Frangipani: A Scalable Distributed File System

Chandramohan A. Thekkath
Timothy Mann
Edward K. Lee

Systems Research Center
Digital Equipment Corporation
130 Lytton Ave, Palo Alto, CA 94301

Abstract

The ideal distributed file system would provide all its users with coherent, shared access to the same set of files, yet would be arbitrarily scalable to provide more storage space and higher performance to a growing user community. It would be highly available in spite of component failures. It would require minimal human administration, and administration would not become more complex as more components were added.

Frangipani is a new file system that approximates this ideal, yet was relatively easy to build because of its two-layer structure. The lower layer is Petal (described in an earlier paper), a distributed storage service that provides incrementally scalable, highly available, automatically managed virtual disks. In the upper layer, multiple machines run the same Frangipani file system code on top of a shared Petal virtual disk, using a distributed lock service to ensure coherence.

Frangipani is meant to run in a cluster of machines that are under a common administration and can communicate securely. Thus the machines trust one another and the shared virtual disk approach is practical. Of course, a Frangipani file system can be exported to untrusted machines using ordinary network file access protocols.

We have implemented Frangipani on a collection of Alphas running DIGITAL Unix 4.0. Initial measurements indicate that Frangipani has excellent single-server performance and scales well as servers are added.

1 Introduction

File system administration for a large, growing computer installation built with today's technology is a laborious task. To hold more files and serve more users, one must add more disks, attached to more machines. Each of these components requires human administration. Groups of files are often manually assigned to particular disks, then manually moved or replicated when components fill up, fail, or become performance hot spots. Joining multiple disk drives into one unit using RAID technology is only a partial solution; administration problems still arise once the system grows large enough to require multiple RAIDs and multiple server machines.

Frangipani is a new scalable distributed file system that manages a collection of disks on multiple machines as a single shared pool of storage. The machines are assumed to be under a common administration and to be able to communicate securely. There have been many earlier attempts at building distributed file systems that scale well in throughput and capacity [1, 11, 19, 20, 21, 22, 26, 31, 33, 34]. One distinguishing feature of Frangipani is that it has a very simple internal structure—a set of cooperating machines use a common store and synchronize access to that store with locks. This simple structure enables us to handle system recovery, reconfiguration, and load balancing with very little machinery. Another key aspect of Frangipani is that it combines a set of features that makes it easier to use and administer Frangipani than existing file systems we know of.

1. All users are given a consistent view of the same set of files.

2. More servers can easily be added to an existing Frangipani installation to increase its storage capacity and throughput, without changing the configuration of existing servers, or interrupting their operation. The servers can be viewed as "bricks" that can be stacked incrementally to build as large a file system as needed.

3. A system administrator can add new users without concern for which machines will manage their data or which disks will store it.

4. A system administrator can make a full and consistent backup of the entire file system without bringing it down. Backups can optionally be kept online, allowing users quick access to accidentally deleted files.

5. The file system tolerates and recovers from machine, network, and disk failures without operator intervention.

Frangipani is layered on top of Petal [24], an easy-to-administer distributed storage system that provides *virtual disks* to its clients. Like a physical disk, a Petal virtual disk provides storage that can be read or written in blocks. Unlike a physical disk, a virtual disk provides a sparse 2^{64} byte address space, with physical storage allocated only on demand. Petal optionally replicates data for high availability. Petal also provides efficient snapshots [7, 10] to support consistent backup. Frangipani inherits much of its scalability, fault tolerance, and easy administration from the underlying storage system, but careful design was required to extend these properties to the file system level. The next section describes the structure of Frangipani and its relationship to Petal in greater detail.

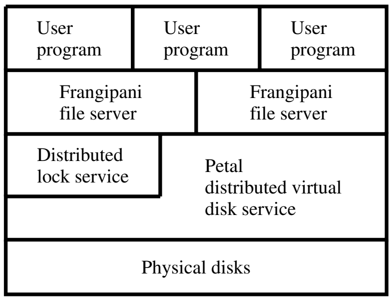

Figure 1: **Frangipani layering.** *Several interchangeable Frangipani servers provide access to one set of files on one Petal virtual disk.*

Figure 1 illustrates the layering in the Frangipani system. Multiple interchangeable Frangipani servers provide access to the same files by running on top of a shared Petal virtual disk, coordinating their actions with locks to ensure coherence. The file system layer can be scaled up by adding Frangipani servers. It achieves fault tolerance by recovering automatically from server failures and continuing to operate with the servers that survive. It provides improved load balancing over a centralized network file server by splitting up the file system load and shifting it to the machines that are using the files. Petal and the lock service are also distributed for scalability, fault tolerance, and load balancing.

Frangipani servers trust one another, the Petal servers, and the lock service. Frangipani is designed to run well in a cluster of workstations within a single administrative domain, although a Frangipani file system may be exported to other domains. Thus, Frangipani can be viewed as a *cluster file system*.

We have implemented Frangipani under DIGITAL Unix 4.0. Due to Frangipani's clean layering atop the existing Petal service, we were able to implement a working system in only a few months.

Frangipani is targeted for environments with program development and engineering workloads. Our tests indicate that on such workloads, Frangipani has excellent performance and scales up to the limits imposed by the network.

2 System Structure

Figure 2 depicts one typical assignment of functions to machines. The machines shown at the top run user programs and the Frangipani file server module; they can be diskless. Those shown at the bottom run Petal and the distributed lock service.

The components of Frangipani do not have to be assigned to machines in exactly the way shown in Figure 2. The Petal and Frangipani servers need not be on separate machines; it would make sense for every Petal machine to run Frangipani as well, particularly in an installation where the Petal machines are not heavily loaded. The distributed lock service is independent of the rest of the system; we show one lock server as running on each Petal server machine, but they could just as well run on the Frangipani hosts or any other available machines.

2.1 Components

As shown in Figure 2, user programs access Frangipani through the standard operating system call interface. Programs running on different machines all see the same files, and their views are coherent; that is, changes made to a file or directory on one machine

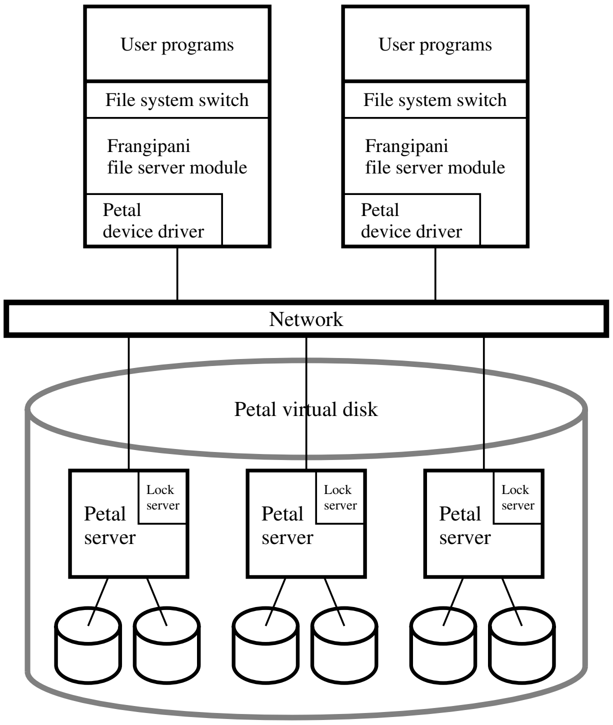

Figure 2: **Frangipani structure.** *In one typical Frangipani configuration, some machines run user programs and the Frangipani file server module; others run Petal and the distributed lock service. In other configurations, the same machines may play both roles.*

are immediately visible on all others. Programs get essentially the same semantic guarantees as on a local Unix file system: changes to file contents are staged through the local kernel buffer pool and are not guaranteed to reach nonvolatile storage until the next applicable `fsync` or `sync` system call, but metadata[1] changes are logged and can optionally be guaranteed non-volatile by the time the system call returns. In a small departure from local file system semantics, Frangipani maintains a file's last-accessed time only approximately, to avoid doing a metadata write for every data read.

The Frangipani file server module on each machine runs within the operating system kernel. It registers itself with the kernel's file system switch as one of the available file system implementations. The file server module uses the kernel's buffer pool to cache data from recently used files. It reads and writes Petal virtual disks using the local Petal device driver. All the file servers read and write the same file system data structures on the shared Petal disk, but each server keeps its own redo log of pending changes in a distinct section of the Petal disk. The logs are kept in Petal so that when a Frangipani server crashes, another server can access the log and run recovery. The Frangipani servers have no need to communicate directly with one another; they communicate only with Petal and the lock service. This keeps server addition, deletion, and recovery simple.

The Petal device driver hides the distributed nature of Petal, making Petal look like an ordinary local disk to higher layers of the operating system. The driver is responsible for contacting the

[1] We define *metadata* as any on-disk data structure other than the contents of an ordinary file.

correct Petal server and failing over to another when necessary. Any Digital Unix file system can run on top of Petal, but only Frangipani provides coherent access to the same files from multiple machines.

The Petal servers run cooperatively to provide Frangipani with large, scalable, fault-tolerant virtual disks, implemented on top of the ordinary physical disks connected to each server. Petal can tolerate one or more disk or server failures, as long as a majority of the Petal servers remain up and in communication and at least one copy of each data block remains physically accessible. Additional details on Petal are available in a separate paper [24].

The lock service is a general-purpose service that provides multiple-reader/single-writer locks to clients on the network. Its implementation is distributed for fault tolerance and scalable performance. Frangipani uses the lock service to coordinate access to the virtual disk and to keep the buffer caches coherent across the multiple servers.

2.2 Security and the Client/Server Configuration

In the configuration shown in Figure 2, every machine that hosts user programs also hosts a Frangipani file server module. This configuration has the potential for good load balancing and scaling, but poses security concerns. Any Frangipani machine can read or write any block of the shared Petal virtual disk, so Frangipani must run only on machines with trusted operating systems; it would not be sufficient for a Frangipani machine to authenticate itself to Petal as acting on behalf of a particular user, as is done in remote file access protocols like NFS. Full security also requires Petal servers and lock servers to run on trusted operating systems, and all three types of components to authenticate themselves to one another. Finally, to ensure file data is kept private, users should be prevented from eavesdropping on the network interconnecting the Petal and Frangipani machines.

One could fully solve these problems by placing the machines in an environment that prevents users from booting modified operating system kernels on them, and interconnecting them with a private network that user processes are not granted access to. This does not necessarily mean that the machines must be locked in a room with a private physical network; known cryptographic techniques for secure booting, authentication, and encrypted links could be used instead [13, 37]. Also, in many applications, partial solutions may be acceptable; typical existing NFS installations are not secure against network eavesdropping or even data modification by a user who boots a modified kernel on his workstation. We have not implemented any of these security measures to date, but we could reach roughly the NFS level of security by having the Petal servers accept requests only from a list of network addresses belonging to trusted Frangipani server machines.

Frangipani file systems can be exported to untrusted machines outside an administrative domain using the configuration illustrated in Figure 3. Here we distinguish between Frangipani client and server machines. Only the trusted Frangipani servers communicate with Petal and the lock service. These can be located in a restricted environment and interconnected by a private network as discussed above. Remote, untrusted, clients talk to the Frangipani servers through a separate network and have no direct access to the Petal servers.

Clients can talk to a Frangipani server using any file access protocol supported by the host operating system, such as DCE/DFS, NFS, or SMB, because Frangipani looks just like a local file system on the machine running the Frangipani server. Of course, a protocol that supports coherent access (such as DCE/DFS) is best, so that Frangipani's coherence across multiple servers is not thrown away at the next level up. Ideally, the protocol should also support failover from one Frangipani server to another. The protocols just mentioned do not support failover directly, but the technique of having a new machine take over the IP address of a failed machine has been used in other systems [3, 25] and could be applied here.

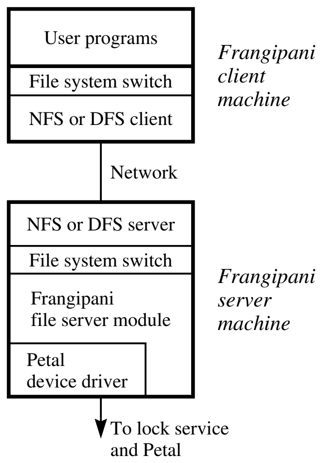

Figure 3: **Client/server configuration.** *A Frangipani server can provide file access not only for the local machine, but also for remote client machines that connect via standard network file system protocols.*

Apart from security, there is a second reason for using this client/server configuration. Because Frangipani runs in the kernel, it is not quickly portable across different operating systems or even different versions of Unix. Clients can use Frangipani from an unsupported system by accessing a supported one remotely.

2.3 Discussion

The idea of building a file system in two layers—a lower level providing a storage repository and a higher level providing names, directories, and files—is not unique to Frangipani. The earliest example we know of is the Universal File Server [4]. However, the storage facility provided by Petal is substantially different from earlier systems, leading to a different higher level structure as well. Section 10 contains detailed comparisons with previous systems.

Frangipani has been designed to work with the storage abstraction provided by Petal. We have not fully considered the design changes needed to exploit alternative storage abstractions like NASD [13].

Petal provides highly available storage that can scale in throughput and capacity as resources are added to it. However, Petal has no provision for coordination or sharing the storage among multiple clients. Furthermore, most applications cannot directly use Petal's client interface because it is disk-like and not file-like. Frangipani provides a file system layer that makes Petal useful to applications while retaining and extending its good properties.

A strength of Frangipani is that it allows transparent server addition, deletion, and failure recovery. It is able to do this easily by combining write-ahead logging and locks with a uniformly accessible, highly available store.

Another strength of Frangipani is its ability to create consistent backups while the system is running. Frangipani's backup

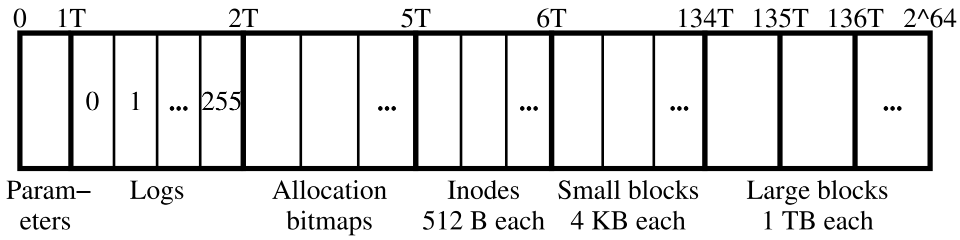

Figure 4: **Disk layout.** *Frangipani takes advantage of Petal's large, sparse disk address space to simplify its data structures. Each server has its own log and its own blocks of allocation bitmap space.*

mechanism is discussed in Section 8.

There are three aspects of the Frangipani design that can be problematic. Using Frangipani with a replicated Petal virtual disk implies that logging sometimes occurs *twice*, once to the Frangipani log, and once again within Petal itself. Second, Frangipani does not use disk location information in placing data—indeed it cannot—because Petal virtualizes the disks. Finally, Frangipani locks entire files and directories rather than individual blocks. We do not have enough usage experience to evaluate these aspects of our design in a general setting, but despite them, Frangipani's measured performance on the engineering workloads we have tested is good.

3 Disk Layout

Frangipani uses the large, sparse disk address space of Petal to simplify its data structures. The general idea is reminiscent of past work on programming computers with large memory address spaces [8]. There is so much address space available that it can be parcelled out generously.

A Petal virtual disk has 2^{64} bytes of address space. Petal commits physical disk space to virtual addresses only when they are written. Petal also provides a *decommit* primitive that frees the physical space backing a range of virtual disk addresses.

To keep its internal data structures small, Petal commits and decommits space in fairly large chunks, currently 64 KB. That is, each 64 KB range of addresses $[a \cdot 2^{16}, (a+1) \cdot 2^{16})$ in which some data has been written and not decommitted has 64 KB of physical disk space allocated to it. Thus Petal clients cannot afford to make their data structures *too* sparse, or too much physical disk space will be wasted through fragmentation. Figure 4 shows how Frangipani divides up its virtual disk space.

The first region stores shared configuration parameters and housekeeping information. We allow one terabyte (TB) of virtual space for this region, but in fact only a few kilobytes of it are currently used.

The second region stores logs. Each Frangipani server obtains a portion of this space to hold its private log. We have reserved one TB (2^{40} bytes) for this region, partitioned into 256 logs. This choice limits our current implementation to 256 servers, but this could easily be adjusted.

The third region is used for allocation bitmaps, to describe which blocks in the remaining regions are free. Each Frangipani server locks a portion of the bitmap space for its exclusive use. When a server's bitmap space fills up, it finds and locks another unused portion. The bitmap region is 3 TB long.

The fourth region holds inodes. Each file needs an inode to hold its metadata, such as timestamps and pointers to the location of its data.[2] Symbolic links store their data directly in the inode. We have made inodes 512 bytes long, the size of a disk block, thereby avoiding the unnecessary contention ("false sharing") between servers that would occur if two servers needed to access different inodes in the same block. We allocate one TB of inode space, allowing room for 2^{31} inodes. The mapping between bits in the allocation bitmap and inodes is fixed, so each Frangipani server allocates inodes to new files only from the portions of the inode space that corresponds to its portions of the allocation bitmap. But any Frangipani server may read, write, or free any existing file's inode.

The fifth region holds small data blocks, each 4 KB (2^{12} bytes) in size. The first 64 KB (16 blocks) of a file are stored in small blocks. If a file grows to more than 64 KB, the rest is stored in one large block. We allocate 2^{47} bytes for small blocks, thus allowing up to 2^{35} of them, 16 times the maximum number of inodes.

The remainder of the Petal address space holds large data blocks. One TB of address space is reserved for every large block.

Our disk layout policy of using 4 KB blocks can suffer from more fragmentation than a policy that more carefully husbands disk space. Also, allocating 512 bytes per inode is somewhat wasteful of space. We could alleviate these problems by storing small files in the inode itself [29]. What we gain with our design is simplicity, which we believe is a reasonable tradeoff for the cost of extra physical disk space.

The current scheme limits Frangipani to slightly less than 2^{24} (16 million) large files, where a large file is any file bigger than 64 KB. Also, no file can be larger than 16 small blocks plus one large block (64 KB plus 1 TB). If these limits prove too small, we could easily reduce the size of large blocks, thus making a larger number available, and permit large files to span more than one large block, thus raising the maximum file size. Should the 2^{64} byte address space limit prove inadequate, a single Frangipani server can support multiple Frangipani file systems on multiple virtual disks.

We have chosen these file system parameters based on our usage experience with earlier file systems. We believe our choices will serve us well, but only time and usage can confirm this. The design of Frangipani is flexible enough that we can experiment with different layouts at the cost of a backup and restore of the file system.

4 Logging and Recovery

Frangipani uses write-ahead redo logging of metadata to simplify failure recovery and improve performance; user data is not logged.

[2] In this section the word *file* includes directories, symbolic links, and the like.

Each Frangipani server has its own private log in Petal. When a Frangipani file server needs to make a metadata update, it first creates a record describing the update and appends it to its log in memory. These log records are periodically written out to Petal *in the same order* that the updates they describe were requested. (Optionally, we allow the log records to be written synchronously. This offers slightly better failure semantics at the cost of increased latency for metadata operations.) Only after a log record is written to Petal does the server modify the actual metadata in its permanent locations. The permanent locations are updated periodically (roughly every 30 seconds) by the Unix `update` demon.

Logs are bounded in size—128 KB in the current implementation. Given Petal's allocation policy, a log will be composed of two 64 KB fragments on two distinct physical disks. The space allocated for each log is managed as a circular buffer. When the log fills, Frangipani reclaims the oldest 25% of the log space for new log entries. Ordinarily, all the entries in the reclaimed area will refer to metadata blocks that have already been written to Petal (in a previous `sync` operation), in which case no additional Petal writes need to be done. If there are metadata blocks that have not yet been written, this work is completed before the log is reclaimed. Given the size of the log and typical sizes of Frangipani log records (80–128 bytes), the log can fill up between two periodic `sync` operations if there are about 1000–1600 operations that modify metadata in that interval.

If a Frangipani server crashes, the system eventually detects the failure and runs *recovery* on that server's log. Failure may be detected either by a client of the failed server, or when the lock service asks the failed server to return a lock it is holding and gets no reply. The recovery demon is implicitly given ownership of the failed server's log and locks. The demon finds the log's start and end, then examines each record in order, carrying out each described update that is not already complete. After log processing is finished, the recovery demon releases all its locks and frees the log. The other Frangipani servers can then proceed unobstructed by the failed server, and the failed server itself can optionally be restarted (with an empty log). As long as the underlying Petal volume remains available, the system tolerates an unlimited number of Frangipani server failures.

To ensure that recovery can find the end of the log (even when the disk controllers write data out of order), we attach a monotonically increasing log sequence number to each 512-byte block of the log. The end of the log can be reliably detected by finding a sequence number that is lower than the preceding one.

Frangipani ensures that logging and recovery work correctly in the presence of multiple logs. This requires attention to several details.

First, Frangipani's locking protocol, described in the next section, ensures that updates requested to the same data by different servers are serialized. A write lock that covers dirty data can change owners only after the dirty data has been written to Petal, either by the original lock holder or by a recovery demon running on its behalf. This implies that at most one log can hold an uncompleted update for any given block.

Second, Frangipani ensures that recovery applies only updates that were logged since the server acquired the locks that cover them, and for which it still holds the locks. This is needed to ensure that the serialization imposed by the locking protocol is not violated. We make this guarantee by enforcing a stronger condition: recovery never replays a log record describing an update that has already been completed. To accomplish the latter, we keep a version number on every 512-byte metadata block. Metadata such as directories, which span multiple blocks, have multiple version numbers. For each block that a log record updates, the record contains a description of the changes and the new version number. During recovery, the changes to a block are applied only if the block version number is less than the record version number.

Because user data updates are not logged, only metadata blocks have space reserved for version numbers. This creates a complication. If a block were used for metadata, freed, and then reused for user data, old log records referring to the block might not be skipped properly after the version number was overwritten with arbitrary user data. Frangipani avoids this problem by reusing freed metadata blocks only to hold new metadata.

Finally, Frangipani ensures that at any time only one recovery demon is trying to replay the log region of a specific server. The lock service guarantees this by granting the active recovery demon an exclusive lock on the log.

Frangipani's logging and recovery schemes assume that a disk write failure leaves the contents of a single sector in either the old state or the new state but never in a combination of both. If a sector is damaged such that reading it returns a CRC error, Petal's built-in replication can ordinarily recover it. If both copies of a sector were to be lost, or if Frangipani's data structures were corrupted by a software bug, a metadata consistency check and repair tool (like Unix *fsck*) would be needed. We have not implemented such a tool to date.

Frangipani's logging is not intended to provide high-level semantic guarantees to its users. Its purpose is to improve the performance of metadata updates and to speed up failure recovery by avoiding the need to run programs like *fsck* each time a server fails. Only metadata is logged, not user data, so a user has no guarantee that the file system state is consistent from his point of view after a failure. We do not claim these semantics to be ideal, but they are the same as what standard local Unix file systems provide. In both local Unix file systems and Frangipani, a user can get better consistency semantics by calling `fsync` at suitable checkpoints.

Frangipani's logging is an application of techniques first developed for databases [2] and later used in several other *log-based* file systems [9, 11, 16, 18]. Frangipani is not a *log-structured* file system [32]; it does not keep all its data in the log, instead maintaining conventional on-disk data structures, with a small log as an adjunct to provide improved performance and failure atomicity. Unlike the other log-based file systems cited above, but like the log-structured file systems Zebra [17] and xFS [1], Frangipani keeps multiple logs.

5 Synchronization and Cache Coherence

With multiple Frangipani servers all modifying shared on-disk data structures, careful synchronization is needed to give each server a consistent view of the data, and yet allow enough concurrency to scale performance as load is increased or servers are added. Frangipani uses multiple-reader/single-writer locks to implement the necessary synchronization. When the lock service detects conflicting lock requests, the current holder of the lock is asked to release or downgrade it to remove the conflict.

A *read lock* allows a server to read the associated data from disk and cache it. If a server is asked to release its read lock, it must invalidate its cache entry before complying. A *write lock* allows a server to read or write the associated data and cache it. A server's cached copy of a disk block can be different from the on-disk version only if it holds the relevant write lock. Thus if a server is asked to release its write lock or downgrade it to a read lock, it must write the dirty data to disk before complying. It can retain its cache entry if it is downgrading the lock, but must invalidate it if releasing the lock.

Instead of flushing the dirty data to disk when a write lock is released or downgraded, we could have chosen to bypass the disk and forward the dirty data directly to the requester. We did not do this for reasons of simplicity. First, in our design, Frangipani servers do not need to communicate with each other. They communicate only with Petal and the lock server. Second, our design ensures that when a server crashes, we need only process the log used by that server. If dirty buffers were directly forwarded and the destination server with the dirty buffer crashed, log entries referring to the dirty buffer could be spread out across several machines. This would pose a problem both for recovery and in reclaiming log space as it fills up.

We have divided the on-disk structures into logical segments with locks for each segment. To avoid false sharing, we ensure that a single disk sector does not hold more than one data structure that could be shared. Our division of on-disk data structures into lockable segments is designed to keep the number of locks reasonably small, yet avoid lock contention in the common case, so that the lock service is not a bottleneck in the system.

Each log is a single lockable segment, because logs are private. The bitmap space is also divided into segments that are locked exclusively, so that there is no contention when new files are allocated. A data block or inode that is not currently allocated to a file is protected by the lock on the segment of the allocation bitmap that holds the bit marking it as free. Finally, each file, directory, or symbolic link is one segment; that is, one lock protects both the inode and any file data it points to. This per-file lock granularity is appropriate for engineering workloads where files rarely undergo concurrent write-sharing. Other workloads, however, may require finer granularity locking.

Some operations require atomically updating several on-disk data structures covered by different locks. We avoid deadlock by globally ordering these locks and acquiring them in two phases. First, a server determines what locks it needs. This may involve acquiring and releasing some locks, to look up names in a directory, for example. Second, it sorts the locks by inode address and acquires each lock in turn. The server then checks whether any objects it examined in phase one were modified while their locks were released. If so, it releases the locks and loops back to repeat phase one. Otherwise, it performs the operation, dirtying some blocks in its cache and writing a log record. It retains each lock until the dirty blocks it covers are written back to disk.

The cache coherence protocol we have just described is similar to protocols used for client file caches in Echo [26], the Andrew File System [19], DCE/DFS [21], and Sprite [30]. The deadlock avoidance technique is similar to Echo's. Like Frangipani, the Oracle data base (Oracle Parallel Server), also writes dirty data to disk instead of using cache-to-cache transfers between successive owners of the write lock.

6 The Lock Service

Frangipani requires only a small, generic set of functions from its lock service, and we do not expect the service to be a performance bottleneck in normal operation, so many different implementations could fill its requirements. We have used three different lock service implementations in the course of the Frangipani project, and other existing lock services could provide the necessary functionality, perhaps with a thin layer of additional code on top.

The lock service provides multiple-reader/single-writer locks. Locks are sticky; that is, a client will generally retain a lock until some other client needs a conflicting one. (Recall that the clients of the lock service are the Frangipani servers.)

The lock service deals with client failure using *leases* [15, 26]. When a client first contacts the lock service, it obtains a lease. All locks the client acquires are associated with the lease. Each lease has an expiration time, currently set to 30 seconds after its creation or last renewal. A client must renew its lease before the expiration time, or the service will consider it to have failed.

Network failures can prevent a Frangipani server from renewing its lease even though it has not crashed. When this happens, the server discards all its locks and the data in its cache. If anything in the cache was dirty, Frangipani turns on an internal flag that causes all subsequent requests from user programs to return an error. The file system must be unmounted to clear this error condition. We have chosen this drastic way of reporting the error to make it difficult to ignore inadvertently.

Our initial lock service implementation was a single, centralized server that kept all its lock state in volatile memory. Such a server is adequate for Frangipani, because the Frangipani servers and their logs hold enough state information to permit recovery even if the lock service loses all its state in a crash. However, a lock service failure would cause a large performance glitch.

Our second implementation stored the lock state on a Petal virtual disk, writing each lock state change through to Petal before returning to the client. If the primary lock server crashed, a backup server would read the current state from Petal and take over to provide continued service. With this scheme, failure recovery is more transparent, but performance for the common case is poorer than the centralized, in-memory approach. We did not fully implement automatic recovery from all failure modes for this implementation before going on to the next one.

Our third and final lock service implementation is fully distributed for fault tolerance and scalable performance. It consists of a set of mutually cooperating lock servers, and a clerk module linked into each Frangipani server.

The lock service organizes locks into *tables* named by ASCII strings. Individual locks within tables are named by 64-bit integers. Recall that a single Frangipani file system uses only one Petal virtual disk, although multiple Frangipani file systems can be mounted on the same machine. Each file system has a table associated with it. When a Frangipani file system is mounted, the Frangipani server calls into the clerk, which opens the lock table associated with that file system. The lock server gives the clerk a *lease identifier* on a successful open, which is used in all subsequent communication between them. When the file system is unmounted, the clerk closes the lock table.

Clerks and the lock servers communicate via asynchronous messages rather than RPC to minimize the amount of memory used and to achieve good flexibility and performance. The basic message types that operate on locks are *request, grant, revoke,* and *release*. The *request* and *release* message types are sent from the clerk to the lock server, whereas the *grant* and *revoke* message types are sent from the lock server to the clerk. Lock upgrade and downgrade operations are also handled using these four message types.

The lock service uses a fault-tolerant, distributed failure detection mechanism to detect the crash of lock servers. This is the same mechanism used by Petal. It is based on the timely exchange of heartbeat messages between sets of servers. It uses majority consensus to tolerate network partitions.

Locks consume memory at the server and at each clerk. In our current implementation, the server allocates a block of 112 bytes per lock, in addition to 104 bytes per clerk that has an outstanding or granted lock request. Each client uses up 232 bytes per lock. To avoid consuming too much memory because of sticky locks, clerks discard locks that have not been used for a long time (1 hour).

A small amount of global state information that does not change often is consistently replicated across all lock servers using Lamport's Paxos algorithm [23]. The lock service reuses an implementation of Paxos originally written for Petal. The global state information consists of a list of lock servers, a list of locks that each is responsible for serving, and a list of clerks that have opened but not yet closed each lock table. This information is used to achieve consensus, to reassign locks across lock servers, to recover lock state from clerks after a lock server crash, and to facilitate recovery of Frangipani servers. For efficiency, locks are partitioned into about one hundred distinct *lock groups*, and are assigned to servers by group, not individually.

Locks are occasionally reassigned across lock servers to compensate for a crashed lock server or to take advantage of a newly recovered lock server. A similar reassignment occurs when a lock server is permanently added to or removed from the system. In such cases, the locks are always reassigned such that the number of locks served by each server is balanced, the number of reassignments is minimized, and each lock is served by exactly one lock server. The reassignment occurs in two phases. In the first phase, lock servers that lose locks discard them from their internal state. In the second phase, lock servers that gain locks contact the clerks that have the relevant lock tables open. The servers recover the state of their new locks from the clerks, and the clerks are informed of the new servers for their locks.

When a Frangipani server crashes, the locks that it owns cannot be released until appropriate recovery actions have been performed. Specifically, the crashed Frangipani server's log must be processed and any pending updates must be written to Petal. When a Frangipani server's lease expires, the lock service will ask the clerk on another Frangipani machine to perform recovery and to then release all locks belonging to the crashed Frangipani server. This clerk is granted a lock to ensure exclusive access to the log. This lock is itself covered by a lease so that the lock service will start another recovery process should this one fail.

In general, the Frangipani system tolerates network partitions, continuing to operate when possible and otherwise shutting down cleanly. Specifically, Petal can continue operation in the face of network partitions, as long as a majority of the Petal servers remain up and in communication, but parts of the Petal virtual disk will be inaccessible if there is no replica in the majority partition. The lock service continues operation as long as a majority of lock servers are up and in communication. If a Frangipani server is partitioned away from the lock service, it will be unable to renew its lease. The lock service will declare such a Frangipani server dead and initiate recovery from its log on Petal. If a Frangipani server is partitioned away from Petal, it will be unable to read or write the virtual disk. In either of these cases, the server will disallow further user access to the affected file system until the partition heals and the file system is remounted.

There is a small hazard when a Frangipani server's lease expires. If the server did not really crash, but was merely out of contact with the lock service due to network problems, it may still try to access Petal after its lease has expired. A Frangipani server checks that its lease is still valid (and will still be valid for t_{margin} seconds) before attempting any write to Petal. Petal, however, does no checking when a write request arrives. Thus, if there is a sufficient time delay between Frangipani's lease check and the arrival of the subsequent write request at Petal, we could have a problem: The lease could have expired and the lock been given to a different server. We use a large enough error margin t_{margin} (15 seconds) that under normal circumstances this problem would never occur, but we cannot rule it out absolutely.

In the future we would like to eliminate this hazard; one method that would work is as follows. We add an *expiration timestamp* on each write request to Petal. The timestamp is set to the current lease expiration time at the moment the write request is generated, minus t_{margin}. We then have Petal ignore any write request with a timestamp less than the current time. This method reliably rejects writes with expired leases, provided that the clocks on Petal and Frangipani servers are synchronized to within t_{margin}.

Another method, which does not required synchronized clocks, is to integrate the lock server with Petal and include the *lease identifier* obtained from the lock server with every write request to Petal. Petal would then reject any write request with an expired lease identifier.

7 Adding and Removing Servers

As a Frangipani installation grows and changes, the system administrator will occasionally need to add or remove server machines. Frangipani is designed to make this task easy.

Adding another Frangipani server to a running system requires a minimal amount of administrative work. The new server need only be told which Petal virtual disk to use and where to find the lock service. The new server contacts the lock service to obtain a lease, determines which portion of the log space to use from the lease identifier, and goes into operation. The administrator does not need to touch the other servers; they adapt to the presence of the new one automatically.

Removing a Frangipani server is even easier. It is adequate to simply shut the server off. It is preferable for the server to flush all its dirty data and release its locks before halting, but this is not strictly needed. If the server halts abruptly, recovery will run on its log the next time one of its locks is needed, bringing the shared disk into a consistent state. Again, the administrator does not need to touch the other servers.

Petal servers can also be added and removed transparently, as described in the Petal paper [24]. Lock servers are added and removed in a similar manner.

8 Backup

Petal's snapshot feature provides us with a convenient way to make consistent full dumps of a Frangipani file system. Petal allows a client to create an exact copy of a virtual disk at any point in time. The snapshot copy appears identical to an ordinary virtual disk, except that it cannot be modified. The implementation uses copy-on-write techniques for efficiency. The snapshots are *crash-consistent;* that is, a snapshot reflects a coherent state, one that the Petal virtual disk could have been left in if all the Frangipani servers were to crash.

Hence we can backup a Frangipani file system simply by taking a Petal snapshot and copying it to tape. The snapshot will include all the logs, so it can be restored by copying it back to a new Petal virtual disk and running recovery on each log. Due to the crash-consistency, restoring from a snapshot reduces to the same problem as recovering from a system-wide power failure.

We could improve on this scheme with a minor change to Frangipani, creating snapshots that are consistent at the file system level and require no recovery. We can accomplish this by having the backup program force all the Frangipani servers into a barrier, implemented using an ordinary global lock supplied by the lock service. The Frangipani servers acquire this lock in shared mode to do any modification operation, while the backup process requests it in exclusive mode. When a Frangipani server receives a request to release the barrier lock, it enters the barrier by blocking all new file system calls that modify data, cleaning all dirty data in its cache and then releasing the lock. When all the Frangipani servers have entered the barrier, the backup program is able to acquire the exclusive lock; it then makes a Petal snapshot and releases the lock. At this point the servers reacquire the lock in shared mode, and normal operation resumes.

With the latter scheme, the new snapshot can be mounted as a Frangipani volume with no need for recovery. The new volume can be accessed on-line to retrieve individual files, or it can be dumped to tape in a conventional backup format that does not require Frangipani for restoration. The new volume must be mounted read-only, however, because Petal snapshots are currently read-only. In the future we may extend Petal to support writable snapshots, or we may implement a thin layer on top of Petal to simulate them.

9 Performance

Frangipani's layered structure has made it easier to build than a monolithic system, but one might expect the layering to exact a cost in performance. In this section we show that Frangipani's performance is good in spite of the layering.

As in other file systems, latency problems in Frangipani can be solved straightforwardly by adding a non-volatile memory (NVRAM) buffer in front of the disks. The most effective place to put NVRAM in our system is directly between the physical disks and the Petal server software. Ordinary PrestoServe cards and drivers suffice for this purpose, with no changes to Petal or Frangipani needed. Failure of the NVRAM on a Petal server is treated by Petal as equivalent to a server failure.

Several aspects of Frangipani and Petal combine to provide good scaling of throughput. There is parallelism at both layers of the system: multiple Frangipani servers, multiple Petal servers, and multiple disk arms all working in parallel. When many clients are using the system, this parallelism increases the aggregate throughput. As compared with a centralized network file server, Frangipani should have less difficulty dealing with hot spots, because file system processing is split up and shifted to the machines that are using the files. Both the Frangipani and Petal logs can commit updates from many different clients in one log write (group commit), providing improved log throughput under load. Individual clients doing large writes also benefit from parallelism, due to Petal's striping of data across multiple disks and servers.

9.1 Experimental Setup

We are planning to build a large storage testbed with about 100 Petal nodes attached to several hundred disks and about 50 Frangipani servers. Petal nodes will be small array controllers attached to off-the-shelf disks and to the network. Frangipani servers will be typical workstations. This testbed would allow us to study the performance of Frangipani in a large configuration. Since this is not yet ready, we report numbers from a smaller configuration.

For the measurements reported below, we used seven 333 MHz DEC Alpha 500 5/333 machines as Petal servers. Each machine stores data on 9 DIGITAL RZ29 disks, which are 3.5 inch fast SCSI drives storing 4.3 GB each, with 9 ms average seek time and 6 MB/s sustained transfer rate. Each machine is connected to a 24 port ATM switch by its own 155 Mbit/s point-to-point link. PrestoServe cards containing 8 MB of NVRAM were used on these servers where indicated below. The seven Petal servers can supply data at an aggregate rate of 100 MB/s. With replicated virtual disks, Petal servers can sink data at an aggregate rate of 43 MB/s.

9.2 Single Machine Performance

This subsection compares how well Frangipani's code path compares with another Unix vnode file system, namely DIGITAL's Advanced File System (AdvFS).

We used AdvFS for our comparison rather than the more familiar BSD-derived UFS file system [27] because AdvFS is significantly faster than UFS. In particular, AdvFS can stripe files across multiple disks, thereby achieving nearly double the throughput of UFS on our test machines. Also, unlike UFS, which synchronously updates metadata, AdvFS uses a write-ahead log like Frangipani. This significantly reduces the latency of operations like file creation. Both AdvFS and UFS have similar performance on reading small files and directories.

We ran AdvFS and Frangipani file systems on two identical machines with storage subsystems having comparable I/O performance. Each machine has a 225 MHz DEC Alpha 3000/700 CPU with 192 MB of RAM, which is managed by the unified buffer cache (UBC). Each is connected to the ATM switch by its own point-to-point link.

The Frangipani file system does not use local disks, but accesses a replicated Petal virtual disk via the Petal device driver. When accessed through the raw device interface using block sizes of 64 KB, the Petal driver can read and write data at about 16 MB/s, saturating the ATM link to the Petal server. CPU utilization is about 4%. The read latency of a Petal disk is about 11 ms.

The AdvFS file system uses a storage subsystem that has performance roughly equivalent to the Petal configuration we use. It consists of 8 DIGITAL RZ29 disks connected via two 10 MB/s fast SCSI strings to two backplane controllers. When accessed through the raw device interface, the controllers and disks can supply data at about 17 MB/s with 4% CPU utilization. Read latency is about 10 ms. (We could have connected the AdvFS file system to a Petal virtual disk to ensure both file systems were using identical storage subsystems. Previous experiments [24] have shown that AdvFS would have been about 4% slower if run on Petal. To present AdvFS in the best light, we chose not to do this.)

It is not our intention to compare Petal's cost/performance with that of locally attached disks. Clearly, the hardware resources required to provide the storage subsystems for Frangipani and AdvFS are vastly different. Our goal is to demonstrate that the Frangipani code path is efficient compared to an existing, well-tuned commercial file system. The hardware resources we use for Petal are non-trivial, but these resources are amortized amongst multiple Frangipani servers.

Tables 1 and 2 compare performance of the two systems on standard benchmarks. Each table has four columns. In the **AdvFS Raw** column, the benchmark was run with AdvFS directly accessing the local disks. In the **AdvFS NVR** column, the benchmark was rerun with NVRAM interposed in front of the local disks. In the **Frangipani Raw** column, the benchmark was run with Frangipani accessing Petal via the device interface. In the **Frangipani NVR** column, the Frangipani configuration was retested with the addition of an NVRAM buffer between Petal and the disks. All numbers are averaged over ten runs of the benchmarks. Standard deviation is less than 12% of the mean in all cases.

		AdvFS		Frangipani	
Phase	**Description**	**Raw**	**NVR**	**Raw**	**NVR**
1	Create Directories	0.69	0.66	0.52	0.51
2	Copy Files	4.3	4.3	5.8	4.6
3	Directory Status	4.7	4.4	2.6	2.5
4	Scan Files	4.8	4.8	3.0	2.8
5	Compile	27.8	27.7	31.8	27.8

Table 1: **Modified Andrew Benchmark with unmount operations.** *We compare the performance of two file system configurations: local access (with and without NVRAM) to the DIGITAL Unix Advanced File System (AdvFS), Frangipani, and Frangipani with an NVRAM buffer added between Petal and the disks. We unmount the file system at the end of each phase. Each table entry is an average elapsed time in seconds; smaller numbers are better.*

Table 1 gives results from the Modified Andrew Benchmark, a widely used file system benchmark. The first phase of the benchmark creates a tree of directories. The second phase copies a 350 KB collection of C source files into the tree. The third phase traverses the new tree and examines the status of each file and directory. The fourth phase reads every file in the new tree. The fifth phase compiles and links the files.

Unfortunately, it is difficult to make comparative measurements using the Modified Andrew Benchmark in its standard form. This is because the benchmark does not account for work that is deferred by the file system implementation. The work deferred during one phase of the benchmark can be performed during a later phase and thus inappropriately charged to that phase, while some work can be deferred past the end of the benchmark and thus never accounted for.

Like traditional Unix file systems, both AdvFS and Frangipani defer the cost of writing dirty file data until the next `sync` operation, which may be explicitly requested by a user or triggered in the background by a periodic update demon. However, unlike traditional Unix file systems, both AdvFS and Frangipani are log-based and do not write metadata updates synchronously to disk. Instead, metadata updates are also deferred until the next `sync`, or at least until the log wraps.

In order to account properly for all sources of deferred work, we changed the benchmark to unmount the file system after each phase. We chose to unmount rather than to use a `sync` call because on Digital Unix, `sync` queues the dirty data for writing but does not guarantee it has reached disk before returning. The results, shown in Table 1, indicate that Frangipani is comparable to AdvFS in all phases.

Table 2 shows the results of running the Connectathon Benchmark. The Connectathon benchmark tests individual operations or small groups of related operations, providing more insight into the sources of the differences that are visible in the Andrew benchmark. Like the Andrew benchmark, this benchmark also does not account for deferred work, so again we unmounted the file system at the end of each phase.

Frangipani latencies with NVRAM are roughly comparable to

		AdvFS		Frangipani	
Test	**Description**	**Raw**	**NVR**	**Raw**	**NVR**
1	file and directory creation: creates 155 files and 62 directories.	0.92	0.80	3.11	2.37
2	file and directory removal: removes 155 files and 62 62 directories.	0.62	0.62	0.43	0.43
3	lookup across mount point: 500 getwd and stat calls.	0.56	0.56	0.43	0.40
4	setattr, getattr, and lookup: 1000 chmods and stats on 10 files.	0.42	0.40	1.33	0.68
5a	write: writes a 1048576 byte file 10 times.	2.20	2.16	2.59	1.63
5b	read: reads a 1048576 byte file 10 times.	0.54	0.45	1.81	1.83
6	readdir: reads 20500 directory entries, 200 files.	0.58	0.58	2.63	2.34
7	link and rename: 200 renames and links on 10 files.	0.47	0.44	0.60	0.50
8	symlink and readlink: 400 symlinks and readlinks on 10 files.	0.93	0.82	0.52	0.50
9	statfs: 1500 statfs calls.	0.53	0.49	0.23	0.22

Table 2: **Connectathon Benchmark with unmount operations.** *We run the Connectathon Benchmark with a unmount operation included at the end of each test. Each table entry is an average elapsed time in seconds, and smaller numbers are better. Test 5b is anomalous due to a bug in AdvFS.*

that of AdvFS with four notable exceptions. Tests 1, 4, and 6 indicate that creating files, setting attributes, and reading directories take significantly longer with Frangipani. In practice, however, these latencies are small enough to be ignored by users, so we have not tried very hard to optimize them.

File creation takes longer with Frangipani partly because the 128 KB log fills up several times during this test. If we double the log size, the times reduce to 0.89 and 0.86 seconds.

Frangipani is much slower on the file read test (5b). AdvFS does well on the file read test because of a peculiar artifact of its implementation. On each iteration of the read test, the benchmark makes a system call to invalidate the file from the buffer cache before reading it in. The current AdvFS implementation appears to ignore this invalidation directive. Thus the read test measures the performance of AdvFS reading from the cache rather than from disk. When we redid this test with a cold AdvFS file cache, the performance was similar to Frangipani's (1.80 seconds, with or without NVRAM).

We next report on the throughput achieved by a single Frangipani server when reading and writing large files. The file reader sits in a loop reading a set of 10 files. Before each iteration of the loop, it flushes the contents of the files from the buffer cache. The file writer sits in a loop repeatedly writing a large (350 MB) private file. The file is large enough that there is a steady stream of write traffic to disk. Both read and write tests were run for several minutes and we observed no significant variation in the throughput. The time-averaged, steady state results are summarized in Table 3. The presence or absence of NVRAM has little effect on the timing.

A single Frangipani machine can write data at about 15.3 MB/s,

	Throughput (MB/s)		CPU Utilization	
	Frangipani	AdvFS	Frangipani	AdvFS
Write	15.3	13.3	42%	80%
Read	10.3	13.2	25%	50%

Table 3: **Frangipani Throughput and CPU Utilization.** *We show the performance of Frangipani in reading and writing large files.*

which is about 96% of the limit imposed by the ATM link and UDP/IP software on our machine. Frangipani achieves good performance by clustering writes to Petal into naturally aligned 64 KB blocks. It is difficult make up the last 4% because Frangipani occasionally (e.g., during `sync`) must write part of the data out in smaller blocks. Using smaller block sizes reduces the maximum available throughput through the UDP/IP stack. The Frangipani server CPU utilization is about 42%, and the Petal server CPUs are not a bottleneck.

A single Frangipani machine can read data at 10.3 MB/s with 25% CPU utilization. We believe this performance can be improved by changing the read-ahead algorithm used in Frangipani. Frangipani currently uses a read-ahead algorithm borrowed from the BSD-derived file system UFS, which is less effective than the one used by AdvFS.

For comparison, AdvFS can write data at about 13.3 MB/s when accessing large files that are striped over the eight RZ29 disks connected to the two controllers. The CPU utilization is about 80%. The AdvFS read performance is about 13.2 MB/s, at a CPU utilization of 50%. Neither the CPU nor the controllers are bottlenecked, so we believe AdvFS performance could be improved a bit with more tuning.

It is interesting to note that although Frangipani uses a simple policy to lay out data, its latency and write throughput are comparable to those of conventional file systems that use more elaborate policies.

Frangipani has good write latency because the latency-critical metadata updates are logged asynchronously rather than being performed synchronously in place. File systems like UFS that synchronously update metadata have to be more careful about data placement. In separate experiments not described here, we have found that even when Frangipani updates its logs synchronously, performance is still quite good because the log is allocated in large physically contiguous blocks and because the NVRAM absorbs much of the write latency.

Frangipani achieves good write throughput because large files are physically striped in contiguous 64 KB units over many disks and machines, and Frangipani can exploit the parallelism inherent in this structure. Frangipani has good read throughput for large files for the same reason.

Recall from Section 3 that individual 4 KB blocks for files smaller than 64 KB may not be allocated contiguously on disk. Also, Frangipani does not do read-ahead for small files, so it cannot always hide the disk seek access times. Thus it is possible that Frangipani could have bad read performance on small files. To quantify small read performance, we ran an experiment where 30 processes on a single Frangipani machine tried to read separate 8 KB files after invalidating the buffer cache. Frangipani throughput was 6.3 MB/s, with the CPU being the bottleneck. Petal, accessed through the raw device interface using 4 KB blocks, can deliver 8 MB/s. Thus Frangipani gets about 80% of the maximum throughput achievable in this case.

9.3 Scaling

This section studies the scaling characteristics of Frangipani. Ideally, we would like to see operational latencies that are unchanged and throughput that scales linearly as servers are added.

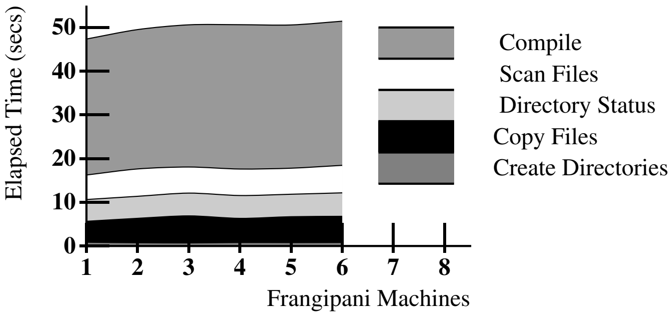

Figure 5: **Frangipani Scaling on Modified Andrew Benchmark.** *Several Frangipani servers simultaneously run the Modified Andrew Benchmark on independent data sets. The y-axis gives the average elapsed time taken by one Frangipani machine to complete the benchmark.*

Figure 5 shows the effect of scaling on Frangipani running the Modified Andrew Benchmark. In this experiment, we measure the average time taken by one Frangipani machine to complete the benchmark as the number of machines is increased. This experiment simulates the behavior of several users doing program development on a shared data pool. We notice that there is minimal negative impact on the latency as Frangipani machines are added. In fact, between the single machine and six machine experiment, the average latency increased by only 8%. This is not surprising because the benchmark exhibits very little write sharing and we would expect latencies to remain unaffected with increased servers.

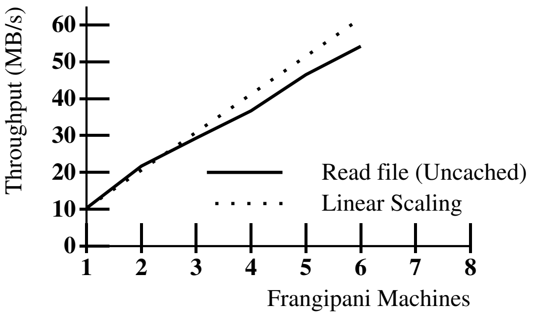

Figure 6: **Frangipani Scaling on Uncached Read.** *Several Frangipani servers simultaneously read the same set of files. The dotted line shows the linear speedup curve for comparison.*

Figure 6 illustrates Frangipani's read throughput on uncached data. In this test, we replicate the reader from the single-server experiment on multiple servers. The test runs for several minutes, and we observe negligible variation in the steady-state throughput. As indicated in the figure, Frangipani shows excellent scaling in this test. We are in the process of installing Frangipani on more machines, and we expect aggregate read performance to increase until it saturates the Petal servers' capacity.

Figure 7 illustrates Frangipani's write throughput. Here the writer from the single-server experiment is replicated on multiple servers. Each server is given a distinct large file. The experiment

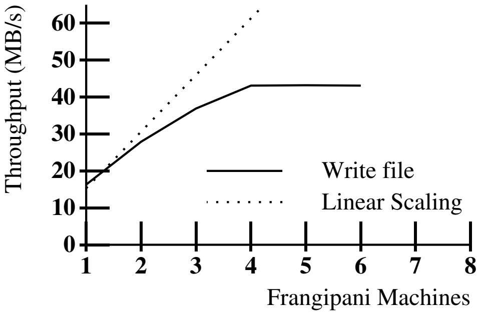

Figure 7: **Frangipani Scaling on Write.** *Each Frangipani server writes a large private file. The dotted line shows the linear speedup curve for comparison. Performance tapers off early because the ATM links to the Petal servers become saturated.*

runs for several minutes, and we observe little variation in the steady-state throughput during this interval. Since there is no lock contention in the experiment, the performance is seen to scale well until the ATM links to the Petal servers are saturated. Since the virtual disk is replicated, each write from a Frangipani server turns into two writes to the Petal servers.

9.4 Effects of Lock Contention

Since Frangipani uses coarse-grained locking on entire files, it is important to study the effect of lock contention on performance. We report three experiments here.

The first experiment measures the effect of read/write sharing on files. One or more readers compete against a single writer for the same large file. Initially, the file is not cached by the readers or the writer. The readers read the file sequentially, while the writer rewrites the entire file. As a result, the writer repeatedly acquires the write lock, then gets a callback to downgrade it so that the readers can get the read lock. This callback causes the writer to flush data to disk. At the same time, each reader repeatedly acquires the read lock, then gets a callback to release it so that the writer can get the write lock. This callback causes the reader to invalidate its cache, so its next read after reacquiring the lock must fetch the data from disk.

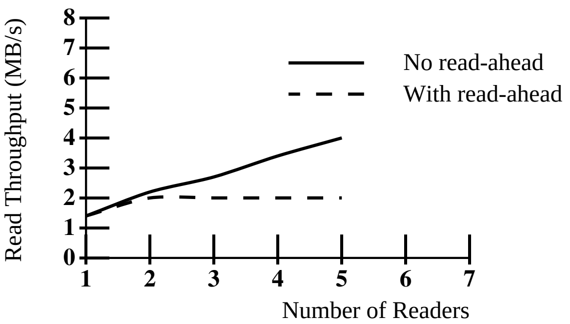

Figure 8: **Frangipani Reader/Writer Contention.** *One or more Frangipani servers read a shared file while a single Frangipani server writes the same file. We show the effect of read-ahead on the performance.*

The first results we observed in this experiment were unexpected. Our distributed lock manager has been designed to be fair in granting locks, and simulations show that this is true of the implementation. If the single writer and the n readers were to make lock requests at a uniform rate, they would be serviced in a round-robin fashion, so successive grants of the write lock to the writer would be separated by n grants of the read lock to the readers. During the interval between two downgrade callbacks, one would expect the number of read requests and the aggregate read throughput to increase as readers were added. In the limit when n is large, the scaling would be linear. However, we did not observe this behavior in our experiment. Instead, read throughput flattens out at about 2 MB/s after two readers are running, as shown by the dashed line in Figure 8. As indicated earlier in Figure 6, this is only about 10% of what two Frangipani servers can achieve when there is no lock contention.

We conjectured that this anomalous behavior was caused by read-ahead, so we repeated the experiment without read-ahead to check. Read-ahead is disadvantageous in the presence of heavy read/write contention because when a reader is called back to release its lock, it must invalidate its cache. If there is any read-ahead data in the cache that has not yet been delivered to the client, it must be discarded, and the work to read it turns out to have been wasted. Because the readers are doing extra work, they cannot make lock requests at the same rate as the writer. Redoing the experiment with read-ahead disabled yielded the expected scaling result, as shown by the solid line in Figure 8.

We could make this performance improvement available to users either by letting them explicitly disable read-ahead on specific files, or by devising a heuristic that would recognize this case and disable read-ahead automatically. The former would be trivial to implement, but would affect parts of the operating system kernel beyond Frangipani itself, making it inconvenient to support across future releases of the kernel. The latter approach seems better, but we have not yet devised or tested appropriate heuristics.

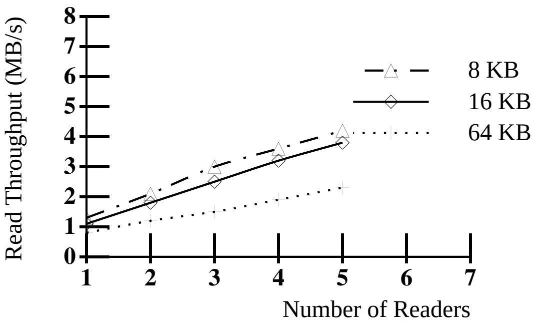

Figure 9: **Effect of Data Size on Reader/Writer Contention.** *One or more Frangipani readers share varying amounts of data with a Frangipani writer. Readahead is disabled in this experiment.*

The second experiment is a variation of the first. Here, the readers run as before, but the writer modifies different amounts of file data. Since Frangipani locks entire files, readers will have to invalidate their entire cache irrespective of the writer's behavior. However, readers will be able to acquire a lock faster when the writer is updating fewer blocks of data because the writer must flush only smaller amounts of data to disk. Figure 9 shows the performance of Frangipani (with read-ahead disabled) when readers and the writer concurrently share differing amounts of data. As expected, when the shared data is smaller, we get better performance.

The third experiment measures the effects of write/write sharing on files. As the base case, a Frangipani server writes a file in isolation. We then added Frangipani servers that wrote the same file and measured the degradation in performance. Writers modify file data in blocks of 64 KB. Since Frangipani does whole-file locking, the offsets that the writers use are irrelevant for this test. We found that the aggregate bandwidth seen by all the writers dropped from 15 MB/s for the single-writer case to a little over 1 MB/s with two or more writers. This is not surprising, because with multiple writers trying to modify a file, nearly every `write` system call will cause a lock revocation request. This revocation request causes the lock holder to flush its dirty data to Petal. Since locks are being revoked on every `write` system call and each call dirties only 64 KB of data, throughput is quite limited. With smaller block sizes, throughput is even smaller.

We do not have much experience with workloads that exhibit concurrent write sharing. If necessary, we believe it would be straightforward to extend Frangipani to implement byte-range locking [6] or block locking instead. This would improve the performance of workloads that read and write different parts of the same file, making it similar to the performance of writing different files in the current system. Workloads in which multiple machines concurrently read and write the same blocks of the same file—where the filesystem is being used as an interprocess communication channel—would perform as indicated above. Frangipani is simply not targeted for such workloads.

10 Related Work

Like Frangipani, the Cambridge (or Universal) File Server takes a two-layered approach to building a file system [4, 28]. The split between layers is quite different from ours, however. CFS, the lower layer, provides its clients with two abstractions: *files* and *indices*. File systems built above CFS can use these abstractions to implement files and directories. A major difference between CFS and Petal is that in CFS a single machine manages all the storage.

NFS [31, 33] is not a file system in itself, but simply a remote file access protocol. The NFS protocol provides a weak notion of cache coherence, and its stateless design requires clients to access servers frequently to maintain even this level of coherence. Frangipani provides a strongly coherent, single system view, using a protocol that maintains more state but eliminates unnecessary accesses to servers.

The Andrew File System (AFS) [19] and its offshoot DCE/DFS [21] provide better cache performance and coherence than NFS. AFS is designed for a different kind of scalability than Frangipani. Frangipani provides a unified cluster file system that draws from a single pool of storage, and can be scaled up to span many disk drives across many machines under a common administration. In contrast, AFS has a global name space and security architecture that allows one to plug in many separate file servers and clients over a wide area. We believe the AFS and Frangipani approaches to scaling are complementary; it would make good sense for Frangipani servers to export the file system to wide-area clients using the AFS or DCE/DFS name space and access protocol.

Like Frangipani, the Echo file system [5, 18, 26, 35] is log-based, replicates data for reliability and access paths for availability, permits volumes to span multiple disks, and provides coherent caching. Echo does not share Frangipani's scalability, however. Each Echo volume can be managed by only one server at a time, with failover to one designated backup. A volume can span only as many disks as can be connected to a single machine. There is an internal layering of file service atop disk service, but the Echo implementation requires both layers to run in the same address space on the same machine, and experience with Echo showed the server CPU to be a bottleneck.

The VMS Cluster file system [14] offloads file system processing to individual machines that are members of a cluster, much as Frangipani does. Each cluster member runs its own instance of the file system code on top of a shared physical disk, with synchronization provided by a distributed lock service. The shared physical disk is accessed either through a special-purpose cluster interconnect to which a disk controller can be directly connected, or through an ordinary network such as Ethernet and a machine acting as a disk server. Frangipani improves upon this design in several ways: The shared physical disk is replaced by a shared scalable virtual disk provided by Petal, the Frangipani file system is log-based for quick failure recovery, and Frangipani provides extensive caching of both data and metadata for better performance.

The Spiralog file system [20] also offloads its file system processing to individual cluster members, which run above a shared storage system layer. The interface between layers in Spiralog differs both from the original VMS Cluster file system and from Petal. The lower layer is neither file-like nor simply disk-like; instead, it provides an array of stably-stored bytes, and permits atomic actions to update arbitrarily scattered sets of bytes within the array. Spiralog's split between layers simplifies the file system, but complicates the storage system considerably. At the same time, Spiralog's storage system does not share Petal's scalability or fault tolerance; a Spiralog volume can span only the disks connected to one machine, and becomes unavailable when that machine crashes.

Though designed as a cluster file system, Calypso [11] is similar to Echo, not to VMS Clusters or Frangipani. Like Echo, Calypso stores its files on multiported disks. One of the machines directly connected to each disk acts as a file server for data stored on that disk; if that machine fails, another takes over. Other members of the Calypso cluster access the current server as file system clients. Like both Frangipani and Echo, the clients have caches, kept coherent with a multiple-reader/single-writer locking protocol.

For comparison purposes, the authors of Calypso also built a file system in the shared-disk style, called PJFS [12]. Calypso performed better than PJFS, leading them to abandon the shared-disk approach. PJFS differs from Frangipani in two main respects. First, its lower layer is a centralized disk server, not a distributed virtual disk like Petal. Second, all file server machines in PJFS share a common log. The shared log proved to be a performance bottleneck. Like Frangipani, PJFS locks the shared disk at whole-file granularity. This granularity caused performance problems with workloads where large files were concurrently write-shared among multiple nodes. We expect the present Frangipani implementation to have similar problems with such workloads, but as noted in Section 9.4 above, we could adopt byte-range locking instead.

Shillner and Felten have built a distributed file system on top of a shared logical disk [34]. The layering in their system is similar to ours: In the lower layer, multiple machines cooperate to implement a single logical disk. In the upper layer, multiple independent machines run the same file system code on top of one logical disk, all providing access to the same files. Unlike Petal, their logical disk layer does not provide redundancy. The system can recover when a node fails and restarts, but it cannot dynamically configure out failed nodes or configure in additional nodes. Their file system uses careful ordering of metadata writes, not logging as Frangipani does. Like logging, their technique avoids the need for a full metadata scan (*fsck*) to restore consistency after a server crash, but unlike logging, it can lose track of free blocks in a crash, necessitating an occasional garbage collection scan to find them again. We are unable to compare the performance of their system with ours at present, as performance numbers for their file system layer are not available.

The xFS file system [1, 36] comes closest in spirit to Frangipani. In fact, the goals of the two systems are essentially the same. Both try to distribute the management responsibility for files over multiple machines and to provide good availability and performance. Frangipani is effectively "serverless" in the same sense as xFS—the service is distributed over all machines, and can be configured with both a Frangipani server and Petal server on each machine. Frangipani's locking is coarser-grained that xFS, which supports block-level locking.

Our work differs from xFS in two principal ways:

First, the internal organization of our file system and its interface to the storage system are significantly different from xFS's. Unlike Frangipani, xFS has a predesignated manager for each file, and its storage server is log-structured. In contrast, Frangipani is organized as a set of cooperating machines that use Petal as a shared store with a separate lock service for concurrency control. Ours is a simpler model, reminiscent of multithreaded shared memory programs that communicate via a common store and use locks for synchronization. This model allows us to deal with file system recovery and server addition and deletion with far less machinery than xFS requires, which has made our system easier to implement and test.

Second, we have addressed file system recovery and reconfiguration. These issues have been left as open problems by the xFS work to date.

We would have liked to compare Frangipani's performance with that of xFS, but considerable performance work remains to be completed on the current xFS prototype [1]. A comparison between the systems at this time would be premature and unfair to xFS.

11 Conclusions

The Frangipani file system provides all its users with coherent, shared access to the same set of files, yet is scalable to provide more storage space, higher performance, and load balancing as the user community grows. It remains available in spite of component failures. It requires little human administration, and administration does not become more complex as more components are added to a growing installation.

Frangipani was feasible to build because of its two-layer structure, consisting of multiple file servers running the same simple file system code on top of a shared Petal virtual disk. Using Petal as a lower layer provided several benefits. Petal implements data replication for high availability, obviating the need for Frangipani to do so. A Petal virtual disk is uniformly accessible to all Frangipani servers, so that any server can serve any file, and any machine can run recovery when a server fails. Petal's large, sparse address space allowed us to simplify Frangipani's on-disk data structures.

Despite Frangipani's simple data layout and allocation policy and coarse-grained locking, we have been happy with its performance. In our initial performance measurements, Frangipani is already comparable to a production DIGITAL Unix file system, and we expect improvement with further tuning. Frangipani has shown good scaling properties up to the size of our testbed configuration (seven Petal nodes and six Frangipani nodes). The results leave us optimistic that the system will continue to scale up to many more nodes.

Our future plans include deploying Frangipani for our own day-to-day use. We hope to gain further experience with the prototype under load, to validate its scalability by testing it in larger configurations, to experiment with finer-grained locking, and to complete our work on backup. Finally, of course, we would like to see the ideas from Frangipani make their way into commercial products.

Acknowledgments

We thank Mike Burrows, Mike Schroeder, and Puneet Kumar for helpful advice on the Frangipani design and comments on this paper. Fay Chang implemented an early prototype of Frangipani as a summer project. The anonymous referees and our shepherd, John Wilkes, suggested many improvements to the paper. Cynthia Hibbard provided editorial assistance.

References

[1] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.

[2] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] Anupam Bhide, Elmootazbellah N. Elnozahy, and Stephen P. Morgan. A highly available network file server. In *Proceedings of the Winter USENIX Conference*, pages 199–205, January 1991.

[4] A. D. Birrell and R. M. Needham. A universal file server. *IEEE Transactions on Software Engineering*, SE-6(5):450–453, September 1980.

[5] Andrew D. Birrell, Andy Hisgen, Charles Jerian, Timothy Mann, and Garret Swart. The Echo distributed file system. Research Report 111, Systems Research Center, Digital Equipment Corporation, September 1993.

[6] Michael Burrows. *Efficient Data Sharing*. PhD thesis, University of Cambridge, September 1988.

[7] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: A high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9, Hewlett-Packard Laboratories, November 1992.

[8] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[9] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode file system. In *Proceedings of the Winter USENIX Conference*, pages 43–60, January 1992.

[10] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The logical disk: A new approach to improving file systems. In *Proc. 14th Symp. on Operating Systems Principles*, pages 15–28, December 1989.

[11] Murthy Devarakonda, Bill Kish, and Ajay Mohindra. Recovery in the Calypso file system. *ACM Transactions on Computer Systems*, 14(3):287–310, August 1996.

[12] Murthy Devarakonda, Ajay Mohindra, Jill Simoneaux, and William H. Tetzlaff. Evaluation of design alternatives for a cluster file system. In *Proceedings of the Winter USENIX Conference*, pages 35–46, January 1995.

[13] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *Proceedings of the ACM International Conference on Measurements and Modeling of Computer Systems (Sigmetrics '97)*, pages 272–284, June 1997.

[14] Andrew C. Goldstein. The design and implementation of a distributed file system. *Digital Technical Journal*, 1(5):45–55, September 1987. Digital Equipment Corporation, 50 Nagog Park, AK02-3/B3, Acton, MA 01720-9843.

[15] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. 12th Symp. on Operating Systems Principles*, pages 202–210, December 1989.

[16] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. 11th Symp. on Operating Systems Principles*, pages 155–162, November 1987.

[17] John H. Hartman and John K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems*, 13(3):274–310, August 1995.

[18] Andy Hisgen, Andrew Birrell, Charles Jerian, Timothy Mann, and Garret Swart. New-value logging in the Echo replicated file system. Research Report 104, Systems Research Center, Digital Equipment Corporation, June 1993.

[19] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[20] James E. Johnson and William A. Laing. Overview of the Spiralog file system. *Digital Technical Journal*, 8(2):5–14, 1996. Digital Equipment Corporation, 50 Nagog Park, AK02-3/B3, Acton, MA 01720-9843.

[21] Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Ben A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Antony Mason, Shu-Tsui Tu, and Edward R. Zayas. DEcorum file system architectural overview. In *Proceedings of the Summer USENIX Conference*, pages 151–164, June 1990.

[22] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. VAXclusters: A closely-coupled distributed system. *ACM Transactions on Computer Systems*, 4(2):130–146, May 1986.

[23] Leslie Lamport. The part-time parliament. Research Report 49, Systems Research Center, Digital Equipment Corporation, September 1989.

[24] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed virtual disks. In *Proc. 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, October 1996.

[25] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In *Proc. 13th Symp. on Operating Systems Principles*, pages 226–238, October 1991.

[26] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems*, 12(2):123–164, May 1994.

[27] Marshal Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[28] James G. Mitchell and Jeremy Dion. A comparison of two network-based file servers. *Communications of the ACM*, 25(4):233–245, April 1982.

[29] Sape J. Mullender and Andrew S. Tanenbaum. Immediate files. *Software—Practice and Experience*, 14(4):365–368, April 1984.

[30] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[31] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and David Hitz. NFS version 3 design and implementation. In *Proceedings of the Summer USENIX Conference*, pages 137–152, June 1994.

[32] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[33] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implemention of the Sun network filesystem. In *Proceedings of the Summer USENIX Conference*, pages 119–130, June 1985.

[34] Robert A. Shillner and Edward W. Felten. Simplifying distributed file systems using a shared logical disk. Technical Report TR-524-96, Dept. of Computer Science, Princeton University, 1996.

[35] Garret Swart, Andrew Birrell, Andy Hisgen, Charles Jerian, and Timothy Mann. Availability in the Echo file system. Research Report 112, Systems Research Center, Digital Equipment Corporation, September 1993.

[36] Randy Wang, Tom Anderson, and Mike Dahlin. Experience with a distributed file system implementation. Technical report, University of California, Berkeley, Computer Science Division, June 1997.

[37] Edward Wobber, Martin Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.