TR03-16

May, 2002

Venti FS: A Hash Based File System

Siva Kollipara, Srinivasan Badrinarayanan

Supervisor: Dr. John Hartman

Venti FS: A Hash Based File System

Siva Kollipara, Srinivasan Badrinarayanan
{ksskumar,srinivas}@cs.arizona.edu
Department of Computer Science
The University of Arizona

Abstract

Venti, a network storage system, is a write-once archival repository. It provides a block level interface and uses unique hashes (SHA1 hash) to identify these blocks. These hashes can be used to identify duplicate blocks and thus eliminate redundant duplication of blocks and reduce storage consumption. A direct mapping between hashes and disk addresses is not possible, therefore an indexer is used to map the hashes to Venti disk addresses. Venti can be used as a building block for constructing a variety of storage applications such as logical backup, physical backup and snapshot file systems.

Most file systems to date use fixed size blocks. Fixed size blocks limit the amount of duplication possible. However by breaking files into variable sized blocks based on the identification of anchor or break points, duplication can be increased and cross file similarities can be exploited efficiently.

We have built a file system on top of Venti and investigated the implications of several design decisions:

1. *Use of different searching algorithms (B – Tree / Binary Tree) in indexer;*
2. *Variable Sized blocks instead of fixed size blocks.*

We present some performance results that measure the average execution time for various operations of the file system and also analyze the impact of the design decisions made.

Key Words: Venti, File System, Rabin Fingerprint, SHA 1 Hash, B – Tree, Binary Tree

1. Introduction

Applications rely on file systems to store data on and retrieve data from mass storage devices. File systems provide the underlying support that applications need to create and access files and directories on the individual volumes associated with the devices.

A file system is a hierarchical structure (file tree) of files and directories. This file tree uses directories to organize data and programs into groups, allowing the management of several directories and files at one time. The file system consists of one or more drivers and supporting dynamic-link libraries that define the data formats and features of the file system. These determine the conventions used for file names, the level of security and recoverability available, and the general performance of operations.

Most file systems to date use fixed size blocks to store and retrieve data from device. Fixed size blocks limit the amount of duplication possible. However by breaking files into variable sized blocks based on the identification of anchor or break points, duplication can be increased and cross file similarities can be exploited efficiently.

Venti [1] is a block level storage system that can be used as a building block for constructing a variety of storage applications such as logical backup, physical backup and file systems. It is a write once storage that prohibits data to be deleted or modified once it has been written to Venti. A unique hash (SHA1 hash), of the contents of the block, identifies the blocks in Venti. These hashes are also used as the address of the block on the storage device. The hash size being large, a direct mapping between hashes and disk addresses is not possible, therefore an indexer is used to map the hashes to Venti disk addresses. The indexer in [1] uses a binary search tree algorithm to search through mappings.

This paper describes the design and implementation of Venti FS – A Hash Based File System. The goal of Venti FS is to develop a filesystem using Venti that achieves a reasonable performance. The file system efficiency is improved by using 1) B – Tree search [6] in the Venti indexer, 2) variable sized blocks that reduce the block usage and help exploit cross – file similarities.

The remainder of the paper is organized as follows. In section 2 we look at the motivations behind the design decisions made. Section 3 presents the File System issues that arise because of using Venti as the storage device. The File System Organization and the File System components are discussed in Section 4 followed by the File System prototype in Section 5. Section 6 highlights the performance measurements. Section 7 talks about related work. Section 8 lists areas of Future work. In section 9 we present our conclusion.

2. Design Motivations

Unlike a binary-tree, each node of a b-tree may have a variable number of keys and children.
Since each node tends to have a large branching factor (a large number of children), it is typically necessary to traverse relatively few nodes before locating the desired key. If access to each node requires a disk access, then a b-tree will minimize the number of disk accesses required. The minimization factor is usually chosen so that the total size of each node corresponds to a multiple of the block size of the underlying storage device. This choice simplifies and optimizes disk access. Consequently, a b-tree is an ideal data structure for situations where all data cannot reside in primary storage and accesses to secondary storage are comparatively time consuming (or expensive).

When file blocks are identified by file contents, rather than on position within a file, cross – file similarities can be better exploited. Using variable sized blocks, where the block boundaries are identified by identification of anchor or break points, avoids the problem of sensitivity to shifting file offsets. This decreases update latency as now insertions and deletions affect only the neighboring blocks. Similar techniques has been successfully used in LBFS [3] to save file bandwidth .

Such a scheme, therefore, improves occurrence of duplicate blocks, because updates change fewer blocks, and makes efficient usage of the disk. Though fixed size blocks can also exploit similarities, the scope is very limited. The block consolidation is poor and not as much when variable size chunks are used.

Based on these observations, we proposed to use a B – Tree search algorithm in the Venti indexer and variable sized blocks. This can significantly improve the efficiency of the indexer lookup and reduce the disk usage.

3. The Venti Disk and File System Issues

The Venti Storage has several unique features different from conventional disks. This could pose potential problems to the File System design and implementation. It is, therefore, necessary to understand and analyze the various file system issues that could arise as a result of the new archival storage system.

3.1 Block Address and Block Contents

In Venti a unique hash of the block contents identifies the blocks. This hash is also used to address the block. This tightly couples the block address with the block contents unlike traditional disks where the block address is independent of the contents of the block. This unique feature of Venti is a major concern when designing a file system as it leads to other related issues.

3.2 Immutable writes

The tight coupling between the block contents and address makes in place updates of data impossible. When a content of a block changes, its hash value changes and therefore the address where the block is stored also changes. Thus one cannot modify a block without changing its address. So writes are immutable. This means that once a block is written to the storage it cannot be changed. This is an issue because now

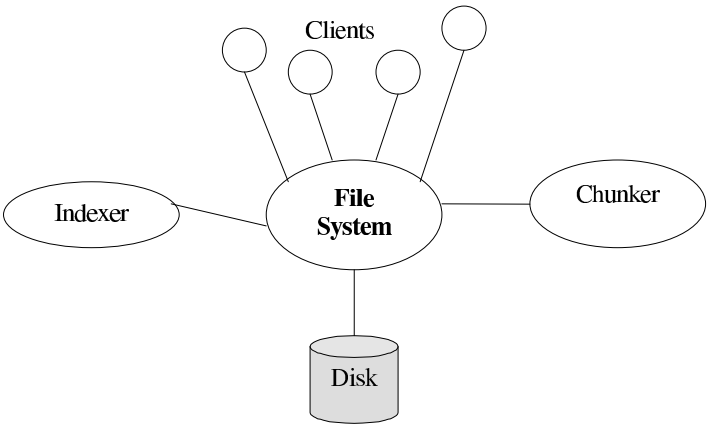

Figure 1: File System Organization

to modify a block a new copy of the block with the desired changes has to be written to a new address. As a consequence:

3.3 Old Data Preserved

The old data is not lost and can be recovered or used if we retain the address of the previous version of the block.

3.4 Floating Inode Table

An Inode Table is stored on the Venti Storage and when the table is updated it has to be stored at a new address. The Venti File System, hence, has a floating Inode Table. The Inode Table is an implementation detail and will not be an issue if it is not a part of the system.

4. File system Organization

The File System is composed of five main components as shown in figure 1.

4.1 The Client

The Client is a simple shell that accepts commands from the user. The commands are parsed and the request is mapped to a File System interface call.

4.2 File System

The File System has 2 layers.

4.2.1 The File System Interface

This provides an API to the File System calls supported. It is this File System Interface that is exported to the clients and used by the clients.

4.2.2 File System Core

The File System Core is the crux of the entire system. It implements all the file system functionalities and integrates and interacts with the various components.

4.3 Indexer

The indexer maps hashes to disk addresses. The indexer exists as a separate entity and is not a part of the Venti Server as in the prototype. The reason behind this is to make the Indexer independent of Venti. De coupling the indexer from Venti gives it more flexibility. If the indexer is a part of Venti then any new indexing schemes devised should also make use of the Venti interface. This could pose a limitation on the different potential indexing schemes.

4.4 Chunker

The Chunker is responsible to divide the file into variable sized blocks. The Chunker uses the Rabin Fingerprint [4] to determine the block boundaries. A Rabin fingerprint is the polynomial representation of the data modulo a predetermined irreducible polynomial. The block boundaries are identified using Rabin by computing the fingerprint on every overlapping 48 bytes of the file. If the low – order 13 bits of the fingerprint equals a pre determined constant value, then that identifies the end of a block.
The Rabin Fingerprint is efficient to compute on a moving window in a file. There is however a maximum and minimum block size of 8K and 2K respectively imposed by the file system. This is to keep the Rabin Algorithm to be within bounds and eliminate the possibility of pathological cases.

4.5 The Venti Server

Simulates the storage device. It stores the file system data and metadata. A simple API is used to interact with the disk.

5. File System Prototype

The design of the Venti File System is similar to the Unix File System. The file system is organised as a hierarchy of directories starting from a single directory called *root* which is represented by a **/** (slash). Immediately below the root directory are several system directories

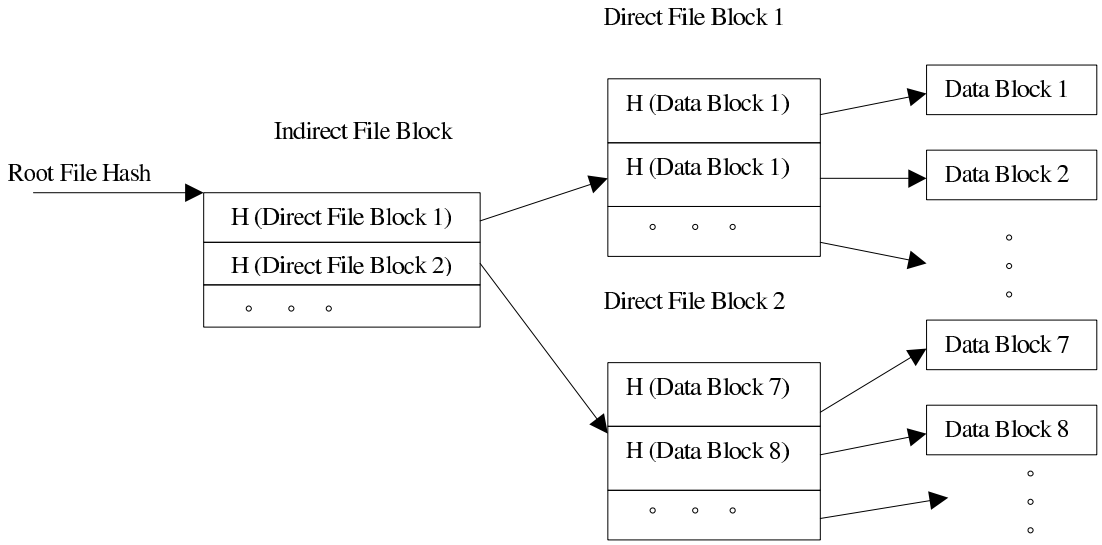

Figure 2: Recursive Hashing of Data Blocks

that contain information required by the operating system. Like Unix a directory is also a file that holds other files and directories.

From the perspective of the Venti server, all the files are represented and stored as blocks. The files are divided into blocks and the blocks are stored in the disk. A Hash of the Block Contents identifies these blocks. A file can consist of many blocks and to enable data to be retrieved from all blocks of the file, the file system must store the block hashes. These hashes are stored in additional blocks that are also written to the Venti server. This process is repeated recursively until a single hash is obtained. This hash represents the root of a tree of blocks and corresponds to a root file hash of the file. This is shown in figure 2.

Once blocks are written they are immutable and cannot be changed. To update a block, therefore, a new block is created with the changes and the pointer to the old block is replaced with a pointer to the new block.

5.1 Venti Block Types

All blocks consist of a header and the data. The header information has block type information that is used to distinguish between the various different blocks that are present. There are 4 types of blocks.

5.1.1 The Inode Table

The Inode Table is the entry point to the file system. It is a list of <inode no, hash> tuples. The hash in the tuple is the root file hash of the file. The inode table uses the inode number to retrieve the root file hash for a file. Inode number zero always represents the hash of the root directory (/) of the file system. The hash of the inode table block is stored at a fixed location from where it can be retrieved at startup.

5.1.2 Directory Block

The directory block stores the mapping between logical file names to inode numbers.

5.1.3 File Block

The file block should not be confused with a data block. This contains a list of data block entries. The entries in the file block are represented by <hash, offset, size>. The hash is the hash of the data block. The offset gives the starting offset of the data in the file and the size is the number of bytes of data in the data block.

File Blocks have 2 sub types – *Direct* and *Indirect*. The Direct File Block has the list of block hashes for the file. If all the data block entries cannot be placed in a single file block then more blocks are used and a single hash for the file blocks is obtained by recursive hashing. The intermediate hashes of the file blocks are stored in Indirect File blocks. These types of File Block store the hash values of the direct file blocks.

5.1.4 Data Block

The Data block has the actual file data corresponding to one block of the file.

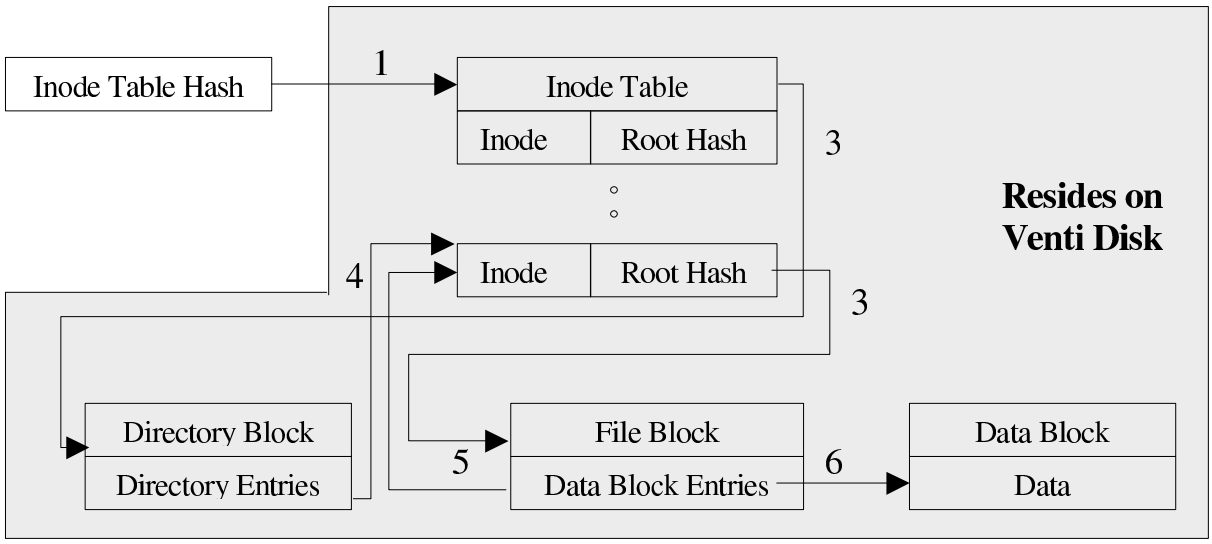

Figure 3: Path Resolution/Data Blocks Retrieval – Schematic Representation

Any file system operation starts with the Inode Table Hash. Path names are resolved one component of the path at a time. Given a path, it has to be resolved to a hash. This hash is the hash of the last component of the path, if such a path exists.

5.2 Path Resolution/Data Blocks Retrieval

1. Retrieve the inode table block, using the Inode Table Hash.
2. Fetch the hash value for the given inode number. (For the root directory inode number is 0, so the first time default inode number of 0 is used.)
3. Fetch Block from Venti Server for the given hash.
4. If Block is a Directory Block:
    a. Search the next component in the path through the entries of the directory block.
    b. If the file name doesn't exist then path is invalid. If an entry is found, then the corresponding inode number is returned.
    c. Return to step 2
5. If Block is an indirect File Block:
    a. For every hash value fetch the Direct File Blocks → i.e. loop back to step 3
6. If Block is a direct File Block
    a. Use offset, size values in each File Block entry to find out the data blocks required.
    b. For every hash required fetch the data blocks → i.e. loop back to step 3
7. If Block is Data Block
    a. Return contents of the data

A schematic representation of the same is shown in figure 3.

5.3 Inode Table Indirection

The directory block entries store inode numbers. This is an extra level of indirection because what is required is a mapping from a file name to its hash. In the prototype, however, a file name maps to an inode number and the inode number is used to obtain the file hash. The straightforward approach is to store the hash values of the file names instead of the inode numbers. This will remove the extra level of indirection, but presents many drawbacks and inefficiencies. The Inode table is currently stored as a single block; instead it can be stored as a normal venti file. The benefits of saving space, faster processing etc., would then apply here too.

	Virgin Writes	Sequential Reads	Duplicate Writes
Binary Tree without Rabin	2.5	957	10
Binary Tree with Rabin	2.53	796.2	5
B Tree Without Rabin	2.67	1278.5	9.4
B Tree with Rabin	3.8	669.77	5.3

Table 1: The performance of read/write operations in Kbytes/sec

5.3.1 Cascading Updates

When directory entries have a mapping of file names to hash values it leads to cascading updates: When a file is modified, its hash and address changes. The meta – data structures have to be updated to reflect this change and keep the system consistent. The directory entry of the parent directory of the file is modified to hold the new hash value. 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 Indirection eliminates and resolves this problem. Now when a file is modified, its hash value changes and the inode table is modified so that the inode number to file hash mapping is changed. An inode table hash is now computed because the inode table has been updated. This hash is then 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 very less compared to the previous scheme and completely eliminates cascading updates. There is a possibility of inode table blocks being recursively hashed. Calculations, however, show that the maximum depth of such an indirection is limited to 2 or 3. In the previous case, the indirection was dependent on the depth of the file in the file system tree.

5.3.2 The "." And ".." Problem

A traditional drawback in using hash based schemes for file systems is the "." and ".." problem. In any traditional file system like UNIX, when a directory is created it has two default entries: "." that refers to the current directory and ".." that points to the parent directory. If the directory entry block has mappings between filenames to hashes it makes the implementation of "." and ".." impossible. This is because, the directory entry for "." should be a mapping from "." to the directory's hash value and ".." to the parent directory's hash. Since hashes depend on block contents, it is impossible to know the hash values before the relevant entries are made in the directory. Thus, "." and ".." implementation is a problem.

In our system, the directory entries now have inode numbers. The directory entries for "." and ".." ,now, maps to an inode number. Since directory blocks and inode numbers don't change when files are updated, the problem is resolved.

6. Performance

Table 1 gives the preliminary performance results for the read and writes operations in various situations. Though the performance is poor compared to any other system. It is justified and acceptable by the fact that the measurements were done on a simulation environment on top of the Unix File System and no caching.

The results shown are not for a comparison of how our file system performs against other file systems. It just highlights the performance relative to the various design alternatives. It is seen that when variable sized blocks are used the latency increases because of the time taken to break the file into variable chunks. When B – Trees are used instead of Binary Trees in the Indexer the performance increases as expected. For virgin writes the entire data along with the meta – data is written to the disk. For duplicate writes, there is no write to the disk since the data already exists. Thus the latency for virgin writes is higher than duplicate writes. The duplicate writes have higher bandwidth than virgin writes. Table 2 gives you the block usage when Variable sized blocks and fixed size blocks are

used. As expected the chunking reduces number of blocks used.

The performance of indexer lookups is shown in the Table 3. The B – Tree indexing scheme is more efficient than the traditional Venti approach of using Binary Trees. The performance however deteriorates when Variable Sized blocks are used. This is because, now, the Chunking module takes more time to break the file into variable sized blocks.

The B – Tree used in the experiments has an order 2 (a branching factor of 5). This could be the reason why the performance of Binary Tree is close to that of B – Tree. Increasing the order of the B – Trees, can increase the performance gap between the Binary and B – Trees.

The experiments conducted just analyze the possibility of such a system. The experiments show that though Rabin Fingerprinting is a good idea of reducing block usage, it contributes to latency. This extra latency reduces the performance of the Indexer and the system in general. So there should be a judicious balance between the use of variable and fixed sized blocks. This requires in depth analysis and further experimentation.

7. Related Work

The Venti storage system uses hashes to identify blocks. The indexer used maps the hashes to addresses. The indexer is divided into buckets. Buckets contain a list of hashes. Given a hash, a binary search tree is used to search through a bucket. The Venti Indexer in our current implementation makes the indexing scheme more efficient by using a B – Tree instead of the Binary Tree approach.

The Plan 9 File System uses Venti as the storage device. The Venti Server comprising the indexer, and Venti Disk is used to store the permanent data while the magnetic disk is used as a cache to speed up operations. The Plan 9 File System has influenced the Venti File System design. Our system, however, improves on the performance and design by making some clever design choices.

The Read Only Secure File System (SFSRO) consists of a database that is a read only server and can be accessed by many clients securely. It uses the SHA1 hash to identify and retrieve blocks from the server that is similar to the Venti FS block identification scheme. In this it recursively hashes on the contents of the data to build a complex structure. SFSRO provides strong data integrity between the client and

	Without Rabin		With Rabin	
40k Read/Write Average per operation in usec		GetINode	UpdateIndex	
	Binary Tree	255.43	18592.674	
	B Tree	160.85	8242.734	
		GetINode	UpdateIndex	
	Binary Tree	144.198	8865.834	
	B Tree	318.2	17649	
80k Read/Write Average per operation in usec		GetINode	UpdateIndex	
	Binary Tree	175.1	14546.45	
	B Tree	114.1	10946.57	
		GetINode	UpdateIndex	
	Binary Tree	143.7	12880.624	
	B Tree	141.8	13311.958	
151k Read/Write Average per operation in usec		GetINode	UpdateIndex	
	Binary Tree	35.2	4591.536	
	B Tree	40.4	6712.941	
		GetINode	UpdateIndex	
	Binary Tree	57.4	8487.62	
	B Tree	52.6	8920.95	

Table 3. Indexer Lookup Performance

servers. The database is digitally signed and access is granted to the system through a secure public key. Our system emulates a SFS – Read Write File System.

LBFS is a network file system designed for low bandwidth networks. It exploits cross-file similarities and takes advantage of the fact that the same chunks of data often appear in multiple files or multiple versions of the same file. This makes LBFS work well over low bandwidth. Venti FS exploits similar cross-file similarities to overcome the limitation posed by fixed size data blocks and help reduce the block usage on the storage system.

8. Future Work

The Venti File System developed is very primitive and there is scope of further research.

The Venti[1] prototype uses a Binary Tree Indexer with Fixed Size Blocks. This is the first of the four possibilities that are possible (refer Table 1). Based on system attributes, like system usage, load, age etc, the system can make a clever choice of which configuration to use and shift to that configuration dynamically.

The Global File System [7] uses stuffed dinodes, which allows both file system information and real data to be included in the dinode file system block. As an offshoot from this approach, stuffed fingerprints could be used in which the hash value and the file data could be stored in the File Block. This would reduce the number of updates and lookups.

In the current prototype, when the Inode Table Hash changes the old hash is lost. If, however, we store the old inode table hashes along with an associated timestamp indicating when the hash was valid, it could be used to keep track of earlier images of the File System. This makes the implementation of snapshots and a versioning file system very easy. The old inode table hashes and the timestamp can be stored on the local disk using the local file system. So the Disk Based File System coupled with Venti File System could provide a strong extensive File System backbone.

The current Venti indexing scheme has been improved by using a B – Tree. In the future if other faster indexing schemes are designed, these can be plugged into the system. Potential systems that could provide indexing functionalities include database servers, token servers, DNS.

9. Conclusion

We have proposed a new file system based on the Venti archival storage system. The design decisions made of using a B – Tree based Venti Indexer and having variable sized blocks improved the performance of the file system. The File system incorporates ideas from the Log Structure File System[9].

Several unique features of the Venti File System include the Inode Table Indirection – that reduces the file update latency and eliminates the "." And ".." problem. Keeping track of earlier root hashes along with a timestamp can provide a consistent snapshot of the file system. The system is very similar to the secure read only file system except that it has also now write capability.

A prototype version of the file system has been developed and work is under progress to improve the performance and also to incorporate many more features that are currently not supported.

Acknowledgements

We wish to thank Dr. John Hartman for his valuable feedback and comments. We also thank the faculty in the Computer Science Department at The University of Arizona and friends for their help and support.

References

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

[2] "Finding similar files in a large file system", Udi Manber

[3] "A low-bandwidth network file system", Athicha Muthitacharoen, Benjie Chen, and David Mazières

[4] “Fingerprinting by random polynomials”, Michael O. Rabin
[5] Cormen, Leiserson, and Rivest
[6] http://www.public.asu.edu/~peterjn/btree
[7] Kenneth W. Preslan et al. A 64-bit, shared disk file system for linux. *IEEE Symposium on Mass Storage Systems*, pages 22–41, San Diego, CA, March 1999.
[8] FU, K., KAASHOEK, M. F., AND MAZI `ERES, D. Fast and secure distributed read-only file system. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Oct. 2000), pp. 181–196.
[9] [Rosenblum92] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *Proc. of the 13th Symp.on Operating System Principles*, pages 1-15, October 1991.