Vol. 46 No. SIG 12(ACS 11) IPSJ Transactions on Advanced Computing Systems Aug. 2005

Regular Paper

Arbre: A File System for Untrusted Remote Block-level Storage

Tomonori Fujita[†] and Masanori Ogawara[†]

Arbre is a file system that guarantees the integrity of the entire file system on an untrusted remote block-level storage system with a small amount of trusted storage. Arbre does not require changes to the remote block-level storage systems commercially available today. Even if an unauthorized person manages to get complete access to the storage system, they cannot modify or replay any data without being detected. This is achieved by organizing all file system blocks as a single tree and storing their hashes as a part of metadata to later verify the correctness of their contents.

1. Introduction

To cope with the rapidly growing volume of data, many companies have started to consider the iSCSI protocol[1] used to build inexpensive Storage Area Network (SAN) environments by using Ethernet networks. The iSCSI protocol encapsulates the SCSI protocol into the TCP/IP protocol, and carries packets over IP networks. SAN environments over IP networks are called IP-SAN.

IP-SAN started a new trend, called storage outsourcing. Storage service providers (SSPs) offer computer storage space and related management to their customer companies. The customers access storage via high-speed, low-latency networks (e.g., Gigabit Ethernet networks in metropolitan area).

While IP-SAN and storage outsourcing may bring about advantages, they also increase concern about data security: a malicious intruder may get more opportunities to access data in storage because a host computer and the storage can be connected over IP networks; a malicious person may work for your SSP and administer your storage.

Nowadays, computerized data is vital for all companies, so malicious data modifications may cause a substantial loss. For example, trade based on bogus information may directly result in huge losses. Or maliciously modifying metadata, which is information about directories or some file system structures such as free inodes and blocks, may cause an operating system to crash. Consequent downtime might lead to the loss of business opportunities.

With IP-SAN, data modifications are more possible compared with traditional storage architectures. Therefore, file systems that can detect data modifications are desirable. Unfortunately, traditional file systems do not provide such protections. To address this problem, we have developed Arbre[2], a file system designed for untrusted block-level storage systems.

Arbre is not a file system for Network Attached Storage (NAS), because NAS commonly refers to a storage architecture that uses file-oriented delivery protocols such as NFS. Arbre assumes block-level storage systems that use block-oriented delivery protocols such as iSCSI[☆]. Arbre works with commercial block-level storage systems without modifying them.

A host computer views the iSCSI driver as a general SCSI host bus adapter driver that manages the disk drives directly connected by system buses. No differences can be found between the *remote* iSCSI storage system and the *local* hard disk. Thus, Arbre does not have any functionality for accessing storage over IP networks unlike traditional remote file systems such as NFS.

Arbre protects the integrity of the entire file system, rather than the integrity of each block or each file individually. It uses a tree-structured collection of collision-resistant hashes for file system blocks. The synergy between the tree structure and the control of the write order always guarantees the integrity of an entire file system and the consistent state of the file system data, including metadata and file data, in the event of a system crash.

The outline of the rest of this paper is as follows. Section 2 describes our security assumptions. Section 3 covers the issues related to pro-

[†] NTT Cyber Solutions Laboratories

[☆] For this work, we used the iSCSI protocol even though Arbre works with other block-level protocols.

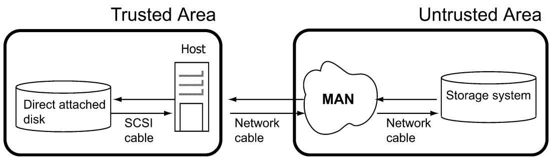

Fig. 1 Arbre architecture (using the iSCSI protocol).

tecting data integrity. Section 4 provides an overview of the Arbre architecture. Section 5 discusses some of the detailed implementations. Section 6 presents our performance results. Section 7 summarizes related work, and Section 8 summarizes the points.

2. Security Model

As shown in **Fig. 1**, we assume the following conditions.

- The host computer is physically protected from unauthorized people. Thus, all programs including the operating system running on it and their volatile state are protected from being read or modified by anyone who is unauthorized.
- The networks between the host computer and remote block-level storage are not trusted, thus a packet may be modified or replayed.
- Remote storage is untrusted. That is, an unauthorized person may gain complete access to it. An administrator, who is expected to manage it properly, may also be corrupt. Thus, data stored in it may be modified or replayed☆.

3. Data Integrity Issues

Under our assumptions, we present a taxonomy of the ways that one can protect data integrity in a file system, and the vulnerabilities of each approach.

3.1 Granularity in Integrity

All prior file systems that address data integrity use a collision-resistant hash function, such as SHA-1[3], which produces a short-length output from an arbitrary-length input. It is computationally intractable for finding any two inputs that produce the same output. Therefore, the output can be used to verify the integrity of the input.

The same technology is used to protect data integrity. However, there are several ways to protect the integrity in a file system. From the perspective of granularity in the integrity, we categorize them into three levels: the block, the file, and the file system.

3.1.1 The Block Level

The integrity of individual blocks is protected. A potential implementation would be to store the hash of each block, index them by file system block number, and verify each block individually.

This approach is vulnerable to replay attacks. Suppose that a file system stores all file system blocks and their keyed hashes in untrusted remote storage. Keyed hash schemes such as Keyed-Hashing for Message Authentication (HMAC)[4] are based on a cryptographic hash and a secret key, therefore an person who does not know the secret key cannot modify data without being detected. However a malicious person who has complete access to the storage can save the set of a block and its keyed hash, and then later replace it with this old set. Consequently, the file system consists of new blocks and an old block. This inconsistency leads to a system crash or a bogus file. Unfortunately, the file system cannot detect the replay, because the maliciously-written old block has its valid hash.

3.1.2 The File Level

The integrity of individual files is protected. A potential implementation would be to store the hash of blocks that constitute a file in the inode structure of the file, and verify each file individually.

This approach is also vulnerable to replay attacks. Suppose that a file system stores all file system blocks and the keyed hash of each file in untrusted remote storage. A malicious person can save blocks constituting a file and the file's keyed hash, and then replace the newer version of the file and its keyed hash with them later without being detected.

3.1.3 The File System Level

The integrity of the entire file system is protected. A potential implementation would be to store a single hash of all file system blocks at some point.

This approach is also vulnerable to replay attacks. Suppose that a file system stores all file system blocks and one keyed hash of them on untrusted remote storage. A malicious person can save all file system blocks and the keyed hash of them, and then replace all newer blocks and its keyed hash with them later without being detected.

☆ It means that a malicious administrator may rolls back stored data.

3.2 Consistency

File systems protecting data integrity have to address three kinds of consistency: hash consistency[☆], metadata consistency, and file-data consistency.

Hash consistency means consistency between a block and its hash. Take for example the implementation of a file system protecting data integrity at the block level in Section 3.1.1. When the file system modifies a block, it has to compute its hash and write the block and the hash to storage in an atomic way. That is, the file system must guarantee that either both modifications would be committed, or neither would. Without this guarantee, inconsistency between a block and its hash would arise after a system crash. Unfortunately, the file system has no way of knowing what caused the inconsistency, i.e., a system crash or malicious modifications.

File systems protecting data integrity must address both metata and file-data consistency, although only file-data consistency has also been addressed by general-purpose file systems. Take for example the implementation in Section 3.1.1 again. Suppose a single system call modifies two file-data blocks — **A** and **B**. The file system computes the hash of each block, and then commits the block **A** and its hash. However, the system crashes before the block **B** and its hash reach storage. After the system reboots, the file system sees the file that consists of the new contents of the block **A** and the old contents of the block **B**. This is the identical situation caused by the replay attacks explained in Section 3.1.1.

4. Architecture

4.1 Design Goals

Under our assumptions described in Section 2, the Arbre file system guarantees the following two things.

- An unauthorized person cannot modify or replay a block without being detected.
- The user never sees any kind of inconsistency because of a system crash.

The second guarantee means that hash, metadata, or file-data inconsistency always means malicious modification or replays.

Note that authentication between the host computer and remote storage is still necessary,

[☆] In the case that a file system stores a hash as a part of metadata, metadata consistency includes hash consistency.

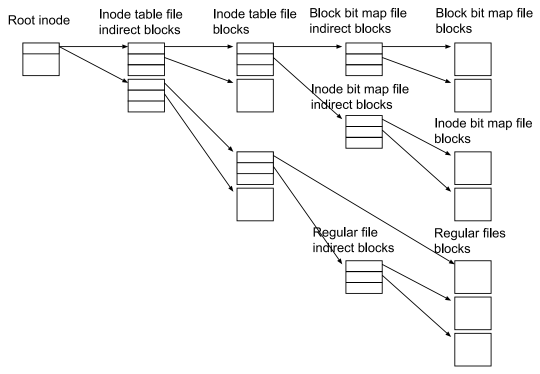
Root inode
Inode table file indirect blocks
Inode table file blocks
Block bit map file indirect blocks
Block bit map file blocks
Inode bit map file indirect blocks
Inode bit map file blocks
Regular file indirect blocks
Regular files blocks

Fig. 2 Arbre disk layout.

although Arbre is designed for untrusted remote storage. This is because Arbre does not guarantee that data stored in remote storage are maliciously modified or replayed. It only guarantees detecting modification and replay attacks and helping recovery process after detecting them. The iSCSI protocol supports several authentication schemes.

4.2 On-disk Data Structures

The Arbre disk layout, shown in **Fig. 2**, is comparable to that of a Write Anywhere File System (WAFL)[5]. Arbre structures all blocks as a tree and stores metadata, the *inode table file*, the *block bitmap file*, and the *inode bitmap file* in files. The *root inode*, the root of a tree, represents the inode table file, which contains inodes for the rest of the files. The block bitmap file and inode bitmap file keep track of free blocks and inodes, respectively. Unlike a WAFL, Arbre uses a bitmap structure for them.

The file system block size is 4 KB.

Like the Fast File System (FFS), the Arbre inode structure contains the attributes of a file and pointers, i.e., disk block number, to indicate which blocks belong to the file. Unlike an FFS, the pointers in an Arbre inode point to blocks at the same level.

Arbre stores a pointer that points to a block and its hash together. To compute a hash, Arbre uses the SHA-1 collision-resistant hash function, which produces a 160-bit hash value. The Arbre inode structure contains eight sets of a pointer and a hash. The size of an inode is 256 bytes.

4.3 Reading a Block

If a file system block is not found in memory, Arbre issues a read request to the block I/O subsystem. On receiving the block, Arbre computes its hash and compares it with the hash stored alongside a pointer that points to the block to verify its integrity. Arbre structures all blocks as a single tree. Therefore, the root of a tree, i.e., the root inode, can be used to verify the integrity of the entire file system recursively.

The hash of the root inode, called *the root hash*, encapsulates the hashes of all blocks constituting the file system at some point. By using a combination of the root hash and a small number of other hashes, Arbre can verify the integrity of any part of the entire file system in a recursive fashion without reading all the blocks. This scheme uses the technique of maintaining the tree-structured collection of hashes, called a *hash tree* 6).

The hash tree is not sufficient for protecting data integrity in untrusted storage, because a person getting complete access to the storage can compute and write the hash of a block. Arbre makes use of HMAC to protect the root hash, so that a person who does not know the secret key cannot forge the root hash. Therefore, Arbre can detect the modification of the root hash in untrusted storage unless the secret key stored in the host computer is revealed.

Though HMAC can prevent an unauthorized person from modifying the root hash without being detected, it cannot prevent replays of an old root hash. In order to prevent replay attacks, upon updating the root hash on untrusted storage, Arbre also stores it in the host computer. When the file system is mounted, Arbre uses the root hash stored in the host computer to ensure that the root inode stored in untrusted storage is not replaced with an old root inode. This scheme enables Arbre to detect any kinds of replay attacks. We describe the detailed procedure later.

The current implementation handles inconsistency between a block and its hash like an I/O error (e.g., disk drive failure). A system call that read a maliciously-modified block receives an EIO error and the system prints an alert message on the console and produces a beep. Thus, the user can prevent further damage.

After Arbre finds a maliciously-modified block, there is no way to recover the original data of the block unless the user saved backup copies. The amount of doubtful data depends on the place of the maliciously-modified block. That is, blocks that descend from the modified block become doubtful. Reading regular files need to access metadata files, thus if the block of the metadata files is maliciously modified, multiple files can be doubtful.

4.4 Writing a Block: Phase Tree Algorithm

Certain system calls modify several blocks. Arbre performs such modifications in an atomic way. That is, Arbre guarantees that after a system crash, either all modifications will reach storage, or none will. We refer to a set of modifications due to a single system call as an *atomic operation*.

We adopted the proposed *phase tree* algorithm of the TUX2 file system 7) to ensure that each system call is performed in an atomic way. Arbre can guarantee hash, metadata, and file-data consistency by using this algorithm.

The phase tree algorithm is similar to *no-overwrite* techniques used by a WAFL or Venti 8). It exploits a tree-structured file system and uses two techniques: the controlling of the order of writes and the writing of a modified block in a new location.

The phase tree algorithm writes the root block after all modified blocks due to an atomic operation are written in a new location. **Figure 3** shows how it works.

Figure 3 (a) represents a file system that has no modified blocks. The file system has only one tree, called the *recorded tree*, representing a persistent version of the file system stored in storage. This is a simplified version of Fig. 2.

Figure 3 (b) represents the state of the file system after an atomic operation modifies the block D3. Before modifying the block D3, the phase tree algorithm makes a copy of the recorded root inode, i.e., the root inode of the recorded tree. Then, the duplicate root inode becomes the new root of a tree, called the *branching tree*, and it represents the latest version of the file system. The phase tree algorithm never overwrites a block belonging to the recorded tree. A modified block is always written in a newly allocated location.

The contents of the modified block D3 are written in a newly allocated location, the block D4. Then the block P1, which originally has a pointer to the block D3, is modified and written in a newly allocated block P2. Now the block P2 has pointers to the allocated block D4 and the block D2, and the modified root inode points to the block P2. After the blocks P2 and D4 reach storage, the modified root inode, which has pointers to the block P0 and P2, is written. It becomes the new version

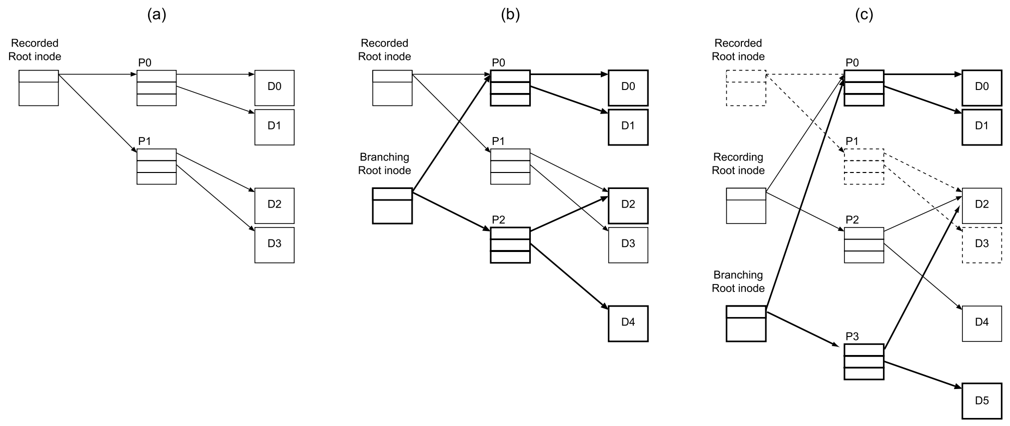

Fig. 3 Tree transition.

of the recorded root inode. In the phase tree algorithm, the file system always has to make changes all the way up the root inode upon every atomic operation.

If an unexpected event happens before the root inode reaches storage, the phase tree algorithm goes back to the recorded root inode stored in storage.

4.4.1 Performance Optimization

Intuitively, the phase tree algorithm has two clear performance bottlenecks. First, each atomic operation results in modifications all the way up the root of the tree. This means that the file system has to write more data than update-in-place file systems like FFS. Secondly, while a modified block is being written to storage, it cannot be modified. That is, if a system call is trying to modify the block, it has to wait for the block to reach storage. This is problematic particularly for a frequently modified block, e.g., a block of the inode table file.

For performance improvement, the phase tree algorithm enables a separate solution for each of these two bottlenecks.

4.4.1.1 Clustering

To decrease the number of writes, the phase tree algorithm can push several atomic operations into a single branching tree. This enables the file system to avoid creating a new branching tree for every atomic operation. This amortize the cost of the operations to create a new branching and commit a recording tree to disk. Moreover, this enables Arbre to commit multiple pending I/O operations within a single block by writing a single block to disk. We refer to this optimization as *clustering*.

A similar technique is used in the ext3 journaling file system, which is a journaling file system widely used in Linux. The ext3 file system merges metadata changes due to several system calls within a certain period of time into a single transaction. When several system calls change the same metadata block, the ext3 file system overwrites them. When the transaction finishes, the ext3 records the last state of the metadata block in the log.

4.4.1.2 Copy Out

As described, if an atomic operation tries to modify a block that is being written to storage, it must wait for the I/O completion of the block. Instead of suspending a system call, the phase tree algorithm copies the contents of the block into a newly allocated buffer and permits the system call to modify the buffer.

Figure 3 (c) shows how this optimization works. It represents the state of the file system receiving a new atomic operation during writing modifications due to a finished atomic operation. Suppose that the file system is writing modifications (the block D4, P2, and the root inode) due to a finished atomic operation and a new atomic operation is about to modify the block D4. We call the tree having blocks that are being written the *recording tree*.

The file system creates a new branching tree by duplicating the recording root inode, and then copies the contents of the block D4 into a newly allocated block D5. The system call can modify the block D5 immediately without waiting for the I/O completion of the block D4.

With this optimization, the phase tree algorithm usually has three trees at the same time: the recorded tree, the recording tree, and the branching tree.

We call this optimization *copy out*, which is one of the biggest differences between the phase tree algorithm and the WAFL algorithm. In the same situation, a WAFL suspends the system call until the I/O completion.

4.4.2 Updating the Root Inode and Its Hash

Arbre stores the set of the root inode and its hash in a fixed location on remote storage to find it when mounting the file system. In addition, Arbre requires 256-byte space of trusted storage (typically a disk directly attached to the host computer) for the copy of the root inode to prevent replay attacks.

We assume that the hard disk drive guarantees the writing of a single 512-byte sector atomically with respect to crashes. Arbre writes a set of the root inode and its hash to a single 512-byte sector; thus the root inode and its hash are updated atomically.

Arbre performs the following two operations upon updating the root inode and its hash. First, Arbre writes the root inode of the recording tree and its hash in the location of the *remote root inode* in remote storage. Secondly, the root inode stored in the location of the *local root inode* in trusted storage is updated after the recording root inode and its hash reach remote storage.

In Section 4.7, we describe the recovery procedure upon system restart.

Atomic operations are not blocked during updating the remote root inode or the local root inode. That is, atomic operations can safely modify the branching tree while the recording tree is committed. The details of schemes to modify blocks that are being committed are explained in Section 5.

4.5 Block Allocation and Reclamation

With the clustering optimization, Arbre asynchronously performs an atomic operation. That is, not all modifications due to a system call reach storage when the system call returns. This enables Arbre to not allocate a new location to a block belonging to a branching tree. Thus, Arbre avoids needless allocations of disk space for a short-lived file to improve performance. After the branching tree becomes a recording tree, new locations are allocated. Similar techniques to delay the allocation of disk space are used in the log-structured file system (LFS) [9].

In the current Arbre implementation, block-allocation policy is simple. It allocates physically sequential blocks to dirty buffers in turn without having concepts about high-level information such as files and directories. This policy may result in good performance in writing but may also result in poor performance in reading. We examine the effect in Section 6.

Unlike no-overwrite file systems such as a WAFL or Venti, Arbre soon reclaims a block that becomes useless. In Fig. 3 (c), the blocks written with dotted lines are reclaimed when the root inode of the recording tree reaches storage and becomes the root inode of the new recorded tree. The branching tree keeps track of blocks that are made unnecessary by replacement with a newly allocated block or by the deletion of files. Then, when the branching tree becomes the recording tree, the block bitmap file of the new recording tree is modified appropriately.

4.6 Computing Hashes

Performing an atomic operation asynchronously also enables Arbre to perform hash computation efficiently.

Arbre asynchronously computes hashes to avoid needless hash re-computations. That is, Arbre does not compute hashes of blocks belonging to a branching tree. Like the approach to allocating a block, when a branching tree becomes a recording tree, Arbre computes hashes of modified blocks all the way to the root inode.

4.7 Recovery

4.7.1 The Root Inode

The Arbre recovery procedure consists of two phases. First, Arbre restores the root inode.

Arbre verifies the set of the remote root inode and its hash. Then, Arbre computes the hash of the root inode, and compares it to the hash stored alongside the remote root inode. The verification failure means malicious modification.

If the hash verification is successful, Arbre compares the set with the local root inode, which is stored in the host computer, to ensure that the remote root inode represents the latest file system. If they are identical, Arbre mounts the file system by using the remote root inode. When they are different, there are two possible cases. If the timestamp in the remote root inode is newer than that in the local root inode, it means that Arbre committed the remote root inode, but a system crash before committed the local root inode. In this case, Arbre can safely mount the file system by using the remote root inode because with HMAC it is impossible for a person who does not know the secret key to forge the set of a root inode containing future timestamp and its valid hash.

When timestamp in the remote root inode is older than that in the local root inode, the past root inode and its hash were maliciously replayed.

4.7.2 Freeing Unused Inodes

After mounting the file system, Arbre performs the second phase to reclaim unused inodes and blocks.

Due to last close semantics in UNIX operating systems, Arbre may mark free blocks and inodes as “in use” even when they are actually unused. This happens when an inode is deleted from the directory, but held open at the time of a crash. Arbre adopts the scheme that the ext3 file system uses to guarantee recovery to a consistent state after a system crash.

When a branching tree becomes a recording tree, the inode number of a file that is deleted but held open is recorded in the root inode. If there is another file deleted but held open, its inode number is saved in the inode structure whose inode number is stored in the root inode. Therefore, Arbre can trace these inodes in turn from the root inode to correct the inode bitmap file and the block bitmap file after a system crash.

5. Detailed Implementation

5.1 Tree Transition

A kernel thread, called *phase thread*, deals with all of Arbre's operations: tree state transitions, asynchronous hash computations, and the write of cached file system data. Arbre does not use the Linux standard *pdflush* service, which writes dirty cached data periodically☆.

To simplify the implementation, we changed the phase tree algorithm slightly. The original phase tree algorithm does not limit the number of recording trees. Thus, the file system usually has one recorded tree, several recording trees, and one branching tree. However, Arbre has one recording tree at most.

We explain how the phase thread works in subsequent paragraphs.

5.1.1 Creating a Branching Tree

When Arbre starts an atomic operation, it first allocates a data structure representing a tree, and sets its state to BRANCHING. Possibly, Arbre already has a branching tree since Arbre pushes several atomic operations into a single branching tree. In such a case, Arbre does not create a new tree and uses the current branching tree.

☆ Though the traditional block buffer cache does not exist in the Linux kernel version 2.5 series, we use the word *buffer* to represent a cached disk block.

Second, Arbre increases the counter called *operation counter* in the data structure of the branching tree. This counter prevents a branching tree from becoming a recording tree while the branching tree manages an unfinished atomic operation.

5.1.2 Modifying Data

Before the atomic operation modifies a buffer, Arbre checks the state of the buffer. There are three possible cases.

(1) The buffer is clean.
(2) The buffer is dirty and linked to the branching tree.
(3) The buffer is dirty and linked to a recording tree.

In the first case, Arbre links the buffer with the branching tree. In addition, Arbre links an inode object to which the buffer belongs with the *dirty inode list* in the data structure of the branching tree. Then, Arbre modifies the buffer.

The second case means that the buffer is already modified and managed by the branching tree. In this case, Arbre can modify the buffer immediately.

In the third case, the buffer is also dirty like the second case; however, the buffer is linked not with the branching tree, but a recording tree. This means the buffer might be being written to storage. If the buffer has not yet been handed over to the block I/O subsystem, Arbre allocates memory, and then copies the contents of the buffer into it. The newly allocated memory called *frozen data* is linked with the buffer. Frozen data is used later when the buffer is actually handed over to the block I/O subsystem. Arbre can safely modifies the buffer because its contents is saved. If the buffer has been handed over to the block I/O subsystem already, Arbre waits for the operation to complete.

When the atomic operation finishes, Arbre decreases the operation counter.

5.1.3 Transition from Branching to Recording State

The phase thread changes the state of a tree from BRANCHING to RECORDING in three cases: when sync or fsync system call is issued[☆], when too much time has elapsed since a branching tree was created, or when the number of dirty buffers exceeds a threshold. In such a case, the phase thread performs the following operations to guarantee that a recording tree does not include the results of unfinished atomic operations.

(1) The phase thread sets the state of the branching tree to LOCKED. This prohibits the branching tree from receiving a new atomic operation.
(2) If the branching tree manages atomic operations that are already in progress, the phase thread waits for these atomic operations to finish.
(3) The phase thread modifies the block bitmap file to free unused blocks, and then links deleted but open inodes together from the branching inode.
(4) The phase thread allocates new locations for dirty buffers linked with the locked tree.
(5) The phase thread modifies the inode table file by using inode objects linked with the dirty inode list in the locked tree and then unlinks the inode objects from the dirty inode list.

After finishing these operations, the phase thread changes the state from LOCKED to RECORDING. This means that the phase thread resumes accepting a new atomic operation.

These operations take a short time because they require few disk I/O. Thus there are few atomic operations blocked due to a locked tree.

5.1.4 Computing Hashes

The phase thread computes the hashes of the dirty buffers and then modifies dirty buffers by using these hashes. It starts from the leaf to the root inode because the contents of a block except file-data blocks contain a hash of a block at the lower level.

If a dirty buffer has frozen data, the phase thread computes the hash of the frozen data instead of the buffer. This is because a new branching tree modifies the buffer before the phase thread starts computing its hash.

5.1.5 Committing

The phase thread starts writing the buffers to storage. If a buffer has its frozen data, the frozen data is written to storage instead of the buffer. After all modified buffers linked with the recording tree reach storage, the phase thread writes the recording root inode in the way described in Section 4.4.2.

6. Performance

6.1 System Comparison

We compared Arbre's performance to that of ext3. We configured ext3 as "writeback" mode. Arbre guarantees metadata and file-data consistency, although ext3 in this mode guarantees only metadata consistency. Both update their data asynchronously. A kernel thread called *kjournald* logs metadata every five seconds by default for ext3. Like kjournald, we configured the phase daemon to write modified blocks every five seconds.

6.2 Experimental Infrastructure

We present the performance of Arbre running on Linux kernel version 2.5.63. We report the averages of three runs for all experiments.

The host computer is the Dell Precision Workstation 530, which uses a 2 GHz Xeon processor with 1 GB of PC800 RDRAM main memory, an Intel 860 chipset and an Intel Pro/1000 MT Server Adapter connected to a 66-MHz 64-bit PCI slot. It runs our iSCSI initiator implementation based on the source code of the IBM iSCSI initiator. Fujitsu-MAM3184MP 18 GB 15000 RPM SCSI disks are directly connected to the host via an Adaptec 7892 Ultra 160 SCSI chip.

The Remote storage is the IBM TotalStorage IP Storage 200i, which is an iSCSI appliance on two 1.13 GHz Pentium III processors with 1 GB of PC133 SDRAM main memory and an Intel Pro/1000 F network adapter connected to a 66-MHz 64-bit PCI slot.

The file systems examined have a 18 GB partition on the remote disk.

To evaluate the network delay effects in a WAN environment on the performance, we used the National Institute of Standards and Technology Net (NIST Net)[10)], software package that emulates performance dynamics in IP networks. We set up another server as a router, on which NIST Net runs, between the server and the remote disk. It had the same specification as the host computer. We set the one-way network delay to 0, 2, and 4 ms[☆☆].

[☆] Effectively, a fsync system call is identical to a sync system call with Arbre, because there is now way of committing only a particular file.

[☆☆] We had measured network delays for one week using

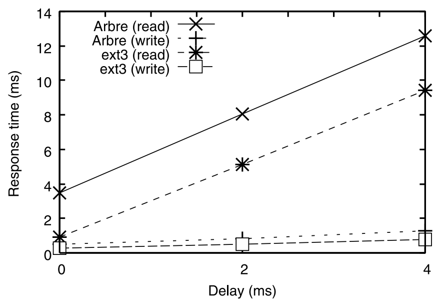

Fig. 4 Response time.

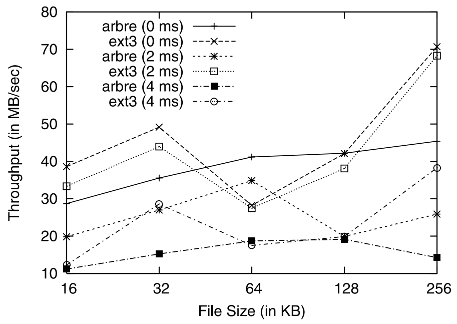

Fig. 5 Throughput (MB/s) creates.

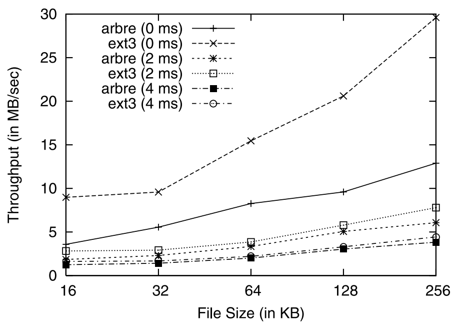

Fig. 6 Throughput (MB/s) reads.

6.3 Microbenchmark Results

To get the basic performance characteristics of Arbre, we (1) measured the response time of reading and writing a single block, and (2) used microbenchmarks similar to those used in the recent file system papers[11),12)] on the remote disk. We started each test with a cold cache.

Figure 4 shows the results of the response time. As expected, the read performance is severely impacted by network delay (the response time has increased linearly), although the write peformance is not due to page cache. Arbre performs worse mainly because it, which stores metadata in files, takes a longer time to modify an inode than ext3, which stores metadata at fixed places in disk.

The microbenchmarks create (including write), read, and delete files with the sizes ranging from 16 KB to 512 KB. The total amount of data is 256 MB. To avoid the big overheads of pathname lookup, the files were distributed over directories, and there were fewer than 50 files in each directory.

Figure 5 shows the results of the create microbenchmark. Arbre is slower than ext3 with file sizes up to 64 KB, for the same reason as before. With 64 KB, Arbre is faster than ext3, because it allocates physically sequential blocks to dirty buffers and writes them at a time without distinction of metadata and file-data blocks. Therefore it can use most of the disk's bandwidth. On the other hand, over 48 KB, ext3 has to log the modifications of indirect blocks[☆]. Ext3 tries to allocate physically continuous blocks to indirect and file-data blocks. However file-data blocks, which ext3 does not log, are passed to the I/O subsystem soon, though ext3 waits to write indirect blocks until they are committed in the log. This causes more disk seeks, which lower the performance.

For large file sizes, Arbre was expected to achieve high performance by allocating physically continuous blocks, however, it did not (especially with large delays). This is mainly because Arbre must serialize most updates.

Figure 6 shows the results of the read microbenchmark. As expected, ext3 greatly outperforms Arbre. Though Arbre suffers the overhead for computing hashes, the main problem is Arbre's simple block-allocation policy, i.e., sequential allocation without high-level notion of a file and a directory. As a result, on Arbre, this benchmark works like random read. On the other hand, ext3 divides the partition into several groups, tries to distribute directories uniformly to the groups, and attempts to allocate all files of a directory to the same group. This policy is favorable to this benchmark, which reads files in each directory recursively. One possible solution would be for Arbre to use a similar block-allocation policy though it might slightly decrease the write performance.

Figure 7 shows the results of the delete microbenchmark. With file sizes up to 64 KB, we see that ext3 performs better than Arbre, be-

our mass-market fiber network services in a metro area. The average one-way delay was 3.71 ms.

☆ An ext3 inode has twelve direct pointers.

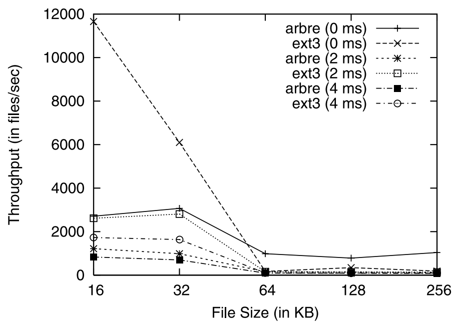

Fig. 7 Throughput (files/s) delete.

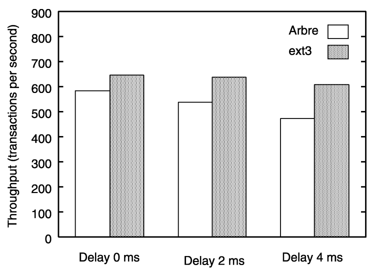

Fig. 8 PostMark benchmark results.

cause Arbre, storing metadata in files, suffers higher cost of inode operations. Furthermore, it also suffers overhead for computing hashes. The large performance drop of ext3 at 64 KB results from the overhead of logging an indirect block such as the create and write benchmark as described. The drop of Arbre is due to the overhead of reading an indirect block such as the write benchmark.

6.4 Macrobenchmark Results

To evaluate the performance of Arbre in common workloads, we ran the PostMark benchmark[13] on the remote disk. The PostMark benchmark is designed to measure performance in the ephemeral small-file workloads seen by Internet Service Providers. We set the benchmark to run with 20,000 files, 50,000 transactions, and file sizes between 512 bytes and 16 KB.

Figure 8 shows the results. Ext3 is 10.8%, 18.9%, and 28.3% faster than Arbre respectively. The results show that Arbre is more sensitive to delay conditions because it must serialize most of writes, as explained before. Furthermore, Arbre relies on high-performance sequential disk access to achieve high performance by allocating physically continuous blocks. However, sequential I/O performance is severely impacted by network delay, unlike random I/O performance whose latency can be hidden by the disk random-access latency[14].

7. Related Work

There have been a number of previously developed file systems that have been designed to protect data integrity in a remote untrusted repository. However, there have been few for remote untrusted block-level storage using existing block-oriented delivery protocols. Most of them assume file-oriented delivery protocols or newly designed block-oriented protocols.

As far as we know, the protecting file system (PFS)[15] is the only file system that achieves data integrity for remote storage using existing block-oriented delivery protocols. PFS is most like the potential implementation protecting data integrity at the block level as described in Section 3.1.1 and guarantees hash and metadata consistency by using the journaling approach. As explained, this approach is vulnerable to replay attacks. With the hash tree technique PFS could provide data integrity at the file system level, however, integrating the hash tree technique with the PFS design is not easy because it stores hashes separately from the file system. PFS keeps the *block map*, data structure for mapping a file system block number to its hash. PFS divides up the block map and writes each piece in a log-like structure in a round-robin fashion because of the large size of the block map. The hash tree technique forces PFS to commit the whole block map in a tree-like structure at a time. This operation takes a long time and requires the complex locking scheme to synchronize access to the block map. A large amount of memory for keeping the whole block map is also required to perform this operation with reasonable performance. Arbre, on the other hand, tightly integrates the hash tree into the design to update the file system with touching a small amount of data while keeping the integrity of the entire file system.

The read-only secure file system (SFSRO)[16] is a read-only file system intended for content distribution. The SFSRO inodes contain a hash tree, and an authorized user signs the root hash. SFSRO uses not existing block-oriented delivery protocols but a newly designed protocol. The protocol uses the hash of a file-data block as the block identifier to guarantee data integrity. SFSRO does not address the consistency problems after a system crash since it does not assume frequent updates.

Another system is SUNDR[17], which provides data integrity at the file system level without relying on trust in remote storage. SUNDR designs a new block-oriented delivery protocol. SUNDR remote storage indexes blocks by their hash instead of their block number. The host computer requests blocks from the remote storage by using their hash.

Ivy[18], a peer-to-peer file system, also uses a similar technique known as distributed hash tables (DHT) to index blocks by their hash[19]. It stores changes made to the file system in a log-like structure. Therefore data integrity at the file system level is provided. Ivy is not a general-purpose file system, but a special-purpose file system for sharing data in P2P networks. It does not provide performance comparable with to that of a general-purpose file system.

WAFL[5] and Venti[8] structure all blocks as a single tree like Arbre, and use a technique similar to the phase tree algorithm. WAFL, a file system designed for an NFS appliance, does not address data integrity. Venti uses a newly designed storage protocol to index blocks by their hash to provide a flexible location for archival storage. Venti provides data integrity at the file system level; however, Venti is an archival storage system for special-purpose applications. It does not keep the general file system interface.

The Arbre design, which structures all blocks as a tree, where metadata blocks contain hashes of data blocks, is comparable to that of the Trusted Database (TDB)[20], which is a database designed to be housed in untrusted storage. However, Arbre, which provides a general file system interface, has to deal with problems peculiar to a file system.

The technique used in Cepheus[21] of storing a pointer to a block and its hash together to protect the integrity of the block is comparable to that of Arbre. Cepheus is most like the potential implementation protecting data integrity at the file level as described in Section 3.1.2. It uses a modified version of the NFS protocol, and thus does not protect the integrity of some metadata such as block bitmap data. Furthermore, hash, metadata, or file-data consistency are not addressed.

SNAD[22] is network attached storage providing confidentiality and integrity at the file level. Unlike Arbre, it uses a newly designed block-level storage architecture.

For data confidentiality, Arbre can use device-mapper crypto target feature[23] in the Linux kernel that transparently encrypts and decrypts data at the block level.

An important extension for future work is the reliability of data. Though it is impossible to prevent a person who gets complete access to storage from wiping out data stored in it, Arbre can alleviate the danger of losing data in such a case by replicating data over multiple storage sites. Many works[24] have addressed the issue of data reliability in the presence of a site disaster through replicating storage at the block level.

8. Conclusion

The advent of IP-SAN has increased concern about data security. However, general-purpose file systems can not detect data modification.

This paper describes Arbre, a file system designed for remote block-level storage using the standard IP storage protocols. Without relying on trust in remote storage or changes to the block interface between a host and storage systems, Arbre guarantees the integrity of the entire file system on a large amount of untrusted remote storage by using a small amount of trusted storage. An unauthorized person has no way to modify or replay any part of the blocks that constitute the file system without being detected.

Arbre structures all file system blocks as a tree and storing a pointer to a block and its hash. It achieves data integrity at the file system level by synthesizing the tree structure and the no-overwrite technique.

Arbre alleviates the overheads of updating blocks all the way up to the root by performing system calls asynchronously, delaying allocations of disk space, and making it possible to modify a block that is being transferred to disk. Consequently, it provides acceptable performance.

References

1) Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M. and Zeidner, E.: Internet Small Computer Systems Interface (iSCSI), RFC 3720 (2004).

2) Tomonori, F. and Masanori, O.: Protecting the Integrity of an Entire File System, *First IEEE International Workshop on Information Assurance* (*IWIA '03*), Darmstadt, Germany, pp.95–105 (2003).

3) National Institute of Standards and Technol-

ogy, FIPS 180-1, Secure Hash Standard, US Department of Commerce (Apr. 1995).

4) Bellare, M., Canetti, R. and Krawczyk, H.: Keying Hash Functions for Message Authentication, *Advances in Cryptology — Crypto '96*, Lecture Notes in Computer Science, Vol.1109, Santa Barbara, CA, pp.1–15, Springer (1996).

5) Hitz, D., Lau, J. and Malcolm, M.: File System Design for an NFS File Server Appliance, *USENIX Winter 1994 Technical Conference*, San Francisco, CA, pp.235–245 (1994).

6) Merkle, R.C.: A Digital Signature Based on a Conventional Encryption Function, *Advances in Cryptology — Crypto '87*, Lecture Notes in Computer Science, Vol.293, Santa Barbara, CA, pp.369–378, Springer, (1987).

7) Phillips, D.R.: The Tux2 Failsafe Filesystem for Linux. http://marc.merlins.org/linux/linux.conf.au_2001/Day2/Tux2.html

8) Quinlan, S. and Dorward, D.: Venti: a new approach to archival storage, *USENIX Conference on File and Storage Technologies*, Monterey, CA, pp.89–102 (2002).

9) Rosenblum, M. and Ousterhout, J.K.: The Design and Implementation of a Log-Structured File System, *ACM Trans. Comput. Syst.*, Vol.10, No.1, pp.26–52 (1992).

10) The National Institute of Standards and Technology: NIST Net network emulator. http://snad.ncsl.nist.gov/nistnet/

11) Seltzer, M., Ganger, G., McKusick, M.K., Smith, K., Soules, C. and Stein, C.: Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems, *USENIX Annual Technical Conference*, San Diego, CA, pp.71–84 (2000).

12) Ganger, G.R., Mckusick, M.K., Soules, G.A.N. and Patt, Y.N.: Soft Updates: A Solution to the Metadata Update Problem in File Systems, *ACM Trans. Comput. Syst.*, Vol.18, No.2, pp.127–153 (2000).

13) Katcher, J.: PostMark: A New File System Benchmark, Technical Report TR3022, Network Appliance (1997).

14) Ng, W.T., Hillyer, B., Shriver, E., Gabber, E. and Özden, B.: Obtaining High Performance for Storage Outsourcing, *USENIX Conference on File and Storage Technologies*, Monterey, CA, pp.145–158 (2002).

15) Stein, C.A., Howard, J.H. and Seltzer, M.I.: Unifying File System Protection, *USENIX Annual Technical Conference*, Boston, MA, pp.79–90 (2001).

16) Fu, K., Kaashoek, M.F. and Mazières, D.: Fast and secure distributed read-only file system, *4th USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, pp.181–196 (2000).

17) Mazières, D. and Shasha, D.: Building secure file systems out of Byzantine storage, *21st ACM Symposium on Principles of Distributed Computing*, Monterey, CA, pp.108–117 (2002).

18) Muthitacharoen, A., Morris, R., Gil, T.M. and Chen, B.: Ivy: A Read/Write Peer-to-Peer File System, *4th USENIX Symposium on Operating System Design and Implementation*, Boston, MA, pp.31–44 (2002).

19) Dabek, F., Kaashoek, M.F., Karger, D., Morris, R. and Stoica, I.: Wide-Area Cooperative Storage with CFS, *18th ACM Symposium on Operating Systems Principles*, Banff, Canada, pp.202–215 (2001).

20) Maheshwari, U., Vingralek, R. and Shapiro, W.: How to Build a Trusted Database System on Untrusted Storage, *4th USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, pp.135–150 (2000).

21) Fu, K.: Group Sharing and Random Access in Cryptographic Storage File System, Master's thesis, Massachusetts Institute of Technology (1998).

22) Miller, E., Long, D., Freeman, W. and Reed, B.: Strong Security for Network-Attached Storage, *USENIX Conference on File and Storage Technologies*, Monterey, CA, pp.1–14 (2002).

23) Saout, C.: dm-crypt: a device-mapper crypto target. http://www.saout.de/misc/dm-crypt/

24) Gabber, E., Fellin, J., Flaster, M., Gu, F., Hillyer, B., Ng, W.T., Özden, B. and Shriver, E.: StarFish: Highly-Available Block Storage, *USENIX Annual Technical Conference*, San Antonio, TX, pp.151–164 (2003).

(Received January 19, 2005)
(Accepted May 12, 2005)

Tomonori Fujita received his B.E. and M.E. degrees from Waseda University in 1998 and 2000 respectively. He has worked in Nippon Telegraph and Telephone Corporation since 2000 and has engaged in research on operating systems. He is a member of IPSJ, USENIX, and ACM.

Masanori Ogawara received the B.E. and M.E. degrees in Electrical Engineering from Keio University, Yokohama, Japan in 1992 and 1994, respectively. In 1994, he joined NTT Network Service Systems Laboratories, where he had engaged in research on photonic network communications. He is currently working at NTT Comware Corporation. His research interests include user-oriented service platform on a home network. He is a member of IEICE. He received the paper award from IEICE, Japan, in 1999.