xFS Project Architecture

Curtis Anderson
Doug Doucette
Jeff Glover
Wei Hu
Michael Nishimoto
Geoff Peck
Adam Sweeney

1.0 Introduction

This project is concerned with producing a next generation file system and volume manager for IRIX. Included in this project are changes to the underlying disk drivers, buffer cache, and virtual memory support. Also included are additional semantics added for reasons such as standardization, new products, new markets, etc.

2.0 Goals and Requirements

2.1 High-level goals

- Large systems must be saleable as scientific file and compute servers, as commercial data processing servers, and as digital media servers.
- The same software must be able to run on all supported SGI machines, in particular small machines must be supported well.
- The file system should replace EFS (the current SGI file system) completely, i.e. it should do everything that EFS does.
- The volume manager should replace the current two volume managers completely.
- The file system must out-perform EFS on benchmarks that represent useful activity.
- The file system must support high availability by recovering quickly from failures and by keeping its disk-based data in a consistent state at all times.
- The file system and volume manager design must support future extensions in certain specific areas, i.e. high availability, distributed file systems, and user transactions.

2.2 Detailed requirements

The goals from section 2.1 have guided us in determining a detailed set of requirements for the project. These are broken up into groups below. Items in the *implementation* sections are not really requirements in the same sense that the *functionality* items are; they represent our current

ideas about how to fulfill the functional requirements. In each section, the items are not in any particular order.

2.2.1 File System Requirements

2.2.1.1 File system functionality

- Implement asynchronous I/O, direct I/O, and synchronous I/O as is done in EFS, in addition to normal (buffered) I/O.
- Efficient support for very large files, where very large means a 64 bit size. There must be little or no performance penalty to access blocks in different areas of the file. Some disk space penalty (for indices, for example) is allowed to increase performance. Linear searches through the file system data structures to get to blocks at the end of a large file are unacceptable.
- Efficient support for sparse files. Arbitrary holes must be supported, areas of the file which have never been written and which read back as zeroes. The representation must be disk-space efficient as well as cpu-time efficient in retrieval of old data and insertion of new data. There is no requirement to detect blocks of zeroes being written in order to replace them with holes (nor is it forbidden). This capability is important for some scientific and compute-intensive applications, as well as for Hierarchical Storage Management (HSM).
- Efficient support for very small files, under 1KB or so. A normal root or usr file system has many such files, as does a file system which contains program sources. Most symbolic links fit into this category, as well.
- Efficient support for large directories, both for searches and for insertions and deletions. This implies some index scheme, to avoid linear searches through a long directory.
- The time to recover from failure does not increase with the size of the file system. The time is allowed to increase with the level of activity in the file system at the time of the failure. The recovery scheme must not scan all inodes, or all directories, to ensure consistency. This implies that consistency is guaranteed by use of a log, since the alternative (synchronous behavior as in MS-DOS) is unacceptably slow.
- Recovery never backs out changes that were committed after returning successfully to the user. Some operations must be synchronous, at least as far as the log writes are concerned. Certainly this includes file creation and deletion, and does not include ordinary (buffered) writes.
- Supports ACLs and other POSIX 1003.6 functionality. This includes some form of support for Mandatory Access Controls, Information Labeling, and auditing.
- Supports extents, contiguous regions in a single file. Contiguity is defined with respect to the underlying logical volume, not with respect to the actual disk space. It is not a requirement that the extents be exposed in the programming interface to the user. This is primarily a performance issue but we may choose to make the extent sizes visible or settable per-file.
- Supports multiple logical block sizes, ranging from the disk sector size up to something large like 64KB or 256KB. The block size is set at file system creation time. It is the minimum unit of allocation in the file system (except for inodes, which can be smaller).

- Supports multiple physical sector sizes. This allows us to support different disk hardware without a built-in reliance on a particular formatted sector size. Smaller sector sizes yield less total useable disk space, so more efficient use of current disks can be made by increasing the sector size.
- Allow the file system to change size on-line, possible automatically as well as by administrative command. The file system's underlying space (volume) can grow, so the file system must be able to use the new space. It is also possible to allow communication between the volume manager and the file system so that the file system will ask the volume manager to grow the underlying volume when the file system is getting full (this is a low priority goal, not a requirement). It is not a requirement to allow on-line shrinking of a file system; it is a requirement to allow off-line shrinking.
- Allow the separation of file system space between inodes and data to change on-line. This implies dynamic allocation of the space for inodes as the only reasonable implementation. Note that any mechanism which yields different numbers of inodes in each allocation group implies some sort of indexing scheme to find the inodes.
- It must be possible to attach an *arbitrary attribute* to any object in the namespace. An arbitrary attribute is a (name, value) pair where the name is a printable string and the value is a smallish string of arbitrary bytes.
- High throughput for file server and compute server applications. In particular, the NFS performance must make our system price/performance competitive. Compute server applications need high single-file throughput.
- Extremely high throughput for video server applications. This means that sequential access to large files must be very fast. This will be done via directives from the application about required read/write performance rates.
- Fast, guaranteed response time for digital media and other real-time applications. In part, this requirement will be met by providing portable interfaces for real-time programming defined in POSIX 1003.4; this includes preallocation of blocks to files.
- High throughput for random access to very large databases, via direct I/O and asynchronous I/O. Such applications may elect to bypass the buffer cache and implement their own cacheing.
- Backup and HSM interfaces for Epoch and similar systems are supported. File migration and backup tools are supported or supplied by us. Backup tools allow full and incremental backups in a reasonable length of time, even for systems with very large amounts of online storage.
- It must be possible to restore filesystems from backup media after a disaster, in a reasonably short amount of time. File restore must also allow selection of individual files to restore, and must allow the backup media to be remote from the file system.
- Support low-power machines with the ability to turn off the disk drives when they are not in use, and turn them on again when needed.

2.2.1.2 File system implementation

- The file system is implemented under vnodes, possibly extended from the current ones. Other file systems (excluding EFS) in IRIX continue to run with little or no implementation effort. EFS must continue to run, but may have impaired performance.

- File system is implemented as a journalled file system. This is implied by the requirement for a small recovery time for large filesystems.
- May be implemented using message passing and kernel threads. The former allows later distribution in a network, but costs some performance in the local case. The latter may allow greater ease of implementation.
- Implement so that a user-mode simulation of the file system is functional and usable for debugging and performance modelling.
- Support large, sparse files using a B-tree as the index to the data blocks, replacing the current direct and indirect blocks scheme. This makes the performance of these files acceptable. Any equivalent index scheme will do as well.
- Delay allocation of user data blocks when possible to make blocks more contiguous. This allows us to make extents large without requiring the user to specify extent size, and without requiring a file system reorganizer to fix the extent sizes up after the fact.
- Store symlinks and other small files in the inode when possible. By doing so, we save a disk block and the time to read it.
- Support directories with some form of indexed structure, so that searches are faster for large directories. Some form of B-tree will work.

2.2.2 Volume Manager Requirements

A volume manager is an integral part of the next generation file system. The volume manager provides an operational interface to the system's disks and isolates the higher layers of the file system from the details of the hardware.

2.2.2.1 Volume manager functionality

- Auto-assembly of logical volumes. The volume manager will be able to assemble logical volumes by scanning the hardware on the system and reading all the disk labels.
- Mirroring (plexing) of storage. Flexibility is required, we do not want to require duplexing entire disks, or limiting to two plexes.
- Disk striping. Required for performance on large systems.
- Concatenation of storage sections. Should be able to build arbitrarily large volumes, up to the 64 bit limit.
- On-line configuration changes. The volume manager will support the restructuring of logical volumes (e.g., adding a new plex, growing, shrinking), without requiring the volume to be dismounted.
- Separate logging, data and unreliable sub-volumes. Logging sub-volumes are needed by the file system. Unreliable sub-volumes are needed by multimedia file systems. Each should be sized and organized independently in its portion of the volume.
- High throughput for file server applications. The performance penalty for using the volume manager should be vanishingly small in normal operation.

- Support of RAID devices at their full performance. Implies large transfers generated from the file system code, through the volume manager, down to the RAID driver.
- Support of RAID in software, as opposed to purely hardware RAID implementations, which appear to the system as large disks.
- Fast, guaranteed response time for digital media and other real-time applications.
- Support multiple logical sector sizes (one per volume). This supports the equivalent file system requirement.
- The new and old volume managers (lv, not Veritas) must be able to run in the same system. It is not a requirement to run a Veritas volume in the same system. This allows a gradual changeover to the new file system and volume manager, since the only way to convert data from EFS to xFS is to dump and restore it.
- EFS and non file system applications must be able to run on top of the new volume manager. Ordinary driver interfaces must be presented to these clients, even if xFS doesn't use them.
- Provide for future support of multiple-access (dual-ported) disk controllers. Necessary for real high-availability applications.
- Provide for future dynamic relocation of control for a volume. Necessary for high availability.

2.2.2.2 Volume manager implementation

- The volume manager will store all configuration data on the disk labels themselves. These labels will be replicated so that a logical volume can be assembled even if some pieces are missing.
- Write-logging (on disk) of blocks that will be changed to reduce the need for plex copies after a system restarts after a crash.
- Logging of written blocks (in memory) when volume is incomplete. Necessary for high availability (fast recovery from a disk failure).
- A reduced functionality *lightweight* volume manager may be produced for smaller system configurations. This volume manager would support only concatenation, plexing, strping, and software RAID. Disk media will be completely compatible between the full and lightweight volume manager implementations. [Geoff: what is left out?]

2.2.3 Buffer cache and virtual memory support

- Need to be able to keep blocks from going to disk until prerequisite blocks have gone out. This is required to ensure consistency in a journalled file system.

2.3 Possible requirements

- Support DFS vnode interfaces. This will wait until we figure out what is happening with DFS.
- We may need to allow the user to ensure that their files are contiguous, implying that there is some way to display layout information, and some way to make a file contiguous.
- We may need a file system reorganizer program, to improve file contiguity in the background.

3.0 Design Overview

3.1 xFS File System

The xFS file system is a **journalled** file system. This means that updates to file system metadata (inodes, directories, bitmaps, etc.) are written to a serial log area on disk before the original disk blocks are updated in place. In the event of a crash, such operations can be redone or undone using data present in the log, to restore the file system to a consistent state. This implementation technique replaces the use of a file system check and repair program (fsck) before mounting file-systems that were active when a system crash occurred. Since a full file system check takes an amount of time proportional to the size of the file system, we must avoid it to meet our availability goals. Therefore, we choose the logging mechanism for our implementation.

3.2 xFS Volume Manager

The xFS volume manager (also known as xlv) is intended to replace both the IRIS lv volume manager and the Veritas Volume Manager products. Xlv creates an abstraction of a sequence of logical disk blocks to be used by the xFS file system. The sequence of disk blocks can be assembled by concatenating, plexing (copying), and striping (including RAID) across a number of physical disk drives. Xlv ensures that plexed data is kept consistent across all plexes.

3.3 xFS Growth Path

In the long run, we expect to add user-level transactions to the file system, at which time selected user data will be logged along with the filesystem metadata. This feature will make it practical for user applications to easily implement databases and database-like systems on top of this file system.

The other major future change anticipated is that the file system will be distributed over multiple computers (referred to as nodes), connected by a fast local area network. This technology will allow us to provide high availability to user data in the event of failure of a node. For instance, nodes might share disk controllers via multiporting, where only one node at a time uses the disks attached to the controller. To achieve consistent operation, the nodes must communicate with each other.

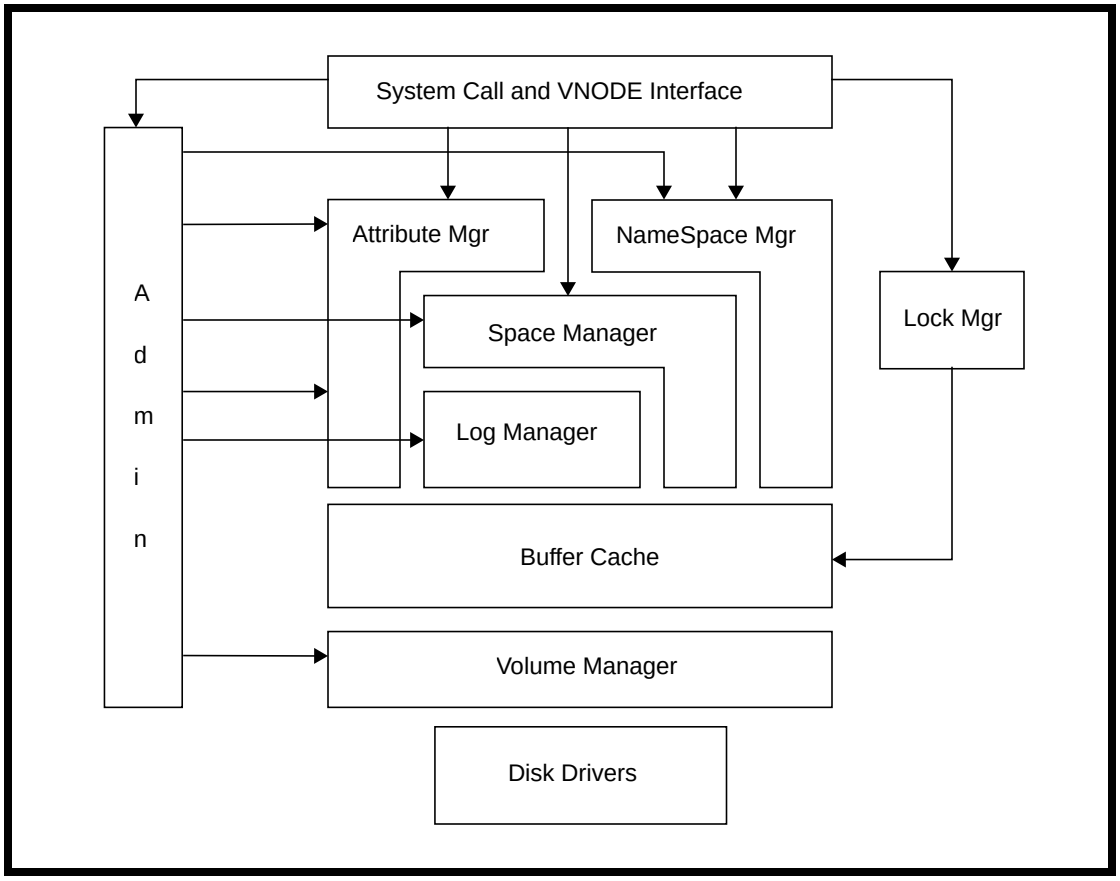
In order to achieve transparent high availability in such an environment, file system naming must be location independent: a file’s name must be the same no matter where it is being named from, or which machine has mounted the file system containing it.

The first release will have neither the distributed capabilities nor transactions, but the architecture is designed to accommodate both. The intent is to make future additions to the implementation easier to do by putting a little more effort in up front.

4.0 The Big Picture

The following is the current architectural diagram for the xFS project.

FIGURE 1. xFS Architectural Diagram



The structure of xFS is similar to that of a conventional file system with the addition of the volume manager between the disk drivers and the file system code and the log manager. The following sections describe the functionality of each component in the architectural diagram.

5.0 System Call and Vnode Interfaces

The file system related calls are implemented here: read, write, open, ioctl, etc., for all file system types. The operations are then vectored out to different routines for each file system type through the vnode interfaces.

Both 32-bit and 64-bit interfaces are supported, where those are supported by the underlying OS and hardware. The semantics of 32-bit applications operating naively on files longer than 2^{32} bytes are defined in the paper “64 Bit File Access”.

The vnode interfaces also allow interoperation with remote clients such as NFS. In a future release the vnode interface may be expanded to support DFS semantics.

When xFS is distributed, references to remote files will come through the vnode layer and be turned into messages if the references cannot be satisfied locally. On receipt of a distributed file system request message, the node owning the file system will send a call through its vnode layer to get data in and out of the file system.

There will be new system call and vnode operations to support Hierarchical Storage Management (HSM) and backup applications; these are currently being designed by an industry-wide working group (DMIG, Data Management Interfaces Group).

There will also be new system call and vnode operations to support extended attributes. There may be a standard proposed for the system call interfaces for these soon, but in all likelihood not before we must ship to customers. One possibility is that we will tell internal customers and ISVs about the new interfaces, with the caveat that they will be replaced with POSIX-compliant interfaces when those exist. Another possibility is that both sets of interfaces will be supported, one set as a library built on the other set.

6.0 Lock Manager

The lock manager implements locking on user files (fcntl and flock calls). For the first release the lock manager is identical to the current implementation. The performance in the non-distributed case must be no worse than that of the current EFS implementation.

In the future, user transactions and distributed locking will be supported. These will cause the lock manager implementation to expand in size and complexity. A distributed two-phase commit protocol will be implemented to allow transactions in the distributed system. Also, some form of distributed deadlock detection will be implemented, possibly using timeouts.

7.0 NameSpace Manager

The namespace manager implements file system naming operations, translating pathnames into file references. A file is identified internal to the file system by its file system and its inode number. The inode is the on-disk structure which holds information about a file; the inode number is the label (or index) of the inode within the particular file system. Files are also identified internally by a numeric value unique to the file, called the file unique id.

File systems may be identified either by a “magic cookie”, typically a memory address of the root inode, or by a file system unique id. File system unique id’s are assigned when the file system is created and are associated uniquely with that file system until the file system is destroyed. An additional temporary unique id, the file system I/O unique id, is created whenever a file system is mounted, and is valid only for the duration of the mount. The file system I/O unique id is used for dynamic reconfiguration in the distributed version of xFS.

The namespace manager manages the directory structures and the contents of the inode that are unrelated to space management, such as file permissions and timestamps. Requests from other systems via NFS come into the system with a *file handle* which the namespace manager uses to find the inode. The file handle includes enough information to deduce the file system, the inode,

and which version of the file is meant (inodes may be reused). Requests from other nodes in a distributed xFS file system would enter with a file system unique id and inode number, and a file unique id, and be validated at this level (to see that the two forms of identification match).

The namespace manager may have a cache to speed up naming operations. The details of the name translation are hidden from the callers.

In the distributed system, we have yet to decide whether the distributed nature of the naming should be transparent (single system view) or visible (multiple system view). (Indeed, different sites may prefer one model over the other, or some customer might even want both models within a single installation.) The issue is whether or not it is necessary to know what machine is managing a file system to name a file within the file system. If the machine name is embedded in the pathname (as in DFS, for example) then the machine controlling access to the file system cannot change without clients being aware of the change. If the machine name is not in the pathname, but rather the file system can move easily from machine to machine, and the file system name and location is broadcast when it moves, then this can change without knowledge of user or user programs.

The current design of the namespace manager's mount semantics have a file system node called a *mount point* which replaces the empty-directory mount points of the current file system. The mount point node contains a file system unique id. When a mount point which refers to a remote file system is encountered during a naming operation, a message is sent to the remote machine which is managing the file system with that file system unique id, along with the naming request, to complete the operation.

It is possible that alternate or extended naming schemes may be implemented in user mode by allowing the entity at the other end of the message queue to be a program. This will not be implemented in the first release of the system.

8.0 Attribute Manager

The attribute manager implements file system attribute operations: storing and retrieving arbitrary user-defined attributes associated with objects in the namespace. Arbitrary attributes are (name, value) pairs where the name is a printable string and the value is a smallish string of arbitrary bytes. Attributes may be controlled either by user applications or by the kernel (ACLs, Trusted IRIX attributes). Certain attributes are pre-defined by the system and may be accessed using both existing UNIX interfaces and the new attribute access system calls. (For example: file access and modification times.) The system call interfaces for controlling attributes have not yet been defined.

An attribute is stored internally by attaching it to the inode of the referenced object. The attribute manager manages the attribute structures that are associated with inodes. However, the attribute manager does not manage those fields which are handled by the namespace manager such as file permissions and timestamps.

No storage for arbitrary attributes is allocated when an object is created, and any attributes that exist when an object is destroyed are destroyed as well. Attributes are not shared between inodes.

Access control lists are handled as a special case of attributes; these can be shared between inodes. There will be no method faster than the equivalent of a ***find*** to locate all the objects in a filesystem that have certain attributes or attribute values.

Some system utility programs will be modified to know about attributes, for example ***cp*** will copy selected attributes of a file when it copies the file. The system backup utility will back up and restore the attributes of an object when that object is backed up or restored.

Standard NFS does not support attributes beyond the traditional UNIX set, so these attributes will not be visible in any way to a client that is accessing an xFS filesystem via standard NFS. NFS mounted file systems will continue to operate as if this feature did not exist.

SGI has an extended version of the NFS protocol that it can speak with other SGI systems. This version may be extended to know about attributes if the distributed system implementation will not be ready in time for the majority of customers of attributes.

For the distributed system case, using the xFS internal protocols, attributes will be accessible in just the same way that other filesystem objects are accessible. There should be no difference between accessing attributes on an object that is managed locally from accessing attributes on an object that is managed remotely.

9.0 Space Manager

The space manager manages the allocation of disk space within a file system. It is responsible for mapping a file (a sequence of bytes) into a sequence of disk blocks. The internal structure of the file system - allocation (cylinder) groups, inodes, and free space management - are controlled by the space manager, as well as the above mapping function.

The space layout choices in the design are influenced by the requirements to support very large files and file systems efficiently, including the possibility of sparse 64 bit files. The space manager is also responsible for optimizing the layout of blocks in a file to avoid seeking during sequential processing, and keeping related files (those in the same directory) close to each other on the disks.

Each file system is divided into log, metadata, data, and real-time sub-volumes. Normally, the data sub-volume and the real-time sub-volume do not exist. If the data sub-volume exists, then ordinary user data is stored in it, otherwise in the metadata sub-volume. Data blocks for real-time files are stored in the real-time sub-volume, if it exists, otherwise in the data sub-volume if it exists, otherwise in the metadata sub-volume. All file system data is stored in the log and metadata sub-volumes.

The space manager divides each file system metadata and data sub-volume into a number of *allocation groups*. When the data sub-volume exists, the allocation group contains blocks from both the metadata and data sub-volumes. Each allocation group has a collection of inodes and data blocks, and data structures to control their allocation. The blocks containing inodes are allocated dynamically from the data block pool, to permit more efficient use of disk space. Knowledge of the location of the sequence of inode blocks for an allocation group is kept the same way it is for ordinary files (in a B-tree).

Free blocks in an allocation group are kept track of via one of two schemes. The first scheme uses a bitmap and a set of counters organized by starting bitmap block and the $\log_2$ size of the free extent. The second scheme uses a pair of B-trees, one indexed by size of the free extent (and secondarily by the starting block), the other indexed by the starting block of the free extent. The scheme to be used will be chosen later, after both methods have been prototyped and their performance analyzed.

The real-time sub-volume is divided into a number of fixed-size extents. The size is chosen at **mkfs** time; it is expected to be large (say on the order of 1MByte). The size does not have to be a power of two, just a multiple of the file system block size; it should be the ideal I/O size for that volume configuration, or a multiple of it. A single extent in the metadata sub-volume contains an allocation bitmap for the real-time sub-volume extents; another extent contains summary information per bitmap block (number of free extents). This alternate method of allocation is chosen for the real-time sub-volume due to the improved performance possible because of the fixed-size extents.

Storage for files is represented in one of three ways, depending on the size and contiguity of the file. For small files, the data in the file is stored in the inode. For medium files, the inode contains pointers to extents containing the file data. For large files, the inode contains the root block of a B-tree indexed by logical position in the file, where the records point to disk extents containing the file data. This storage structure allows for large fragmented and large sparse files to be implemented efficiently, at the cost of some overhead to manage the B-tree indices.

An active file system may be extended by adding more space to the underlying volume. This operation is supported on-line by the space manager, which receives a request to expand the file system and updates on-disk and in-memory structures to implement it.

In a future implementation it may be required to support splitting the control of space management in a single file system over multiple nodes of the system. There will be no effort in the first implementation to take this into account.

10.0 Log Manager

All changes to file system metadata (inodes, directories, bitmaps, etc.) are serially logged to a separate area of the disk space. There is a separate log for each file system. The log allows fast reconstruction of a consistent and correct file system (recovery) if a crash intervenes before the metadata blocks are written to disk. The log space is allocated independently from the file system space for safety; this separation is managed by the underlying volume manager. The log manager utilizes information provided by the space manager to control the sequencing of write operations from the buffer cache, since specific log writes must be sequenced before or after data operations for correctness if there is a crash.

The space and name manager subsystems send logging requests to the log manager. Each request may fill a partial log block or multiple blocks of the log. The log is implemented as a circular sequential list which wraps when writes reach the end. Each log entry contains a log sequence number, so that the end of the log may be found by looking for the highest sequence number. On

plexed volumes, the buffer cache is also responsible for inserting log records for non-metadata blocks, so that the volume manager's write-change log does not need to be used by the file system. This allows the system to keep the plexes of a volume synchronized with each other in the event of a crash between writes.

After a crash, the log must be recovered before the file system can be used. Operations which are recorded and are complete in the log but are not yet stored in the data area of the file system are re-done so that the file system data reflects a correct and consistent state. The log manager's role in this is to identify the log records and to call other pieces of the file system to perform recovery operations.

Log operations are blocked together to get higher throughput on the log portion of the volume. The block is called a log record. Typically, log records are written asynchronously; the log manager can be directed by higher levels of the system to force writing of the current log record as soon as possible. A given log write cannot be started until the previous one finishes.

11.0 Buffer Cache

The buffer cache is a cache of disk blocks for the various file systems local to a machine (node). Read requests may be satisfied from the buffer cache; write requests may write into the cache. Cache entries are flushed when new entries are needed, in an order which takes into account frequency (or recency) of use, and file system semantics. File system metadata as well as file data is stored in the buffer cache. User requests may bypass the cache by setting flags (O_DIRECT); otherwise all file system I/O goes through the cache.

The current buffer cache interfaces will be extended in two ways. First, 64-bit versions of the interfaces will be added to support xFS's 64-bit file sizes. Second, a transaction mechanism will be provided. This will allow buffer cache clients to collect and modify buffers during an operation, send the changed buffers to the log manager, and release all the buffers after successful logging.

In the future distributed system, a buffer cache will hold data for filesystems remote to a machine.

12.0 Guaranteed Rate I/O

xFS will support digital media applications with a guaranteed rate I/O mechanism. This will allow applications to specify "real-time" guarantees for the rate at which they can read or write a file. The application will specify a rate of X bytes per Y seconds for a given file, possibly beginning at some future time, and the file system will reserve the bandwidth of all underlying devices to meet the requested guarantee. If the bandwidth is not available the application's request will be refused. Note that this mechanism is device and topology independent; it can, for example, be extended to a distributed system with no change to application code.

The only new interface to the user will be that used to make a bandwidth reservation. This will take the form of a new ioctl() call on a file or a new system call. All data accesses will be made with standard read() and write() system calls.

Guaranteed rate I/O will have very little impact on the buffer cache, because programs which utilize this mechanism will typically be required to use direct I/O and will thus completely avoid the buffer cache. It will, however, have an impact on the disk drivers. These must be modified to recognize guaranteed rate requests and to schedule them in a real time manner. The disk and volume drivers will also be required to export an interface for acquiring their response time and bandwidth characteristics for use by the reservation scheduling module.

The knowledge of what bandwidth is available for reservation will most likely be pushed out into a user level reservation scheduling daemon. The daemon will have knowledge of the characteristics and configuration of the disks and volumes on the system (including backplane and SCSI bus throughput), and it will be the responsibility of this daemon to track both current and future bandwidth reservations.

13.0 Volume Manager

Within each logical volume, the volume manager implements multiple sub-volumes, which are separate linear address spaces of disk blocks in which the file system stores its data and log blocks. Each subvolume is made of *partitions* (real, physical regions of disk blocks) composed by concatenation, plexing (copying), and striping. The volume manager is responsible for translating logical addresses in the linear address spaces into real disk addresses. Where there are multiple copies of a logical block (plexing), the volume manager writes to *all* copies, and reads from *any* copy (since all copies are identical). The volume manager is responsible for maintaining the equality of plexes across crashes and both temporary and permanent disk failures. In future systems, volumes will be spread across nodes of the system and will be accessible from multiple nodes.

A volume used for file system operations is composed of at least two subvolumes, one for log and one for data. Each subvolume consists of a number of plexes. Plexes are individually organized and map different portions of the subvolume’s address space. A plex consists of a sequence of volume elements each of which maps a portion of the plex’s address space. Volume element can then be striped across a number of disk partitions.

To support plexing on non-xFS volumes, the volume manager will maintain an on-disk write log to keep track of changed blocks. This allows the volume manager to synchronize the data on all the plexes without copying the full contents of the plex. When the volume manager is providing plexing support for an xFS file system, the volume manager interfaces will allow the file system to do the write logging.

14.0 Disk Drivers

Disk drivers are the same as in traditional and current IRIX systems, except for ordering constraints needed by the log and volume managers and rate guarantees. In particular, it must be possible for the volume manager to be certain that a write request has actually been completed, not merely cached for later writing. It should also be possible for the volume manager to specify that a given write not be reordered - that all blocks passed to the driver before this block will be writ-

ten before this block is written, and all blocks passed to the driver after this block will be written after this block. (If non-ordered writes are not available on a particular driver, the volume manager can synthesize this behavior by waiting for completion, but this is much less efficient.)

15.0 Administration

xFS administration includes the utilities needed to create and maintain volumes and file systems. It also includes programmatic interfaces for volume control, file system control (mount, unmount, etc.), backup and restore, hierarchical file systems, etc. In the future, administration support will be expanded to allow remote access to volumes and file systems, for mounting, backup, etc. Graphical interfaces will be provided by the system administration group in MSD for the new tools that need it, i.e. volume administration.

16.0 Threads, Messages, Memory Allocation (Kernel Services)

The project will make use of new kernel services at some point, not necessarily in the first (non-distributed) release. The underlying communication between certain modules will be by message passing in the remote cases in the future implementation, otherwise by ordinary procedural interfaces. The implementation will be based on kernel threads to increase parallelism. The threads will pass messages to each other through message queues. There may also be memory allocation enhancements to support the file system and its messages. None of this will be present in the first, non-distributed, release.

17.0 Inter-module Communication

Communication between major xFS modules will be done by messages where that gives us an advantage. This will happen either if the sender is remote, in a distributed system, or if additional performance (parallelism) can be gained by using messages. We expect to use messages from the vnode layer to the file system implementation layers in the distributed cases, and procedure calls in the local cases. Disk drivers do not need a message interface; the volume managers can communicate with each other. Administrative interfaces must all be message-based so that administrative utilities can be run remotely to the administered object. The log manager interfaces do not need to be message-based, as log accesses are never remote. The name manager needs message interfaces to deal with mount points for remote file systems encountered during pathname parsing.

Communication is addressed to a message queue, identified either by a unique id or by an address; the address is allowed only in the case where the queue is known to be local. The unique id is associated with the queue when it is created. A message queue may be serviced by one thread or by several threads. The message-passing routines are simple synchronous and asynchronous queueing routines. For the remote case we will need to implement a communication manager to identify the machines associated with each unique id, and to packetize and transmit the messages. In the local case this module is a simple routine that handles the translation of unique id’s to message queue addresses, and queues the message appropriately.

Proper assignment of unique id's, whereby the non-timestamp portion of the unique id is administered in a rational way, will mean that the administration tasks will scale. The goal is to have a unique id space that can be interpreted correctly over a wide area network, so that filesystems can be mounted and accessed from very remote systems.