Recovery Protocols For Flash File Systems

Ravi Tandon and Gautam Barua

Indian Institute of Technology Guwahati,
Department of Computer Science and Engineering,
Guwahati - 781039, Assam, India
{r.tandon}@alumni.iitg.ernet.in
{gb}@iitg.ernet.in

Abstract. Supporting transactions within file systems entails very different issues than those in Databases, wherein the size of writes per transaction are smaller. Traditional file systems use a scheme similar to database management systems for supporting transactions resulting in suboptimal performance. Ext[6] based file systems either involve duplication of blocks, resulting in a reduced write throughput or provide only metadata consistency. The performance provided by a Log-structured file system on traditional hard disk drives is poor due to non-sequential reads that require movement of the read head.
This work presents an implementation of transaction support for log-structured file systems on flash drives. The technique makes use of the copy-on-write capabilities of the hitherto existing log- structured file systems. The major improvement is in the reduction in the overall write-backs to the disk. We provide protocols for recovery from transaction aborts and file system crash. The transaction support and recovery has been implemented in a flash file system[1].

Keywords: File Systems, Recovery, Transactions

1 Introduction

A *transaction* can be defined as a unit of work that is executed in an *atomic, consistent, isolated and durable (ACID)* manner. With the advent of larger, more reliable and highly accessible systems, ensuring consistency of system data has become a necessity for modern user applications. DBMS (Database Management Systems) provide ACID properties to user applications through logging. However, databases do not provide uniform interfaces for supporting transactions. Support for transactions at the File System layer can provide a generic solution that can be used by any user level application.

Transaction support in file systems is provided through various logging techniques. Atomicity and durability are ensured by write-ahead logging, wherein updates are first made to the log. Each transaction has a start and a stop. Undo and redo based logging techniques are used to recover from a crash. Redo based recovery makes use of delayed writes, or the concept of aggregation of updates as in Ext3 [6]. Isolation and consistency are ensured by having concurrency control

mechanisms used for ensuring serializability of schedules. Strict 2 phase locking scheme is widely used for ensuring consistency.

Logging schemes such as those used in Ext3 have two copies of data (one copy in the journal and the other copy in the actual file). This leads to a decrease in the bandwidth usage of the storage disk. Metadata logging schemes reduce the writes to the log, but fail to ensure complete transaction semantics can only be used to ensure consistency of a file system after a crash. Atomicity for instance might not be guaranteed by such a scheme. Other file systems such as the Log-structured file system have improved the write speed to the disk, by having sequential writes. Reads, however, are affected due to non-sequential storage of a files data. With the advent of new technology, such as Flash Devices, non-sequential write and read speeds have improved considerably. Also, the inherent *copy-on-write* nature of flash devices suits a log-structured file system well. Such a file system extends naturally to a transactional file system. Log-structured file systems provide easy crash recovery.

Our work focuses mainly on the recovery aspects in a log-structured file system designed for flash based storage devices. Our file system is based on a client-server architecture. The client runs the user application program and the server runs the file system operations. We have designed protocols for recovery from crashes on the server side as well as transaction aborts on the client side. We also evaluate our design and provide a comparative study with a file system that uses a separate journal.

The rest of the paper is organized as follows: Section 2 throws light upon some of the existing works that have helped us understand and propose a design for the transactional file system for Log-structured file systems (LFS). Section 3 provides an overview of the underlying architecture on which transaction support for flash file system works. Section 4 describes the protocols used for recovery from client and server side aborts. Section 5 contains the experiments and the analysis of the results. Section 6 summarizes the work and proposes ideas for future work.

2 Related Work

Transaction support to user applications can be provided within user space or kernel space. The authors in [5] consider trade-offs between implementing transaction support within user space, on top of an exiting file system, or in kernel space as part of a file sysem for a read or a write optimized file system. The dissertation shows that a careful implementation of a transaction support within the kernel in a write optimized file system can provide better performance than all other possible set of implementations, both for a CPU or I/O bound system. We, therefore, have implemented transaction support on a write-optimized file system within the file system. But our implementation of the file system is in user space due to time costraints. Other implementations such as Amino[8] have provided transaction support within user space. Implementations such as Sprite

LFS [3], Ext3 [6], Transactional Flash[4] provide support for transactions in a generic manner.

Sprite LFS [3] is a prototype implementation of a log-structured file system. A log-structured file system writes all modifications to disk in a sequential manner similar to a log, thereby speeding up both write and recovery time. The log is the only structure on the disk, which contains the indexing information that is used to read files from the disk. The primary idea behind the design of Sprite LFS was to buffer writes in a memory cache and then to flush the writes to the disk in a sequential manner. This saves the time the read/write head spends in seeking to the accurate disk location for each random read/write. Sprite LFS essentially uses copy-on-write and therefore does not write data/metadata blocks in place. Traditional storage devices offer poor read performance for non-sequential reads. Solid state drives and flash file systems provide higher read performance and mitigate the effect of random reads.

Ext3 file system [6] provides faster recovery from crashes using metadata journaling on disk. A journal is a circular log stored in a dedicated area of the file system. Whenever updates are made to the disk, the corresponding metadata blocks are written to the journal before being flushed to the disk. The data blocks get updated to the disk first, then the metadata blocks are written to the journal on disk. After the metadata blocks have been updated to the journal, the transaction is assumed to be committed. However, the journal cannot be erased unless and until the metadata blocks are synced back to the disk. Data blocks can also be written to the log, and if this is done, then a transactional file system can be implemented. However, the journal is mainly used only for metadata logging to provide easy recovery from crashes, as a transaction file system has not been implemented in Ext3 and there seems little point in incurring the extra overhead of data block writes to the log in the absence of such an intereface.

Transactional Flash[4] is a write atomic interface implementation on a Solid State Device. It uses a cyclic commit protocol for ensuring atomicity of writes. The main aim of the commit protocol was to do away with the commit records that have to be written to the disk on each transaction commit operation. The authors of transactional flash have come up with a novel cyclic commit protocol, wherein they use page level tagging. The drawback of such a scheme is that each aborted transaction's writes to the disk must be erased from the disk, before a new version of the same page can be written to the disk. This induces a heavy erase penalty on the disk, as erase level granularity on flash disks is a block. To reduce the erase dependency, the authors develop another commit protocol BPCC. Each page points to the last committed version of the page. It, however, imposes certain restrictions on erase order, i.e. any page can be erased only after the pages that it marks aborted are erased. Our recovery protocol uses an idea that is similar to marking pages with some information (transaction identifier), and it does not impose any restrictions on the erase order.

3 Architecture

Our file system design uses a hybrid transaction management approach. Support for locking that provides isolation and consistency, has been implemented in user space[2]. The logging subsystem, that ensures atomicity and durability, has been implemented within the file system, currently also implemented in user space, although it can be easily moved to the kernel.

A client-server architecture is built, wherein the client side runs the user application program and the server runs the file system. The calls at the user level are traced using *LD_PRELOAD* runtime linker [7]. The system calls after being intercepted are sent to the server side using *Remote Procedure Calls*. The server handles the system call by creating a separate handler thread for each transaction. This thread acts as a dispatcher for all the system call requests on this connection.

3.1 Top Level Architecture

The basic architecture was implemented in [2]. In the present architecture each client is a separate process. Each transaction begins with a txn_beg() and ends with a txn_end() or txn_commit(). The transaction specific calls are implemented as a shared library. In this work, we have replaced the Ext3 file system from the server side and we have used a log-structured file system (YAFFS Yet Another Flash File System [1]). Logging and crash management are now handled by the YAFFS.

The transaction reader (TxReader) listens to system calls from the user process (see Fig. 1). The transaction file system manager (TxFS Manager) interprets the system call. If the system call requests opening of a file, then the transaction file system manager sends a lock request to the Lock Manager. If the lock request is granted by the lock manager then the transaction file system manager sends a file open request to the translation layer. The translation layer forwards the request to the YAFFS. The YAFFS library handles the open (or close) request. On a read or a write system call, the transaction file system manager requests the update manager to handle them. The update manager checks whether the read or write system call is possible on the file (it checks whether the file has already been locked or not). Once the check is done, and the read or the write is allowed, then the update manager forwards the request to the translation layer (this is the part that has been implemented in this work). The translation layer converts the system calls for the Ext3 file system to those for YAFFS. These calls are then handled by the YAFFS system call library.

4 Recovery Protocols

Recovery protocols restore a file system to a consistent state by reconstructing the metadata and data of all the objects (files, directories, symbolic and hard links) that had been modified by aborted transactions. Transactions can fail due

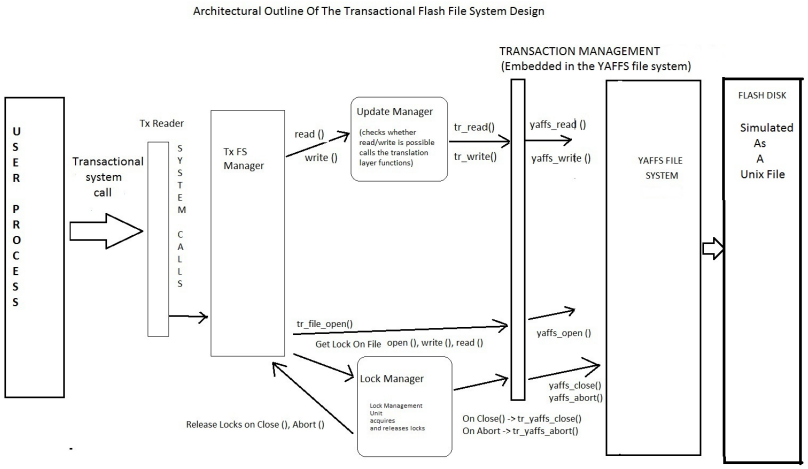

Fig. 1: Overview of the architecture of Transactional Flash File System

to two reasons. Firstly, there can be client side aborts. Secondly, there can be a server crash. Accordingly, the recovery protocols are divided into two broad categories viz. Transaction abort recovery protocol and Server crash recovery protocol.

4.1 Data Structures

YAFFS is a true log-structured file system. The log consists of segments (called blocks) of pages (called chunks). Each chunk consists of an object identifier (inode number), chunk identifier (logical page number), byte count and sequence number. For server crash recovery protocol, we have added a transaction identifier to each chunk. YAFFS maintains a node (called tnode) tree that maps each logical chunk in the file to a physical chunk on disk. We maintain a chunk map list (explained in subsection 4.2) that primarily stores a map that relates modified chunks of each file to their earlier on disk location. For transaction abort recovery protocol, we maintain a list of all the transactions that have modified any file, along with the file metadata (Transaction File Metadata List). For the server crash recovery protocol, a pair of bitmaps (in memory and on disk) is used in order to identify transaction status.

4.2 Transaction Abort Recovery Protocol

,

Transaction abort recovery protocol restores the metadata of objects that have been modified by a user transaction abort. Transaction recovery restores the chunk map tree (chunk map tree is a map which converts logical page indices to physical page indices), file size and the timestamps of objects that have

been modified. A log-structured file system has previous consistent copies of data chunks (chunks are pages in YAFFS terminology) already on disk. Hence, rewrites to disk are avoided.

Data Structures The data structures used for Transaction abort recovery protocol are as follows:

1. **Transaction File Metadata List**: Transaction File Metadata List stores a list of all those transactions that have modified any object. This list stores metadata of all the objects that have been modified by the transaction. The metadata is a snapshot of the previous state of the file before it was modified by the transaction. The recovery protocol switches the metadata state of objects to this consistent state once the transaction that modifies it aborts. The members of this list are as follows:
   - **Transaction Identifier**: Each element in the transaction file metadata list is uniquely identified by a transaction and an object identifier. The transaction identifier uniquely identifies a transaction.
   - **Object Identifier**: The object identifier is the inode identifier for a file (that has been modified by a transaction).
   - **File Length**: The length of each file that has been modified by a transaction is stored. During recovery, the file length is restored to the original file length (that was before a transaction modified it).
   - **Chunk Map List**: To restore a file the chunk map tree has to be restored in a log-structured file system. The chunk map list stores a map of pages that have been modified along with their previous consistent states on disk. The transaction file metadata list stores a pointer to a chunk map list.
2. **Chunk Map List**: Chunk map list stores a mapping from the logical space to the physical (on-disk) space for each chunk. Transaction abort recovery protocol is an undo based protocol. The physical identifier of a chunk in this list is the image of the chunk that was present before the transaction modified it. The list is an in-memory structure. Chunk map list consists of the following members:
   - **Logical Chunk Identifier**: The logical chunk identifier identifies a chunk(page) within a file. Only those chunks that have been modified by a transaction are stored within this list.
   - **Physical Chunk Identifier**: For each modified chunk the previous consistent image is stored in a physical chunk identifier. It translates to on-disk address of a chunk.

The Protocol Transaction abort recovery protocol proceeds in three steps:

1. **Initialization**: The initialization phase initializes the transaction file metadata list by inserting the transaction identifier of each uncommitted transaction and the metadata of objects that each such transaction identifies.

2. **Update**: On a write call, since a copy-on-write takes place, data is written on to a new chunk. Whenever, a chunk is flushed to the disk (because of sync called by the user, use of write through mechanism, cache buffers become full), the logical chunk id to physical chunk id of each modified chunk is inserted in the chunk map list within the transaction file metadata list.
3. **Recovery/Rollback**: The recovery is an undo based rollback mechanism. The file length is updated. The chunk tree is restored to an earlier consistent state using the chunk map list in the transaction file metadata list.

4.3 Server Crash Recovery Protocol

The Server Crash Recovery Protocol is based on the concept of identification of the committed transactions through the on-disk inode (object header in YAFFS). Every time a commit takes place the file is closed and the inode is written to the disk. Each chunk written to the disk (both data and metadata chunk) has a tag field, which identifies the transaction that has written the chunk to the disk. Inodes written to the disk identify committed transactions.

Data Structures The following data structures are maintained for the server crash recovery protocol:

- **In-Memory Transaction Bitmap**: It is an in-memory data structure that maintains the state of all the transactions. It stores the status of each transaction . Currently, the transaction state consists of a committed and an uncommitted state. The transaction identifiers are allocated by the file system itself, so there are no issues of collision of transaction identifiers (handled by keeping a pool of free transaction identifiers). The structure is a bitmap, storing binary information for each transaction identifier i.e. 1 for a committed transaction and 0 for an uncommitted transaction. Whenever a chunk is to be validated for commit or abort status, the transaction bitmap is looked up and the value provides the validity of the chunk.
- **On-Disk Transaction Bitmap**: On-disk transaction bitmaps are required to persist the status of transactions across reboots. Garbage collection in log-structured file systems may lead to intermediate inodes getting cleaned resulting in some data chunks becoming falsely uncommitted. The on-disk transaction bitmap stores the status of all transactions that have been committed. After every boot, a scan takes place that builds the in-memory data structures required by the file system. On encountering an inode-chunk the in-memory transaction bitmap is updated. On the completion of the scan process, the in-memory transaction bitmap is synced with the on-disk transaction bitmap. If the on disk transaction bitmap is stale, we flush the in-memory transaction bitmap to the disk. Thereby ensuring the consistency of all those transactions for which the mapping inode chunks have been invalidated due to re-write of the inode chunk. It is implemented by allocating a set of chunks in a file (t_bmap).

Protocol Implementation The server crash recovery protocol consists of two stages, a scan stage and a sync stage.

1. **The Scan Stage**: The first stage in the recovery is the stage after the crash takes place and the file system boots. The file system scans the data and the metadata chunks in the opposite direction. This ensures that the inode chunks for each committed transaction are encountered first and followed by their respective data chunks . The in-memory transaction bitmap is initialized to all zeros - reflecting that as of now the file system does not know of any committed transaction. On a scan two categories of chunks are encountered:
   (a) **Metadata Chunks**: Each metadata chunk identifies a committed transaction. Therefore, the corresponding bit within the in-memory transaction bitmap is updated to reflect a committed transaction.
   (b) **Data Chunks**: There are basically two kinds of data chunks. They are:
      i. **Normal Data Chunk**: On encountering a normal files data chunk, the recovery protocol performs a validation check. The transaction identifier of the data chunk is checked against the in-memory transaction-bitmap. If the corresponding bit is set to one this data chunk becomes part of a committed transaction else the chunk is marked to be deleted.
      ii. **Bitmap Data Chunk**: On encountering the bitmap data chunk, the bitmap data is read in the t_bmap file as a normal file. This is later opened and read in the sync stage, so that the unmapped committed transactions persist across reboots.
2. **The Sync Stage**: After the scan completes, the t_bmap file is read and all the transactions that are marked committed in the on-disk bitmap are marked valid in the in-memory data chunk. This way the transactions for which the inode gets over-written persist across the reboots and the in-memory transaction-bitmap reflects a consistent view of the transaction-identifier space. The on-disk bitmap is checked for staleness. The on-disk bitmap becomes stale when there is at least a single committed transaction that has not been marked committed on the on-disk bitmap. This occurs when the transaction is committed after the last scan. The transaction bitmap is then written back to disk only if it was earlier found to be stale.

5 Evaluation

The primary objective of the experimental study was to measure the performance of our recovery protocols (implemented in YAFFS) with existing techniques. A file system that writes to a separate log file for journaling data (Separate Log FS) has been modelled. We have considered transaction aborts while measuring performance. Overheads due to data writes to files during the recovery process have been compared.

5.1 Experimental Setup

For the experimental study we have used a client server (file server) architecture. The client sends request to the server through Remote Procedure Calls. The transaction aborts were communicated to the server by the client. Each transaction opens a file, writes data in the file and either aborts or commits. Finally the data is read by the last transaction. We have performed experiments on three sets of data writes per transaction. The sets have been divided according to the amount of data that is written to the disk per transaction. The three categories are:

1. **Small Data Per Transaction**
   For this category we have taken data of the order of 5-10 KB per transaction.
2. **Medium Data Per Transaction**
   For this category we have taken data of the order of 10-20 KB per transaction.
3. **Large Data Per Transaction**
   For this category we have taken data of the order of 20-50 KB per transaction.

For each of the above categories of data writes per transaction, we have simulated transaction aborts on the client side and measured the performance of our transaction abort recovery protocol. The parameter for performance measure is the overhead incurred during data writes to disk per transaction. Consistency of the data written by the client has also been checked. Each experiment consists of five different abort rates (0%, 20%, 32%, 70%, 77%). The results for each category (for a particular abort rate) have been obtained by taking an average over 1000 transactions.

5.2 Separate Log File System

This file system uses a separate log to journal data writes. Recovery is done using undo operations that read consistent image of data from the journal. The earlier image of data blocks gets written to the log. On a transaction abort, all the data blocks that have been written by the aborted transaction are recovered from the log. On a transaction commit the inode block and all the data blocks are written to disk. The journal is a separate file. Unlike a log-structured file system, it supports writing in place. Only data blocks are journaled to the log to ensure consistency.

5.3 Results

In 1 each transaction writes about 7.5 KB of data to the disk and at very high abort rates the data written to the disk falls down to 1.5 KB (effective data written) per transaction. Some of the observations are:

Table 1: Comparison of Overheads: YAFFS VS Separate Log FS for small writes per transaction

Abort Rate (IN %)	Effective Data Written	Overhead Separate Log FS	Overhead YAFFS	Overhead Ratio Separate Log FS:YAFFS
0	7430	0.85	0.57	1.17
20	5922	1.33	0.91	1.21
32	4689	1.94	1.34	1.25
70	3001	3.56	2.48	1.31
77	1499	8.17	5.74	1.36

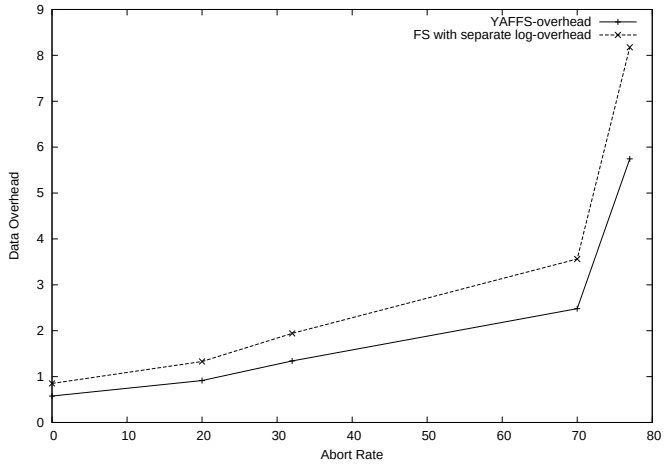

Fig. 2: Comparison of write overhead of the actual results with the theoretical model for Separate Log File System and YAFFS for small writes per transaction

1. The overhead ratio for the separate log scheme was almost 1.25 times more than that for YAFFS. This is because for each write to the disk the previous image of the data chunk is written to the disk. However, this is not close to two. For a sequential write model only a single page is written back to the disk per transaction.
2. YAFFS performs better than the separate log based journaling file system (see Fig. 2) primarily because YAFFS does not write duplicate data to the disk. This reduces writes to the disk and the overall overhead is less.

5.4 Comparison Across Writes

From Fig. 3a and Fig. 3b, the following observations can be made:

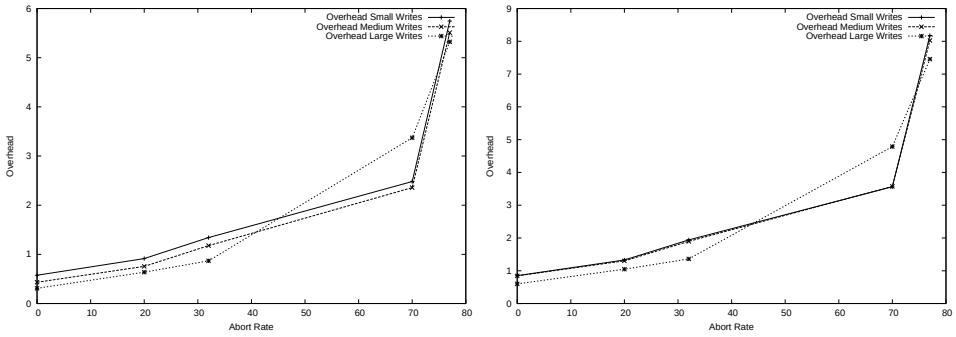

Fig. 3: Comparison of the overhead over different write sizes. (a) Shows the comparison of overheads for YAFFS across the three different write patterns. (b) Shows the comparison of overheads for Separate Log FS across the three different write patterns.

1. The overhead, when the abort rate is low, decreases as the size of data written per transaction increases. For each transaction commit inode is flushed to the disk. Therefore, if the amount of effective data that is written to the disk for each transaction is high, then the overhead cost that is incurred becomes low.
2. At higher abort rates, however the overhead for the large writes increases. This occurs because a large amount of data that belongs to aborted transactions is flushed to the disk. Thus, the overall writes to the disk increase.

The experimental study performed conclusively shows that YAFFS outperforms file system with separate log, which incurs heavy overheads due to writes to the log. The overall data overhead for the file system with separate log was 1.25 to 1.40 to that of YAFFS. The overhead due to writes is affected by the size of data that gets written to the file per transaction and the abort rate. The overhead mainly occurs due to metadata writes, duplicate data writes (write on a partially written block involves rewriting some of the earlier data as the granularity of write is a block) and writes of aborted transactions. For a particular range of data writes per transaction (viz. small, medium, large) as the abort rate increased the overhead increased too. A comparison across write sizes reveals interesting results for a log-structured file system. At lower abort rates the overhead decreases as the writes per transaction increase. At a fixed abort rate variation in the overhead across different write sizes is observed due to metadata writes. As the amount of data written to disk per transaction increases the overhead decreases. As the abort rates increases a different effect is observed. The overhead due to aborted data became the dominating factor over the overhead due to the metadata write. This is because the metadata write is almost constant per transaction and the data written to disk per transaction increases. At

20% abort the overhead decreased from 0.91 (for small writes per transaction) to 0.64 (for large writes per transaction). At 70% abort rate the overhead was actually lesser for smaller writes (overhead $\approx$ 2.48) than that for large writes per transaction (overhead $\approx$ 3.37). This is because the overhead due to aborted writes becomes much larger as compared to the overhead due to metadata write per transaction.

6 Conclusion

This work presents a design for providing transaction support in a log-structured file system for flash devices. The primary idea proposed is to tag pages with the transaction identifiers and to flush file inodes at the time of commit, thus, enabling the transactions to be identified as aborted or committed. We provide a transaction abort and a server crash recovery protocol. Using a comparison based simulation study this work shows that supporting transactions within log-structured file systems is efficient in terms of writes to the storage disk. The copy-on-write feature of log-structured file systems along with high speed random reads in flash file systems can enhance the performance of user applications and at the same time ensure consistency.

Copy-on-write capabilities of a flash file system can be effectively used for an online versioning system. The versioning system would make use of transactional support from the file system. Applications such as an online backup of a transactional log-structured file system present scope for future research and development.

References

1. http://www.yaffs.net/.
2. Lipika Deka. *On-line Consistent Backup in Transactional File Systems*. PhD thesis, Dept of Computer Science and Engineering, IIT Guwahati, April 2012.
3. John K. Ousterhout Mendel Rosenblum. The design and implementation of a log-structured file system. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, February 1992.
4. Vijayan Prabhakaran, Thomas L. Rodeheffer, and Lidong Zhou. Transactional flash. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 147–160, Berkeley, CA, USA, 2008. USENIX Association.
5. Margo I. Seltzer and Michael Stonebraker. Transaction support in read optimizied and write optimized file systems. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 174–185, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.
6. S.C. Tweedie. Journaling the linux ext2fs filesystem. In *The Fourth Annual Linux Expo*, 1998.
7. Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending acid semantics to the file system. *Trans. Storage*, vol. 3(no. 2), June 2007.
8. Charles Philip Wright. *Extending acid semantics to the file system via ptrace*. PhD thesis, Stony Brook, NY, USA, 2006. AAI3238986.