TR03-07

June 26, 2003

SFSRO LITE - A Self-Certifying Read-Write File System

Siva Kollipara

Supervisor: Dr. John Hartman

SFSRO Lite - A Self-Certifying Read-Write File System

Siva Kollipara, John Hartman
{siva, jhh} @cs.arizona.edu
University of Arizona,
Tucson, AZ- 85721

Abstract

This report presents a read-write version of SFSRO. SFSRO [2], a self-certifying read-only file system, is a content distribution system providing secure, scalable read-only access to data. Since the data exists as a database, the read-only file system structure is too tightly coupled with the data [2]. Any change in the data will cause the whole file system to be updated and its hierarchy modified. To get around this problem, SFSRO creates a completely new database for each version of the file system and incrementally updates the replica file servers.

In this report, we present some modifications and extensions to SFSRO. We aim to achieve a file system with SFSRO functionality and guarantees, but without its drawbacks and limitations – suitably called SFSRO Lite (SFSROL). We plan to build a read-write version of SFSRO that uses a 'floating inode table' [5] and supports variable sized blocks. We have currently finished making necessary changes to incorporate the floating inode table. Further research in achieving a read-write version is in progress. Measurements of an implementation show that SFSROL is approximately 0-2% slower than SFSRO.

Introduction

SFSRO [2] is a secure read-only file system designed to be widely replicated on untrusted servers. Each read-only file system has a public key associated with it. A database of file system contents is created offline and digitally signed using the file system's private key. This database is then replicated on untrusted machines.

The data and meta-data blocks are of fixed sizes and are identified by collision-resistant cryptographic hashes called '*handles*'. The file system is formed by recursively hashing the handles to form a tree of handles with the root directory's handle as the tree's root. In such a hierarchy, directory meta-data blocks contain file name to handle bindings and inodes contain handles of a file's blocks. Using the handle of the root inode of the file system, a client can verify the contents of any block by recursively checking hashes.

Such a handle-based file system tree is very tightly coupled with the data since if the data changes at any level in the tree, it will trigger a series of updates and changes working up to the root. Data updates are critical to a read-write file system, thus modifications are necessary to build a read-write file system. The tight link between handle and file data is essential to preserving SFSRO security, but we can decouple the link between handle and file system meta-data (namely filename to handle mapping in directory entries). We do this by means of a '*floating inode table*' [5].

The rest of this report is organized as follows: Section 2 provides a brief summary of SFSRO design; Section 3 provides the motivations for SFSROL, section 4

describes SFSROL; section 5 details our design modifications and extensions to SFSRO; Section 6 describes its implementation; Section 7 compares the performance of SFSROL vs. SFSRO. In section 8 we discuss future work and Section 9 concludes.

2. SFS RO

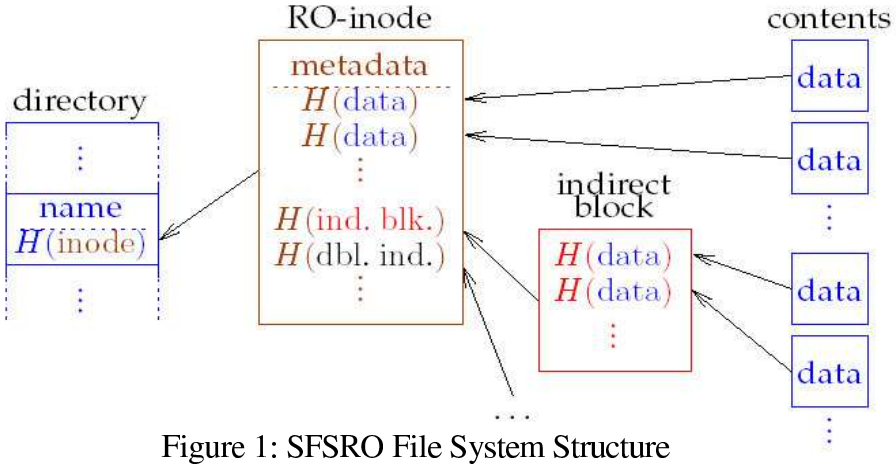

Figure 1: SFSRO File System Structure

A file system has global namespace (implying support for a wide range of applications). Thus, in order to distribute public read-only data securely, the SFSRO Database generator '*sfsrodb*' creates a secure read-only file system database of the contents of the directory passed as the argument. Figure 1 shows the file system hierarchy maintained while forming this read-only file system database. *sfsrodb* takes care of maintaining the relevant file system meta-data structures while forming this database. This database is replicated on several untrusted machines, each of which runs a copy of the SFSRO server daemon '*sfsrosd*'. The server is simple program that looks up the data for a given handle in the replica's database and returns the data to the client. The file system is implemented by the SFSRO client daemon '*sfsrocd*' that runs on the client machine. It handles file system requests from the local operating system. The RO client understands the format of inodes and directories. It parses pathnames, searches directories, looks up blocks of files etc. Figure 2 shows the SFSRO file system components.

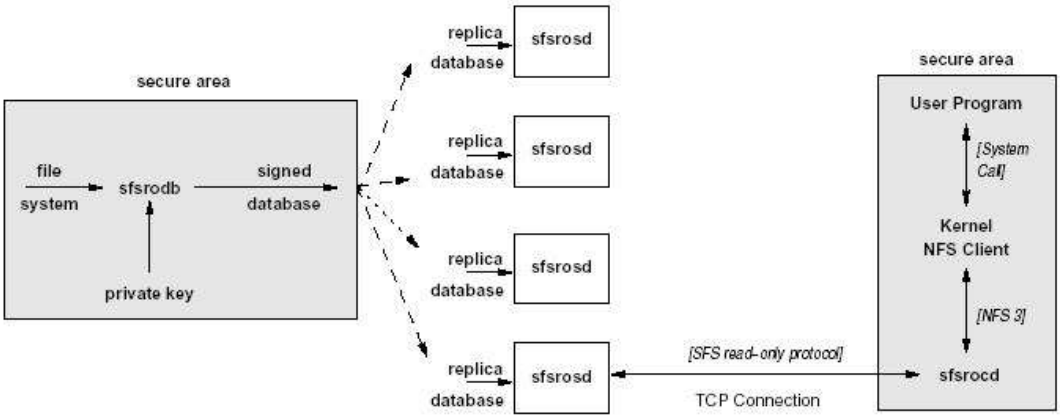

Figure 2: SFSRO File System

The read-only protocol uses two RPCs: *getfsinfo()* and *getdata()*. *getfsinfo()* returns a digitally signed FSINFO structure shown in Figure 3.

```
struct FSINFO {
  sfs_time start;
  unsigned duration;
  opaque iv[16];
  sfs_hash rootfh;
  sfs_hash fhdb;
};
```

Figure 3: FSINFO Structure

```
struct sfsro_inode_reg {
  uint32 nlink;
  uint64 size;
  uint64 used;
  nfstime3 mtime;
  nfstime3 ctime;

  sfs_hash direct<SFSRO_NDIR>;
  sfs_hash indirect;
  sfs_hash double_indirect;
  sfs_hash triple_indirect;
};
```

Figure 4: SFSRO inode structure

```
struct sfsro_dirent {
  sfs_hash fh;
  string name<>;
  sfsro_dirent *nextentry;
};

struct sfsro_directory {
  sfsro_dirent *entries;
  bool eof;
};
```

Figure 5: SFSRO Directory Entry Structure

FSINFO.rootfh is the handle of the files system's root directory. It is a hash of the root directory's inode, which through recursive use of hash specifies the contents of the entire file system. Figure 4 shows the SFSRO inode structure. The inode contains the handles of successive 8KB blocks of file data. If the file has more than eight direct handles, then the indirect handle contains the handles of direct handles. Similarly for double and triple indirect handles. The hash of this inode structure specifies the handle for a file.

The database generator *sfsrodb* traverses the given file system depth-first to build the database that consists of file data blocks and inodes indexed by their handles. For each file *sfsrodb* creates an inode, fills it appropriately and inserts the file data and inode into the database. When all files for a given directory have been inserted, the generator inserts a directory file and an inode corresponding to the directory file. The directory entries are specified as shown in Figure 5. Directory entry structure shows that there is a direct mapping between filename and file handle. The directory file blocks contain lists of <name, handle> pairs and the directory inode contains handles of these blocks. Since a directory's handle depends on the handles of subdirectories (files), a circular dependency makes it impossible to create directory entries for <".", current directory handle> and <"..", parent directory handle>. Thus, during directory inode creation, SFSRO skips "." and ".." entries; these entries are synthesized by the client *sfsrocd*.

Security Guarantees

The key security feature of SFSRO is data integrity. The client verifies that any data sent by the server was signed using the appropriate private key. The client verifies the digitally signed FSINFO structure using the public key embedded in the server's name. The client uses *getdata()* to retrieve data blocks corresponding to hashes and verifies the authenticity of these using the FSINFO structure. Any change to the data will cause a series of changes to the hashes of the data (and meta data) blocks all the way upto the root handle. This will in turn change the FSINFO signature (Figure 3). Thus

SFSRO provides strict security guarantees – the correctness of the signature and hashes ensures data integrity.

More information on SFSRO design and implementation is available [2].

3. Motivations

While SFS provides good functionality and guarantees, it has some bottlenecks for use as a read-write file system. Evident from Figures 1 and 5 is the fact that there is tight mapping between filename and file handle. Thus anytime the file data changes, the associated handle also changes and this triggers a series of cascading updates: When a file is modified, the handles of the data blocks change. The inode meta-data structures for the file have to be updated to reflect this change and keep the system consistent. Thus, the directory entry of the parent directory of the file is modified to hold the new hash value for the inode. This in turn causes the directory block's hash to change, leading to an update of its parent directory block. This continues and cascades all the way to the root directory (/). The cascading halts when we get a new root hash for the file system. Such cascading updates makes writes inefficient. The effect of cascading updates is proportional to the depth of the write in the file system hierarchy. The deeper the write, the larger the number of updates.

Thus we propose certain changes to the file system structures as a first step in realizing a read-write version of SFSRO (SFSROL). Note that SFSROL is not the same as SFS [3]. Some concepts, like '*self-certifying pathnames*', are similar to SFS, but the design of file access control protocol is different. SFS performs key negotiation, user authentication and cryptographically enforces all file access control [3]. SFSROL doesn't use any of these methods but follows SFSRO's approach in preserving data integrity. Sections 2 and 5 provide information on the security guarantees of SFSRO and SFSROL respectively. Certain ideas, like '*floating inode table*', are similar to the work in Venti File System (VFS) [4]. The design of the file system writing access control protocol is closely similar to VFS's design ideas. VFS uses immutable writes and maintains previous file system images (data blocks) to provide a consistent file system image to the client.

4. SFSRO Lite

This tight coupling between file name and file handle can be removed by using an extra level of indirection. A new meta-data structure called '*floating inode table*' stores inode numbers to handle hashes. Each file name is associated with an inode number and the inode number to handle mapping is maintained separately in this table. In this way any changes to the file system are isolated to the inode table. Resolution of a file name to a file handle uses the floating inode table. When the inode for a file is requested, the associated inode number is used as an index into the inode table and retrieve the handle (hash) of the inode. This handle is then used to retrieve the inode. The handle of the inode table fully specifies the file system. Figure 6 shows the inode table indirection mechanism.

The indirection eliminates the cascading updates problem. Now when a file is modified, its handle changes and the inode table is updated so that the inode number for that file reflects the new handle. A new inode table handle is computed and used for further accesses to the file system. This completes all the writes and updates required.

The directory entries are unaffected as they hold inode numbers that do not change. The number of updates is less than in the previous scheme and completely eliminates cascading updates.

#Files	Max. #Meta-data Updates
< 7*256	2
< (7+256)*256	3
< (7+256+256*256)*256	4
< (7+256+256*256+256*256*256)*256	5

Table 1: Maximum Inode Table Updates required for a given file system size

In our implementation, we treat the inode table as a normal file and its data blocks as indirect SFS inode entries. By following SFS design in which each indirect SFS inode entry can have a maximum of 256 handles, each inode table block has 256 entries. Thus, each inode table block occupies 5KB (256*20B). The maximum number of inode table (meta-data) updates required for a given file system size is shown in Table 1. If the number of files is greater than 7 * 256, then the inode table blocks will be recursively hashed. Usually the maximum depth of such an indirection is 3-4. In the previous case, the indirection was dependent on the depth of the file in the file system tree. Table 2 shows the expected space required for the inode table.

#Files	Max. Space Required for inode Table
< 7*256	7 * 5KB (35KB)
< (7+256)*256	(7+256+1) * 5KB (~1.3MB)
< (7+256+256*256)*256	(7+256+256*256+1+256) * 5KB (~323MB)
< (7+256+256*256+256*256*256)*256	(7+256+256*256+256*256*256+1+256+256*256) * 5KB (~81GB)

Table 2: Maximum Inode Table Size required for a given file system size

As a additional benefit, the circular dependency issue for "." and ".." is resolved. Thus, UNIX file system semantics are maintained as now "." and ".." can also be stored in the database. Further, multiple versions of a file system can co-exist by maintaining the previous inode tables [4]. Other benefits of SFSRO are inherited.

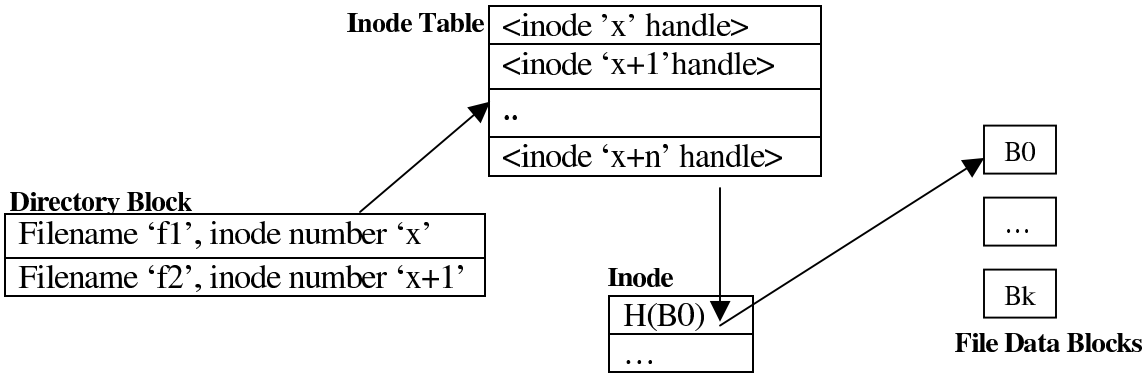

Figure 6: Inode Table Indirection

5. Design Modifications

The necessary data structure changes are shown in Figures 7,8 and 9. The 'imap' structure (Figure 7) implements the floating inode table. This table is a normal file that doesn't appear in the file system namespace. The data blocks of this imap file contain a list of handles. The index of the handle is the inode number associated with the handle. The directory entry (Figure 8) now contains an inode number (shown in bold) rather than a handle (sfs_hash). Figure 9 shows the new FSINFO structure. In addition to the normal fields, FSINFO now contains the handle for the imap file inode – 'imapfh', the index (inode number) of the root inode – 'rootinno'. 'imapfh' and 'rootinno' are used to retrieve the root inode handle (rootfh).

```
struct imap_entry {
  sfs_hash fh;
};

struct imap {
  imap_entry
entries<SFSRO_NFH>;
};
```

Figure 7: SFSROL IMAP Structure

```
struct sfsro_dirent {
  unsigned inode_num;
  string name<>;
  sfsro_dirent *nextentry;
};

struct sfsro_directory {
  sfsro_dirent *entries;
  bool eof;
};
```

Figure 8: SFSROL Directory Structure

```
struct FSINFO {
  sfs_time start;
  unsigned duration;
  opaque iv[SFSRO_IVSIZE];
  sfs_hash fhdb;

  sfs_hash imapfh;
  uint32 rootinno;
};
```

Figure 9: SFSROL FSINFO Structure

Security Guarantees

The hash of the imap file verifies the integrity of the whole file system. Changing a single imap block causes the hash of the inode file to change and hence the FSINFO changes (Figure 9). Thus, as long as the FSINFO signature and hashes are correct, the data integrity is preserved. Correctness of FSINFO signature implies that integrity of the whole imap file is preserved, which in turn implies that the integrity of the whole file system structure is preserved. Correctness of data (and meta-data) block hashes implies that the integrity of block data is preserved. Thus, SFSROL has the same security guarantees as SFSRO. The difference is that changing a single block in SFSROL file system changes the hashes of only that file's inode and the imap file, whereas in SFSRO it changes the hash of every directory up to the root.

6. Implementation

SFSRO implementation has been modified to integrate the new changes into 'sfsrodb' and 'sfsrocd'. The rest of the system is unmodified.

6.1 sfsrodb

'sfsrodb' implementation has been modified to use the new data structures. It walks down the given file system and builds a database with the file system data indexed by the cryptographic hash (handle) of the data. When it encounters a new directory, it passes its inode number ("..." for child) and the next inode number to be used ("." for child) to the main routine that creates the database. This main routine calls

itself recursively for creating the database blocks of files. 'sfsrodb' takes care to use the name inode number for hard links in the file system. As imap block space is exhausted (ie. maximum inode number entries in an imap block), it is stored in the database and the imap file inode is updated. After the given file system has been completely dumped into the database, it dumps the imap inode too and then creates a FSINFO structure, signs it with the private key and stores that too in the database.

6.2 sfsrocd

'*sfsrocd*' implementation was also modified to use the new data structures. The client daemon requires changes in two places – one at interpreting the result of getfsinfo() call and the other at fetching the inode corresponding to a handle. The File System data structure (*filesys*) has been modified to incorporate two additional members – the imap file inode (*imapinode*) and the imap data block cache (*imapcache*).

In addition to the existing *sfsrocd* caches, there is a separate imap data block cache because different keys are used to index these cache sets. Existing *sfsrocd* caches use <hash, data block> as key-value pair whereas *imapcache* uses <inode-number, hash> as key-value pair. During file system calls the hash is used to retrieve a block, but during inode resolution the inode number is used to retrieve the appropriate imap block and imap entry. During inode number (say 'N') to hash (say 'H') resolution, 'N' provides the index (say 'i'='N'/256) of the appropriate imap block to be fetched. The hash 'K' at the position 'i' in the imap inode data block pointers is used to retrieve the imap block (say 'B'). The block 'B' retrieved using this hash 'K' may hit in *sfsrocd* caches, if it were read previously. If 'B' is an imap data block, then 'N'%256 is used as index to retrieve the appropriate imap entry and the hash 'H'. If 'B' is an indirect imap block, then this process is repeated recursively to obtain 'H'.

The current implementation uses an imap cache size of one imap block (256 imap entries). Any first time access to an imap block will miss in imap cache and *sfsrocd* caches. Any access to a previously read imap block will miss in imap cache (unless the block already in the imap cache is being retrieved), but may hit in *sfsrocd* caches. Thus, for ease of implementation we use one imap block as imap cache size.

getfsinfo() – the FSINFO obtained as the result of this call is used for further setup before control returns. FSINFO.imapfh is used to retrieve the imap inode for the imap file, which is then used to retrieve the root handle (*rootfh*). These are stored in the file system (*filesys*) structure.

For SFSROL, inode number to inode resolution replaces SFSRO's handle to inode resolution. Thus, a given inode number is first used to identify the imap data block that contains the required inode handle. If this block is not in the file system imap cache (filesys.imapcache), then it is fetched from the server and cached. The inode number is used to index this block to retrieve the required inode handle. Further processing proceeds as in SFSRO.

6.3 dbwalk

'*dbwalk*' is a standalone program that walks the whole database starting from FSINFO. *dbwalk* performs the same kind of operations that *sfsrocd* performs in interpreting FSINFO. In the case of SFSRO, it uses the handle from the directory entry directly to fetch the inode corresponding to the handle. In the case of SFSROL it has an

intermediate step of resolving the inode number to a handle before fetching the inode. This program measures the time taken to walk the whole file system tree. Using this program, the walk times of SFSRO and SFSROL are compared.

The *dbwalk* program follows the same sequence of events that occurs when a client reads a file from SFSRO or SFSROL. Each retrieval of FSINFO from the database is followed by signature verfication and each retrieval of data (and meta-data) blocks is followed by hash verfication. Figure 10 depicts the flow of the *dbwalk* program. The paths taken by SFSRO and SFSROL have been labeled where necessary.

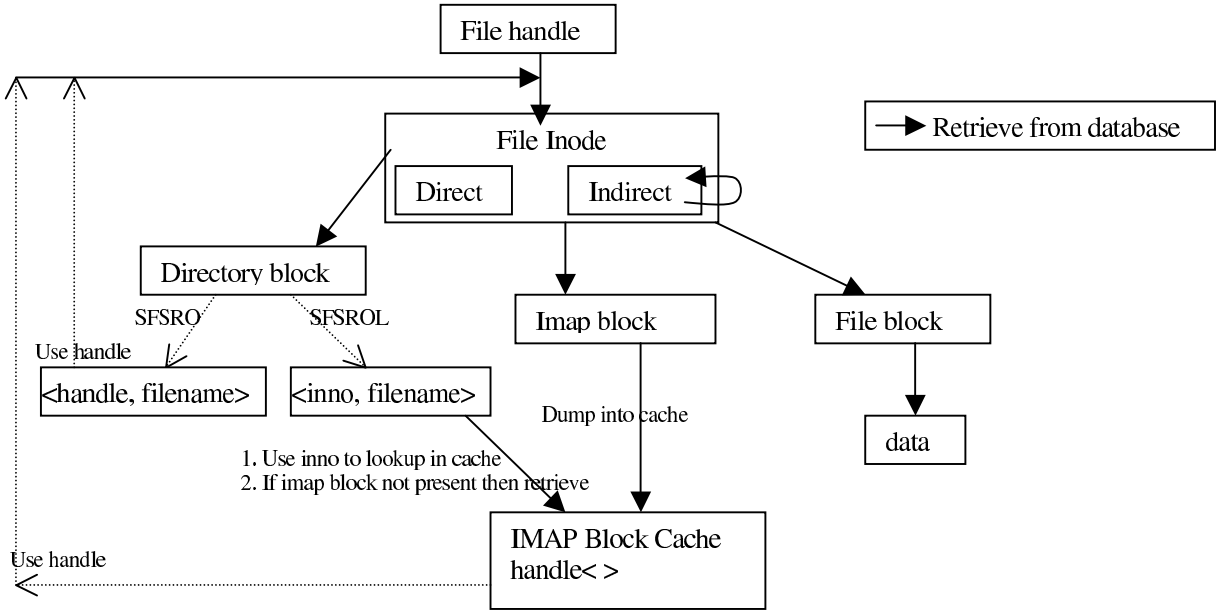

Fig 10: dbwalk program

For SFSRO, the *dbwalk* program functions as follows:

0. The FSINFO is retrieved from the database.
1. The root's handle is obtained from FSINFO.
2. The inode corresponding to the handle is retrieved from the database.
3. If it's an inode for a regular file, then for every direct and indirect entry retrieve a file data block from the database.
4. If it's an inode for directory, then for every direct and indirect entry retrieve the directory block data. This directory data block will contain <handle, filename> mappings. These handles are used to repeat step 2.

For SFSROL, the *dbwalk* program functions as follows:

0. The FSINFO is retrieved from the database.
1. The imap inode handle obtained from FSINFO is used to retrieve the imap inode. Using the information available from this imap inode, the appropriate inode table block corresponding to 'rootinno' (root directory's inode number) is retrieved (and cached). The root handle is obtained by indexing this imap block using '*rootinno*'.
2. The inode corresponding to the handle is retrieved from the database.
3. If it's an inode for a regular file, then for every direct and indirect entry retrieve a file data block from the database.

4. If it's an inode for the imap file, then for every direct and indirect entry retrieve an inode table block. If a handle for an inode number is requested, store the required block into the imap entry cache.
5. If it's an inode for a directory, then for every direct and indirect entry retrieve a directory block data. The directory data block contains <inode number, filename> mappings. The inode number is used to index the required imap block in the cache to obtain the handle. (If an imap block misses, the imap inode is used to retrieve the required one into the cache). These handles (except for "." and "..") are used to repeat step 2.

7. Performance Evaluation

This section presents the results of measurements of performance comparison between SFSRO and SFSROL to support that claim that SFSROL is 0-2% slower than SFSRO. We measure the performance of small and large file benchmarks for both. We also measure the performance of both to do a complete file system walk (*dbwalk*).

7.1 Experimental Setup

We performed the experiments on two machines –
1. sndh : 550x2 MHz Pentium III (Dual P3), 512 KBx2 cache, 512 MB RAM, 20GB HDD.
2. chung : 1.7GHz Pentium 4, 256 KB Cache, 512 MB RAM, 8GB HDD

The file system block size for both machines was 4KB. 'dbwalk' experiments were performed on sndh in standalone mode. Small and large file benchmarks were performed with chung as server and sndh as client. In sfsrocd, the imap cache has a maximum of 256 entries (1 imap block). The remaining cache sizes are as in SFSRO. For all experiments we report the average of ten runs. The Small File Benchmark tests consist of 1K, 4K and 8K test directories. Large File Benchmark test consists of a 54M test directory. The tests are named after the size of a single file in that directory. Each of the 1K, 4K and 8K test directories contain 100 sub-directories with 100 files in each of first 10 directories (remaining 90 are empty directories) – i.e., 100 directories and 1000 files, with each file of 1K, 4K and 8K bytes in size respectively. All files contain random data. For 54M tests, there is a single file of size 54MB in the test directory. For 1K and 4K test suites, 1KB and 4KB blocks are read respectively. For both 8K and 54M tests, 8KB blocks are read in.

7.2 Database Walk Experiment

For the 'dbwalk' experiment, the database size of SFSROL is greater than the database size for SFSRO because for SFSROL even empty directories are stored. This experiment walks the entire the file system tree using only the database. For the 1K, 4K and 8K tests there were 1101 inode number to handle resolves (1 root, 100 directories, 1000 files). For the 54M test there were 2 resolves (1 root, 1 file). Table 3 shows the performance measurement comparison of SFSRO and SFSROL using the dbwalk experiment. The resolve column shows the extra overhead incurred due to inode number to handle resolution. A resolve happens for every file (file or directory) in the file system. When resolve is called, the local imap cache is first checked to find out if the correct imap block is cached. If not then the appropriate imap block is fetched from the

database. The inode number is used to index that block and retrieve the handle for the inode. The last column shows the percentage slowdown of SFSROL with respect to SFSRO. It is calculated as ((SFSROL-SFSRO)/SFSRO). This shows that SFSROL is 0-2% slower than SFSRO.

Test Size (in bytes)	SFSRO (Walk time in secs)	SFSROL		SFSROL vs SFSRO %
		(Walk time in secs)	Total resolve overhead (in usecs)	
1K	0.3099551	0.3143673	4348	1.4
4K	0.5350773	0.5396902	4540	0.9
8K	0.8169375	0.8214546	4456	0.6
54M	0.8913501	0.8913664	140	0.002

Table 3: Walk Bencmark Results

7.3 Microbenchmarks – Small and Large File Benchmarks

For these benchmarks, the 90 empty directories were deleted before the database was formed. The Small File Benchmark for 'x'K tests consists of sequentially reading 1000 files of 'x'KB. The Large File Benchmark consists of sequentially reading 8KB blocks of a large 54MB file.

For the 1K, 4K and 8K tests there were 1011 resolves (1 root, 10 directories, 1000 files). For the 54M test there were 2 resolves (1 root, 1 file). Table 4 shows the performance comparison of SFSRO and SFSROL using the small and large file benchmarks. The resolve column shows the extra overhead incurred due to inode number to handle resolution. A resolve happens based on the upper layer calls (like lookup etc.) to SFSROL. Resolve works in the same as explained above in 'dbwalk' experiment. The last column shows the percentage slowdown of SFSROL with respect to SFSRO. This shows that SFSROL is 0-1% slower than SFSRO.

Test Size (in bytes)	SFSRO (Benchmark read time in secs)	SFSROL		SFSROL vs SFSRO %
		(Benchmark read time in secs)	Total resolve overhead (in msecs)	
1K	3.60	3.63	24	0.8
4K	4.40	4.42	25	0.5
8K	6.16	6.19	24	0.3
54M	24.69	24.7	2	0.04

Table 4: Micro Bencmark Results

In the current implementation, the imap cache has the capacity to hold just one imap block (256 inode number to handle entries). The imap cache size can be increased, to

say 256 blocks. The imap block corresponding to the root inode number is fetched during the initial imap cache miss. Prefetching some imap blocks into the imap cache as soon as the imap inode is obtained, assuming cache hits occur, will remove most of the delays incurred due to imap cache misses. Thus, by using better imap caching mechanisms SFSROL performance can be made even closer to SFSRO.

8. Future Work

Modifying SFSRO to incorporate an inode table is the first step in making SFSRO read-write. Further modifications necessary to support writing are in progress.

RW file system: Normal read operations proceed as before. For write operation new means of writing data (while maintaining authenticity of the user and integrity of the data) has to be devised. We can imagine a Venti style server that never deletes blocks [4]. Thus, data integrity is not an issue since there is no overwriting or deletion of blocks. By maintaining a list of FSINFOs, the client can decide which file system image it wants to use based on whether or not it trusts the file system key.

Variable size blocks: Incorporate variable sized block to exploit cross-file similarities and reduce traffic flow between client and server.

9. Conclusions

As a first step towards making a read-write version of SFSRO, we have embedded a floating inode table into SFSRO. From the performance experiments, we can conclude that this addition doesn't cause significant performance degradation. The flexibility achieved at a slight slowdown is of great significance when we consider the benefits offered by the read-write version.

References

[1] "Venti: a new approach to archival storage", Sean Quinlan and Sean Dorward, *Bell Labs, Lucent Technologies*

[2] Fu, K., Kaashoek, M. F., And Mazieres, D. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation* (Oct. 2000), pp. 181–196.

[3] David Mazi`eres, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawah Island, SC, 1999. ACM.

[4] Venti FS: A Hash Based File System. Siva Kollipara and Srinivasan Badrinarayanan, Department of Computer Science, University of Arizona

[5] Rosenblum, M., and Ousterhout, J. ``The Design and Implementation of a Log-Structured Filesystem." *ACM Transactions on Computer Systems*, *10*(1), February 1992, pp. 26-52.