H-0177 June 25, 2003
Computer Science

IBM Research Report

Lock Free File System (LFFS)

Edya Ladan-Mozes, Ohad Rodeh, Dan Touitou
IBM Research Division
Haifa Research Laboratory
Haifa 31905, Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Lock Free File System (LFFS)

Edya Ladan, Ohad Rodeh, Dan Touitou
IBM Research Labs in Israel
ladan,orodeh@il.ibm.com

Abstract

LFFS is a lock free distributed file system without servers or managers. It makes use of techniques from shared memory field to overcome distributed locking and fault tolerance problems. These techniques allow the file system to scale to petabytes of data and to a theoretically unlimited number of clients. The system is built on top of a set of independent disks with enhanced functionality called *Object Based Storage Devices* (OBSD). The novelties presented in this work are: 1) the use of Read-Modify-Write operations in storage systems for synchronization purposes, 2) the introduction and use of a light-weight variation of the Load-Linked Store-Conditional operation, provided by the OBSD, and 3) the design of a file system using the above techniques. A prototype of LFFS is currently being implemented.

1 Introduction

Industry groups, such as SSPs and Telcos, have introduced a growing need for very large scale data storage systems that can efficiently handle increasing amounts of data and clients. Large storage systems traditionally consist of central servers, which limit scalability in terms of storage space, clients, and bandwidth. In an attempt to circumvent the need for a central server, new distributed storage systems have been introduced. These systems typically reduce server workload by distributing its functionality among other components — server cluster, clients, and disks. Both centralized and distributed systems require some synchronization mechanisms to coordinate between the system components and to keep the system's state consistent, even when failures occur. Usually, lock servers, distributed locking algorithms, and group communication are used for solving the distributed synchronization problems. These approaches do not allow file systems to scale beyond several hundred nodes. This work attempts to build larger systems, using different techniques.

With the advent of SANs (Storage Area Networks), standard disks, once connected to a server, have become distributed independent units that provide services to many clients. As a consequence, functions such as allocation and protection, previously performed by servers, have become complex distributed problems. Hence, standard disk functionality has been recognized as insufficient, and there is an effort to enhance disks to *Object Based Storage Devices* (OBSD) [2]. The OBSD manages its allocation, and provides an object based interface for data access. A client can manipulate objects through operations such as create, delete, truncate, read, and write.

Prior works in the shared memory field [12, 15, 16, 18, 20, 25], have shown the strength of *Read-Modify-Write* (RMW) operations in solving synchronization problems. Every synchronization object can be implemented in an asynchronic environment without using locks. All RMW operations read data, update it and store it back in an atomic manner.

A specific RMW operation called *Load-Linked* (LL) *Store-Conditional* (SC) was defined in [16]. In this work, the standard OBSD is enhanced to support a variation of LL-SC. We denote this variation *Load-Linked OBSD* (LLD) *Store-Conditional OBSD* (SCD), see details in section 3.2.2. In brief, LLD reads a byte-range from an object, and SCD stores a byte-range to that object, only if the range has not been modified in the meanwhile. There are no assumptions as to the success of these operations, and they can fail spuriously. This allows LLD and SCD to be efficiently implemented and integrated in OBSDs.

In this work we introduce a new file system — a *Lock Free File System* (LFFS). LFFS uses the LLD and SCD operations to eliminate synchronization mechanisms, servers and managers. Thus, the system can scale up to hardware limits: network, disks and CPUs. Consequently, the system can scale to petabytes of data and to a theoretically unlimited number of clients.

2 Related work

File systems have been extensively used to provide a convenient way to access stored data. The most basic of these are local file systems, providing access from a machine to its locally attached disks (NTFS [7], Ext2 [23], and more). Since current machines are interconnected by the Internet protocols, and sharing of data stored on remote machines is required, the next step was to provide access through standard protocols (NFS [5] and CIFS [21]) to remote file systems. Storage attached to a single server introduces a bottle-neck in the system for clients' access, and limits performance. Hence, distributed file systems have been built, clustering machines and disks to provide a uniform file-system interface [1, 17, 27].

The major challenge in building large scale file systems is synchronization. Typically, file systems guarantee atomic operations such as read, write, create, truncate and rename on files and directories. Since files can be accessed by many clients, locking is required to maintain atomicity. Since clients may fail, some mechanism is required for releasing held locks. Typically, *leases*, or limited time exclusive locks are employed. Since servers may fail, mechanisms for server failover, recovery, and agreement on lock state are required. These mechanisms are based on consensus style protocols, which limit scalability.

In xFS [27], storage, cache, and control are spread over cooperating workstations. Each station exports its local disk as shared storage space. Each file has a *file-manager* that maintains cache consistency and disk location meta-data. When a client wishes to read a file block, it consults a local table, looks up the file-manager, and asks for the block. The manager either redirects the client to the workstation holding the block, or, redirects the client to another client already holding the block. This is also called *cooperative caching*. A prototype of xFS has been built, demonstrating fairly good scalability. However, at the time of writing, the fault-tolerance architecture has not been implemented in xFS.

The IBM General Parallel File System (GPFS) [1] connects nodes using TCP/IP, and uses the Phoenix [8] group communication system to enable recoverability in case of node failure. A file system manager coordinates access to files on shared disks by granting tokens that convey the right to read or write data or metadata. After granting a token, a node can operate on the file without additional interaction with the file system manager. When another node attempts to operate on the same region, it gets the lists of conflicting tokens. It is the node's responsibility to communicate with all the nodes holding conflicting tokens and "convince" them to relinquish that token. GPFS has supported file systems of tens of terabytes, with IO rates of gigabytes per second.

Frangipani [6] and Petal [9] are two systems built by the Digital Equipment Corporation (DEC). Petal takes a cluster of machines connected to disks and combines them together to a large block-

device. The Paxos [19] agreement protocol is used to solve consistency and agreement problems. Frangipani is a file system layered on top of Petal. It uses servers to provide file system services, and a distributed lock service to synchronize clients access to Petal.

A different approach is used by GFS [26]. GFS allows direct use of a cluster of disks by clients. File system functionality is spread across the set of clients and storage responsibilities across the devices. Consistency of shared files is maintained by a device locking mechanism. A client can acquire a lock on a file system block, and release it at the end of the operation. A lock acquired by a failed client can be cleared by another node, after a period of time in which the lock was not active. This requires synchronization between the clients.

With the emergence of SANs, file systems such as Storage Tank [22] and QFS [3] have been developed. Such file systems separate files metadata from the data itself. In order to gain access to data, the client asks the servers for its location. The server returns a token to the client, who may use it to directly access data stored on the disk. The servers are also responsible for synchronizing file access by different clients.

Modern disks are independent units and have their own computational power, used mainly for high availability capabilities, such as RAID and copy services. Projects such as NASD [11, 14, 13], Object Disk [2], Active Disk [24] and Autonomous Disks [4], exploit this computational power to delegate some file system functionality to the disks. Hence, the amount of work performed by servers and the amount of data transfered over the network are reduced. For synchronization between clients and disks, lock servers are used.

3 The Model

3.1 LFFS system architecture

LFFS architecture consists of clients and OBSDs, connected to the same network and communicating asynchronously via standard reliable protocols, such as TCP. Each client can communicate with all the OBSDs. Clients occasionally fail or get disconnected from the network for extended periods of time. The OBSDs are assumed to be implemented using powerful and highly available disk machines. LFFS, like NFS, does not keep information about the global state of the system (e.g. open files).

3.2 OBSD and its functionality

The OBSD is an entity in the network, that communicates via standard Internet protocols. The OBSD is an enhanced disk supporting an object based interface for data access. Traditional server functionality concerning disk management, such as allocation, is shifted to the OBSD [2][1]. Functions supported by the OBSD are expected to be implemented efficiently, and to be used by higher level storage systems.

3.2.1 Standard OBSD functionality

1. Create (`object_id`) - Create an object identified by `object_id`.
2. Delete (`object_id`) - Delete an object.
3. Read (`object_id, offset, count`) - Reads `count` bytes from `object_id`, starting at `offset`. Returns the byte array read.

[1] An OBSD can be implemented by placing an operating system on top of a box full of standard disks. The OS then provides a thin software interface to the disk volume.

4. Write (`object_id, offset, count, data`) - Writes `count` bytes of `data` to `offset` within the `object_id`. Allocate space if needed.
5. Append (`object_id, count, data`) - Writes `count` bytes of `data` to the end of `object_id`.
6. Truncate (`object_id, offset`) - Deletes `object_id` data from `offset` until the end of the object.
7. Rename (`old_object_id, new_object_id`) - Change the name of the object `old_object_id` to the name `new_object_id`. All other object characteristics remain the same.
8. Length (`object_id`) - returns the object length.

3.2.2 OBSD RMW operation

RMW operations are a class of synchronization primitives. They allow applying a function to a value and changing it atomically. The LLD-SCD operation presented above is a variation of the Load-Linked Store-Conditional operation [16], performed in block or byte-range granularity.

A client issues the LLD operation in an attempt to read a byte range — *count* bytes starting at *offset*, from *object_id*. A *ticket* is supplied to the client by the OBSD for every successful LLD request. This ticket is also saved in volatile memory within the OBSD. When the client issues the SCD operation, it provides the OBSD with tickets that were previously issued by that OBSD, as well as data to be written. The SCD request will successfully write the data only if all provided tickets are still valid — existing in the OBSD memory. A ticket is invalidated by the OBSD (removed from its memory) if another client succeeded to perform a SCD on an overlapping byte range. It can also be invalidated for other reasons, such as memory shortage or a failure.

1. LLD (`object_id, offset, count`), returns a ticket and the content of the byte range - *count* bytes starting at *offset*.
2. SCD (`object_id, tickets_list, offset, data`), returns success value. If successful, invalidates overlapping tickets, except the successful ticket(s).

4 Design Techniques

The model presented above allows clients to be disconnected or unavailable, even during execution of the file system operations. The file system state remains consistent even in such situations. To keep the file system state consistent, the following techniques, borrowed from shared memory [16] are used:

- **No-return stage** - A stage in an operation, such that if the client that started the operation performs it successfully, then the operation must be completed either by the initiating client or by another client, even if the initiating client fails.
- **Publishing** - A shared known byte-range is dedicated for storing information about the operation executed by a client. This information contains all the needed variables, so the operation can be completed by other clients in cases of contention or client failure. This space is rather small and is located close to where the operation is taking place.
- **Helping** - Any available client can continue operations started by an unavailable client, if that client had passed the no-return stage of that operation.
- **Interfering** - Any client that encounters an operation in mid-flight, prior to the no-return stage, should try to stop it. Then, it can initiate its own operation.

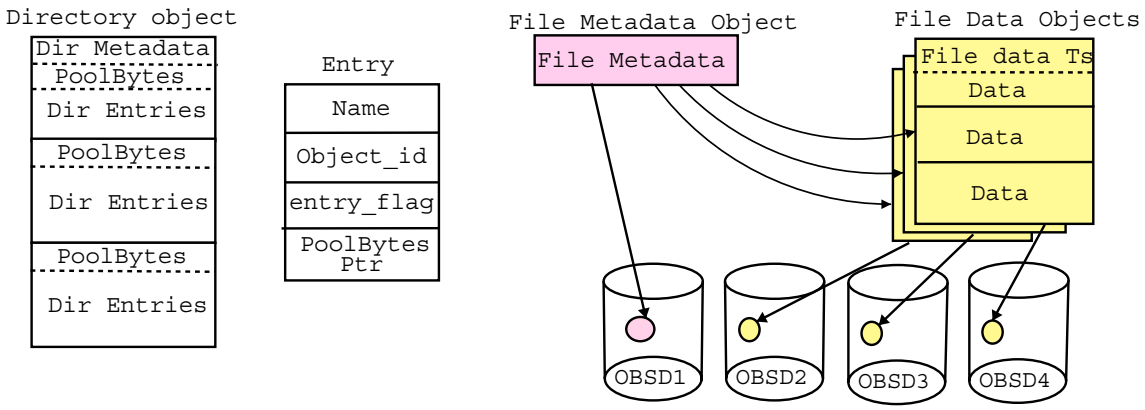


Figure 1: LFFS structures

- **Back-off** - For efficiency, before deciding to help or interfere, a client can hold for a period of time, to enable the completion of the operation by the initiating client or other clients. Currently this technique is not implemented.

5 Representation and Internal Structure

The following sections describe how directories and files are represented in LFFS. The OBSD has no knowledge about the representation, which is managed solely by the LFFS. Figure 1 graphically explains the internal structures representation.

5.1 Directories

A directory is maintained as an object within an OBSD. It is logically divided into numbered virtual blocks (vBlocks), whose size is user defined. A vBlock contains *entries* of files and sub-directories, and some space for publishing operation information, called *PoolBytes*. The beginning of the first vBlock contains the directory *metadata* (attributes). Initially a directory has only one vBlock used to store all entries. However, a directory may dynamically grow and shrink as needed, using an extendible hashing scheme [10]. A map of the available vBlocks is maintained in the directory metadata. The distribution of entries over multiple vBlocks is performed by hashing the entry *name* field to a 32-bit integer. Using the hash value and the vBlock-map, the placement of the entry can be determined. The root directory is placed in a fixed known location.

Each vBlock of a directory includes the following:

1. PoolBytes[n] - Array of n cells, used for publishing. Each cell contains a sufficient amount of bytes to store operation information needed to complete the operation by other clients.
2. split-level - Several bits that are needed for the extendible hashing scheme.
3. Directory and file entries.

The metadata vBlock also includes:

1. dir_flag - A flag that can take any of the following values: Null, BeingDeleted, BeingEnlarged, BeingShrinked

2. map - the vBlock map.
3. standard attributes - e.g. mode, type, uid, gid, size, aTime, cTime, mTime.

5.2 Entries

An entry is a structure that is kept for each file and directory. It binds the name of the file or directory with an object in the OBSD and holds published information for internal use.

Each entry includes the following:

1. name
2. object_id (OBSD + Object)
3. entry_flag - A flag that can take any of the following values: Name, InCreateDir, InCreateFile, InDeleteDir, InDeleteFile, InDeleteSuccessDir, InDeleteSuccessFile, InRenameFrom, InRenameTo, InRenameFromSuccess, InRenameToSuccess, InLink InLinkSuccess, InUnlink, InUnlinkSuccess.
4. ptr - a pointer to a cell in PoolBytes.

5.3 Files

A file is represented by one metadata object, and several data objects, that are spread among several OBSDs. In this way, a file can by larger than an entire OBSD. The data is distributed among the data objects according to a predetermined method and granularity (chunks of several vBlocks). In our system [2], the file is first created with the metadata object only. The first write to the file creates a new data object in another OBSD. When that data object, denoted X, becomes full i.e. the OBSD of that data object is full or the size of the object has reached a predetermined limit, an additional data object Y is created on another OBSD. X will contain data from the beginning of the file until the offset at which it was split. Y will contain the rest of the file. When Y's OBSD becomes full, this procedure is repeated.

Every file metadata object and data objects contains a timestamp. This timestamp is used to indicate that a certain change has been made to the file metadata. The timestamp is handled by the LFFS and does not related to real time. The timestamp is increased by LFFS only when applying operations that decrease file size (such as truncate) or change the file layout. Operations that are executed concurrently and depend on the file size or layout (such as write) use the timestamp to detect such changes. The client must stop the operation, and reload the file metadata. Only then can the operation proceed.

A file metadata object contains:

1. file_ts - A timestamp that is used in some operations, to indicate a change made to file metadata.
2. file_flag - A flag that can take any of the following values: Null, BeingTruncated, BeingDeleted, BeingLinked, BeingUnlinked, BeingSplit.
3. map - A tree of the data objects of the file.
4. dist_method - data distribution system.
5. nLink - the number of hard links to the file.
6. standard attributes - e.g. mode, type, uid, gid, size, aTime, cTime, mTime.

[2] The distribution method can be replaced by another one.

The first vBlock in a file data object contains:

1. data_ts - A timestamp that indicates that a change was made in the file metadata. Basically, it is a copy of the metadata timestamp.

6 Implementation

LFFS operations are divided into five classes, depending on the objects involved, the amount of synchronization needed, and the internal functionality.

1. **Non-interfering** - Operations that involve only one existing object. Includes: mkdir and create.
2. **Interfering** - Operations that involve coordination between two existing objects on different OBSDs. Includes: rename, rmdir, remove, link, and unlink.
3. **Common** - Operations on files that involve the data objects. Includes: truncate, write, and read.
4. **No-sync** - Operations that do not require synchronization. Includes: lookup and readdir.
5. **Internal** - Operations that are called by other operations as subroutines. It includes:
   - enlargeDir - takes care of enlarging a directory if it is full.
   - shrinkDir - takes care of shrinking the directory if it becomes sparse.
   - split - splitting a file data object if there is no more space in the OBSD.

All operations cause a change in the time attributes. Updates to these attributes are issued periodically by an internal mechanism. Each client holds a queue containing buffered updates. Periodically, these updates are sent to the appropriate OBSDs.

The following explains the three first classes, to demonstrate the use of the techniques in implementing the file system. To simplify the pseudo code below, the following conventions and notations are used:

- If an SCD operation fails, the whole operation is restarted unless stated otherwise.
- *metadata_offset, metadata_size* - the first byte and the number of bytes of the metadata.
- *entry_block_offset, entry_block_size* - the block offset within a file or directory and the size of a vBlock.
- *data_ts_offset, data_ts_size* - the position and the number of bytes of data_ts.
- To differentiate between LFFS operations and OBSD operations, all standard OBSD operations will be denoted as *OperationD.* (e.g. *CreateD*)

6.1 Algorithms

Directory creation - mkdir(pathname):

The pathname is looked-up to find the parent directory, and the new directory. The new directory may be created on a different OBSD then the parent. Denote: *new* - the new directory to create, *parent* - the parent of *new*, and *success* - a boolean value.

Method:

1. t, b = LLD(*parent_oid, metadata_offset, metadata_size*).
2. If (b.dir_flag <> Null) then call the appropriate function according to the flag.

3. Calculate the offset of the block entry where *new* should be placed.
4. $t1, b1$ = LLD(*parent_oid, entry_block_offset, entry_block_size*).
5. If $b1$ is full then call an internal function that enlarges the directory (enlargeDir).
6. If *new* exists then:
   - if (b1.entry_flag = InCreateDir) then read PoolBytes[ptr] and go to stage 9 to help complete the operation.
   - if (b1.entry_flag <> Null) then call the appropriate function according to the flag.
   - if (b1.entry_flag == Null) then *new* already exists.
7. If *new* does not exist then update $b1$ with the following:
   - Decide in which OBSD to create the object, and acquire an object_id
   - Create an entry for *new*: {name = *new*, object_id = (new_OBSD, new_oid), entry_flag = InCreate, ptr = i }.
   - PoolBytes[i] = Creation of *new* with all the above details.
8. *success* = SCD(*parent_oid*, $[t, t1]$, *entry_block_offset*, $b1$) - No-return stage.
9. CreateD (*new_oid*) - Create *new* object in new_OBSD with initial attributes.
10. Update $b1$ with the following :
    - Modify *new* entry with: {name = *new*, object_id = (new_OBSD, new_oid), entry_flag = Name, ptr = null }.
    - PoolBytes[i] = Null.
11. *success* = SCD(*parent_oid*, $[t, t1]$, *entry_block_offset*,$b1$).
12. Enqueue a time update request for *parent_oid* metadata.

Explanation

A client who wishes to create a new directory first LLDs the metadata of the parent directory, and the block where *new* should be created. If *new* exists then the $entry\_flag$ is checked to decide what action to take. If *new* does not exist, then we create it and try to SCD its vBlock. Once a client succeeds in performing this SCD (stage 8), it is guaranteed that the mkdir operation will complete. Even if the client fails in the following stages, another client will complete the operation on his behalf later on. This helping technique allows the system as a whole to continue to progress even though clients may fail.

Since clients can help each other, race conditions can occur. Consider a case where client A attempts to create file X and succeeds in stage 8. Client A subsequently sleeps for several minutes. Client B, attempting to perform another operation on X finds X in an interim state. It then follows up A's operation and completes the operation, using the published information in PoolBytes[i]. When A wakes up, it's SCD operation at stage 10 will fail because the tickets it provides are no longer valid. It will restart the operation, and discover that the operation has already been completed. This situation is satisfactory from A's point of view.

Rename File - rename(old, new):

The rename function works on both files and directories, and it associates a new name (*new*) to the object pointed by *old*. If *new* does not exist, it should be created. If it does exist, the

operation fails[3]. On successful completion of rename, an entry should exist for *new*, and the *old* entry should not exist.

Method:

1. t, b = LLD(*old_parent_oid, metadata_offset, metadata_size*).
2. If (b.dir_flag <> Null) then call the appropriate function or try to interfere.
3. Calculate the offset of the block entry where *old* should be.
4. $t1, b1$ = LLD(*old_parent_oid, old_entry_block_offset, entry_block_size*).
5. If *old* does not exist in $b1$, then abort.
6. if (b1.entry_flag <> Null) then call the appropriate function according to the flag.
7. Update $b1$ with the following:
   - *old* entry with: {name = *old*, object_id = (OBSD, oid), entry_flag = InRenameFrom, ptr = i }.
   - PoolBytes[i] = InRenameFrom with *new* information.
8. *success* := SCD(*old_parent_oid*, $[t, t1]$, *old_entry_block_offset*, $b1$).
9. $t2, b2$ = LLD(*new_parent_oid, metadata_offset, metadata_size*).
10. If (b2.dir_flag <> Null) then call the appropriate function or try to interfere.
11. Calculate the offset of the block entry where *new* should be placed.
12. $t3, b3$ = LLD(*new_parent_oid, new_entry_block_offset, entry_block_size*).
13. If *new* exists then abort the operation.
14. Update $b3$ with the following:
    - *new* entry with: { name = *new*, object_id = (OBSD, oid), entry_flag = InRenameTo, ptr = i }.
    - PoolBytes[i] = InRenameTo with *old* information.
15. *success* := SCD(*new_parent_oid*, $[t2, t3]$, *new_entry_block_offset*, $b3$)
16. Update $b1$ with the following:
    - *old* entry with: { name = *old*, object_id = (OBSD, oid), entry_flag = InRenameFromSuccess, ptr = i } .
    - PoolBytes[i] = InRenameFromSuccess with *new* information.
17. *success* := SCD(*old_parent_oid*, $[t, t1]$, *old_entry_block_offset*, $b1$) -No-return stage.
18. Update $b3$ with the following:
    - *new* entry with: { name = *new*, object_id = (OBSD, oid), entry_flag = InRenameToSuccess, ptr = i } .
    - PoolBytes[i] = InRenameToSuccess with *new* information.
19. *success* := SCD(*new_parent_oid*, $[t2, t3]$, *new_entry_block_offset*, $b3$).
20. Update $b1$ with the following:
    - Remove *old* entry.
    - PoolBytes[i] = Null.
21. *success* := SCD(*old_parent_oid*, $[t, t1]$, *old_entry_block_offset*, $b1$)
22. Update $b3$ with the following:

[3]This is a slight deviation from the standard. The standard requires that if *old* and *new* are of the same type (both directories or both files), then *new* is removed, and *old* is renamed to *new*. This is currently not done to simplify the protocol.

- *new* entry with: {name = *new*, object_id = (OBSD, oid), entry_flag = Name, ptr = Null}.
- PoolBytes[i] = Null.

23. *success* := SCD(*new_parent_oid*, [$t2, t3$], *new_entry_block_offset*, $b3$)
24. Enqueue a time update request for *old_parent_oid* and *new_parent_oid* metadata.

Explanation

To rename a file or directory, a client should mark *old* - the current name, as InRenameFrom. Afterwords it checks if *new* name is already exist in the directory it should be created. If *new* already exist then the operation is stopped. Otherwise, *new* entry should by created with entry_flag InRenameTo. Rename succeeds only after the initiator successfully SCDs *old* name with entry_flag InRenameSuccess - the no-return stage (17), and must fail otherwise. Once a client successfully passes stage 17 it is guaranteed that the operation will succeed. Only the initiator may perform the algorithm until the no-return stage. Any client that observes the operation after the no-return stage must help to complete it. On the other hand, if it is in prior stages, the client should interfere with it. Interference is performed by SCD-ing *old* to its normal status and removing the *new* entry. Since rename can be rolled-forward, or rolled-backward, it requires that all clients agree on what operation to take — forward or backward. In any stage of the algorithm, any client that reads *new* or *old* entries knows which step to take.

A peeking client is a client that encounters an operation in mid-flight. A peeking client should act as follows:

1. Encounters *old* with $entry\_flag$ = InRenameFrom (the initiator successfully performed stage 8) - SCD *old* with Name flag. If successful, the peeking client can continue its operation; the rename operation will fail and the initiator will need to restart it.
2. Encounters *new* with $entry\_flag$ = InRenameTo (the initiator successfully performed stage 15) - Check *old*'s $entry\_flag$. If it is InRenameFrom or Name, then SCD *old* with Name flag. If successful, then remove *new* entry. If both operations succeed, the peeking client can continue its operation. The rename operation will fail and the initiator will need to restart it. If it is InRenameFromSuccess then the initiator passed the No-return stage and we must help the rename operation to be completed.
3. Encounters the operation after stage 17 or 19 - It must help the initiator to complete the rename operation.

Truncate File - truncate(object_id, length):

This function truncates the file up to *length* bytes. Denote $fent$ - the file to truncate, *parent* - the parent of $fent$, *metadata_ts* - the metadata timestamp, *data_ts[i]* - the timestamp of data object i, *len* - the length of $fent$ after the operation completes.

Method:

1. t, b = LLD (*fent_oid, metadeta_offset, metadata_count*)
2. If (*file_flag* == BeingTruncated (len')) then continue the operation with len'.
3. If (*file_flag* <> Null) then call the appropriate function.
4. Update b with the following:
   - *new_file_ts* = *file_ts* +1
   - file metadata with: { *file_flag* = BeingTruncated (*len*), *file_ts* = *new_file_ts* }

5. *success* := SCD(*fent_oid*, [*t*], *metadeta_offset*, *b*) - No-return stage.
6. For each data object do:
   (a) $t[i], b[i]$ = LLD(*data_object_oid, data_ts_offset, data_ts_count*)
   (b) $b[i]$ can only be equal or larger than *file_ts*.
   If ($b[i]$ >*new_file_ts*) then the operation was completed.
   If ($b[i]$ == *file_ts*) then
   i. $b[i] = b[i]+1$
   ii. *success* := SCD(*data_object_oid*, [$t[i]$], *data_ts_offset*, $b[i]$).
   iii. If (not *success*) then go to stage 6a.
7. TruncateD and DeleteD data objects as needed.
8. RenameD(*fent_oid*, *new_oid*) - Get *new_oid* from the same OBSD and generate the rename operation with it.
9. Update *b* with the following:
   - file metadata with: { file_flag =Null, size = *len* }.
10. *success* := SCD(*fent_oid*, [*t*], *metadeta_offset*, *b*).
11. If (not *success*) then
    - t, b = LLD(*fent_oid*,*metadeta_offset, metadata_size*)
    - If the size and file_ts have not been updated then goto 10
12. Enqueue a time update request for *fent_oid* metadata.

Explanation

The first stage in truncating a file is to publish the operation arguments (the length and file_flag) and to increment the timestamp in the file metadata. If the client successfully performs the publishing stage (5), then the operation must be completed. Then, the timestamp of each data object is incremented to be equal to the metadata timestamp. A concurrent operation that observes this timestamp change, must be stopped or restarted. To explain stage 7 we provide an example. Assume a file is comprised of five data objects, each containing 10MB, the total file size is 50MB. Suppose a client wants to truncate the file to 32MB. The algorithm deletes the fifth data object, and truncates the fourth to size 2MB. The first three data objects remain untouched. The next stage (8) is to rename the truncated object to prevent a slow client from truncating it after the operation has already been completed. To complete the operation, we update the file length and revert the file_flag to Null.

Write File - write(object_id, offset, count, data):

This function attempts to write *count* bytes starting at *offset* to object_id file. It is possible that less than count bytes will be written, and we return the actual number of bytes that were written. In any case the write is performed atomically.

Method:

1. t, b = LLD(*fent_oid, metadeta_offset, metadata_size*)
2. If (*file_flag* <> Null) then call the appropriate function.
3. If there is no data object, then call an internal function that creates one (split).
4. For each data object that participates in the write do:
   (a) $t1[i], data\_ts[i]$ = LLD(*data_object_oid, data_ts_offset, data_ts_size*)
   (b) $t2[i], b2[i]$ = LLD(*data_object_oid, byte_range_offset, byte_range_size*)

(c) If $data\_ts[i] <> file\_ts$ then: stop operation and retrieve file metadata.

5. For each data object that participates in the write do:
   (a) $success$:= SCD(($data\_object\_oid$,$[t1[i], t2[i]]$, $data\_ts\_offset$, $data$),
   (b) If $success == true$ then $sum = sum +$ size of written data
   (c) Else: return - sum bytes were written.
6. Enqueue a time update request for $fent\_oid$ metadata.

Explanation

To provide atomicity, each write consists of several LLD and SCD. We first LLD the $data\_ts$ of the data objects to whom we want to write and the byte range of the write. We use the ticket issued on the timestamp ($t1[i]$) in order to detect a change in the file metadata. This change may cause the write to fail because the file size has been changed due to an execution of the truncate operation. The second ticket, issued on the byte-range to be written, is used to detect concurrent writes and provide atomicity.

After the LLD, we start to SCD the writes, in sequential order. We try once each SCD. At the first time that a SCD fails, we stop and return the amount of bytes that were written successfully. If all the SCDs succeed, then the write was completed and we return $count$ bytes. We update the file attributes using the update mechanism. If the write enlarges the file, then the size attribute must be updated as well.

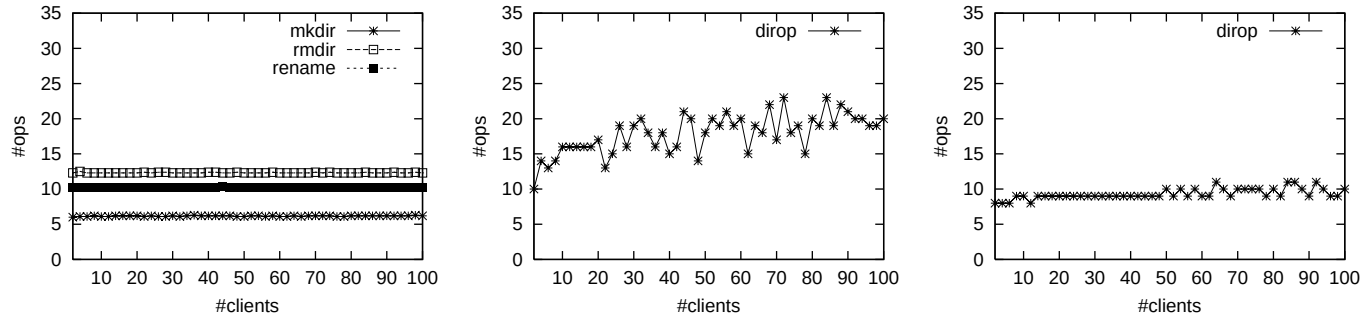
7 Preliminary Results

A Prototype of LFFS was built on top of a simulation of the enhanced OBSD. Our goal was to prove the LFFS concepts and scalability claims in terms of clients and storage capacity. The cost of each LFFS operation was measured under different workloads. The network configuration included a fixed number of OBSDs and a varying number of clients. We have chosen to measure three file system operations - mkdir, rename, and rmdir. Several test conditions were used:

1. No contention: Each client creates it's own directory in the root directory, and in it creates, renames, and removes directories. This that measures the basic costs of these operations.
2. Contention1: clients perform operations on different directories in the root directory. High contention occurs in the root directory.
3. Contention2: clients perform the same set of operations on the same set of directories under the root directory. Contention occurs in the root directory and sub-directories. Furthermore, there is a lot of interaction between the operations.

The graphs in Figure 2 show the costs of performing the above tests as a function of the number of clients. The X axis shows the number of clients. The Y axis shows the average number of OBSD operations issued by a client until it's operation completes. Since we use helping techniques, it is difficult to calculate each operation's cost alone. Therefore, in the Contention1 and Contention2 graphs we applied a weighted-average to the costs of the three measured LFFS operation. The results were found to be the follows:

1. No-contention - Each operation cost is constant, and is between 8 to 12 OBSD operations.
2. Contention1 - The operations cost more than in the No-contention scenario. The cost is a combination of:
   - The basic cost multiplied by the number of entries in each vBlock in the parent directory. In our measurements, each vBlock contains four entries.



No contention Contention1 Contention2

Figure 2: LFFS performance measurements

- The contention on the metadata resulting from enlargeDir and shrinkDir subroutines.

3. Contention2 - Since we use helping techniques and all clients attempt to perform the same operations in the root directory, we can see that the number of OBSD operations issued by the clients is even smaller than the Contention1 scenario.

We have built a non-standard NFS system on top of a set of OBSDs using the same internal structures as LFFS. Preliminary testing shows that LFFS has comparable performance to NFS.

8 Discussion

Standard file systems use servers for locking, meta-data management, and file management. Servers introduce a bottleneck into the system, furthermore, any server is a single point of failure. To support high availability and fault tolerance servers are replicated using consensus style protocols, which limit scalability.

The system model, the use of lock free synchronization, and the use of shared memory techniques enabled us to design a highly scalable file system. In LFFS, lock free synchronization is used to coordinate access to the system. This is supported locally at each OBSD by the LLD and SCD operations. These operations enable to transform complicated distributed problems into simpler local problems obviating the need for servers and managers.

The no-return stage and the helping techniques introduced above ensure that client failures are transparent to other clients and do not affect file system's state. At any stage in which a client fails, others know exactly what action to take in order to complete or interfere with it as needed. There is no knowledge in the system about connected or disconnected clients.

During LFFS design and implementation many questions surfaced. In today's file systems, client-side caching is very important. It is not clear how to add client side caching without breaking the lock-free concept. LFFS is built on top of OBSDs. If an OBSD will support richer functionality than the one assumed, LFFS may improve its performance. Examples of such reacher functionality may be: freeing byte ranges, or atomic transactions.

References

[1] General parallel file system, 1988. `http://www.rs6000.ibm.com/resource/technology/paper1.html`.

[2] Object based storage device, 2001. `http://www.snia.org/English/Work_Group/OSD`.

[3] The quick file system, 2001. http://www.lsci.com/pdf/qfstechnicaloverview.pdf.

[4] C. Akinlar, W. G. Aref, I. Kamel, and S. Mukherjee. Autonomous disks: The building block for a scalable distributed file system.

[5] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. RFC 1813, IETF, June 1995.

[6] C.A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *Symposium on Operating Systems Principles*, pages 224–237, 1997.

[7] H. Custer. *Windows NT File System*. Microsoft Press, Seattle, USA, 1997.

[8] E. Chiakpo. *RS/6000 SP High Availability Infrastructure*. IBM, Internationl Technical Support Organization, Poughkeepsie, USA, November 1996.

[9] E.K. Lee and C.A. Thekkath. Petal: Distributed virtual disks. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, 1996.

[10] R. Fagin. Extendible hashing - a fast access mechanism for dynamic files, 1979.

[11] G. A. Gibson and R. V. Meter. Network attached storage architecture. *Communications of the ACM*, 43(11), November 2000.

[12] G. Barnes. A method for implementing lock-free shared data structures. In *Proc. of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[13] G. Gibson, D. Nagle, K. Amiri, F. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for network-attached secure disks, 1997.

[14] G. A. Gibson, D. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File server scaling with network-attached secure disks. In *Measurement and Modeling of Computer Systems*, pages 272–284, 1997.

[15] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.

[16] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[17] J. Howard, M. Kazar, S. Menees, M. Nichols, S. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[18] J. Anderson and M. Moir. Universal constructions for large objects. In *WDAG: International Workshop on Distributed Algorithms Springer-Verlag*, 1995.

[19] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[20] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, 1991.

[21] P. J. Leach and D. with many others. Common internet file system (cifs). Draft, IETF, 2001. http://www.snia.org/English/Collaterals/Work_Group_Docs/NAS/CIFS/CIFS-SPEC-0P9-REVIEW.pdf.

[22] R. Burns. Data management in a distributed file system for storage area networks. Technical report, Departement of Computer Science, University of California, Santa Cruz, March 2000.

[23] R. Card, T. Ts'o, and S. Tweedie. The design and implementation of the second extended filesystem. In *Dutch International Symposium on Linux*, December 1994.

[24] E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage, 1997.

[25] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[26] S. R. Soltis, T. M. Ruwart, and M. T. O'Keefe. The Global File System. In *Proceedings of the Fifth NASA Goddard Conference on Mass Storage Systems*, College Park, MD, 1996. IEEE Computer Society Press.

[27] T. Anderson, M. Dahlin, J. Neefe, D. Pat-terson, D. Roselli, and R. Wang. Serverless network file systems. In *15th Symposium on Operating System Principles. ACM*, pages 109–126, December 1995.