A HOLOGRAPHIC FILE SYSTEM FOR A MULTICOMPUTER WITH MANY DISK NODES

By

Amnon Barak, Bernard A. Galler and Yaron Farber

Technical Report 88-6

May 1988

המחלקה למדעי המחשב
האוניברסיטה העברית בירושלים

DEPARTMENT OF COMPUTER SCIENCE
THE HEBREW UNIVERSITY OF JERUSALEM
JERUSALEM 91904, ISRAEL

A HOLOGRAPHIC FILE SYSTEM FOR A MULTICOMPUTER WITH MANY DISK NODES

By

Amnon Barak, Bernard A. Galler and Yaron Farber

Technical Report 88-6

May 1988

A HOLOGRAPHIC FILE SYSTEM FOR A MULTICOMPUTER WITH MANY DISK NODES

AMNON BARAK
The Hebrew University of Jerusalem

BERNARD A. GALLER
The University of Michigan

AND

YARON FARBER
The Hebrew University of Jerusalem

ABSTRACT

Future computing systems may involve thousands of networked general-purpose computers, without shared memory or shared devices. The "operating system" for such a configuration must be completely distributed, and it must tolerate the random disappearance and reappearance of nodes, possibly with obsolete control information and/or data. Traditional file systems are not equipped to satisfy these requirements.

We present a "holographic" file system (HFS), for concurrent data retrieval in a computer system with a large number of disks (although it is probable that most nodes will be diskless). In such a file system it is possible to operate on a file by simultaneously utilizing many (or all) of the disks, since the file system is organized to take advantage of the multiplicity of equipment, rather than limiting access to a single disk for each file, as in most existing file systems. The main advantages of the HFS are a speed-up in data retrieval related to the number of disks, and improved availability by allowing access to parts of a file even when other parts of that file are not accessible.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management - *file organization, access methods*; D.4.2 [**Operating Systems**]: Storage Management - *allocation/deallocation strategies; secondary storage devices*; D.4.8 [**Operating Systems**]: Performance - *measurements*; H.3.2 [**Information Storage and Retrieval**]: Information Storage - *file organization*; E.5 [**Files**]: *organization/structure*; D.4.7 [**Operating Systems**]: Organization and design - *distributed systems*;

General Terms: Design, Organization

Additional Key Words and Phrases: Distributed file system, holographic file system, interleaved file system, file servers, file system organization, diskless workstations, distributed systems.

This work was supported in part by the U.S Air Force Office of Scientific Research under grant AFOSR-85-0284.
Authors' addresses: A. Barak and Y. Farber, Department of Computer Science, The Hebrew University of Jerusalem, Jerusalem 91904, Israel; B.A. Galler, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109.

1. INTRODUCTION

Future computing needs, such as in the management of large amounts of data, will require unified computing systems consisting of a large number (possibly thousands) of loosely coupled, independent computers (nodes), interconnected by a communication network, without shared memory or shared devices. With so many processors, however, no one processor can "know" the state of the entire system (because of the overhead and possible congestion in trying to keep it up to date), and we shall no longer be willing to shut down the entire system because of the failure of one or a few processors. The "operating system" for such a configuration must be completely distributed, in some reasonable sense, and it must account for processors dropping out of the system and returning at some later time, possibly with obsolete control information and/or obsolete data.

Unfortunately, existing operating systems are unable to handle more than several dozens of nodes, due to the fact that many commonly used mechanisms and internal algorithms lead to congestion when used in a configuration beyond a certain size. One bottleneck in most multicomputer systems is due to the use of a traditional file-system organization, in which all the records of each file are placed in a single disk. The main drawback of such an organization is that it does not take advantage of the multiplicity of the hardware components by allowing parallel operations on files. Future systems, particularly systems with a large number of disks, must find a way to speed up the access time to files; in particular, by encouraging parallel access to many records simultaneously.

In this paper we present an organization for a file system that scatters records of a file into different nodes, in order to allow parallel read/write operations of different records. A related concept, called disk striping (interleaving), which can use up to 16 partitions, is implemented in the Convex file system [3]. The advantages of both organizations include (a) the elimination of the restriction that a file system reside in a single partition of a disk, (b) enhanced throughput due to concurrent accessing of the same file system by many processors, and (c) the possibility of better disk-drive load balancing.

The main difference between an interleaved file system (IFS) and the holographic file system (HFS) proposed here is the way records are scattered among the different nodes. In the typical IFS, consecutive records of a file are placed in consecutive disk nodes, while in the HFS records are placed in a random fashion, using a hash function on the file and record names. The IFS organization implies that a centralized index-map must be used for the locations of the records of a file, while no such centralized map is needed in the HFS. Instead, any record can be retrieved directly by any process on any node, using the file's known hash function. The centralized index-map of the IFS may be an advantage in a single machine with multiple disk drives, but it is a serious bottleneck in a system with a large number of file servers, all of which may be trying to access the index-map concurrently. Moreover, any single source of critical information leaves the system very vulnerable if that source "goes down".

Another advantage of the HFS is that it increases the availability of files and allows operations on files even when portions of the data are inaccessible due to disk or node crashes. This situation is similar to a holographic image, which remains visible even if some portions of it are removed.

The use of a hash function to spread data across several machines is also recommended in Kitsuregawa et al [8], where hashing is used to prepare the results of one database operation for the action of the next. They rely there on the clustering aspect of hashing to isolate *buckets* of data with similar attributes from other *buckets*, so different processors may act on them independently. This is of course quite different from the use of a hash function to organize a file system, which is the subject of this investigation.

The fundamental assumption that we make is that in future systems many, perhaps most, of the node machines in a large configuration will not have disks. Thus, almost every file-related operation will be performed to or from a remote file server. While there is often concern over the relative cost of remote file access over local access, in Lazowska et al [10], it is argued that "with a well-designed file server ... the cost of remote file access is reasonable even for substantial numbers of client workstations." In effect, then, the holographic file system may be regarded as a redesign of the traditional file

server so that queuing delays and congestion problems are minimized, or avoided altogether.

In the next section we introduce the holographic file system and its main advantages. In section 3 we describe the architecture of a possible target system. In Section 4 we give the details of the organization of the holographic file system. Sections 5 and 6 discuss further issues related to the operations on files, and the management and main features of the HFS. In Section 7 we present analytical and computational results which indicate the speedup in access which the HFS can provide. Conclusions follow in Section 8.

2. THE HOLOGRAPHIC FILE

A holographic file is a file whose records are scattered across different disks such that each record can be addressed directly. Unlike traditional disk-based file systems, e.g., UNIX[1], where the whole file resides on one disk and thus is either available or totally inaccessible due to a node or disk crash, each record in a holographic file can be accessed independently, even if other records in the file are not available at that time. In this sense, the file is holographic, i.e., the file survives the loss of part of itself. One small difference with a physical hologram, of course, is that the loss of a record does not guarantee that that record can be resurrected from information contained in other records. If it is crucial to a particular application that no records ever be totally lost, it is possible to employ various replication and recovery techniques, e.g., the technique developed by Rabin [11].

To illustrate this aspect of the HFS, consider an airline reservation system. At any given time, each agent needs to access only a small portion of the whole data base. If one portion is not available, then only the data of this portion cannot be reached; other parts of the file are not affected.

Since the systems we are considering are specifically aimed at achieving as much concurrency as possible, one of the goals of the HFS is to encourage concurrency in data access as well as in computation. This is accomplished by allowing independent access to individual records of files, in parallel.

[1] UNIX is a trademark of Bell Laboratories.

This implies, of course, that there must be no part of the information needed to access individual records that resides in only one place. Otherwise, on the way to accessing their specific records, many processes would be requesting information simultaneously from that centralized source, causing severe congestion. In particular, information needed to locate a record, given the file name and the number of the particular record, must be accessible to a requesting process without putting it in competition with other processes wishing to access their own records. In the HFS, this is accomplished by deriving location information for individual records directly from the file name and record number. Such a strategy implies that not only the data must be dispersed over many disknodes, but appropriate control and status information must likewise be dispersed and available in parallel. The user interface is unchanged, of course, since the entire file system is hidden by the supporting kernel on each node.

3. ARCHITECTURE

In general, a holographic file system can be implemented in any computer with multiple disk drives. The main benefit of such a file system comes, however, when it is used in a multicomputer with a large number of processing elements (nodes) interconnected by a communication network, and a reasonably large number of nodes with attached disks. In contrast to existing massively parallel architectures for special applications, e.g., the Connection Machine [6], the system under consideration is intended to be used as a general-purpose computing facility, possibly for multiple users. The relevant part of its architecture consists of two main components. First, there is a cluster of full-fledged, general-purpose processing elements, each with its own local memory, and communication channels that permit direct interaction between any pair of nodes. We assume that the network topology is a complete graph, i.e., it is a fully connected network, with sufficient capacity that the communication delays between any pair of nodes are of the same order of magnitude.

The second component of the architecture is the secondary memory, i.e., the disks. Because of technology limitations, power consumption, and financial considerations, it is reasonable to assume that only some fraction of the node machines will possess disk drives. We call these machines *disknodes*.

For example, the TF-1 (Teraflop-1) supercomputer, a massively parallel MIMD machine being designed at IBM T. J. Watson Research Center [4], will have about 1000 disknodes among its 32,768 computational nodes. Thus the ratio of diskless nodes to disknodes in the TF-1 is approximately 31:1. Clearly, other ratios are also possible, including a 1:1 ratio in which all of the node machines are disknodes. (Of course, even in a 1:1 system, one could, through software, arrange for temporary configurations with the behavior of systems with other ratios.) In principle, the disknode machines can function independently, since they possess all the necessary hardware components. The remaining nodes are diskless, however, and must depend on services from the disknode machines for many operations; in particular, file-system-related operations.

An important observation is that regardless of the diskless/disknode ratio, future parallel and/or distributed computing will require intensive interaction between processing elements and remote files. Obviously, all of the diskless nodes must perform all of their file-related operations to and from remote nodes. Thus, the greater the ratio of diskless nodes to disknodes, the more nodes that must perform all of their file operations to and from remote nodes. In the extreme case, even if all the nodes are disknodes, it is difficult if not impossible to guarantee that each node will possess locally all the data files with which it must interact. Thus even in this case, it must be expected that many, if not most of the file-related operations will be to and from remote disks.

As an example, consider typical operations on a large matrix. Using traditional file systems, this matrix will either reside in one disknode, as one file, or will be transferred back and forth to the multicomputer from an attached file server. Assuming that the operations on the matrix are performed in parallel by a cluster of nodes, then in either case the appropriate portion of the matrix must be transferred to each node in a serial manner, using remote file operations. Similar arguments hold for input and output files as well as intermediate files .

It follows that an appropriate file system must be devised to take advantage of this (potentially) massively parallel accessibility. One such file system is presented in the next section.

4. ORGANIZATION OF THE HOLOGRAPHIC FILE SYSTEM

In this section we present the organization of the holographic file system. We begin with a short review of the organization of a traditional file system. For convenience we call it a single-disk file system.

Many single-disk file systems are organized in a three-level hierarchy: directories, index-maps, and data records. In UNIX [1], for example, the directories are arranged as a rooted tree, where each directory includes a list of pairs, each consisting of a file or directory name and its index-map (inode) number. The entries in each directory are arranged sequentially, and normally the user is encouraged to disperse his files into different directories, using some self-imposed logical division.

The second level of the file-system hierarchy is the index-map, called the inode in UNIX. One such structure is created for each file. It contains general identifying information such as the file name, owner, access permission, time of creation, size, etc., and a set of pointers to the actual data blocks of the file. For example, in UNIX if a file has up to ten data blocks, its inode consists of a single data block. (In some recent implementations, the inode structure of small files also includes the data [1]). The motivation behind this organization is to use a cache mechanism, once the file is opened, by loading its inode into main memory and then minimizing the number of disk accesses by maintaining the appropriate inode information in main memory.

The third part of the file-system hierarchy consists of the data records. In UNIX and many other systems these records contain only data, i.e., they do not contain any control information. The goal here is to maximize the amount of data that can be placed in each block, without wasting any space for control or identification information.

In contrast to this organization, the main requirement of the proposed multicomputer file system is concurrent access to a large number of data records, by multiple processes that execute in parallel, most likely in different nodes. Accordingly, the file system should be organized in a way which allows these processes to access different data records concurrently. This is clearly not possible if an entire file

resides in a single disk. Therefore, in the proposed file system we place different data records in different disknodes. Furthermore, since data retrieval from a file is intended to allow a high degree of computational concurrency, we allow a variable record size, defined by the user, based on a logical division of the data in accord with the requirements of his various processes, as well as a fixed-size option.

The organization we suggest could in fact be achieved in a traditional file system (with several disks), by partitioning each file into several smaller files, called records, deliberately placed into different disks. If this were done by the user, however, then in addition to file partitioning and placement, the user would also be responsible for naming, locating, and retrieving each record. The holographic file system relieves users of such burdensome and error-prone tasks.

The top level of the HFS is the directory. Its organization is similar to that of a traditional, single-disk directory, e.g., a rooted tree as in UNIX, in that it includes file and directory names and some additional control information. In general, this information includes all the properties which are attributable to all the records of the file and which are not changed frequently, i.e., a unique file ID, the owner's name, the original date of creation, the hash function to be used to select disknodes for the records of the file, the file type (i.e., whether it has fixed-size or variable-size records), etc. In particular, we note that unlike the directory in UNIX, the HFS directory does not include a pointer to the file's index-map, since no such single map exists for the whole file.

The second level of the HFS includes mechanisms for naming, placement, and location of the records of a given file. Recall that in a single-disk system the necessary information is placed in the (centralized) index-map of the file. In the HFS, individual records of a file are dispersed among the available disks using two hash functions. The first maps record names to disknodes, and the second is used to map record names to specific index-map entries within a node. These entries in turn contain pointers to the disk blocks containing the data (the third level of the HFS organization). The first hash function and the file type are directly obtainable from the directory, and are passed on from process to process as needed without further reference to the directory. They can thus be used independently by

each process. The second hash function is defined independently by each disknode based on its disk configuration, and need not be known by a process requesting disk I/O.

The use of hash functions implies that in the proposed HFS each file and each record of a file must be given a unique identification (ID). For example, the file ID may be a combination of the machine ID in which its directory entry is created and a sequence number, while the record ID may consist of the file ID followed by the record number. Each record can thus be identified without regard to other records. Note that in the case of fixed-size records, the system can allocate sequential numbers to new records.

We now describe in detail the use of the hash functions when performing (read/write) operations on records. Every operation that requires access to a record starts by executing the first hash function on the record ID to identify the target disknode machine on which the record should reside. Assuming that this disknode machine is operational, a message with the required operation (or data) is sent to that machine.

4.1. Writing a record

The second stage of the write operation is internal to the target disknode machine. The main requirements here are: (1) a mechanism that allows quick placement and retrieval of specific records, and (2) efficient global operations when needed on all the records of a file which are in each disknode. To begin with, we require that each disk be partitioned into two parts, the area into which we shall hash, and the data area. The hash area contains pointers to the index-maps of records, with the index-maps for all of the records of a given file organized as a single B-tree (indexed by the record number), for very quick access to any specific record, or to all of the records of that file in sequence. The entry in the hash area, which we may think of as the "root" of the B-tree for a particular file, will also contain control, identification, and access permission information for the file. Each node of the B-tree, representing a record in the file and called an *imap*, since it is somewhat similar in function to the UNIX *inode*, contains the record number and pointers to the data blocks of that record.

To realize the mapping from the file name to the root of the B-tree, we use the second hash function, applied to the name of the file. This will take us to a block of perhaps 16 entries, representing the possible collisions in the hash computation. Each entry contains the file name (to resolve the collision), and a pointer to the B-tree for that file. Depending on the expected number of file names that might hash to the same value, the collision entries can be searched linearly, or if there are many, they can themselves be organized into a B-tree for quick access.

The data area contains the B-trees and the actual data. We note that this allows a high degree of disk-space utilization because disk blocks in the data area can be assigned within each node from a linked-list pool of free blocks.

We indicated earlier that we provide in the HFS for both variable-size and fixed-size records, retaining the file type in the directory so that it is available when the directory information for the file is obtained. The B-tree organization just described is appropriate for both file types. The difference is that for fixed-size records the application can refer to specific fields within the file by byte offsets from the beginning of the file and count on the system to compute the relevant record number, while an application using variable-size records must keep track of the record numbers itself and cannot use byte offsets from the beginning of the file. The variable-size record may be more flexible, however, in terms of the needs of the application.

4.2. Reading a record

When a record is to be read, the first hash function is used to identify the target disknode machine, followed by a read-request message sent to this node. The target node uses the second hash function to locate the imap of the record, as described earlier. The record is then read, and the result is sent to the reading process. As in the write operation, the usual exceptions due to disk failure and subsequent recovery apply. This problem is discussed below.

5. OPERATIONS ON HOLOGRAPHIC FILES

In this section we discuss some of the main issues related to the performance of operations on records and files in the HFS.

5.1. Global operations on files

The HFS is a record-oriented file system. Therefore all global operations on files, e.g., copy, removal, change of ownership, change of access permissions, etc., must be performed independently in each disknode. The use of the hash function associated with the file lets us avoid the problem that in the HFS we do not possess global information about the file size or the specific record numbers used, since these would contradict our requirement for decentralized control. To overcome this lack of global information, each global file operation starts up a system process that generates calls to all the disknodes, somewhat like a broadcast, for the required operation. Then each disknode carries out the operation independently and concurrently. Since we hash within each disknode only on the file name, all records of a particular file will hash to the same B-tree, where they will then be identified by the record number. Thus, one use of the second hash function of the HFS will identify immediately all of the records of a file within each disknode, if any, and the global file operation can be carried out efficiently. (Of course, many global operations, such as change of ownership or access permission, only affects the hash entry for the file, and not the B-tree.)

We note that certain global file operations, initiated by the owner of a file, or by someone to whom proper authority has been delegated by the owner, may override the actions of ongoing processes. For example, an owner's command to "remove" a file, i.e., to eliminate it entirely, is assumed here to take priority over the fact that some process which already knows about the file may later refer to its records by name. Groups of collaborators who wish to avoid such situations are free to create an application-level process to monitor and "approve" such global actions when they are "safe", but that is beyond the scope of this paper.

To illustrate these ideas, consider a file that has only one record at the time of a global file request, say record number 10,000. In order to "know" this, and to make sure that there are no other records, etc., we send the file operation command to all of the disknodes to be executed concurrently. If appropriate, all but one of the disknodes will respond immediately that the command has been carried out, and the one disknode which does contain a record for that file will act accordingly. Since all of this action is concurrent, there is very little lost time in the process.

For another example, in a system that has 1,000 disknodes, a file with 30,000 records would approximate 30 records per disknode. A request to remove the file would cause a process to issue 1,000 commands to all the disknodes, each of which would simply delete a hash entry. Once its entry is deleted, the disknode can contine on to release the local space previously allocated to imaps and data for the records of that file.

It is implied in the previous examples that on some occasions it may be appropriate for disknodes to respond that a global operation is complete, and on other occasions it may not be desirable to do so. These are policy decisions, but we note that in those cases where responses are desired, these, too, may be organized hierarchically. In this way, the number of responses can be minimized, avoiding congestion.

In spite of the possibility of performing most file operations concurrently there are some file operations, e.g., sequential reading, that cannot be performed with such efficiency because no information is available about the number of records and their sequence numbers. One method to speed up the performance of such operations is to use a parallel merge algorithm on all the B-trees of the file, and then to refer to the data blocks in the resulting sorted order. In this procedure we can take advantage of available computational nodes and multiple disknodes to reduce the time required for the command, up to $O(log_2 n)$, for an n-disknode multicomputer.

5.2. Concurrent access to file information

One of the most critical issues in the use of the HFS is how to provide file and record IDs to a possibly large number of processes. Clearly, if the application first generates these processes and then each process attempts to read the file name and other relevant information from the single directory entry, a serious bottleneck may result. To overcome this problem we suggest that first the user activate a small number of processes (preferably one), which would be responsible for obtaining the names of all the files to be used and the directory information for these files. This process would then create the remaining processes, which will inherit its information and access rights. We note that this activity itself can be distributed, so that process creation could be done in a hierarchical manner.

5.3. A degree of redundancy

The design of the HFS includes a provision to guarantee that data can be written on *shadow* disks even when a target disknode is "down" (see the next section for details). This will be seen to provide excellent recoverability for records written after the target node is no longer available, but it cannot guarantee access to records which only exist on the missing node. To guarantee the reading of such data a provision for some redundancy could be added to the file system. For example, operations on records could be performed concurrently on $k > 1$ different disknodes. The hash function could generate the identifiers of k target disknode machines each time it is invoked, which in turn would then be interrogated, in the order given by the hash function, until an operational machine was found. If k were sufficiently large, there would be a high probability that an operational target machine would be found. The main drawback of this scheme is that any value of $k > 1$ would increase the number of disk accesses, potential inconsistencies, etc., since every read/write operation must be performed to all k disknodes. (In Joseph et al [7] a technique is presented for replicating data efficiently, "permitting replicated data items to be updated concurrently with other operations, while at the same time ensuring that correctness is not violated." However, the authors assume that no information survives a failure of a node.) A better solution would be to use the algorithm of Rabin [11] to provide for writing each record,

suitably transformed, to j disknodes, such that the record can be reconstructed from any i pieces, for a fixed value of i, $i < j$. An interesting research problem would be to combine the results of [11] and the file dispersion that results from the design of the HFS proposed in this paper.

5.4. Data balancing and record migration

One optimization strategy which would be possible in an HFS in order to achieve even better performance is record migration. This technique would be particularly attractive in a multicomputer in which many nodes possess disk drives. There are several potential situations in which migration might be helpful. It might happen that a particular set of file names (and record numbers) causes records to cluster in certain disknodes under the specific hash function being used. When this would be detected (because of congestion at those nodes), we might wish to migrate some of the records to alternate disknodes, thus achieving a kind of "data balancing". Another situation which could benefit from record migration would occur when several processes, attempting to access different records, happen to select the same disknodes almost simultaneously over a period of time, even though the records of each file are rather evenly dispersed. When such congestion is detected, one or more of the popular records could be migrated to another disknode, and the activity would be "balanced" correspondingly. Still other occasions for record migration would occur when a (smaller than usual) disk might become full, or when the number of disks in the system was to be changed rather dramatically. In all of these cases, a system monitoring process would detect the problem, and advise the user or automatically start record migration.

To implement record migration, we shall assume that in addition to the standard (default) hash function known throughout the file system, there are available some additional hash functions (with perhaps provision for user-created hash functions as well). We have provided in each file's directory a field which names the hash function to be used for that file, and this information is transmitted to subprocesses as they may be created. When a situation is detected for which migration is desirable, as decribed above, a new hash function is selected, and all of the records of the file are remapped accord-

ing to the new hash function, as a normal global file operation. (We assume that before any global file operation is started, the initiating process will obtain the most recent hash function from the directory.)

To avoid the problem of processes continuing to use the old hash function because they were active when the change was initiated, we specify that in each disknode, a new hash function implies a new "root" for that file, i.e., a new file name hash entry in the disknode pointing to the B-tree for the file. The old root is retained (pointing to no records), so as to intercept requests by processes which were unaware of the hash function change. A request for reading or writing a record which has been migrated would thus get a response indicating that there is a new hash function, and that the request must be regenerated using that new function. Of course, a process would only need to receive one such response per file migration. After a while, there would be a small amount of overhead in keeping the old file hash entries around. At any time when the system became quiescent, so that there were no processes around which might later want to use old hash functions, a garbage collection process could be initiated to remove (in parallel) all unneeded hash entries of this kind. The amount of overhead incurred in data migration would of course have to be taken into account in determining when migration is desirable, but in many cases we expect that this would be the case.

6. MANAGEMENT OF HOLOGRAPHIC FILES

In this section we discuss some implementation details and other issues related to the management of the holographic file system.

6.1. The directories

As indicated in a previous section the HFS file system is a rooted tree with an organization that is somewhat similar to that of a traditional file system except that its directory entries do not include pointers to index maps of files. Each directory itself may be considered an ordinary file, with the usual placement of its records using the hash functions. We note the possibility of replicating the root record of the file system tree to several, or all of the nodes, in order to overcome any single machine failure.

6.2. File accessability and disknode failure

Let us now examine the accessibility of a file in an HFS. To begin with, since different data records of a file are expected to reside on different disknodes, the probability that the whole file would become inaccessible is dramatically reduced. Assuming that a file is evenly dispersed over n disks, each disk failure isolates, on the average, only $1/n$ of the total number of records in the file. As a result, based on reasonable disk failure rates, we may conclude that at any given time most of the records of a file are accessible. More formally, if the probability of a disk or node crash is p, then for a file which is dispersed over n disks, the probability that the entire data portion of a file is not reachable is p^n. The probability that any single record of the file is inaccessible is $1-(1-p)^n$.

We now consider how to treat a disknode failure that results in disconnecting a portion of the file system. We generally expect that disknode to return to the system, but we do have to be careful that operations on files and records that should have affected its contents have been taken into account, and to recognize that data already there may have become obsolete and shouldn't be used. We therefore associate with each disknode a small set of disknodes, to be referred to as its *shadow* disknodes. These are the next 10 disknodes, according to the standard ordering of the nodes of the system, with appropriate wrap-around for the highest-numbered nodes. The reason for using more than one disknode for this purpose is to resolve problems caused by these nodes themselves leaving and returning to the system with possibly obsolete data. As we describe their function, it will become clear that designating 10 such machines as the shadow disknodes should be sufficient for our needs without incurring an intolerable overhead.

Suppose disknode 35 does not respond to a write request, and after the usual number of retries, etc., it is determined that disknode 35 is in fact "down". The process trying to write the record in question would immediately request that the record be written on all of the shadow nodes of disknode 35. These nodes could be intelligent about this activity, in that a previous shadow version of that same record could be replaced thereby. A request to read a record which had been written on the shadow

nodes could be honored, since their version would represent the most current version of the record. (Of course, a request to read a record which still existed only on the missing disknode 35 could not be honored.) In addition, some global file operations could be accommodated by the shadow nodes on behalf of disknode 35. In particular, a request to copy a file to another file (name) could be honored at least with respect to the records available on the shadow nodes. Whether this should be done would be a matter of policy, since this could lead to some complications. For example, after such a copy was made, it would be difficult to know which records were missing because they only existed on disknodes which were "down".

When disknode 35 is ready to return to active duty, it must relieve the shadow nodes of their responsibility, and prepare to begin taking requests. In order to carry out this transition, disknode 35 must receive from the shadow nodes the most up-to-date copies of records they have received, as well as requests from them to carry out any global operations which they had to defer, such as to delete records which only existed on disknode 35 itself. Once disknode 35 has thus updated itself, it can begin business as usual. During this updating process, the shadow nodes may accumulate new requests for disknode 35, and forward them for processing as soon as the update is completed.

The use of perhaps 10 nodes is to try to make sure that this part of the HFS is as reliable as possible for new write requests. A technique such as that in [11] could be used, in addition, to ensure that even if a few of these nodes were down, the probability would be very high that enough nodes would remain to fully recover the saved information when disknode 35 reappears.

6.3. Disk size vs. number of disks

The organization of the HFS provides a means to access and retrieve information concurrently from multiple disks. The degree of this concurrency depends on the one hand on the application level, which should be tuned to take advantage of the available concurrency; at the same time it depends on the physical partitioning of the file into records that are written on different disk drives (on different disknode machines). Assuming that the application requires a high degree of concurrent record access,

then as we shall see in the next section, an increase in the number of available disks decreases the access time (provided the communication speed is high enough). In other words, a file organization which results in a larger number of (smaller) records can imply more efficient disk retrieval.

The main implication of this discussion is that for a fixed amount of disk space, a large number of smaller disks might very well be better than a small number of larger disks. Note that we would also benefit in this case from less disk-head movement, thus improving the average retrieval speed.

6.4. Free-space handling

In the HFS, each disk is controlled by the disknode machine to which it is attached, and is partitioned into two areas, the hash area and the data area. For the data area, a bit map of the free space (disk blocks) would be maintained. This map is used to allocate free blocks to B-trees and data and for consistency checks of the disk.

6.5. File system maintenance

The organization of the HFS implies that a set of programs must be developed for maintaining its correctness and integrity. While the details of these programs are beyond the scope of this paper, we note that many of the required operations can be done in parallel.

7. PERFORMANCE ANALYSIS AND SIMULATION OF THE HFS

In this section we analyze the performance of the HFS and provide some simulation results. In practice, operations on disknodes are performed by many processors. To simplify the analysis we assume that all I/O operations are performed by a single node, which in turn generates the total number of operations on behalf of all the processors. Further, we assume that the communication overhead is essentially zero compared to the time required to satisfy a request by a disknode and that the requesting processor is sufficiently fast so that all the I/O operations are requested simultaneously.

Let the number of disknodes be n, and let r be the number of I/O requests. Assuming that the first hash function distributes the requests to the disknodes uniformly with equal probability $1/n$, then the time required to read r records from n disknodes is proportional to the expected maximal number of records requested from any single disknode, which is denoted in the sequel by $\tau(r, n)$. We note that this problem is an instance of the occupancy problem [5]. Also note that since we assume that there is no communication overhead, there will be a maximal load on the file system, and our results can be considered to be "worst case". Finally, we note that for convenience, in the remainder of this section we shall use the terms and notation of probability theory.

Consider an equiprobable scheme of allocating r balls to n cells. The probability that a specific cell contains exactly k balls is given by:

$$P_k = \binom{r}{k} 1/n^k \, (1 - 1/n)^{r-k} \ , \ k = 1, 2, \ldots, r \ .$$

Assuming that $r, n >> k$, then P_k can be approximated by

$$\frac{r^k}{k!} \, 1/n^k \, e^{-r/n},$$

which is the Poisson distribution,

$$p(k; \lambda) = e^{-\lambda} \frac{\lambda^k}{k!} \ , \ \lambda = r/n.$$

An approximation for the value of $\tau(r, n)$ can be obtained by finding the maximal value of k for which $E(k; \lambda)$, the expected number of cells having k balls, satisfies

$$E(k; \lambda) = p(k; \lambda) \, n = 1,$$

which is equivalent to finding the value of k for which

$$p(k; \lambda) = e^{-\lambda} \frac{\lambda^k}{k!} = 1/n \ , \ \lambda = r/n. \qquad (1)$$

In Table I, we give values of $\tau(r, n)$ for several values of r and n, using the Poisson approximation (1) to find the maximal value of k. We note that the missing values in the first few columns result from

values of k which are not smaller than r or n. Also note that the Γ function was used for non-integer values of k.

	n							
r	8	16	32	64	128	256	512	1024
2	1.27	1.25	1.21	1.19	1.16	1.14	1.13	1.11
4	1.71	1.60	1.50	1.42	1.35	1.31	1.27	1.24
8	2.42	2.14	1.91	1.74	1.61	1.52	1.45	1.39
16	3.61	3.01	2.54	2.20	1.97	1.81	1.68	1.59
32	5.63	4.45	3.53	2.91	2.49	2.20	1.99	1.84
64	-	6.94	5.16	4.00	3.26	2.76	2.42	2.18
128	-	11.40	7.95	5.79	4.45	3.59	3.02	2.64
256	-	-	12.91	8.82	6.37	4.87	3.91	3.28
512	-	-	21.99	14.14	9.60	6.91	5.27	4.23
1024	-	-	-	23.81	15.22	10.32	7.43	5.66

Table I: Values of $\tau(r, n)$ using the Poisson approximation.

A comprehensive study of the limit distribution of $\tau(r, n)$ is given in Kolchin et al [9]. In particular, it is shown that the limit distribution of the maximum in the equiprobable scheme coincides with the limit distribution of the maximum of n independent random variables having a Poisson distribution with parameter λ. We present here the relevant theorem from [9] and then use it to obtain sample values of $\tau(r, n)$.

Theorem 3: If $r/(n \ln n) \to c_1$ as $r, n \to \infty$ and $s = s(r/n, n)$ is chosen so that $n\, p(s; \lambda) \to c_2$, where c_1 and c_2 are positive constants, then

$$P(\eta(r, n) \leq s + k) \to \exp\left\{ -\frac{c_2 \gamma^{k+1}}{1-\gamma} \right\}$$

where $\eta(r, n)$ is a random variable denoting the maximal number of balls in any cell in one experiment, and γ is the root, in the interval $0 < \gamma < 1$, of the equation

$$\gamma + c_1(\ln \gamma - \gamma + 1) = 0 \ .$$

We note that $\tau(r, n) = E(\eta(r, n))$. Based on the proof of the theorem in [7], we assume as an approximation, $\gamma = r/(n\,s)$. Using this approximation, we list in Table II sample values of $\tau(r, n)$, in particular, for $r = n$. These values agree reasonably well with the values on the diagonal (i.e., $r = n$) in Table I.

n	8	16	32	64	128	256	512	1024
$\tau(n, n)$	1.97	2.53	3.07	3.60	4.07	4.58	5.01	5.50

Table II: Asymptotic values of $\tau(n, n)$

In the following, we show how the expected behavior of the HFS, under the assumptions stated above, can be predicted quite accurately by simply carrying out repeated trials and measuring the lengths of the queues. Then we relax the assumption that there is no communication delay, using a simulation package. The results for both cases are quite similar.

We present next the results of 5000 independent experiments in which r balls are placed into n cells. Let $\eta_i(r, n)$ denote the maximal number of balls in any cell in experiment i. In Table III we list values of $\tau(r, n)$, the average value of $\eta_i(r, n)$, $i = 1, \ldots, 5000$.

	n										
r	1	2	4	8	16	32	64	128	256	512	1024
2	2.00	1.48	1.24	1.12	1.07	1.03	1.01	1.00	1.00	1.00	1.00
4	4.00	2.73	2.12	1.64	1.36	1.18	1.09	1.04	1.02	1.01	1.00
8	8.00	5.09	3.56	2.60	2.07	1.66	1.37	1.20	1.10	1.04	1.02
16	16.00	9.54	6.11	4.18	3.06	2.39	1.98	1.65	1.37	1.21	1.10
32	32.00	18.23	11.01	7.07	4.80	3.51	2.73	2.22	1.94	1.65	1.38
64	64.00	35.16	20.24	12.25	7.88	5.41	3.94	3.07	2.45	2.12	1.88
128	128.00	68.45	37.94	21.87	13.36	8.68	5.97	4.38	3.38	2.74	2.25
256	256.00	134.26	72.27	40.20	23.38	14.37	9.38	6.47	4.75	3.67	3.02
512	512.00	265.00	139.79	75.62	42.32	24.79	15.34	10.04	7.00	5.15	4.00
1024	1024.00	524.69	272.72	144.30	78.49	44.23	26.08	16.20	10.70	7.45	5.53

Table III: Values of $\tau(r, n)$, each an average over 5000 independent cases.

Finally, we have simulated the performance of the HFS, using the Nest [2] network simulation package. In these simulations various ratios between the communication delay and the disk delay were used, in contrast with the earlier assumption of no communication delay. The results of this simulation for the ratio 1: 256 are given in Table IV.

	n										
r	1	2	4	8	16	32	64	128	256	512	1024
2	2.00	1.50	1.25	1.11	1.05	1.02	1.00	1.00	1.00	1.00	1.00
4	4.00	2.78	2.22	1.69	1.35	1.23	1.12	1.06	1.02	1.00	1.00
8	8.00	5.16	3.44	2.51	1.92	1.52	1.27	1.10	1.04	1.02	1.02
16	16.00	9.64	6.08	4.08	3.01	2.38	1.98	1.58	1.32	1.14	1.07
32	32.00	18.31	11.02	7.00	4.87	3.52	2.75	2.21	1.91	1.61	1.32
64	64.00	35.11	20.31	12.54	8.04	5.69	4.09	3.05	2.42	2.13	1.97
128	128.00	68.97	38.70	22.69	13.77	8.98	6.20	4.47	3.43	2.77	2.30
256	256.00	133.88	72.63	41.11	24.16	14.86	9.43	6.66	4.84	3.71	3.02
512	512.00	264.64	140.39	75.95	42.83	24.81	15.29	9.99	6.88	5.16	4.06
1024	1024.00	521.51	270.66	143.62	78.78	44.29	26.24	16.24	10.49	7.30	5.37

Table IV: Values of $\tau(r, n)$ using the Nest simulation.

Let the speedup of the HFS, $S(r, n)$, be defined as:

$$S(r, n) = \tau(r, 1)/\tau(r, n).$$

In Table V we list sample values of this speedup, based on the results of Table III and some additional experiments for the higher values of r.

	n						
r	1	2	4	16	64	256	1024
2	1.00	1.35	1.61	1.87	1.98	2.00	2.00
4	1.00	1.47	1.89	2.94	3.67	3.92	4.00
16	1.00	1.68	2.62	5.23	8.08	11.68	14.55
64	1.00	1.82	3.16	8.12	16.24	26.12	34.04
256	1.00	1.91	3.54	10.95	27.29	53.89	84.77
1024	1.00	1.95	3.75	13.05	36.26	95.70	185.17
2048	1.00	1.96	3.86	13.80	44.52	119.97	258.18
4096	1.00	1.98	3.87	14.40	49.02	144.17	344.06
8192	1.00	1.98	3.91	14.83	52.72	166.80	440.10

Table V: Sample values of the speedup of the HFS.

7.1. Observations

On the whole, there is good agreement between the Poisson approximation derived from the theoretical model, the asymptotic behavior of $\tau(r, n)$, the computed values based on repeated experiments, and the Nest simulations. More specifically, our computations and the results of Tables I-V show that:

1. The approximation of $\tau(r, n)$ by the Poisson distribution appears to be most accurate when $r > n$, for relatively large values of n. The computed asymptotic values of $\tau(r, n)$ are not stable for $r >> n$, and they are most stable for $r = n$. The computed values given in Table III appear to be the most stable values, and perhaps are the most accurate approximation. These values are further validated by the Nest simulations, which also take into account some network delay.

2. From Table V, as the number r, of disk requests increases (for a fixed n), the speedup appears to asymptotically approach n. Also, as the number of disk nodes increases (for a fixed r), the speedup approaches r, although more slowly. These results are encouraging, since we would expect the HFS to perform better when a large number of processes are generating requests, and when a large number of disknodes are available, or both!

3. Based on the Nest simulation runs, we have observed that if the ratio of the disk response time to the communication delay is too small, the speedup tends to be very small. On the other hand, if this ratio is sufficiently large, i.e., over 256, then as r increases, the speedup approaches n.

4. The time interval to read n records from n disknodes is linearly proportional to log n.

8. CONCLUSIONS

We have presented here an architecture for a file system appropriate for a large multicomputer system with many disknodes. Such a file system avoids the bottlenecks usually associated with centralized directories and storage maps, and encourages processes executing in parallel to access their data concurrently as well. In addition, the HFS is designed to tolerate and recover as well as possible from the disappearance and reappearance of disknodes, without an assumption that the data on such disknodes must inevitably be lost.

Accompanying these advantages of the HFS is a decided speedup in access time for data stored in the system. If we assume that communication and CPU overhead is (or will eventually be) much less than the disknode processing time, there is a definite advantage to using the HFS, especially when many processes can be expected to access different records spread over many disknodes. As the ratio of the CPU and communication speed to the disknode speed goes up, and as the number of requesting processes and their disk I/O requests increases, the better the speedup. In the future, these should prove to be the prevailing technology characteristics, and the HFS should prove to be an appropriate design for a distributed file system.

REFERENCES

[1] Bach, M.J. The design of the UNIX operating system. Prentice-Hall Software Series, 1986.

[2] Bacon, D.F., Schwartz, J. and Yemini, Y. Nest: A network simulation and prototyping tool. Proc. USENIX Conference, Winter 1988, pp. 71-77.

[3] Convex Computer Corporation, UNIX Programmer's Manual, Version 6.1, Feb., 1988.

[4] Denneau, M.M. Hochschild, P.H. and Shichman, G. The switching network of the TF-1 parallel supercomputer. Supercomputing Magazine, Winter 1988, pp. 7-10.

[5] Feller, W. An Introduction to Probability Theory and Its Applications. Wiley, New York, 1968.

[6] Hillis, D. The Connection Machine. MIT Press, Cambridge, MA, 1985.

[7] Joseph, T.A. and Birman, K. P. Low cost management of replicated data in fault-tolerant distributed systems. ACM Trans. on Computer Systems, 4, 1, Feb. 1986, pp. 54-70.

[8] Kitsuregawa, M., Tanaka, H. and Moto-oka, T. Application of Hash to Data Base Machine and Its Architecture. New Generation Computing, 1, 1, 1983, pp. 63-74.

[9] Kolchin, V.F., Sevast'yanov, B.A. and Chistyakov, V.P. Random allocations. Wiley, New York, 1978.

[10] Lazowska, E. D., Zahorjan, J., Cheriton, D. R. and Zwaenepoel, W. File access performance of diskless workstations. ACM Trans. on Computer Systems, 4, 3, Aug. 1986, pp. 238-268.

[11] Rabin, M.O. Efficient dispersal of information for security, load balancing and fault tolerance. Center for Research in Computer Technology, Harvard University, TR-02-87, April 1987.