $\boldsymbol{T}$ and t_b the backup transaction, and given that t_b is* mutually serializable *with every $t_x \in \mathbf{T}$,* **if** *in π_{Cb}, $t_b << t_1 < t_2 .. < t_m$,* **then** *$t_b << t_m$.*

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

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

$$< a_1, a_2 > \in \{< r_1, w_2 >, < w_1, r_2 >, < w_1, w_2 >\}. \tag{3.1}$$

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 $t_b << t_1$, $r_b(f_1) << a_1(f_1)$ must be in the schedule, where

$$< r_b, a_1 > \in \{< r_b, w_1 >, < r_b, r_1 >\}. \tag{3.2}$$

From 3.2 we get $r_b(f_1) << a_1(f_1)$. From 3.1 we get $a_1(f_1) < a_2(f_1)$. Therefore, by 3.1, 3.2 and the definition of *mutual serializability*, $r_b(f_1) << a_2(f_1)$ holds. $a_2(f_1) << r_b(f_1)$ cannot hold as t_b reads every file at most once.

Thus,

$$\textbf{\textit{if}}\ t_b << t_1 < t_2\ \textbf{\textit{then}}\ t_b << t_2. \tag{3.3}$$

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

Theorem 3.3.1. *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, $t_b << t_1 < t_2 << t_b$.

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 $\mathbf{F}$ atmost once and it is also given that t_b is *mutually serializable* with all $t_x \in \mathbf{T}$, $F_b = \mathbf{F}$. Therefore

$$t_b \textit{ reads } f_1 \textit{ exactly once and so } r_b(f_1) \textit{ occurs only once in the schedule } \pi_{Cb}. \tag{3.4}$$

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

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

But this implies, $r_b(f_1) << a_1(f_1) < a_2(f_1) << r_b(f_1)$ is part of the schedule π_{Cb}. But, this is not possible by 3.4 above.

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

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

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

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

But by Lemma 3.3.1, $t_b << t_m$ and so the cycle $t_b << t_m < t_{m+1} << t_b$ 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+1 transactions. So, $t_b << t_1 < t_2.... < t_m < t_{m+1} << t_b$ is not possible.

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

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 taking into consideration read-read conflicts in the *mutual serializability* protocol, may result in a possible cycle. For example, if $a_1 = r_1$, then it is no longer necessary that $r_b(f_1) << a_1(f_1)$ and 3.4 is not violated. However, $t_b << t_1 < t_2 << t_b$ 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 $r_b(f_i) << w_1(f_i)$ must hold and so $t_b << t_1$ will hold. Similarly, not considering read-read conflicts may still result in a serialized schedule as can be seen from the following example: $\{r_b(f_1), r_b(f_2), w_1(f_1), r_2(f_2),$

$w_2(f_3)$, $r_b(f_3)\}$. *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.

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

Chapter 4

Hierarchical File System Formal Model

In Chapter 3, by considering a transactional flat file system with transactions consisting of *read* and *write* operations, we have shown that by using *mutual serializability* as the backup program specific concurrency control algorithm, an online consistent backup-copy is ensured. Now, practical file systems are much more complex then the simple flat file systems of our model, 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 through operations such as *create* and *delete*. 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 chapter, we formally show that the proposed *mutual serializability* scheme will continue to work when considering 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.

4.1 Hierarchical File System

To extend the theory of online consistent 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 multilevel hierarchy called a directory tree, with regular files always occupying 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 which allows access to the file. It is called an inode number in ext2fs and is a unique file identifier in a flat inode number space. 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 ext2. Children of the same directory are called "siblings".

At the very top of the file system hierarchy is a single directory called "root" which is represented by a /(slash). A directory can be the child of only one parent directory, whereas 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. A *pathname* gives a unique name to a file and it also specifies how to traverse the directory hierarchy in order to find the files unique identifier. Absolute pathnames gives the file location from the root directory whereas relative pathnames gives the files location from a processes current working directory. Thus, 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 4.2 and Section 4.3, 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* on the consistency of a backup shall be dealt with in more detail in Section 4.5

A hierarchical file system like most practical file systems do not consist of a fixed set of files throughout its lifetime, i.e. they are not static file system. A hierarchical file system is dynamic with new files being inserted and existing once deleted. A hierarchical file system transaction accesses files through a number of operations that keep the file system static e.g., *link*, *unlink*, *rename* etc., and those that grow/shrink the file system e.g., *creat*, *unlink* etc.,

each accessing one or more files. Now, the file system model we used to prove our proposed, backup transaction specific concurrency control protocol namely *mutual serializability*, assumed the file system to consist of a static set of files and files to be accessed by only *read* and *write* operations. 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 4.3. Now, coming to the operations that grow/shrink the file system, though the *mutual serializability* protocol extends normally for the file *delete* operations, but not so for the operations that *inserts* new files. We notice that 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, the *mutual serializability* protocol shall hold correct in the presence of file inserts. These results are formally stated and proved in Section 4.4.

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, access operation to a directory maps directly to an access operation to a file and no matter how directory operations conflict within user transactions but with respect to backup transactions any access to a directory by a user transaction conflicts with the read access of the directory by the backup transaction. Thus, a hierarchical file system's transaction schedule too consist of sequences of read and write operations to files, where a file can either be a regular file or a directory.

Every file in a file system has a unique file identifier which identifies the file descriptor or inode of the file. An inode contains a file's metadata describing its attributes, such as file type, file length, user id etc. A file's meta data can be accessed and modified through inode operations like *chmod*, *stat*, etc. In our model we consider a file's descriptor to be a part of the file and any inode operation is thus an operation to the file itself.

A backup transactions *schedule* of a hierarchical file system too consists of a sequence of *read* operations to every file in the file system where a file is either a *regular file* or a *directory*. Moreover, as in the case of flat file systems, every file in the file system hierarchy is read *atmost* once by the backup transaction. We do not assume any particular order in which the backup transaction reads and backs up the files in a file system, but owing to the *parent* $\rightarrow$ *child* relationship existing between a directory and its children, the backup transaction traverses this relationship to reach each file. The *parent* $\rightarrow$ *child* relationship

may be traversed in either the forward direction (file system hierarchy traversed top-down and a parent is read before its children) or backward direction (file system hierarchy traversed bottom-up and a child is read before its parent).

4.2 The Formal Model

A *Hierarchical File System* $\mathbf{F^h}$ is a set of distinct files, $\{f_1, f_2, ...f_n, d_1, d_2, .., d_p\}$ where f_i, i=1,2,..n is a regular file and d_j, j=1,2,..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 regular file or directory. As stated above, in the present model, symbolic links are also treated as regular files. Thus, $file_k \in F^h$, k=1,2,..n+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 stated, files in a hierarchical file system exhibit an inter file hierarchical relationship also termed as $parent \rightarrow child$ 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 $\mathbf{T^h} = \{t_1, t_2..., t_m\}$ is the set of user transactions and t_b is the *backup transaction*. A transaction $t_x, x = 1, 2...m$ accesses a file $file_k, k = 1, 2, ..n + p$ 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 an element of the set $\{r_x(file_k), w_x(file_k) | \forall t_x \in \mathbf{T^h}, \forall file_k \in \mathbf{F^h}\}$. 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 $F_{id} = \mathbf{F^{hb}}$. 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 $t_i | \forall t_i \in T^h, t_i << t_b\}$, and $\{F^{insert}$ is the set of files created by $t_j | \forall t_j \in T^h, t_b << t_j\}$.
- 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 $\sum_{id} = \{r_{id}(file_k), w_{id}(file_k) | \forall file_k \in F_{id}\}$.

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

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 an online consistent backup of F^h be obtained by keeping t_b *mutually serializable* with each $t_x \in \mathbf{T^h}$.

4.3 Mapping of Hierarchical File System Operations

Mappings for the major file system calls in Linux to an equivalent sequence of *read* and *write* operations are given below. It can be easily seen that other minor system calls will have similar mappings and hence have not been explicitly given. We broadly divide the file system operations into two categories, ones that keep the data set fixed or static and ones that grow/shrink the data set.

4.3.1 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 3.3.1.

4.3.2 Operations Maintaining a Fixed Data Set

Regular file operations :-

read(): The read of a regular file, f_i directly maps to the read operation of our model i.e $r_x(f_i)$. It is to be repeated that this may involve the reading of the inode as well as the

data, and since this operation will be part of a transaction, atomicity is assured. The same reasoning applies to all the other operations that follow.

write(): A regular file write operation can similarly be directly mapped to a write operation of our model i.e. $w_x(f_i)$.

truncate(): The operation *truncate* modifies the size of a regular file to a given length by setting the "size" field in the file's inode to the required size and releasing data blocks no longer part of the file. So, *truncate* of a file f_i is therefore mapped to a single write operation, $w_x(f_i)$.

Directory Operations :-

readdir(): Directories are but specially formatted files in most file systems including ext2fs. The directory read access operation, *readdir* is thus similar to a regular file's read operation and is equivalent to a file read operation of the flat file system model i.e., $r_x(d_j)$ where d_j is the directory being read.

link: The *link* operation 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}. The $link()$ function 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 f_i 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 $\{(w_x(d_{j2}), w_x(f_i))\}$.

rename/move: 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. 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. So, in some implementations, the st_ctime (creation time) file of the inodes 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 seems the right thing to do. The *rename/move* operation is hence mapped as $\{(w_x(d_{old}), w_x(file_k), w_x(d_{new})\}$

Inode Operations :-

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 is considered as a single entity. Now, most inode operations can be directly mapped to either a *read* or *write* of the file and we shall look at the mapping of only a few inode operations here.

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., $w_x(file_k)$.

chown() is used to change the owner of a file and hence chown() acting on a file $file_k$ directly corresponds to a *write* of the file i.e., $w_x(file_k)$.

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., $r_x(file_k)$

4.3.3 Operations that Grow/Shrink the Data Set

So far we have only considered operations that do not add or delete files. Regular files are deleted via the *unlink()* call and *rmdir()* deletes a directory. Similarly, a regular file is created by the *creat()* call and directories are created by the *mkdir()*. We now map these operations 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 :

***regular file* removal:** 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* deleting operation maps as $\{w_x(d_j), w_x(file_k)\}$.

***directory* removal:** The rmdir() function removes a directory d_{j2} whose name is given by a path passed as an argument to the function. rmdir() removes a directory only if it is an empty directory. Removing a directory results in updating the parent directory by removing the reference to the directory and then removing the directory. Hence, mapping of the *directory* removal operation, is $\{w_x(d_{j1}), w_x(d_{j2})\}$, where d_{j1} is the parent directory. The fact that a directory can be removed only if it is empty 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 users 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 (t_x is a "before" transaction) and $file_k$ is never copied to backup or t_x is

serialized after t_b (t_x is an "after" transaction) and $file_k$ has been copied to backup, since $file_k$ is not accessed by any other user transaction after its deletion, the *mutual serializability* protocol extends comfortably to ensure a serializable $\pi_{C^h b^h}$.

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 suggested by the following example: As before, symbol " $<$ " is used to denote "precedes" in any schedule and when the backup transaction is involved the symbol is " $<<$ ". Say, $t_x << t_b$. Hence 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 $t_y << t_b$. But, $t_y << t_b$ cannot hold because t_b will never read $file_k$, as it has been created "$after\ t_b$.

The following section delves into the issue of capturing a consistent file system backup in the presence of file *create* operations. We revisit the *mutual serializability* protocol and modify it to handle file *creates*.

4.4 Enhanced *Mutual Serializability*

To extend the *mutual serializability* backup specific concurrency control protocol so that it efficiently handles insertion of new file into the 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 $\{w_x(d_j), creat\_node_x(file_k)\}$.

The set of file access operation now includes *creat_node* apart from the original *read* and *write* operation. Formally, $a_x(file_k)$ is an element of the set $\{r_x(file_k)$,$w_x(file_k)$, $creat\_node_x(file_k) | \forall t_x \in \mathbf{T^h}, \forall file_k \in \mathbf{F^h}\}$.

On including a new operation, the set of conflicting operations must be extended to include interactions with the new operation in a way such that the definition of conflicting operation is not violated. Considering two operations to conflict if they belong to different transactions, act on the same file and both are not *read* operations [18], 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 needs to be modified. Again, we redefine *mutually serializable* with the only difference from its original definition being the set of access operations which now include *creat_node*.

Mutually serializable: Let concurrently executing transactions $t_x, t_y \in (\mathbf{T^h} \cup t_b)$ access(read,write or creat_node) into the set of files $F_x, F_y \in \mathbf{F^h}$ 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* transaction

OR

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

  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 $t_x << t_y$ then the following sequence of operations cannot occur: $w_x(file_i) << creat\_node_y(file_i)$ and $r_x(file_i) << creat\_node_y(file_i)$. Similarly if $t_y << t_x$, the following sequence of operations cannot occur: $w_y(file_i) << creat\_node_x(file_i)$ and $r_y(file_i) << creat\_node_x(file_i)$. It is simply because, a file cannot be read or written into before it is created. Given these basic definitions we observe that, $\pi_{C^h b^h}$ 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 $t_b << t_y$ and $creat\_node_y(file_k)$ is in π_y **if** there is a $t_x \in \mathbf{T^h}$ such that t_x reads or writes into $file_k$ and $t_x << t_b$. Thus, this gives rise to a modified version of the protocol *mutual serializability*. We state and prove that we get serializable schedules by using the modified *mutual serializability* protocol in Theorem 4.4.1. The proof of Theorem 4.4.1 is similar to the proof of Theorem 3.3.1, once we show that Lemma 3.3.1 continues to hold with the set of file access operations now including *creat_node* and set of conflicting operations extended to include the conflicts with the *creat_node* access operation.

Theorem 4.4.1. *Given π_{C^h} is a serializable schedule, $\pi_{C^h b^h}$ is serializable* ***if*** $\forall t_x \in \mathbf{T^h}$ *the following hold:*

1. t_b, t_x *are* mutually serializable *and*

2. *if* $t_x << t_b$, *then there is no* $t_y \in \mathbf{T^h}$ *such that*

   (a) $t_b << t_y$, *and*

   (b) *there is a* $creat\_node_y(file_k)$ *in* π_y

   (c) $creat\_node_y(file_k) < a_x(file_k)$ *for some* $a_x(file_k)$ *in* π_x *where 'a'* $\in \{r, w\}$.

Proof. We prove Theorem 4.4.1 by first showing that Lemma 3.3.1 continues to hold when condition 2 is added. Thus, we first show that the following statement holds:

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

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

Let t_1, t_2 access (read ,write or creat_node) 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

$$< a_1, a_2 > \in \{< r_1, w_2 >, < w_1, r_2 >, < w_1, w_2 >, < creat\_node_1, r_2 >, < creat\_node_1, w_2 >\}. \tag{4.1}$$

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_z such that $t_b << t_z << t_1$ 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 $t_b << t_1$, $r_b(f_1) << a_1(f_1)$ must be in the schedule, where

$$< r_b, a_1 > \in \{< r_b, w_1 >, < r_b, r_1 >\}. \tag{4.2}$$

From 4.2 we get $r_b(f_1) << a_1(f_1)$. From 4.1 we get $a_1(f_1) < a_2(f_1)$. Therefore, by 4.1, 4.2 and the definition of *mutual serializability*, $r_b(f_1) << a_2(f_1)$ holds. $a_2(f_1) << r_b(f_1)$ cannot hold as t_b reads every file at most once.

Thus,

$$\textbf{\textit{if}}\ t_b << t_1 < t_2\ \textbf{\textit{then}}\ t_b << t_2. \tag{4.3}$$

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 $t_b << t_1$ 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 *mutual serializability*, t_b is *mutually serializable* with t_2 as there is no dependency between t_b and t_2. So $t_b << t_2$ holds.

If F_b intersect F_2 is not null, and is say f_2, then depending when t_b reads f_2, $t_b << t_2$ or $t_2 << t_b$ will hold. But if $t_2 << t_b$ 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, $t_b << t_2$ 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 statement **A** continues to hold.

Having shown this, we now prove the theorem:

We use mathematical induction to show that $\mathrm{SG}(\pi_{C^h b^h})$ is acyclic and hence $\pi_{C^h b^h}$ is a serialized schedule.

We proceed with the knowledge that π_{C^h} is a serialized schedule and hence there cannot exist a cycle in $\mathrm{SG}(\pi_{C^h b^h})$ involving only transactions from the set $\mathbf{T^h}$.

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

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

Let t_1, t_2 access (read, write, creat_node) 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 $< a_1, a_2 > \in \{< r_1, w_2 >, < w_1, r_2 >, < w_1, w_2 >, < creat\_node_1, r_2 >, < creat\_node_1, w_2 >\}$.

We need to consider two separate cases:

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

Now, by the definition of t_b, t_b reads every file in $\mathbf{F^{hb}}$ at most once (and it reads f_1 since it has not been created by t_1 or t_2; the case of another transaction t_z such that $t_b << t_z << t_1$

and t_z creates f_1, is covered by Case II below).

$$So\ r_b(f_1)\ occurs\ only\ once\ in\ the\ schedule\ \pi_{C^h b^h}. \tag{4.4}$$

By definition of *mutual serializability*, since $t_b << t_1$, t_b reads f_1 before t_1, i.e $r_b(f_1) << a_1(f_1)$ is part of the schedule $\pi_{C^h b^h}$.

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

But this implies, $r_b(f_1) << a_1(f_1) < a_2(f_1) << r_b(f_1)$ is part of the schedule $\pi_{C^h b^h}$. But, this is not possible by 4.4 above.

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

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 $t_b << t_1$ 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 *mutual serializability* t_b is *mutually serializable* with t_2 as there is no dependency between t_b and t_2. So $t_b << t_2$ holds and there is no cycle involving two transactions and t_b.

If F_b intersect F_2 is not null, let this be a single file f_2. For a cycle to exist $t_2 << t_b$, but, this violates the second condition of the theorem where $t_x = t_2$, and $t_y = t_1$. Therefore, even in this case, there is no cycle involving two transactions and t_b.

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.

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

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

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

But by **A** above, $t_b << t_m$ and so the cycle $t_b << t_m < t_{m+1} << t_b$ 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+1 transactions. So, $t_b << t_1 < t_2.... < t_m < t_{m+1} << t_b$ is not possible.

Thus, Theorem 4.4.1 follows.

□

4.5 Path Lookups

One of the issues that need to be considered in a hierarchical file system is, given the pathname of a file, the process of traversing the file system tree to find the inode number of the file or the *path lookup* operation in Linux terminology. The issue is whether the reads of directories during such traversals (or searches, as they are called in Linux terminology) can conflict with other reads of, and writes to, these directories. We discuss this issue with attention on the possible interactions between the backup transaction t_b and a user transaction t_x. We assert that there is no need to consider conflicts between a directory's read by t_b and a search in that directory during regular reads and writes of t_x to ensure *mutual serializability* between the two transactions. It is so because, path lookup is done by the kernel always within one system call and never by user programs by reading directories one by one and t_b captures the backup of the file system and not of any particular file. So it is not given any file pathname to back up neither 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.

4.6 Realizable Schedule

Our model has assumed that t_b reads every file exactly once, because this is what a backup means. In an actual implementation, transactions including t_b may have to abort (roll back) and redo operations. 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.

A realizable schedule is one in which t_b should never have to *roll back*. To understand why, we explain in terms of the following example: Consider the following schedule, $\{...., w_1(f_1), r_b(f_1), r_2(f_1), w_1(f_2), r_b(f_2), w_2(f_3),...\}$. t_1 precedes t_b in serialization order and once t_b reads the modified data of t_1, t_1 aborts rendering t_b inconsistent. t_b only *reads* every file in the file system at most once and hence t_b's consistency can be reestablish by *rolling back* t_b's read of only the files accessed by t_1 i.e., f_1 and f_2. Unlike in the case of a regular transaction, a total abort of t_b is not necessary. *Rolling back* t_b means *undoing* and then *redoing* the read of the files accessed by an aborted transaction. Now, looking at the example again we see that t_b's *roll back* does not end here. On *rolling back* the read of f_1 and f_2 by t_b, transaction t_2 which earlier succeeded t_b, now loses its *mutually serializable* relation with t_b. To reestablish t_2's *mutually serializable* relationship with t_b, either t_2 is aborted or t_b *rolls back* the read of the other files accessed by t_2 and in this case it is f_3. If t_b *rolls back* then it can easily be seen that in the worst case scenario t_b has to *roll back* all

its read 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 a *realizable schedule*, $\pi_{C^h b^h}$ is one in which t_b never needs to roll back its reads of files.

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 committed transaction or itself. The resulting serializable schedule is called a *cascadeless schedule* [18]. Formally, a schedule is said to be *cascadeless* if whenever t_x reads $file_k$ from t_y, $(x \neq y)$, $commit_y < r_x(file_k)$. Where $commit_y$ denotes the commit of a transaction $t_y \in \mathbf{T^h}$. 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 4.6.1.

Theorem 4.6.1. *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 $t_x \in \mathbf{T^h}$, $commit_x < r_b(file_k)$ 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.

Chapter 5

Implementation

Given the formal framework for consistent backup of an online file system, in the present chapter we put forward algorithms and describe our practical implementation of an online backup solution designed to realize the theoretically proven results. The backup utility and the solutions for ensuring *mutual serializability* (MS) of each user transaction with the backup transaction are designed by assuming a hierarchically organized transactional file system.

Transactions provide a clean approach for concurrent access of shared data apart from its many other benefits. Owing to these obvious advantages, and fueled by the positive performance evaluations of past research [59, 61, 62], recent years have seen accelerated research in the direction of incorporating transactional properties in file systems [45, 51, 67, 69, 12, 60, 72]. As a result several prototypes as well as fully functional transactional file systems have been designed and implemented to provide transactional interface to files. But, these systems either exist as research projects still under development and evaluation [51, 67] or closed-source systems limited to specific file system and operating system [69]. Thus, we were faced with the challenge of not having any underlying transactional file system for implementing our proposed consistent online backup approach. Under such circumstances, we were left with the only option of implementing our own basic transactional file system. We refer to our file system as TxnFS. It 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. The online backup specific concurrency control protocol, *mutual serializability* is then evaluated for correctness and the amount of performance degradation incurred due to the online backup. Knowledge gathered from

studies of file system access patterns such as temporal and spatial locality of references are then used to improve performance of the *mutual serializability* protocol.

The current chapter is organized in the following manner: the design and implementation of TxnFS is described in Section 5.1. Section 5.2, gives details of the implementation of mutual serializability and an online backup utility. .

5.1 Design and Implementation of TxnFS

The primary goal while implementing a transactional file system was to implement an online file system backup utility and evaluate the performance and correctness of the MS concurrency control protocol. While achieving the said objective, the present study has also successfully contributed an alternative transactional file system implementation which without much complexity can easily be added over existing non-transactional file systems and can provide transactional security to application programs. We now present in detail the design of TxnFS by first introducing and discussing various alternative designs, as well as the level in the software hierarchy where best to insert the TxnFS prototype.

Let us first define a few basic concepts. The term *file* refers to a storage data object for storing data and is associated with attributes such as, the owner of the file, its size, the time it was last accessed etc. The on-disk organization and management of these *files* and the procedures needed to query and modify these data objects are collectively referred to as the *file system*.

Transactional file systems, have been built in the past as an entirely new file system based around transactions from the ground level, such as Window's native transactional NTFS [69], or modifying and building a prototype on top of some existing system as done in the Sleepycat Software [12] and by *Wright et al.* [72], *Seltzer et al.* [61, 62] and *Spillane et al.* [67]. The complexity and amount of code required to build an entirely new file system as well as the debugging and testing involved pushed us to code around some existing system. Common approaches are to implement transactional file systems around systems providing transactional properties such as the two systems over the Berkeley Database [72, 12] or around an existing file system as proposed by *Seltzer et al.* [62, 61] and *Spillane et al.* [67]. A file system interface built around an existing transactional system is faced with the issue of choosing between following the path through the standard VFS layer or to bypass the VFS. The first approach requires keeping track of all cached objects like inodes and directories, so as to facilitate their rolling back in scenarios of a transaction aborting. Bypassing the VFS requires the new system to duplicate tasks like caching thus increasing the amount as well as complexity of the code. Implementing using a DBMS may not yield good performance as transactional systems like databases are not tuned for file system based applications.

Developers have added transactional properties around existing file systems with positive results benefitting from the stability of the tried and tested underlying file system. We have therefore opted to develop TxnFS over an existing file system. Researchers have augmented existing file systems to benefit from transactional semantics by modifying or extending kernel code [67] or by developing functionality in user space over an underlying file system [61, 62]. User-space file systems incur overhead due to multiple context switches, but on the other hand, extending an existing file system in the kernel involves complex coding and debugging and the resulting system is not portable across different platforms.

Keeping in mind our main goal of showing the feasibility of our proposed backup scheme, and the constraints of time, we designed and develop our transactional file system in a user level process. The implementation is not very efficient as it is implemented at the user level. Further, the logging process has been modeled on the DBMS logs, and as already mentioned in an earlier chapter, this is not very efficient for large writes to files. Figure 5.1 illustrates the top level TxnFS architecture.

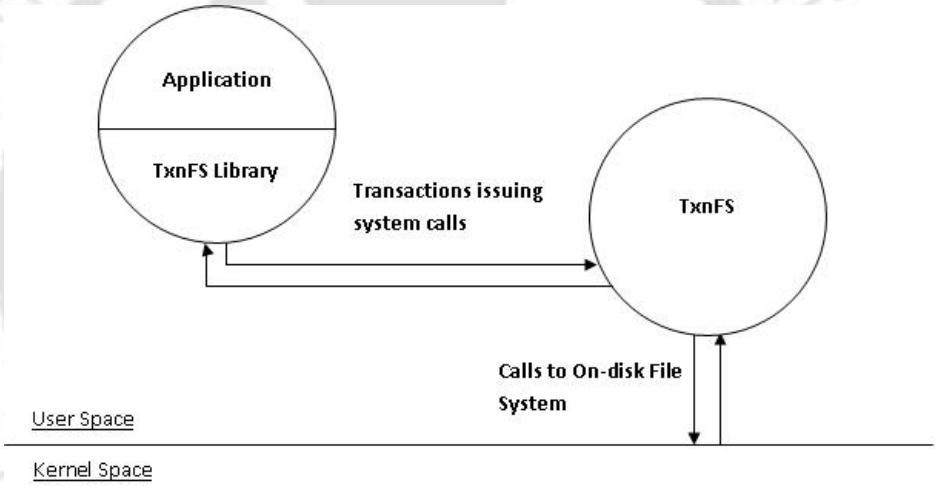


Figure 5.1: Top Level TxnFS Architecture

5.1.1 Transactional Model

5.1.1.1 TxnFS Specific System Calls

The transactional interface provides three additional function calls, namely *txn_begin*, *txn_commit* and *txn_abort* and applications desiring to protect groups of one or more system calls with transactional semantics should wrap them with these function calls.

txn_begin signals the beginning of a transaction and it must be called before any other operations belonging to the transaction.

txn_commit signals the successful termination of a transaction. It signals that all updates

made by the transaction must now be made permanent.

txn_abort is used to request the transaction to be aborted and none of the updates made by it should be recorded in the file system.

5.1.1.2 TxnFS's Transactional Guarantees

A system providing transactional semantics should ideally support the ACID (Atomicity, Consistency, Isolation and Durability) properties. The *atomicity* property requires that either all or none of the modifications done by a transaction be applied to the file system. The *consistency* property requires that the file system state is left consistent by a transaction which either aborts or commits, in the presence of other concurrent transactions. *Isolation* requires that the updates made by one transaction does not affect concurrently executing transactions. In other words the effect of concurrently executing transactions should be equivalent to the transactions executing sequentially. The *durability* property requires the effect of committed transactions to be permanently recorded across any form of system failure [61].

TxnFS guarantees the transactional properties of atomicity, isolation and consistency to application transactions running on top of it. Though the infra-structure required for supporting the property of durability is in place but TxnFS at present cannot guarantee durability. Adding the durability property is possible in the present design and shall be part of future work.

5.1.1.3 Realizing TxnFS's Transactional Guarantees

TxnFS supports the above stated transactional properties through the techniques of *locking* and *logging*as detailed in the current section.

Locking: TxnFS ensures the property of *isolation* through the technique of *locking*. User applications do not need to make explicit requests to "lock" and "unlock" files. Files are locked implicitly by TxnFS in *shared* or *exclusive* mode before files are "opened" for reading or updating respectively. By releasing all locks held by a transaction only at the time of commit or abort, TxnFS thus implements the *strict 2PL concurrency control protocol* to achieve the transactional property of *consistency* in presence of multiple accesses. Locking is done at the granularity of an entire file including its inode. Finer granularity locking such as at the level of per page [67] or more sophisticated locking performed to reduce contention [61] is beyond the scope of the present implementation. The Strict version of the 2PL protocol was chosen to avoid rollback of the backup transaction, as discussed in the previous chapter. Other concurrency control schemes like optimistic concurrency control and timestamp ordering were not considered due to time constraints. But, it must be stressed

here that the *mutual serializability* concurrency control method specific to the backup utility will work perfectly even when accompanied with other strict concurrency control methods.

Logging: Atomicity and durability (not implemented in the current system) of transactions are ensured through TxnFS's write-ahead logging mechanism. The logging protocol records a transactions begin and end (commit/abort) operations, every update operation, and the after-image of the data acted upon by the operation in case of a *write* operation. Log records of all transactions are appended to a single log file stored on a separate partition. TxnFS applies updates in-place only at the time of a transaction commit, after a commit record is appended to the log and the log is forced to stable storage. We use the term *deferred write* to refer to the technique of writing dirty data to the file system only at the time of commit. As the log file stores the after-image of updated data, the resulting log is a redo log and in case of a system crash, updates of all committed transactions not yet written in-place can be redone from the log. Many logging schemes store before-images of the updated data to facilitate rollback of modifications done by transactions that later abort or end ungracefully because of a system or process crash [67, 61]. TxnFS's method is the same as that used in journaling of file systems, such as that used by JBD in the ext4 file system [46]. Our proposed *deferred write* mechanism also does away with the complex issues highlighted by *Spillane et al.* [67] and *Wright et al.* [72], of rolling back dirty pages in the VFS cache as well as stale in-memory data structures such as inodes, dentries etc., when a transaction aborts.

5.1.2 Module Architecture and Implementation

In this section we detail the entire system including each module and data structure required to augment an ext2 file system with an interface providing transactional properties to all applications running on it. Portions of TxnFS's architecture design follows from that of the transactional library, LibTB [61], but the two libraries have significant differences specially in the *logging* and *concurrency control* protocols. Moreover, LibTB's access interface is that of 4.4BSD databases whereas TxnFS's interface is the standard POSIX compliant API apart from the inclusion of three calls ($txn\_beg, txn\_commit, txn\_abort$) required to support transactions. The TxnFS user space file system is implemented as a multi-threaded server process, where each transaction is processed by a single thread which terminates when a transaction commits or aborts. A pictorial representation of TxnFS's modular architecture is depicted in Figure 5.2. We note here that arrows depict function calls from one module to another and also to the underlying file system store and most function calls return values (including errors) to the calling modules. We do not explicitly show the return flow of control in the concerned figure for the sake of clarity.

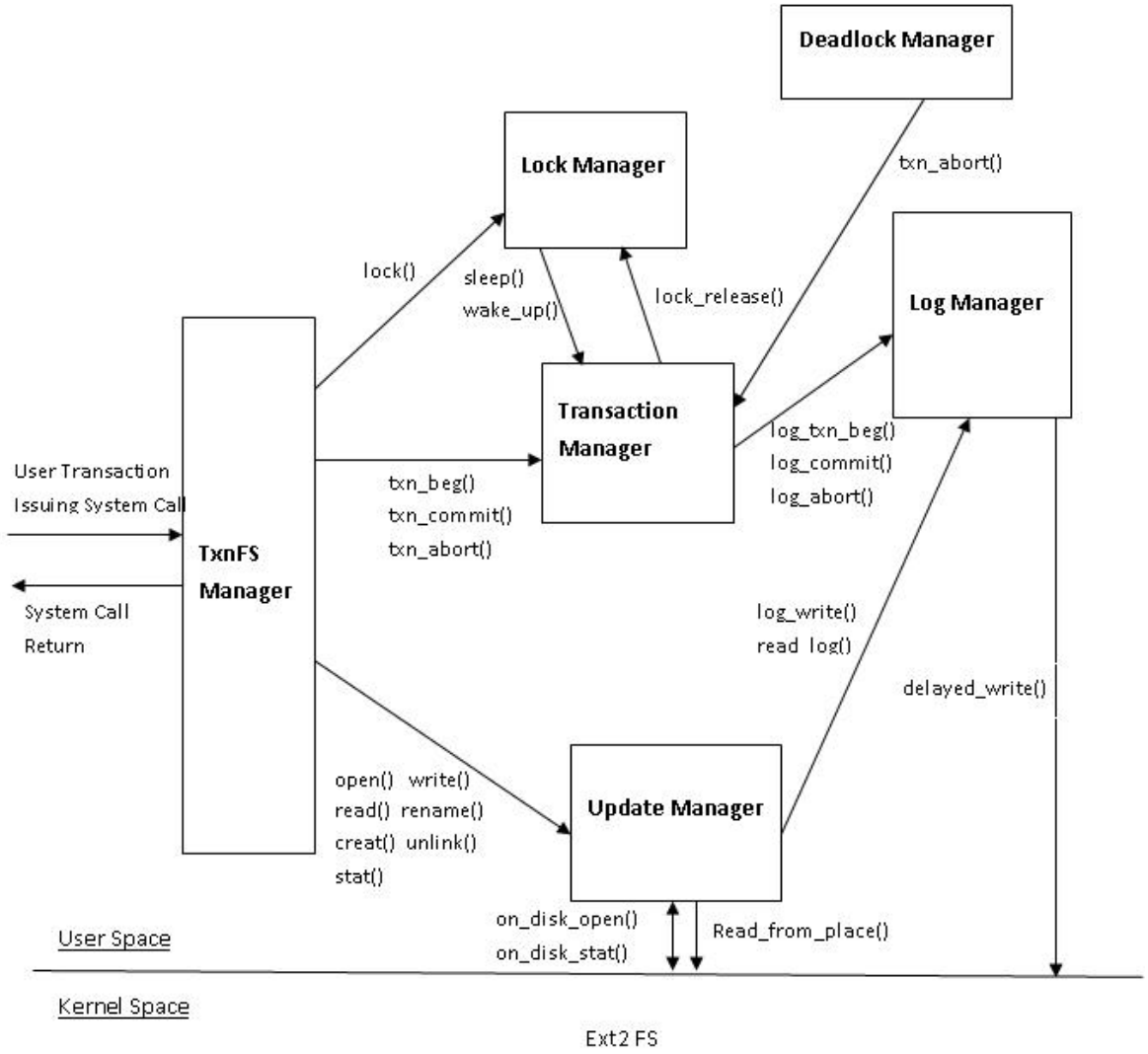


Figure 5.2: Modular Architecture of TxnFS

5.1.2.1 TxnFS Manager

Every transactional system call issued by user applications enters the Transactional File System(TxnFS) through the **TxnFS manager** and the return value of each system call is returned to the concerned application transaction by this module. The **TxnFS manager** manages the flow of control within the system by calling appropriate routines in the other modules. For example, on receiving an *open*() call to a file from an application transaction, the **TxnFS manager** calls routines in the **Lock manager** to check and either assign the appropriate file lock to the calling transaction or to put the transaction in a waiting list if the lock associated with the requested file is not available. Basic error handling is also performed by the **TxnFS manager**, such as checking whether a file exists before sending a lock request to the **Lock manager** or checking whether a file has been opened for writing

on receiving a write() from a user transaction.

5.1.2.2 Log Manager

As mentioned, a write-ahead redo log has been implemented to ensure the transactional property of atomicity. Towards this end, the **Log manager** is responsible in maintaining a chronologically ordered log file and it logs each update operation and the *txn_begin*, *txn_commit*, *txn_abort* *txn_end* operations. The **Log manager** is directed to make log entries by the **Update manager** and the **Txn manager**.

A log record in the log contains a monotonously increasing *Log Record Number*(LRN), *txnid* of the transaction responsible for the entry, pointers to the previous and next record of the same transaction to facilitate *deferred write* at commit time, type of record(*txn_begin*, *txn_commit*, *txn_abort*, *txn_end*, *write*, *rename*, *creat* and *unlink*), the entire system call, and an after-image for *write* operations.

The **Log manager** facilitates executing the *strict 2PL* concurrency control protocol by performing the *deferred write* operations from the log to the file system after a transaction commits and the log file has been forced to disk. At commit time, the **Txn manager** directs the **Log manager** to perform *deferred write* of every update operation of the transaction. On receiving the direction, the **Log manager**, forces the log to disk, beginning at the first record of the committing transaction (whose LRN is stored in the *txn_list* entry of the transaction) and performing *deferred write* of every update operation in the chronological order in which they were issued. On completion, control is returned to the **Txn manager**. On receiving an *txn_abort* signal, the **Log manager** simply appends a record to that effect. Similarly it appends a *txn_end* record to the log against the just committed/aborted transaction when directed to by the **Txn manager**.

5.1.2.3 Txn Manager

The **Txn manager** (or Transaction manager) services the *txn_beg*, *txn_commit* and *txn_abort* application transaction calls when it needs to start a new transaction, commit a running transaction or abort a running transaction respectively. These calls are normally made by the **TxnFS manager** and when deadlocks occur, *txn_abort* is called by the **Deadlock Detection** module.

The **Txn manager** keeps track of all "live" transactions in the system whether they are running or blocked, with the help of an in-memory data structure referred to as *txn_list*. There is one entry per "live" transaction and each entry in the *txn_list* is indexed by the identity number of the transaction (txnid). An entry further stores a pointer to the list of locks held by this transaction, the first LRN and the last LRN of this transaction, the

transactions current state (running, blocked, aborting, or committing) and inode number of the file it is waiting for if its state is currently *blocked*.

On a *txn_beg*, the **Txn manager** assigns a monotonously increasing unique identifier (txnid) to the transaction and inserts a new node for the transaction in *txn_list*. It then calls the **Log manager** to append in the log a *txn_beg* record. On receiving a txn_commit the **Txn manager** directs the **Log manager** to initiate a *commit* by performing *deferred writes*. On a *txn_abort*, the **Log manager** simply appends a record to the log. On return from the **Log manager**, the **Txn manager** calls the *release_lock* function of the **Lock manager** to release all locks held by the transaction. After this, the **Txn manager** removes the transaction from the *txn_list* and requests the **Log manager** to record a *txn_end* record against the commited/aborted transaction, and returns control to the application through the **TxnFS manager**. A transaction can also be aborted due to a deadlock situation. In such a case *txn_abort* is called from within TxnFS from the **Deadlock Detection** module and after aborting, the aborted transaction is informed and its execution thread terminated.

5.1.2.4 Lock Manager

The **Lock manager** is responsible for assigning locks to requesting transactions and releasing those locks when the transaction commits or aborts. Locks are requested and assigned at the granularity of an entire file including its inode. The **Lock manager** is invoked from the **TxnFS manager** through the interface *insert_lock_list* requesting for a lock and from the **Txn manager** through the interface *release_lock* to release locks on commit or abort. The **Txn manager** directs the **Lock manager** to release locks only after the log file has been forced to disk and *deferred write* is completed by the **Log manager**.

A file can be locked in either exclusive or shared mode. A transaction requesting an exclusive lock is called a writer and one requesting a shared locked is called a reader. The locking protocol supports either a single writer or multiple readers per lockable entity and by releasing all locks held by a transaction only at the time of commit or abort, the **Lock manager** enforces the *strict2PL* concurrency control protocol. If a request for an incompatible lock is received, the request is placed in a queue and the thread servicing the requesting transaction is blocked. When a lock is released by a transaction on commit or abort, any blocked transaction is woken up. For a file currently locked in shared mode, to prevent a *writer* from starvation, a requesting reader is made to wait in queue behind the writer.

The **Lock manager** keeps track of all locks through a *lock_list* in-memory data structure. Each entry in the list contains the *inode number* of the locked entity, the file descriptor if the entity has been opened for reading or writing, file offset position (updated on write, read and lseek by **Update manager**), identity of the parent transaction and the locked mode.

An entity in the *lock_list* can be locked in either exclusive or shared mode. Upgrading and down-grading a lock is not supported currently, but can easily be included. If more then one transaction holds a shared lock, the list contains as many entities as the number of transactions sharing the lock. Locks of any particular transaction are chained together and a pointer to it is stored against the concerned transactions data structure in the *txn_list*. This facilitates in releasing locks during a commit or abort operation. Entries sharing a lock are also chained together and this facilitates in deciding whether to wake-up any transaction when any transaction releases its shared lock. The entry in *lock_list* of an entity locked in exclusive mode contains a pointer to a list of *log record numbers* where the corresponding update operations are logged.

5.1.2.5 Update Manager

The **Update manager** is responsible for orchestrating the various operations on regular files, directories and inodes once they have been locked. It receives directions from the **TxnFS manager**.

On receiving a *read* request, the **Update manager** extracts the *offset* position within the file from the concerned *lock_list* entry and "gathers" the requested bytes of data from either the log or from the original file or both. Reading may take place from the log either partially or fully if the requested data has been written by an earlier write from the same transaction. The "gathered" data is returned to the application through the **TxnFS manager**.

On receiving an update request, the **Update manager** directs the **Log manager** to log the update operation and the data to be written (after-image). To avoid redundant writes to the log, if the same location is rewritten in at least three consecutive calls to *write*, the *writes* are buffered in memory and written to the log in a single record. By more aggressive caching these writes to the log can be further reduced, but this has not been implemented in our prototype. The **Update manager** also updates a files *offset* position if necessary in the in-memory data structure *lock_list*.

It also performs preliminary error checks such as if a write on a file is requested, the **Update manager** checks if the file is a regular file and has been opened for writing. Another example of an error check is whether a file requested to be renamed, exists or not. If such checks fail, the caller transaction is informed and is forced to abort.

5.1.2.6 Deadlock Detection

A *deadlock* situation arises when two or more concurrent transactions have successfully *locked* one or more files and are *blocked* while waiting in a circular chain for another *blocked* transaction to release a *lock*. A deadlock situation can either be prevented, avoided or

detected and resolved. We resort to detecting and resolving a deadlock in TxnFS.

A thread is dedicated to deadlock detection, which wakes up at regular intervals to detect whether a deadlock has occurred and in case of a deadlock the execution thread resolves it. Let us call this thread the **Deadlock manager**. Resolving a deadlock is done by aborting a randomly chosen transaction which is a participant of the deadlocked circular chain.

Deadlock detection is done by constructing a *wait-for-graph* of the *blocked* transactions and then checking for cycles in the graph. The presence of a cycle indicates the presence of a deadlock. In a *wait-for-graph* the nodes represent the *blocked transactions* and an edge from t_x to t_y implies that t_x is *waiting* to lock a file currently being held by t_y in a conflicting mode. A *blocked* transaction's entry in the *txn_list* stores the inode of the file for which the transaction is currently waiting. Each entry in the *lock_list* stores the *txnid* of the transaction currently holding the corresponding lock to a file. The **deadlock detection** module uses the data structures *txn_list* and *lock_list* to construct a *wait-for-graph*. The *wait-for-graph* is then traversed to detect any cycles. If present, a transaction in the cycle is chosen at random and *aborted* to resolve the deadlock.

5.1.3 Transactional Handling of File Operations

In this section, we briefly look into how TxnFS handles various file operation requests. As mentioned above, preliminary error checks for each file operation is performed within the **TxnFS manager** before calling appropriate routines within other TxnFS modules. Such error checks include checking the existence of a file, whether the file type is compatible with the corresponding file operation, and issues like whether a file on which a write request has been made is *opened* for writing or not.

open(): On *open()*, the **Lock manager** attempts to lock the file (sent as an argument to open()) in either shared mode or exclusive mode as requested. On successful locking, a call to *open* the on-disk file is made to the underlying file system through the *on_disk_open*() function by the **Update manager**. The return value (the file descriptor on success or -1 otherwise) is sent to the calling application transaction through the **TxnFS manager**. The calling transaction's execution thread is blocked if the file could not be successfully locked. In our experiments a file is opened for possible succeeding calls to read(), write(), stat(), and lseek() .

close(): A call for a file *close()* is deferred by TxnFS to the time of *txn_commit*() or *txn_abort*() as locks are released and files are closed only at the time of commit or abort in accordance to the implemented strict 2PL protocol.

lseek(): On receiving an *lseek()* call the **Update manager** updates the *file offset* value stored in the corresponding entry within the *lock_list* in-memory data structure to the requested value after performing necessary error checks. This value is used by a succeeding *read* or *write* to the file by the same transaction.

read(): On receiving a *read()* call, the **Update manager** checks the *file offset* corresponding to the file descriptor (sent as an argument in *read*) stored in the *lock_list* data structure. With the help of this value, the **Update manager** "collects" the to-be-read contents from the on-disk file (if required portion has not been overwritten or the file was opened only for reading) or the *log* (if the file has been opened for both reading and writing and the transaction has previously written on the requested portion) or from both (if the file has been opened for reading and writing and the requested portion has been partially written by the same transaction) and passes the "collected" read data back to the calling transaction.

write(): The **Update manager** requests the **Log manager** to log the call as well as the "after" image and then updates the *file offset* in the concerned files entry in the *lock_list* together with other related updates.

creat(): On receiving a call to *creat()* a file, the **TxnFS manager** directs the **Lock manager** to lock in exclusive mode, the parent directory under which the request to create the new file has been made. Only on successful locking of the parent directory the *creat* request is processed by the **Update manager** by directing the **Log manager** to log the call and requesting an in-place creation of the file in the underlying file system at the time of commit.

unlink(): On receiving a call to *unlink()* a file, the *unlink()* operation is processed only after the **Lock manager** successfully locks in exclusive mode the to-be-unlinked file and its parent directory. The **TxnFS manager** only sends lock requests to the Lock manager if the to-be-unlinked file is a regular file. After locks have been successfully acquired, the **Update manager** directs the **Log manager** to log the *unlink()* call and performs an in-place *unlink* upon transaction commit.

rename(): The call to *rename* a file is handled by the TxnFS after the concerned file and its old and new parents are locked in exclusive mode and if the file is a directory all of the file's descendants are recursively locked in shared mode. This is the accepted norm of handling concurrent *renames* [67]. Once all files are successfully locked, the **Update manager** logs the call through the **Log manager** and performs an actual *rename* at the time of commit.

stat(): A *stat()* call to the underlying file system is made through the *on_disk_stat* function by the **Update manager** after the file sent as an argument is successfully locked in shared mode. This call is not logged.

5.1.4 Application Programmers Interface to TxnFS

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. During runtime of an application program the LD_PRELOAD environment variable facilitates the loading of this library before any other including the standard C library, libc.

5.1.5 Contributions and Limitations of TxnFS

Contributions: Through the implementation of the TxnFS prototype, we have successfully added a transactional layer that converts any existing file system into a file system providing transactional protection to all applications running on it. The underlying existing file system is not modified in any way and hence, TxnFS benefits fully from the established stability and performance of the file system.

In providing a transactional file system interface, one of the main challenges is the handling of the VFS cache on an *abort*, which necessitates the *roll back* of dirty pages as well as the stale data structures such as inodes, deentries etc. We tackled this issue with the help of a write-ahead log which implements a *deferred write* policy in which all updates done by a transaction are written in place only at the time of commit. Till commit time, it is the log which holds all updates. The disadvantage here is the time taken to commit a transaction and also the overheads during reads of modified data, but it saves the time and complexity involved in *rolling back* in case of an abort or during a system crash when the system is left inconsistent by uncommitted transactions.

Limitations: While designing and implementing TxnFS, our focus was on a correct, working prototype of a transactional file system and not much attention was paid to the performance of the system. Thus it is not surprising that TxnFS has a number of limitations. A user space implementation of TxnFS leads to a degradation in performance which can be avoided through an in-kernel implementation. Moreover, TxnFS's coarse granularity locking at the granularity of a whole file increases contention and decreases throughput. But, it should also be noted that implementation and execution of such coarse granularity locking is simple and its disadvantages are not very visible when there are very infrequent inter-process file sharing. But, file sharing will definitely increase in future and in-kernel implementations

will be required.

5.2 Implementation of Consistent Online Backup

With a functional transactional file system now in place, we turn to designing and implementing an online backup utility and ways of realizing the *mutual serializability* concurrency control protocol. In Section 5.2.1, we present a conceptual understanding of how the backup utility and the backup specific concurrency control solutions are seamlessly integrated on top of the TxnFS implementation. In-depth design and implementation details are presented in Section 5.2.2

5.2.1 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. Now, 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. We distinguish transactions as read-only and update transactions. 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 files 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 et al.* [53]. A *read* bit of value 0 indicates that the file has as yet not been read by the backup transaction and 1 indicates that it has. This bit of all files is initialized to 0 before a backup program starts. But, initializing the *read bit* of the entire file system before every backup may not be very efficient and so a sequence number of the backup transaction may be used instead, where the sequence number of the present backup transaction is one greater then its immediate predecessor. As our implementation is at the user level, we cannot make changes to the inode structure and so we use a single bit by utilizing the sticky bit of an inode.

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 reading it and on successful reading, unlocks the file 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 a file, the user transaction verifies whether it continues to be mutually serializable with respect to the backup transaction by comparing the *read* bit of the file with its own *before-after* bit. The following table enumerates mutually serializable checking .

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

Table 5.1: Establishing Mutual Serializability.

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, as already discussed in Chapter 4, although this solution does not hamper the execution of user transactions, it may lead to a cascading roll back of backup "reads" as roll backs may render previously consistent transactions to now be mutually non serializable. In the worst case scenario, the backup transaction has to be restarted.

Another method of handling the problem is to abort and restart t_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 1 and it attempts to access a file whose *read bit* is 0. 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, as a conflict resolving technique.

When a file $file_k$ is created by a user transaction, $file_k$'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 so doing so we note that if the *read bit* of $file_k$ is 0, t_b will read it and if its *read bit* is 1, the file will not be read by t_b (even though $file_k$ is not actually in the backup). It can easily be seen that such a treatment ensures that the transaction which created $file_k$ is *mutually serializable* to t_b as per the definition given in Chapter 4. Moreover, the implementation ensures consistency when $file_k$ is accessed by a succeeding transaction.

We recall as pointed out in Section 4.6 of Chapter 4, that an implementation realizing the *mutual serializability* protocol must ensure that t_b does not become a victim of cascading roll backs as a result of a concurrent user transaction aborting. It has been shown in Theorem 4.6.1 that as long as t_b reads committed data and t_b is *mutually serializable* with each $t_x \in \mathbf{T}$, 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.

5.2.2 Implementation Details

We implement the online backup utility as a service provided by the file system and 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 of file system root*. The backup utility runs as a transaction and is thus wrapped between the transactional constructs of *txn_begin* and *txn_commit*. The backup utility algorithm is implemented by a module titled **Backup manager** which is described in Section 5.2.2.2. Methods used to enforce the *mutual serializability* protocol is built into the module titled **MS manager** and is described in the Section 5.2.2.3. Its integration into the TxnFS to facilitate the *Backup manager* to capture a consistent image is detailed in Section 5.2.2.4. But before going into the details of these modules, in Section 5.2.2.1 we describe the data structures added into the TxnFS design to help **MS manager** in enforcing *mutual serializability*.

5.2.2.1 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 structure. 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. The "read bit" mentioned in Section 5.2.1, required to indicate whether a file has been read or not by the backup process is practically implemented through the *sticky bit* present within an ext2fs inode data structure [20]. 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 otherwise. The backup utility thus "sets" the *sticky bit* after successfully reading a file to the backup medium. Thirdly, an integer field is added to each transactions housekeeping data structure in the *txn_list*. This field is referred to as the "before-after bit" as mentioned in Section 5.2.1 and a value of 0 indicates that the concerned transaction is ordered "before" and a value of 1 indicates that the concerned transaction is ordered "after" the backup transaction in the global serializability order. It is set to 0 when a transaction begins.

5.2.2.2 Backup Manager

The **Backup manager** is responsible for traversing the file system hierarchy visiting and copying each node on its way, starting at the node whose path has been passed to it during its invocation by the **TxnFS manager**. The implemented *mutual serializability* concurrency control protocol will work correctly to facilitate the **Backup manager** to capture a consistent file system copy, irrespective of the order in which the **Backup manager** traverses the file system hierarchy. Now, 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 disjoint subtree. Considering such access patterns, if the **Backup manager** 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, the probability of conflict between user and backup transaction increases to a large extent. Considering all these, currently we have implemented the **Backup manager** to traverse the file system hierarchy in a depth-first manner, thus facilitating access of highly probable related groups of files, temporally closer to each other. As each file is visited the module performs the following tasks in order,

1. Locks the file in exclusive mode (exclusive mode is required because the sticky bit on the inode has to be written into).

2. Reads and copies it to a backup file on a separate partition (can also be another device)

3. Marks the file as "read" by setting the *sticky bit*

4. Unlocks the file before traversing to the next node.

We now describe the algorithm followed by the **Backup manager** to backup each file in the file system rooted at *root_path* which is passed to the backup module as an argument. The **Backup manager** 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 and each string represents 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". After which "top" is again updated to point to the next element in the *stack*. The procedure is detailed in Algorithm 1:

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 an "after" transaction "moves" an ancestor directory to a new location in the hierarchy as explained by the following example: Let $d_6 \rightarrow d_1$, $d_1 \rightarrow d_2$, $d_2 \rightarrow d_3$ be the *parent* $\rightarrow$ *child* relationships in the file system hierarchy. t_b has read d_6,

d_1 and d_2. So, on the stack we have the file name $/d_6/d_1/d_2/d_3$. Now t_x moves d_2 to d_6. t_x is *mutually serializable* with t_b as d_6, d_1 and d_2 are all *read* by t_b. Hence t_x is allowed to go ahead by the *mutual serializability* protocol. But, the path name of d_3 now becomes $/d_6/d_2/d_3$ and the path name $/d_6/d_1/d_2/d_3$ stored on the *stack* becomes invalid.

If the **Backup manager** could store the inode of a file on the stack instead of the *file* name and use it to open the corresponding file, the above stated inconsistency would not have arisen. Ours being an user level implementation, opening a file by its inode is not possible. Hence, to prevent such inconsistencies, we used a dedicated thread which is woken up by the **Update manager** 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.

Now, from the time we implemented the TxnFS prototype and the *mutual serializability* protocol described in this chapter, new system calls have been released 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*() [24]. The *name_to_handle_at*() call, takes the name of the *file* and returns the associated *file handle* in the *handle* structure. The *handle* can then be passed as argument to *open_by_handle_at*() to open the corresponding file. Thus, the implementation of the **Backup manager** can be modified by requiring each element of the *stack* to now be pointers to *file handles*, so when a parent directory is read by t_b, each child files name is passed to *name_to_handle_at*() and the *handle* returned is stored in the *stack* and is later passed to *open_by_handle_at*() to open the file for reading.

5.2.2.3 MS Manager

The **MS manager** is responsible for serializing the backup transaction with application transactions by enforcing that each user transaction is *mutually serializable* with the concurrently executing backup transaction. We must remember here that the application transactions are serialized among themselves using the standard *strict 2PL* concurrency control protocol by the **Lock manager**. The primary interface to the **MS manager** is *IS_MS()* and its input arguments are *file*, a string representing the file name of the file for which a lock has been requested by the **TxnFS manager** on behalf of the user transaction and the unique *txnid* of the parent user transaction. In our current implementation, in the scenario of a compromise in the *mutually serializable* relationship the following conflict resolution schemes are used.

1. If the user transaction t_u is serialized "after" the backup transaction and it requests for

a lock on a file that has not yet been backed up, then the **MS manager** "pauses" t_u for a short random period "hoping" that in the mean while t_b backs up the concerned file.

2. If the user transaction t_u is serialized "before" the backup transaction and it requests for a lock on a file that has already been backed up then the **MS manager** "aborts" t_u to resolve the conflict.

The algorithm used by the **MS manager** to check and enforce a user transactions *mutually serializable* relation with the backup transaction is described in Algorithm 2. $txnid \rightarrow before\_after\_bit$ refers to the $before\_after\_bit$ of transaction $txnid$ and $file \rightarrow read\_bit$ 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 had to be *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".

Algorithm 2 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
```

In addition, if a user transaction t_u makes a *creat()* call to create a new file, the **MS manager** also sets the newly created file's "sticky bit" if t_u's *before-after bit* is 1 and leaves it unset otherwise. For this service it exposes the interface *set_on_creat*, using which the **Update manager** directs the **MS manager** to perform the said task at the time of commit.

5.2.2.4 Integrating the MS Manager into TxnFS

To enforce the *mutually serializable* relationship of each application transaction with the backup transaction, the interface *IS_MS()* provided by the **MS manager** is utilized by the **Lock manager**. The **Lock manager** on receiving each request to *lock* a file in either exclusive or shared mode from the **TxnFS manager** calls the IS_MS interface of the **MS manager**. If the **MS manager** successfully retains a *mutually serializable* relationship between the transaction requesting the lock and the backup transaction, the **Lock manager** continues with the process of locking the file in accordance with the *strict 2PL* scheme to serialize the application transaction with other user transactions. If the return value is "fail", the **Lock manager** "knows" that the transaction has been aborted and hence sends the message back to the **TxnFS manager** so that it can initiate the termination of the just aborted transactions executing thread. If the requesting transaction was "paused", the **Lock manager** receives as a return value "try again" and it calls the IS_MS again (t_b by now may or may not have read the file and MS manager checks again to confirm).

If a user transaction t_u makes a *creat()* call to create a new file, at the time of commit, after the **Update manager** has made a successful *creat()* call to the underlying file system, it directs the **MS manager** to set the newly created file's "sticky bit" if t_u's *before-after bit* is 1 and leave it unset otherwise. The **Update manager**sends as an argument the path of the newly created file to the **MS manager** through the function call *set_on_creat*.

5.2.2.5 Consistent Backup

In the current section we show that the implemented *mutual serializability* protocol ensures consistency in the backup copy and is hence correct. Now, the *mutual serializability* protocol attains overall serializability and hence backup consistency by establishing pair-wise *mutually serializable* relationships between the backup transaction and each concurrent user transaction. We show that our implemented *mutual serializability* protocol is correct by showing the implemented algorithm of establishing pairwise *mutually serializable* relationship is correct. In this regard, we consider a user transaction t_x to comprise of a single file system operation. If the implemented technique is correct for every file system operation, it will continue to be so with transactions comprising of a sequence of more then one file system operation.

Now, the conflicting operations that guide in establishing the said pair-wise *mutually serializable* relationship is in accordance with the proposed mapping of each file system operation to an equivalent sequence of read and write operations and the *creat_node* operation as given in Chapter 4. The mapping of each file system operation is incorporated into our implementation in the following manner: consider the file operation *unlink* of f_i from

directory d_{parent} which is mapped to $\{w_x(f_i), w_x(d_{parent})\}$. The implementation handles this mapping by locking in appropriate mode (shared lock for read and exclusive lock for write and creat_node) the files addressed in the mapped sequence (f_i and d_{parent} in exclusive mode in the current context) before the corresponding file operation is processed (*unlink* in this case). While establishing the *mutually serializable* relationship, the locks by user transactions conflict with t_b's lock operations issued before it reads the concerned files.

We demonstrate the correctness of our technique with a note on the following methods used in the implementation:

1. Checking and establishing the pair-wise *mutually serializable* relationship is done using a bit (termed *read bit*) associated with each file and another bit (termed *before after bit*) associated with each transactions data structure.

2. In a scenario where t_x is "after" t_b (*before-after bit* of t_x is 1) and is attempting to access an unread (*read bit* is 0) file, a compromise in the *mutually serializable* relationship is detected and is then prevented by "pausing" t_x.

3. In a scenario where t_x is "before" t_b (*before-after bit* of t_x is 0) and is attempting to access a read (*read bit* is 1) file, a compromise in the *mutually serializable* relationship is detected and is then prevented by aborting t_x.

4. t_b traverses the file system hierarchy using a *depth first traversal* technique. This leads to a node being always read before its descendants.

5. Upon successful reading of a file which is a directory, say d_j, t_b records in its internal data structure (the traversal stack) all references to d_j's children.

Let us now consider each file system operation with each being a transaction in itself and show that the **MS manager** ensures consistency in all possible cases. Though we say all possible cases we avoid similar cases for the sake of clarity of exposition.

Now, *read* and *write* accesses to files are assumed to be atomic in the formal framework and are also implemented as atomic operations by requiring each transaction (including the backup transaction) to *lock* a file including its inode. Hence, consistency of the backup copy when taken concurrently with an application transaction which consists of either a single *read* or a single *write* operation will hold. Thus, we do not explicitly consider file operations which accesses single files and are equivalent to a *read* or *write* access, while proving the correctness of implementation. These file operations include, *read*(), *write*(), *lseek*() and *stat*(). The remaining section considers the rest of the implemented operations.

creat(): Let t_x comprise of the single file operation $creat(f_i, d_j)$ which creates a regular file f_i under parent directory d_j. *creat*() has been mapped to $\{w_x(d_j), creat\_node_x(f_i)\}$. Thus,

Case 1 t_b has read d_j before t_x issues the *creat()* call. Hence, when t_b reads d_j, a reference to f_i has not yet been written into it and by 5 above we see that t_b does not traverse to f_i even after it is created. This ensures a consistent backup copy which does not include f_i.

Case 2 t_x locks d_j and issues the *creat* operation call. If t_b attempts to lock d_j before t_x commits, t_b waits as t_x releases locks only at commit/abort time (according to the implemented strict2PL). Say t_x commits (thus *creat* has been processed successfully) and t_b goes ahead to lock d_j. As f_i has been created, by 5 above t_b reads f_i too, leading to a consistent backup. Similarly say t_x aborts (thus *creat* has not been processed successfully and any modifications by t_x was undone) and t_b goes ahead to lock d_j. t_b reads d_j and it does not have any reference to the intended-to-be-created file f_i and the resulting backup is consistent.

unlink(): Let t_x comprise of the single file operation *unlink*() of a regular file f_i from directory d_j. The *unlink*() operation has been mapped to $\{w_x(d_j), w_x(file_k)\}$ and according to the mapping we list the possible interleaving cases with t_b and the **MS managers** handling of each case to ensure backup consistency.

Case 1 t_b reads both d_j and f_i before t_x issues the *unlink*() call. Hence, when t_x locks d_j its *before-after bit* is initialized to 1 and as t_x goes on to lock f_i the **MS manager** ascertains that t_x and t_b are still *mutually serializable* as the *read bit* of f_i is also 1. t_x proceeds to *unlink*. We see that the backup is consistent with the file system state before the *unlink*.

Case 2 d_j and f_i have not been read as yet by t_b. t_x locks both d_j and f_i, performs the *unlink* operation, commits and releases the locks. On locking the first file d_j, t_x's *before-after* bit is initialized to 0 and on locking the next file f_i, MS ascertains that *mutually serializable* relationship is not compromised as f_i's *read bit* is also 0. Now when t_b reads d_j, t_b does not see any reference to f_i and hence does not traverse to it. Now, even if f_i is still *linked* to some other parent the backup copy will not have any reference to it from d_j and this is a consistent result. And if the *link* from d_j was the only link to f_i then f_i will not appear in the backup copy and hence this is also a consistent result.

Case 3 t_b reads d_j (its *read bit is set to 1.* t_x then locks d_j for the purpose of *unlink* and hence its *before-after bit* is initialized to 1, t_b now reads f_i (its *read bit* set to 1), t_x then attempts to lock f_i, and the **MS manager** finds that the *mutually serializable* relationship is not compromised and allows t_x to lock f_i and the *unlink* is processed.

We see that when d_j is copied, the reference to f_i is copied and consistency prevails as f_i is also copied as expected.

Case 4 t_b reads d_j (its *read bit is set to 1*). t_x then locks d_j for the purpose of *unlink* and hence its *before-after bit* is initialized to 1. t_x now attempts locking f_i but its *read bit* is 0 and **MS manager** detects inconsistency. By 2 above, t_x is "paused" and t_b goes ahead and reads f_i. After a random period f time when t_x "resumes", f_i has been read, no inconsistencies are detected and t_x proceeds to lock f_i and process the *unlink* operation.

Case 5 Now, consider a case where another directory d_k has a hard link to f_i and t_b has read f_i by traversing this particular hard link. t_x now locks d_j for the *unlink* and as d_j has not yet been read by t_b, t_x's *before-after bit* is initialized to 0. t_x now attempts to lock f_i (*read bit* is 1) and the **MS manager** detects inconsistency. By 3 above, t_x is "aborted" without the *unlink* not taking place. t_b reads d_j with the reference to f_i in it and f_i is also in the backup, thus showing that consistency prevails.The case where d_j is read by t_b and but not f_i when t_x locks them is handled in a similar manner.

rename(): Let t_x comprise of the single file operation *rename*() which "moves" $file_k$ from d_{old} to d_{new}. The *rename/move* operation is mapped as $\{(w_x(d_{old}), w_x(file_k), w_x(d_{new})\}$. t_x and t_b may interleave and *mutually serializable* relationship established in the following possible cases. In each case backup copy consistency is ensured and the backup copy represents the file hierarchy before or after the *rename()*.

Case 1 t_b has read d_{old}, $file_k$ and d_{new} before t_x issues the *rename()* system call. t_x's *before-after* bit will be set to 1 on locking the first file say d_{old} and when it attempts to lock $file_k$ and then d_{new} both already read, no conflicts arise and *rename()* is processed. As t_x is "after" t_b, the backup copy will see the file system before the "move" operation and by 5 above the reading of $file_k$ ensures that its descendants, if it is a directory will certainly be read.

Case 2 t_b reads d_{old}, $file_k$ and d_{new} after t_x processes the *rename* call and commits leading to it releasing the locks on the concerned files. The backup copy will represent the file system after the "move" and all three files are clearly copied after they have been written to.

Case 3 t_b has read d_{old}, t_x locks d_{old} (*before-after bit* is set to 1) and then attempts to lock $file_k$ whose *read bit* is still 0. By 2 above, the **MS manager** "pauses" t_x, and t_b reads $file_k$. t_x is then "un-paused" and proceeds to locks $file_k$. If t_x now attempts to lock d_{new} before it is read by t_b, the **MS manager** "pauses" t_x for the second time.

We see that the **MS manager** ensures *mutual serializability* and in this case d_{old}, $file_k$ and d_{new} is read by t_b before *rename* is processed. By 5 above as $file_k$ is read, its descendants are also read. Even if the descendants are read after the "move" the backup copy will still have them as descendants of d_{old} as d_{old} and $file_k$ was copied before the "move".

Case 4 t_x locks d_{old} (*before-after bit* is set to 0), and then t_b reads d_{new} (its *read bit* set to 1). Then t_x locks $file_k$ (no conflicts is detected as $file_k$'s *read bit* is still 0), and t_x attempts to lock d_{new} and the **MS manager** detects a conflict as the *read bit* of d_{new} is 1 and the *before-after bit* of t_x is 0. By 3 above, t_x is aborted, t_b goes on to read d_{old} and d_{new}. The intended "move" did not take place and t_b read a consistent copy.

We note here that if t_x locks a file $file_k$, it can never be accessed by t_b or any user transaction till t_x commits or aborts because user transactions serialize using *Strict2PL* and locks are released only after commit (equivalent to making all modifications by t_x permanent by writing to disk) or abort (which means no modifications of t_x is written to disk). This ensures that t_b does not read dirty data and hence does not need to be backtracked.

Chapter 6

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. Considering that user transactions are already serialized through some standard concurrency control protocol(such as the *stict2PL* in our implementation), they further have to now serialize with respect to the backup transaction through an additional concurrency control protocol, the *mutual serializability* protocol.

In this chapter we shall evaluate the effect of obtaining an online consistent backup using the *mutual serializability* protocol on the performance of user transactions as well as the backup transactions. The metrics used for evaluation are the percentage of transactions conflicting with backup transaction, time taken for the backup to complete, and throughput of user transactions during the duration of the backup transaction. Experiments were run with and without enabling the *mutual serializability* protocol to evaluate the overhead of capturing a consistent backup copy over an inconsistent one. We shall further propose and evaluate techniques to reduce conflicts of user transactions with the backup transaction and thus explore methods to reduce cost incurred due to the *mutual serializability* protocol. To the best of our knowledge the online concurrency control method proposed in this thesis 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.

To evaluate the overhead incurred on deploying the backup specific concurrency control protocol, we simulated a number of transactional file system workloads and ran them on our implementation. The workload characteristics are described in details in Section 6.1. Section 6.2 describes the configuration of the system on which the prototype was implemented and finally evaluated, as well as configurations of the simulated environment. Various simulation results are presented and analyzed in Section 6.3, which also includes the implemented

method used to improve performance in Section 6.4. This chapter is finally concluded in Section 6.5 with a discussion on other possible ways of improving performance while obtaining a consistent backup.

6.1 Workload Models

We ran TxnFS on which the *mutual serializability* protocol is implemented through various types of transactional file system workloads to perform the performance studies. 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 is currently no available real transactional file system traces. Hence, we have evaluated our proposed protocol using synthetic transactional file system traces. Synthetically generated traces allow us to isolate individual behaviors and test a system on these separate behaviors. Moreover through synthetic traces we can model access behaviors 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 [43, 30, 71, 55]. The generated workloads are modeled such that transactions comprise of sequences of file system calls wrapped in transactional constructs of *txn_begin* and *txn_commit*/*txn_abort*. The file system calls made by each transaction in these experiments 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 entire file system hierarchy is evenly accessed by the generated workloads with the exception of the *hot-cold* workload. Such a behavior may not portray a realistic scenario but helps us in making a conservative evaluation of the system. This leads us to believe that the actual performance of the system will be much better than the evaluated results. To better represent a more realistic scenario, the *hot-cold* workload has been generated in which only a small portion of the entire file system is actively accessed. With these basic configurations, we generated nine sets of traces with characters as described below.

6.1.1 Uniformly Random Pattern(global)

In this workload set, the traces consist of transactions which accessed files randomly from the entire file system hierarchy. This access pattern does not exhibit any form of access locality. Each file in the hierarchy has an equal probability of being selected and the access

operation too is selected randomly with each operation having an equal likelihood of being selected. Such an access pattern is highly unrealistic, but we simulate our system with this workload for evaluation under the worst case scenario and to provide a performance lower bound . We refer to this workload as the **global** workload, indicating that transactions access across the file system hierarchy.

6.1.2 Spatial Locality Access Pattern(local)

File system access patterns are seen to exhibit a strong spatial locality. For example, whenever a file is accessed and modified, it is highly likely that its parent directory and inode are also accessed and modified. Moreover, Filesystem Hierarchy Standard's [57] provide guidelines for area specific placement of files in Unix like systems, thus directing logically related files to be located spatially close to each other in the file system hierarchy, and in all probability under a single subtree. To capture this behavior, the current suite of traces has been generated such that a transactions access pattern exhibits spatial locality. During trace generation, a transaction randomly selects a subtree and then performs operations mostly on files of this particular subtree. Within the selected locality, every file has an equal probability of being accessed. Within this category we generate workload sets that exhibit varying degrees of inter transactional sharing as detailed in Section 6.1.2.1 and access patterns exhibiting preference to certain file access operations as detailed in Section 6.1.2.2. The **hot-cold** trace too exhibits locality of reference and is detailed in Section 6.1.2.3. We note here that all sets of traces apart from the **global** exhibits access locality, and we collectively refer to all the trace sets that exhibit access locality as the **local** workloads.

6.1.2.1 Inter-Transactional Sharing

Studies [43] 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 direction 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. To generate these workload sets, the files in the file system are divided into two sets, the "to-be-shared" and "not-to-be-shared" sets. The number of files in

each set is in accordance with the degree of sharing of each workload set as already described. Files from the "to-be-shared" set can be accessed by any transaction and access of files from the "not-to-be-shared" set are mutually exclusive among concurrent transactions. During workload generation, each transaction randomly selects its locality of access and accesses any file from this locality with equal probability. The selected "locality" can either fall entirely within the "to-be-shared" set or entirely within the "not-to-be-shared" set or partially in both. Now, the event of concurrent access to files from the "to-be-shared" set is a matter of chance when the traces are replayed.

6.1.2.2 Access pattern with high percentage of *stat* calls(stat)

File access pattern studies have also reported user access to comprise a higher percentage of *read*(file attribute or data or both) access as compared to other operations. For example, *Roselli et al.* [55] has reported that most workloads show a much higher probability of *stat()* calls in comparison to the other file access operations, while *Leung et al.* [43] reports a higher read-to-write ratio. We study the affect on performance of such access patterns on our implemented protocol by modeling our next set of workloads to exhibit the just described access pattern. As higher degree of stat calls is equivalent to higher read access of files in the context of our evaluation, we do not separately model the higher read-to-write pattern. The evaluation results obtained by running experiments with the **stat** workload applies directly to the higher read-to-write access pattern.

We generate two sets of workload within this category, the **50%share-stat** and the **0%share-stat**(will collectively be referred to as the **stat** workloads). Both sets are generated to consist of transactions making *stat()* calls at most 70% of all its file system calls. The percentage value chosen is based on the findings by *Roselli et al.* [55]. The large number of stat() calls is attributed to calls for listing directories and also as a result of programs checking a files attributes before opening and accessing a file. These two sets of workload also model the locality of access with the **50%share-stat** trace set consisting of transactions sharing up to 50% of the files and the **0%share-stat** trace set consist of transactions not sharing any files among themselves.

6.1.2.3 Hot and Cold Pattern(hot-cold)

The entire file system hierarchy is not actively accessed at all times. Only about 10% is accessed 90% of the time and the rest 90% remains mostly "cold" by being accessed only 10% of the time [71]. To model this behavior the file system is divided into two groups, the "hot" group containing about 500 files and the "cold" group containing the rest of the files. The transactional workload was generated such that the "hot" set is accessed 90%

of the time and the "cold" set accessed only 10% of the time. This workload also models the locality of access with each transaction accessing files close to each other in the file system hierarchy whether it is accessing the "hot" or "cold" set. Within this category too 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). The **50%share-hot-cold** trace set consist of transactions sharing up to 50% of the files and the **0%share-hot-cold** set consist of transactions not sharing any files among themselves.

It is on the **hot-cold** suite of traces that we shall apply heuristics to improve performance of the deployed *mutual serializability* protocol as shall be detailed in Section 6.4.1.

6.2 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 exact 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 6.1 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.

6.3 Simulation Results and Performance Analysis

In order to understand the effects of capturing a consistent online backup using the *mutual serializability* protocol, on the performance of user transactions as well as the backup transaction, we tested the system under the different workloads described in Section6.1. We measured the overhead incurred on enabling *mutual serializability* by evaluating the percentage of transactions that conflict with the backup transaction(referred to as "con-

flict percentage") and time taken for the completion of the backup transaction(referred to as "backup time"). We performed these tests by running each set of workload and a concurrent backup transaction in two scenarios as illustrated below:

MS_enabled With the *mutual serializability* concurrency control protocol enabled to ensure an overall serializability and hence a consistent backup.

MS_disabled Without enabling *mutual serializability* protocol for ensure consistency of the backup. Hence with this experimental configuration, we obtain an inconsistent backup copy.

The simulation results of each workload set under both **MS_enabled** and **MS_disabled** on the metrics "backup time" were compared to evaluate the time overhead incurred to capture a consistent backup copy. Longer the duration needed to perform backup, more skewed the backup copy becomes. A longer backup "window" also increases the vulnerability of the system. On the other hand, the simulation results of the workloads in the **MS_enabled** class, on the metrics "conflict percentage" makes us aware of the degree of overhead incurred by the user transactions as a result of the *mutual serializability* protocol. We studied the cost incurred by user transactions from a different angle, by comparing the simulation results obtained under **MS_enabled** and **MS_disabled** with respect to the throughput of user transactions during the duration of the backup transaction. Higher the percentage of conflicts, lower is the throughput, as conflicting transactions are either aborted or "paused" to recover from a possible compromise in the consistency of the backup copy. A lower decrease in the throughput value during an **MS_enabled** run as compared to an **MS_disabled** run for any workload set is considered to be a positive result as this suggests that the user transactions are less affected by our *mutual serializability* protocol.

6.3.1 *Conflict Percentage* under different workloads

In the present context, a pair of transactions are said to *conflict* if they access a file using *conflicting operations* and the order in which the operations were issued in the corresponding test set was such that the serializability of the schedule is compromised. Detection of conflicts are followed by measures to resolve these conflicts and such measures lead to performance degradation. The current section presents and analyzes the simulation results in terms of the percentage of user transactions conflicting with the backup transaction under the different workloads.

See Figure 6.1 and its corresponding Table 6.1 for the percentage of transactions that conflict with the backup transaction in each of the different workloads. We provide only the results of the **MS_enabled** run as in the **MS_disabled** run, conflicts between the user

Workload \ Metrics	Increase in Percentage Conflict 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

Table 6.1: Change in Conflicts Between The MS_disabled and MS_enabled Run.

and backup transactions are neither detected nor prevented. So, the values in Table 6.1 as well as Figure 6.1 are effectively the % increase of conflicts in the **MS_enabled** run as compared to the **MS_disabled** run. 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 modeled 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.

In Figure 6.1 we also 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.

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. The **50%share-stat** and **0%share-stat** trace sets are modeled to illustrate equal degree of sharing with the **50%share** and **0%share** workloads respectively, but the **stat** workloads have a higher percent of *stat()* file system calls which are basically *read* accesses to a files attributes. Backup

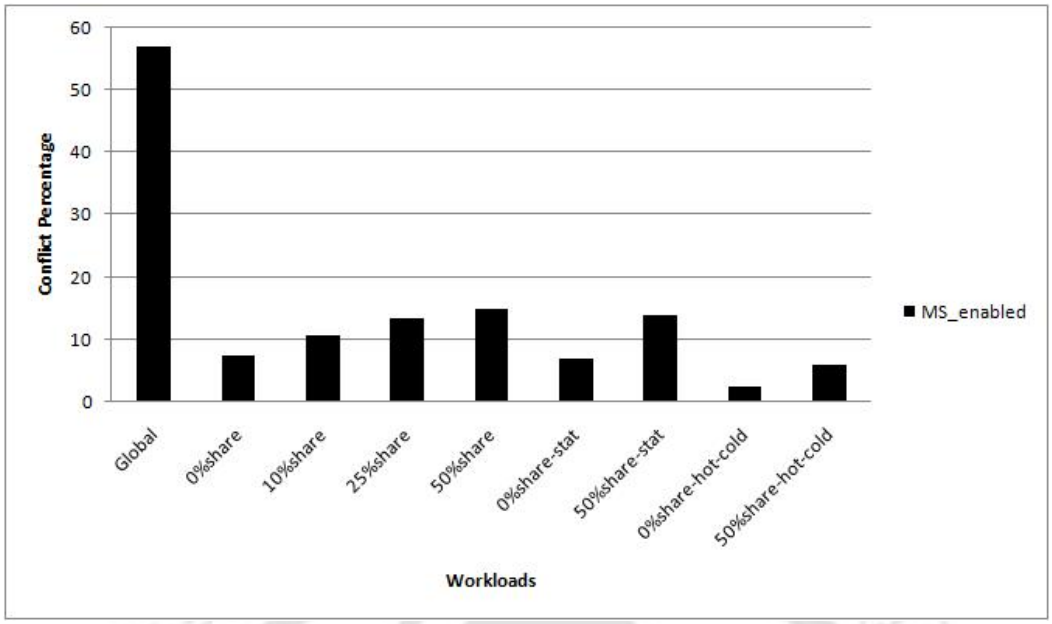


Figure 6.1: Percentage Of Transactions Conflicting In Each Set Of Workload

transaction serializes with the concurrently executing user transactions by establishing a *mutually serializable* relationship separately with each concurrent user transaction. 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** workload by issuing a much higher percentage of read accesses, the total percentage of conflicts in the corresponding two workloads is almost the same. It may be noted that if transactions could be identified as read-only then such transactions will not conflict with the backup transaction. We have not shown results with such transactions as we were more interested in assessing the worst case behaviors.

The simulation results of the **50%share-hot-cold** workload set reports that 6% of the user transactions conflict with the backup transaction. This is much lower then the 15% *conflict percentage* obtained from the **50%share** workload, which is similar to the **hot-cold** workload in terms of the inter transactional percentage of sharing. The two differ in the fact that **50%share** workload accesses the entire file system whereas the **50%share-hot-cold** workload accesses a very small subset of the entire file system. The much lesser *conflict percentage* in case of the **50%share-hot-cold**workload is because user transactions access only a small part of the file system thus decreasing the probability of 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.

6.3.2 *Backup Time*

The *backup time* benchmark measures the time in microseconds taken for the backup transaction to complete in the many experiments. *Backup time* measurements were taken for different workload sets in both the **MS_enabled** and **MS_disabled** scenarios. See Figure 6.2 and the corresponding Table 6.2 for the time in microsecond for a backup transaction to complete for each workload under **MS_enabled** and **MS_disabled**. The difference in the results obtained in the **MS_enabled** and **MS_disabled** scenario for each workload represents the overhead incurred in terms of time on obtaining a consistent backup copy. See Table 6.2 for the percentage increase in *backup time* while taking a consistent backup(under **MS_enabled**) as opposed to an inconsistent backup(under **MS_disabled**).

Workload	Backup time under MS_disabled (microsec)	Backup time under MS_enabled (microsec)	Percentage increase in Backup time with MS_enabled(%)
global	260.5	680	161
0%share	105	119.5	13.8
10%share	150.5	210	39.5
25%share	177	250	41.24
50%share	235.6	340.5	44.5
0%share-stat	90	101	12.2
50%share-stat	165	236	43
0%share-hot-cold	88	93	5.7
50%share-hot-cold	105	113	7.6

Table 6.2: Change in Backup Time Between The MS_disabled and MS_enabled Run.

The time taken by the backup transaction during an **MS_enabled** run of the **global** workload is considerably greater(161%) than its **MS_disabled** run as well as in comparison to the other workloads. This result is attributed to the large(above 50%) number of conflicts of user and backup transaction as is seen from Figure 6.1.

The percentage increase in the backup duration of the **MS_enabled** in comparison to the **MS_disabled** runs of the **50%share**, **25%share**,**10%share** and **0%share** increases though slightly as the degree of sharing among transactions increases. 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. 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. The difference in backup time between a **MS_disabled** and **MS_enabled** run decreases with the degree of sharing because of the corresponding decrease of conflict percentage.

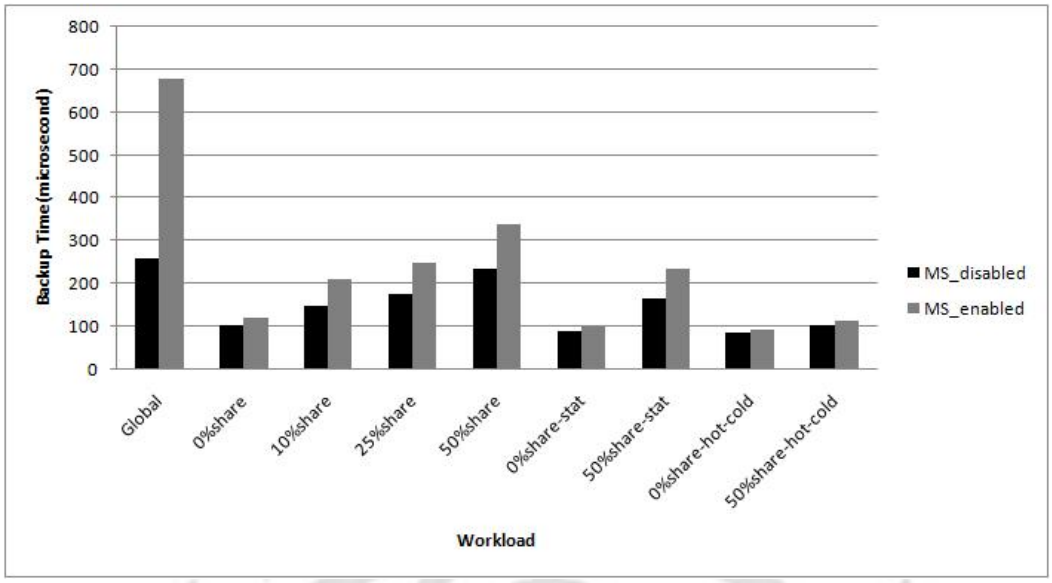


Figure 6.2: Duration Of Backup For Each Set Of Workload

Another interesting and encouraging result is seen from the simulation result of the **stat** workloads(both **50%share-stat** and **0%share-stat**). In the degree of sharing the **50%share-stat**(or **0%share-stat**) is equivalent to the **50%share**(or **0%share**) workload, but it has a much higher percentage of *stat()* file system calls and hence *read* accesses. As a result, inter user transactional conflicts 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**) workload. But, the difference in backup time of the **MS_enabled** and **MS_disabled** runs in both the **50%share**(or **0%share**) and the **50%share-stat**(or **0%share-stat**) workload is approximately same, mainly because of approximately equal number of conflicts between the user and backup transaction are seen in the two workloads as depicted in Figure 6.1.

The simulation results of the **50%share-hot-cold** workload on metrics *backup time* with **MS_enabled** is approximately 7% more than that reported from a **MS_disabled** run. The considerably low overhead incurred is attributed to the user transactions accessing the file system sparsely which results in lesser contention with the backup transaction. Contention arises only when backup is accessing the current "hot" regions of the file system hierarchy and "hot" areas are a mere 10% of the entire file system. The low overhead result is re-established by the **0%share-hot-cold** workload where we see a mere 5.7% increase in backup time to capture a consistent backup

6.3.3 Overall Throughput

By recording the number of user transactions committing during the duration of the backup transaction, we could also measure and compare the **MS_enabled** and **MS_disabled** simulation results for each workload on the *throughput* metric. The throughput value depicts the number of transactions completing to *commit* per microsecond during the duration the backup utility is active. Let us refer to Figure 6.3 and Table 6.3.

Metrics / Workload	Throughput under MS_disabled (No. of txns/microsec)	Throughput under MS_enabled (No. of txns/microsec)	Percentage decrease in Through-put with MS_enabled(%)
global	3	0.98	67.33
0%share	8.32	7.5	9.85
10%share	5.48	4	27
25%share	4.3	3.15	27.44
50%share	3.4	2.3	32.35
0%share-stat	9	7.9	12.2
50%share-stat	5	3.31	33.8
0%share-hot-cold	9.5	9.15	3.68
50%share-hot-cold	6.17	5.9	4.37

Table 6.3: Change in Throughput Between The MS_disabled and MS_enabled Run.

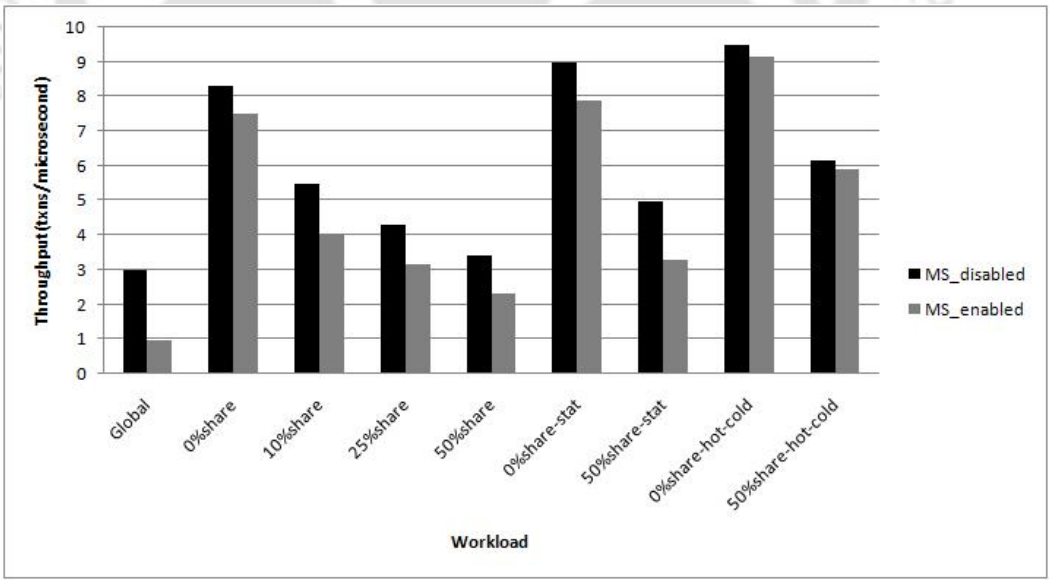


Figure 6.3: Throughput Of User Transaction(Measured for the duration the backup was active)

We see a decrease of approximately 67% in *throughput* for the **global** workload from the

MS_disabled to the **MS_enabled** run. This huge difference is due to more then 50% of the user transactions conflicting with the backup in the **global** workload as seen from Figure 6.1.

We also see from the results of the **50%share**, **25%share**, **10%share**, **0%share** that, as sharing increases a marked decrease of throughput is noted under both **MS_disabled** and **MS_enabled**. For example, throughput decreases from 7.5% in the **0%share** to 2.3% in the **50%share** workload under **MS_enabled**. The reason behind this decrease in throughput 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 analysis is reinstated by the decrease in throughput with increase in sharing in the corresponding runs under **MS_disabled**.

We also notice an increase in the overhead(difference between the **MS_disable** and **MS_enabled** run) of capturing a consistent backup with the increase of inter transactional sharing. For example, on comparing the results under **MS_disabled** and **MS_disabled** as seen in Table 6.3, we see that the overhead in terms of throughput increases from 9.85% in the **0%share** to 32.35% in the **50%share** workload. This increase in overhead can be attributed to the increase in the *conflict percentage* with increased inter-transactional file sharing.

The **50%share-stat** and **0%share-stat**workloads for both **MS_enabled** and **MS_disabled** scenarios show a higher throughput as compared to the **50%share** and **0%share** workloads respectively. Given that the percentage of conflicts with the backup transactions in both the workloads are almost the same as seen from Figure 6.1, again the reason behind a higher throughput for the **50%share-stat** and **0%share-stat** workloads is the lesser degree of conflicts among user transactions in these workloads as compared to the **50%share** and **0%share** workloads respectively. This may be better appreciated by studying the varying degree of throughput among the workloads under the **MS_disabled** scenario as depicted in Figure 6.3 and also Table 6.3. In the **MS_disabled** tests, *mutual serializability* has no effect and the varying degree of throughput is attributed to the degree of sharing within user transactions and access pattern of each transaction in terms of access locality and file system operations. With more sharing, contention increases and throughput decreases as seen by comparing the results of **50%share**, **25%share**, **10%share** and **0%share** workloads. Similarly, with a higher percentage of *read* access, contention among user transactions decrease and throughput increases.

Now, from the percentage decrease in throughput during an **MS_enabled** run with respect to the **MS_disabled** run as seem from Table 6.3, we see that the two **stat** workloads have a higher change, for example the **50%share-stat** shows a 33.38% decrease as opposed to a 32.35% seen from **50%share** workload. The higher overhead incurred by the **stat**

workloads is attributed to the fact that the **stat** and its equivalent **50%share** and **0%share** workloads show almost equal conflict percentages but the **stat** workloads report a lower value of backup time. Thus, slightly lesser transactions commit during the backup duration contributes to a higher decrease in throughput.

The most promising result is seen from both the **hot-cold** workloads where the throughput under **MS_enabled** is much higher than the **50%share**(increase by approximately 156% in the **50%share-hot-cold**) and **0%share**(increase by 22% in the **0%share-hot-cold**) workloads. As conflicts are rare in the **hot-cold** workloads, more transactions can commit successfully during the backup duration. From Table 6.3, 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 online backup attractive. This low overhead is as a result of low conflict percentage during the **MS_enabled** run

6.4 Performance Improvement

Consider the **hot-cold** workloads and its associated simulation results reported in Section 6.3.1, Section 6.3.2 and Section 6.3.3. 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 workload model we also know that these "active" regions are a very small chunk in the entire file system. Hence, the heuristics we apply to improve system performance during an **MS_enabled** run is 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". The logic behind choosing this performance improving heuristic is that, transactions exhibit per transaction locality of access hence the probability of its next access being spatially close is very high. Moreover, the files in that locality in the file system hierarchy probably belong to the current "hot" set of files, with currently active transactions accessing from this set. Hence, 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 again "hoping" the region is no longer 'hot". We describe the technique used to implement the performance enhancing heuristics in the following section.

6.4.1 Algorithm and Implementation

The backup transaction traverses the file system hierarchy starting from its mount point(root), in a depth-first manner and its implementation makes use of a stack data structure to keep track of the traversal path. The algorithm is detailed in Section 5.2.2. The original backup

traversal algorithm utilized a single stack with the reference to the *root* being the first to be *push*'ed into the stack. The enhanced backup traversal algorithm implemented to improve the system performance utilizes as many stacks as the number of child files contained in the root directory. After reading the root directory, the backup traversal algorithm *push*'es one child into each stack and then continues its traversal and reading by picking up a child subtree for traversal and utilizing the corresponding stack to aid in traversal of this subtree. Now, whenever a *mutual serializability* conflict is detected, the backup transaction is diverted to another part of the file system tree and this is practically implemented by the backup transaction now switching to another stack for its traversal aid. The point to which it had traversed in the subtree it was just diverted from is stored on "top" of the stack corresponding to that subtree and hence the backup transaction is never "lost". The stacks and hence the corresponding subtree each rooted at a child of the root are arranged in a circular list. The backup transaction completes reading a subtree in depth-first manner before moving to the next subtree if it is not diverted before completion. In such a case, the just traversed subtree's traversal stack is deleted from the circular list. On being diverted the backup transaction traverses the subtree whose stack is next in the circular list. Backup of the entire file system hierarchy ends when the circular list of traversal stacks become empty.

6.4.2 Simulation Result and Performance Analysis

We simulated the enhanced online backup prototype with the **50%share-hot-cold** workload as well as the **0%share-hot-cold**. To present the simulation results and study performance, for ease of presentation we shall refer to an **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).

See Figure 6.4 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 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-**

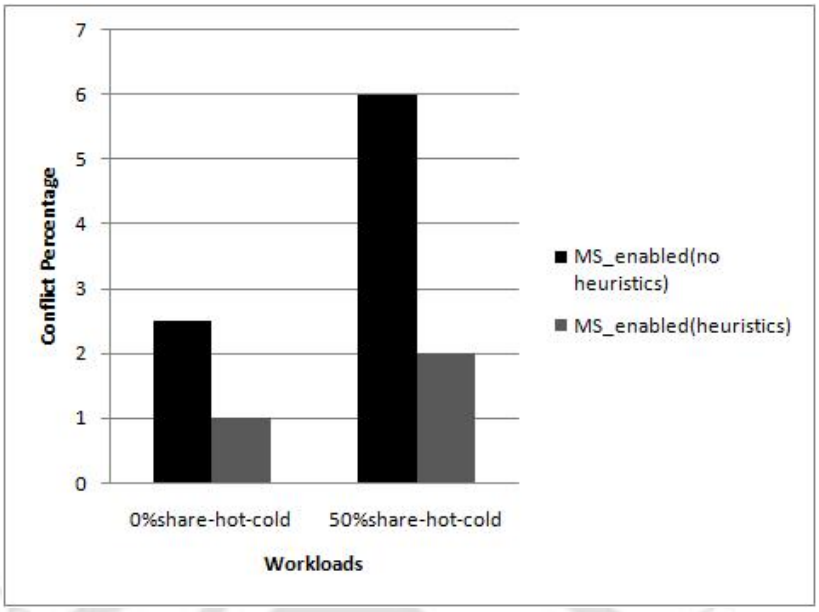


Figure 6.4: Conflict Percentage On Applying Heuristics To Improve Performance

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 diverting to another locality, is still a conflict and the corresponding user transaction has to be either aborted or "paused".

See Figure 6.5, 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.

Figure 6.6 compares the throughput of the user transactions during the backup duration

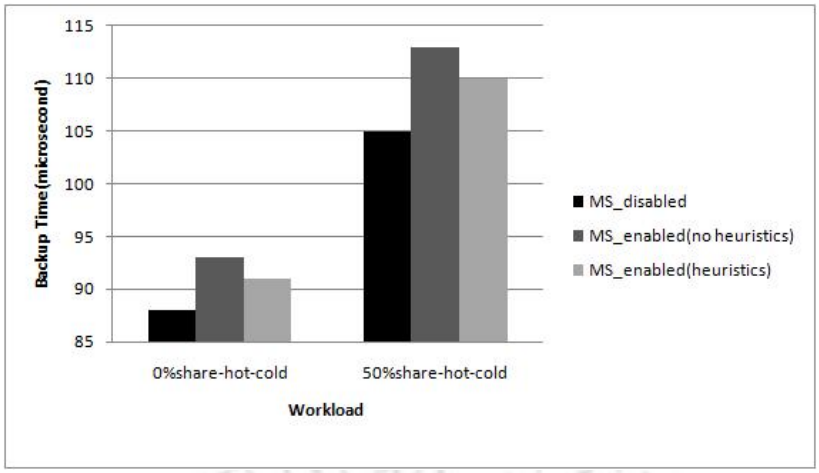


Figure 6.5: Backup Time On Applying Heuristics To Improve Performance

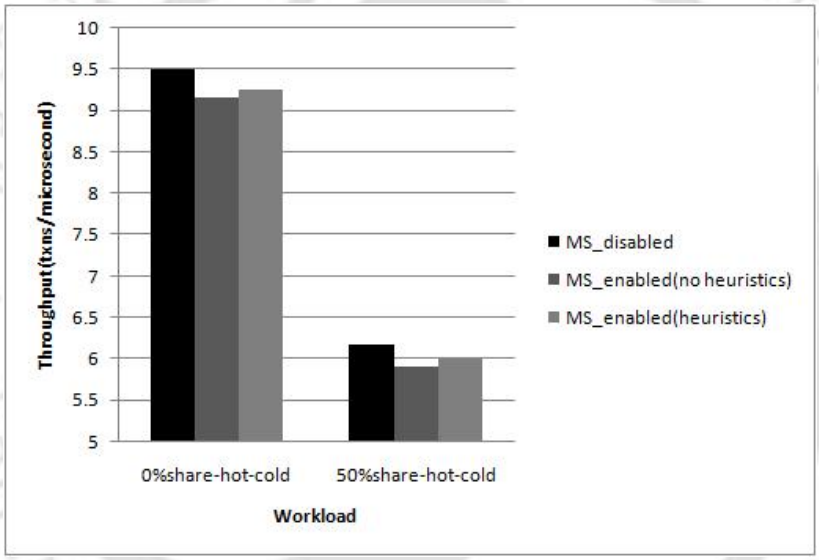


Figure 6.6: Throughput On Applying Heuristics To Improve Performance

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 decrease in conflicts, hence contributing to the increased throughput. 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**

6.5 Discussion

This chapter presented the performance evaluation of TxnFS when an online backup utility is serialized by *mutual serializability* concurrency control protocol with the concurrently executing user transactions. The metrics used to evaluate were the percentage of user transactions conflicting with backup transaction, duration of a backup transaction run and throughput of the user transactions during the duration of the backup transaction. The results obtained on simulation with the different workload sets showed that the overhead incurred in reading a consistent backup is minimum for workloads which rarely shared files concurrently and overhead increases as sharing increases. Significantly lesser overhead is incurred by workloads that model locality of access as opposed to workloads accessing randomly from any where in the file system hierarchy. On the other hand we also saw that an increased *read* access as compared to *writes* as seen in most real workloads, does not change the performance of the system and we report simulations results are equivalent to another workload in which *read* and *write* accesses are made with equal probability. We further see that very less overhead is incurred by the **hot-cold** workload set. This improves further when we apply heuristics that involve diverting the backup transaction to currently "cold" regions of the file system to improve performance.

Overall performance during an online backup using our approach can be considerably improved if conflicts between user transactions and the backup transaction can be kept low. In the current study we have implemented and tested only one performance enhancement method. But, 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. 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 in Section 6.4.1. Another method to improve performance is to backup up 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 by *Leung et al.* [43]. 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 signifcant in number, and so the results obtained point to even better performance in such environments.

Chapter 7

Conclusion

Data Backup is extremely important for the protection of valuable 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. This thesis has considered the issue on 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. 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 in most cases resolves conflicts. We termed this concurrency control protocol as *mutual serializability*.

To implement and evaluate the *mutual serializability* protocol we 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. API of file access operations remain unaltered, just requiring the applications to encapsulate sequences of operations logically belonging together in a txn_begin and txn_commit/txn_abort pair. TxnFS ensures atomicity and durability through its *deferred write* logging mechanism. Future work must implement checkpointing into the log mechanism to reduce the vulnerability window for long transactions. Isolation and consistency is achieved through the strict 2phase locking protocol. On this transactional file system platform, the backup transaction was implemented and it establishes a *mutually*

serializable relationship with every application transaction through a bit(read-bit) in each files metadata to indicate its backup status and another bit(before-after bit) embedded in each transactions metadata to indicate its serialization order with respect to the backup. Compromise in *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 backup transaction to go ahead till *mutual serializability* can be established.

We evaluated the overhead incurred by the *mutual serializability* protocol in terms of conflicts between user and backup transaction , backup transaction runtime and throughput of the user transactions. Evaluation was performed by comparing the results obtained with and without enforcing *mutual serializability* using a number of synthetically generated transactional file system workloads, modelled as closely as possible on realistic file access patterns together with injecting patterns likely to occur in the future, such as increased inter transactional sharing.

On evaluation we saw workloads modeling locality of access, incurred considerably lesser overhead then workloads accessing randomly from the entire file system hierarchy. On the evaluation metrics of *conflict percentage*, the **global** workload comprising of transactions accessing from across the file system hierarchy, has an overhead of 66% over the workload exhibiting locality along-with 0% file sharing. For workloads exhibiting no file sharing among concurrent transactions, the overhead incurred due to the *mutual serializability* protocol was low and increases as the degree of inter-transactional sharing increased with the value here ranging from 7.5% to 15% transactions conflicting respectively. What was interesting to note was that the performance does not differ much for workloads having a higher percentage of read access as compared to workloads having approximately equal read and write access.

On evaluating the cost in terms of backup time for capturing a consistent backup as opposed to an inconsistent backup, we saw that on the 0% sharing workload exhibiting locality, the overheads of a consistent backup was just 13.8% over an inconsistent backup. Similarly, there was an equal percentage of decrease in throughput on capturing a consistent backup as opposed to an inconsistent backup. We believe that it is indeed a small price to pay for the sake of correctness in the backup copy.

The most promising results were seen on simulation with workloads modeling transactions actively accessing only about 10% of the total files in the files system while rest of the files remain relatively un-accessed. The transactions in one such workload set exhibits locality of access and models 0% inter-transactional file sharing. With this workload we saw that the guarantee of a consistent backup comes with just 5.7% overhead on backup time over an inconsistent backup, with only 2.5% of the transactions conflicting with the backup transaction. We then applied heuristic performance enhancement techniques on this work-

load, which involved diverting the backup transactions to "cold" regions to avoid conflicts with concurrent transactions accessing the "hot" areas. We noticed that number of conflicts dropped significantly by about 60% from the simulation results of the same workload but without the application of heuristics. A similar decrease in backup time and increase in throughput was recorded on applying heuristics.

7.1 Future Work

In this section, we explore possible improvement of the performance of our *mutual serializability* consistent online file system backup protocol and interesting avenues for further research in the area of online file system backup of the file system

7.1.1 Performance improvement

The protocol in its present state can benefit tremendously from heuristically driven performance enhancement techniques. One such technique has already been tested in section 6.4 and a few others listed in section 6.5. We plan to further explore performance improvement through parallelizing the backup transaction itself. Existing file system studies have shown that files are rarely shared. Now, from a file system access history if we identify sets of files in a file system that are always accessed in a mutually exclusively manner, backup of two or more such sets can be taken in parallel. The implemented *mutual serializability* protocol remains as it is and user transactions serialize themselves with the backup transaction, just that the backup transaction is now a multi-threaded implementation with each thread backing up a mutually exclusive set of files.

7.1.2 Mutual Serializability vis-a-vis Non Transactional File Systems

We have proposed and shown the correctness of the *mutual serializability* protocol, theoretically as well as practically by assuming a transactional file system. But, existing file systems have no support for transactions. However, approaches such as *journaling* and *soft updates* have been introduced in existing file systems to support the atomic execution of system calls and to protect on-disk file system integrity after a crash or power outage. In order to implement our backup strategy on existing file systems with no notion of transactions, such per system call consistency preserving approaches can be utilized to capture a backup, preserving consistency at the level of single system calls. The goal can be achieved by considering user transactions to be system calls and the *mutual serializability* algorithm can then be applied between. But, to implement our algorithm we need to have the ability to undo, or abort a system call that affects more than one file (for example deleting a file, where

both the parent directory and the deleted file are written into in the system call) whenever required to resolve *mutual serializability* conflicts. Adding this facility to a journaling file system such as the ext4 file system will be fairly easy, requiring minor changes to the ext4 implementation which facilitates a "transaction" in the journal to be metadata and data blocks of a single system call.

The scheme could be further extended to automatically considering the opening and closing of files as the start and end of a transaction. Doing this will ensure that the backup transaction does not copy files that have been opened for writing. If it is further assumed that sharing of files is handled at the application level by using the file locking system calls (flock, lockf), then our scheme will provide a consistent online backup in even existing file systems. More work is required to formally show this and to implement this scheme in an existing file system such as ext4. This scheme is likely to be superior to snapshot based schemes where file fragmentation is a major cause of concern.

7.1.3 Improving TxnFS

The TxnFS prototype implementation needs to be strengthened to improve performance. Durability needs to be incorporated into the implementation. Further performance improvements can be made by moving the implementation into the kernel. With the use of flash memory as a logging device, efficient transactional file systems can be built and Txnfs can be the basis for such a system.

Bibliography

[1] cpio. http://www.gnu.org/software/cpio/manual/html_mono/cpio.html. Accessed on 9/04/2012.

[2] dd. http://pubs.opengroup.org/onlinepubs/9699919799/utilities/dd.html. Accessed on 9/04/2012.

[3] dump. http://dump.sourceforge.net/. Accessed on 9/04/2012.

[4] Encrypting Critical Backup Data. http://www.symantec.com/business/resources/articles/article.jsp?aid=encrypting_critical_backup_data Accessed on 30/09/2011.

[5] 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.

[6] The Health Insurance Portability and Accountability Act (HIPAA) of 1996. http://en.wikipedia.org/wiki/Health_Insurance_Portability_and_Accountability_Act. Accessed on 09/04/2012.

[7] PostgreSQL 8.1.23 Documentation. The PostgreSQL Global Development Group.http://www.postgresql.org/docs/8.1/static/. Accessed on 09/04/2012.

[8] 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. Accessed on 09/04/2012.

[9] tar. http://www.gnu.org/software/tar/. Accessed on 9/04/2012.

[10] VERITAS File System(TM) System Administrator's Guide. Release 3.2 SCO UnixWare. http://uw714doc.sco.com/en/ODM_FSadmin/fssag-1.html. Accessed on 9/04/2012.

[11] Quantifying Performance Loss: IT Performance Engineering and Measurement Strategies, October 2000. `http://www.metagroup.com/cgi-bin/inetcgi/jsp/displayArticle.do?oid=18750`. Accessed on 12/12/2008.

[12] Sleepycat Software, Inc. Berkeley DB Reference Guide, December 2004. 4.3.27 edition `www.oracle.com/technology/documentation/berkeley-db/db/api_c/frame.html`. Accessed on 09/04/2012.

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

[14] HP Data Protector A.06.1x Zero Downtime (Split-Mirror) Backup Support Matrix for EMC Arrays Version: 3.2, October 2011. `http://bizsupport.austin.hp.com/bc/docs/support/SupportManual/c01631151/c01631151.pdf`. Accessed on 09/04/2012.

[15] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A Five-Year Study of File-System Metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies.*, 2007.

[16] P. Ammann, S. Jajodia, and P. Mavuluri. On-The-Fly Reading of Entire Databases. In *IEEE Transactions on Knowledge and Data Engineering*, volume 7, pages 834–838, 1995.

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

[18] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[19] S. Bhalla and S. E. Madnick. Parallel On-The-Fly Reading of an Entire Database Copy. In *Practical Parallel Computing*, pages 149–162. Nova Science Publishers, Inc., 2001.

[20] D. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002.

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

[22] A.L. Chervanak, V. Vellanki, and Z. Kurmas. Protecting the File System: A Survey of Backup Techniques. In *Proceeding of the Joint NASA and IEEE Mass Storage Conference*, pages 17–32, March 1998.

[23] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The Episode File System. In *Proceedings of the USENIX Annual Technical Conference*, pages 43–60, 1992.

[24] J. Corbet. Open by handle, February 2012. `http://lwn.net/Articles/375888/`. Accessed on 09/04/2012.

[25] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: a User-Level Versioning File System for Linux. In *Proceedings of the USENIX Annual Technical Conference*, pages 27–27, 2004.

[26] O. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 285–298, 2002.

[27] J. Damoulakis. Continuous Protection. In *Storage Technology Magazine*, volume 3, pages 33–39, June 2004.

[28] L. Deka and G. Barua. On-line Consistent Backup in Transactional File Systems. In *Proceedings of the first ACM Asia-Pacific Workshop on Systems*, pages 37–42, August 2010.

[29] M. D. Flouris and A. Bilas. Clotho: Transparent Data Versioning at the Block I/O Level. In *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies*, pages 315–328, 2004.

[30] T. Gibson, E. L. Miller, and D. D. E. Long. Long-Term File Activity and Inter-Reference Patterns. In *Proceedings of the 24th International Computer Measurement Group Conference*, pages 976–987, 1998.

[31] D. K. Gifford, P. Jouvelotl, M. A. Sheldon, and J. W. Otoole. The Cedar File System. In *Communications of the ACM*, pages 288–298, March 1988.

[32] J. Gray. Notes on Data Base Operating Systems. In *Lecture Notes in Computer Science*, volume 60, pages 393–481, 1978.

[33] J. Gray. Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King. In *Storage Guru Gong Show*, December 2006.

[34] R. J. Green, A. C. Baird, and J. C. Davies. Designing a Fast, On-line Backup System for a Log-Structured File System. In *Digital Technical Journal of Digital Equipment Corporation*, pages 32–45, October 1996.

[35] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–246, 1994.

[36] 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. In *US Patent No. 5,819,292*, 1998.

[37] J. H. Howard, M. L. Kazar, S. G. Menees, A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and Performance in a Distributed File System. In *ACM Transactions on Computer Systems*, volume 6, pages 51–81, 1988.

[38] J. S. Howard. Online Backups Using the VxVM Snapshot Facility. In *Enterprise Engineering Sun BluePrints OnLine*, September 2000. `http://www.filibeto.org/sun/lib/blueprints/2002-vol4-cd/0900/vxvmfac.pdf`.

[39] N. C. Hutchinson, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. Physical File System Backup. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 239–249, 1999.

[40] J.E. Johnson and W.A. Laing. Overview of the Spiralog File System. In *Digital Technical Journal*, volume 8, pages 5–14, 1996.

[41] G. Laden, P. Ta-shma, E. Yaffe, M. Factor, and S. Fienblit. Architectures for Controller Based CDP. In *Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 107–121, 2007.

[42] K. Lam and C. S. Lee Victor. On Consistent Reading of Entire Databases. In *IEEE Transaction on Knowledge and Data Engineering*, volume 18, pages 569–572, April 2006.

[43] A. W. Leung, S.Pasupathy, G. Goodson, and E. L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proceedings of the USENIX Technical Conference*, pages 213–226, June 2008.

[44] A. J. Lewis. LVM HOWTO. `http://tldp.org/HOWTO/LVM-HOWTO/`. Accessed on 09/04/2012.

[45] B. Liskov and R. Rodrigues. Transactional File Systems Can Be Fast. In *Proceedings of the 11th Workshop on ACM SIGOPS European Workshop*, Septemper 2004.

[46] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier. The New EXT4 Filesystem: Current Status and Future Plans. In *Proceedings of the Ottowa Linux Symposium*, pages 21–34, 2007.

[47] M. K. Mckusick and G. R. Ganger. Soft updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *USENIX Annual Technical Conference, FREENIX Track*, pages 1–17, June 1999.

[48] M.G.Oxley. (H.R.3763) Sarbanes-Oxley Act of 2002, February 2002. `http://fl1.findlaw.com/news.findlaw.com/cnn/docs/gwbush/sarbanesoxley072302.pdf`. Accessed on 09/04/2012.

[49] R. H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. Snapmirror: File-System-Based Asynchronous Mirroring for Disaster Recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, 2002.

[50] Z. Peterson and R. Burns. Ext3cow: a Time-Shifting File System for Regulatory Compliance. In *ACM Transactions on Storage*, volume 1, pages 190–212, May 2005.

[51] D.E. Porter, O.S. Hofmann, C.J. Rossbach, A. Benn, and E.Witchel. Operating Systems Transactions. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 161–176, October 2009.

[52] W. Curtis Preston. *Backup and Recovery - Inexpensive Backup Solutions for Open Systems: Covers Windows, Linux, Unix, and OS X*. O'Reilly, 2007.

[53] C. Pu. On-The-Fly, Incremental, Consistent Reading of Entire Databases. In *Proceedings of the 11th International Conference on Very Large Data Bases*, volume 11, pages 369–375, 1985.

[54] C. Pu, C. H. Hong, and J. M. Wha. Performance Evaluation of Global Reading of Entire Databases. In *Proceedings of the First International Symposium on Databases in Parallel and Distributed Systems*, pages 167–176, 1988.

[55] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–54, 2000.

[56] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *ACM Transactions on Computer Systems*, volume 10, pages 26–52, 1992.

[57] R. Russell, D. Quinlan, and C. Yeoh. Filesystem Hierarchy Standard, 2004. Technical Report, File system Hierarchy Standard Group, `http://www.pathname.com/fhs/`.

[58] D. J. Santry, M. J. Feeley, and N. C. Hutchinson. Elephant: The File System that Never Forgets. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 2–7, 1999.

[59] F. Schmuck and J. Wylie. Experience with Transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 239–253, October 1991.

[60] R. C. Sears. *Stasis: Flexible Transactional Storage*. PhD thesis, EECS Department, University of California, Berkeley, Jan 2010.

[61] M. Seltzer and M. Olson. LIBTP: Portable, Modular Transactions for UNIX. In *Proceedings of the 1992 Winter USENIX*, pages 9–25, 1992.

[62] M. I. Seltzer. Transaction Support in a Log-Structured File System. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 503–510, 1993.

[63] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Technical Conference*, pages 307–326, 1993.

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

[65] 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. In *Symantec Yellow Books*, 2006. `http://www.symantec.com/en/uk/theme.jsp?themeid=yellowbooks`. Accessed on 09/04/2012.

[66] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata Efficiency in a Comprehensive Versioning File System. In *Proceedings of USENIX Conference on File and Storage Technologies*, pages 43–58, 2002.

[67] R. P. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling Transactional File Access via Lightweight Kernel Extensions. In *Proceedings of the 7th Conference on File and Storage Technologies*, pages 29–42, 2009.

[68] A. Tridgell. Efficient Algorithms for Sorting and Synchronization. In *PhD Thesis, The Australian National University*, 1999.

[69] S. Verma, T. J. Miller, and R. G. Atkinson. Transactional File System. In *US Patent No.6,856,993*, 2005. `http://www.google.com/patents/US20050149525`. Accessed on 09/04/2012.

[70] K. Vidyasankar. Serializable Graphs. In *Proceedings of the 14th International Workshop Graph-Theoretic Concepts in Computer Science*, Lecture Notes in Computer Science, pages 107–121, 1989.

[71] J. Wang and Y. Hu. 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*, pages 47–60, 2002.

[72] C. P. Wright. *Extending ACID Semantics to the File System via ptrace*. PhD thesis, Computer Science Department, Stony Brook University, May 2006. Technical Report FSL-06-04.

[73] W. Xiao, J. Ren, and Q. Yang. A Case for Continuous Data Protection at Block Level in Disk Array Storages. In *IEEE Transaction in Parallel Distributed Systems*, volume 20, pages 898–911, 2009.

[74] L. You, K. Pollack, and D. E. Long. Deep Store: An Archival Storage System Architecture. In *Proceedings of the 21st International Conference on Data Engineering*, pages 804–815, April 2005.

[75] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: a ZFS Case Study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, pages 29–42, 2010.

[76] E. D. Zwicky. Torture-Testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not. In *Proceedings of the Fifth Large Installation System Administrators Conference, USENIX*, 1991.

[77] E. D. Zwicky. Further Torture: More Testing of Backup and Archive Programs. In *Proceedings of The 17th Annual Large Installation Systems Administration Conference*, 2003.