Design of the Server for the Spiralog File System

Christopher Whitaker
J. Stuart Bayley
Rod D. W. Widdowson

The Spiralog file system uses a log-structured, on-disk format inspired by the Sprite log-structured file system (LFS) from the University of California, Berkeley. Log-structured file systems promise a number of performance and functional benefits over conventional, update-in-place file systems, such as the Files-11 file system developed for the OpenVMS operating system or the FFS file system on the UNIX operating system. The Spiralog server combines log-structured technology with more traditional B-tree technology to provide a general server abstraction. The B-tree mapping mechanism uses write-ahead logging to give stability and recoverability guarantees. By combining write-ahead logging with a log-structured, on-disk format, the Spiralog server merges file system data and recovery log records into a single, sequential write stream.

The goal of the Spiralog file system project team was to produce a high-performance, highly available, and robust file system with a high-performance, on-line backup capability for the OpenVMS Alpha operating system. The server component of the Spiralog file system is responsible for reading data from and writing data to persistent storage. It must provide fast write performance, scalability, and rapid recovery from system failures. In addition, the server must allow an on-line backup utility to copy a consistent snapshot of the file system to another location, while allowing normal file system operations to continue in parallel.

In this paper, we describe the log-structured file system (LFS) technology and its particular implementation in the Spiralog file system. We also describe the novel way in which the Spiralog server maps the log to provide a rich address space in which files and directories are constructed. Finally, we review some of the opportunities and challenges presented by the design we chose.

Background

All file systems must trade off performance against availability in different ways to provide the throughput required during normal operations and to protect data from corruption during system failures. Traditionally, file systems fall into two categories, careful write and check on recovery.

- Careful writing policies are designed to provide a fail-safe mechanism for the file system structures in the event of a system failure; however, they suffer from the need to serialize several I/Os during file system operations.
- Some file systems forego the need to serialize file system updates. After a system failure, however, they require a complete disk scan to reconstruct a consistent file system. This requirement becomes a problem as disk sizes increase.

Modern file systems such as Cedar, Episode, Microsoft's New Technology File System (NTFS), and Digital's POLYCENTER Advanced File System use logging to overcome the problems inherent in these two approaches.[1,2] Logging file system metadata removes the need to serialize I/Os and allows a simple

and bounded mechanism for reconstructing the file system after a failure. Researchers at the University of California, Berkeley, took this process one stage further and treated the whole disk as a single, sequential log where all file system modifications are appended to the tail of the log.[3]

Log-structured file system technology is particularly appropriate to the Spiralog file system, because it is designed as a clusterwide file system. The server must support a large number of file system clerks, each of which may be reading and writing data to the disk. The clerks use large write-back caches to reduce the need to read data from the server. The caches also allow the clerks to buffer write requests destined for the server. A log-structured design allows multiple concurrent writes to be grouped together into large, sequential I/Os to the disk. This I/O pattern reduces disk head movement during writing and allows the size of the writes to be matched to characteristics of the underlying disk. This is particularly beneficial for storage devices with redundant arrays of inexpensive disks (RAID).[4]

The use of a log-structured, on-disk format greatly simplifies the implementation of an on-line backup capability. Here, the challenge is to provide a consistent snapshot of the file system that can be copied to the backup media while normal operations continue to modify the file system. Because an LFS appends all data to the tail of a log, all data writes within the log are temporally ordered. A complete snapshot of the file system corresponds to the contents of the sequential log up to the point in time that the snapshot was created. By extension, an incremental backup corresponds to the section of the sequential log created since the last backup was taken. The Spiralog backup utility uses these features to provide a fast, on-line, full and incremental backup scheme.[5]

We have taken a number of features from the existing log-structured file system implementations, in particular, the idea of dividing the log into fixed-sized segments as the basis for space allocation and cleaning.[6] Fundamentally, however, existing log-structured file systems have been built by using the main body of an existing file system and layering on top of an underlying, log-structured container.[3,7] This design has been taken to the logical extreme with the implementation of a log-structured disk.[8] For the Spiralog file system, we have chosen to use the sequential log capability provided by the log-structured, on-disk format throughout the file system. The Spiralog server combines log-structured technology with more traditional B-tree technology to provide a general server abstraction. The B-tree mapping mechanism uses write-ahead logging to give stability and recoverability guarantees.[9] By combining write-ahead logging with a log-structured on-disk format, the Spiralog server merges file system data and recovery log records into a single, sequential write stream.

The Spiralog file system differs from existing log-structured implementations in a number of other important ways, in particular, the mechanisms that we have chosen to use for the cleaner. In subsequent sections of this paper, we compare these differences with existing implementations where appropriate.

Spiralog File System Server Architecture

The Spiralog file system employs a client-server architecture. Each node in the cluster that mounts a Spiralog volume runs a file system clerk. The term clerk is used in this paper to distinguish the client component of the file system from clients of the file system as a whole. Clerks implement all the file functions associated with maintaining the file system state with the exception of persistent storage of file system and user data. This latter responsibility falls on the Spiralog server. There is exactly one server for each volume, which must run on a node that has a direct connection to the disk containing the volume. This distribution of function, where the majority of file system processing takes place on the clerk, is similar to that of the Echo file system.[10] The reasons for choosing this architecture are described in more detail in the paper "Overview of the Spiralog File System," elsewhere in this issue.[11]

Spiralog clerks build files and directories in a structured address space called the file address space. This address space is internal to the file system and is only loosely related to that perceived by clients of the file system. The server provides an interface that allows the clerks to persistently map to file space addresses. Internally, the server uses a logically infinite log structure, built on top of a physical disk, to store the file system data and the structures necessary to locate the data. Figure 1 shows the relationship between the clerks and the server and the relationships among the major components within the server.

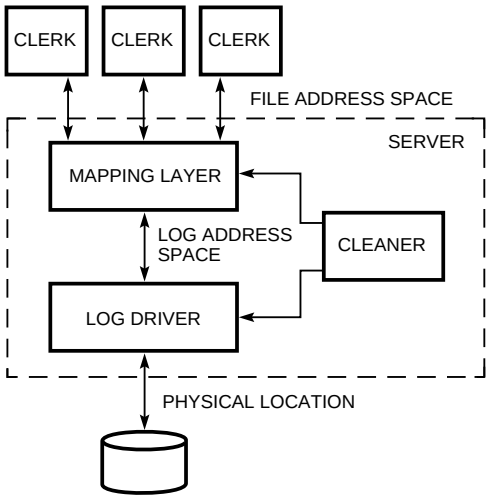

Figure 1
Server Architecture

The mapping layer is responsible for maintaining the mapping between the file address space used by the clerks to the address space of the log. The server directly supports the file address space so that it can make use of information about the relative performance sensitivity of parts of the address space that is implicit within its structure. Although this results in the mapping layer being relatively complex, it reduces the complexity of the clerks and aids performance. The mapping layer is the primary point of contact with the server. Here, read and write requests from clerks are received and translated into operations on the log address space.

The log driver (LD) creates the illusion of an infinite log on top of the physical disk. The LD transforms read and write requests from the mapping layer that are cast in terms of a location in the log address space into read and write requests to physical addresses on the underlying disk. Hiding the implementation of the log from the mapping layer allows the organization of the log to be altered transparently to the mapping layer. For example, parts of the log can be migrated to other physical devices without involving the mapping layer.

Although the log exported by the LD layer is conceptually infinite, disks have a finite size. The cleaner is responsible for garbage collecting or coalescing free space within the log.

Figure 2 shows the relationship between the various address spaces making up the Spiralog file system. In the next three sections, we examine each of the components of the server.

Mapping Layer

The mapping layer implements the mapping between the file address space used by the file system clerks and the log address space maintained by the LD. It exports an interface to the clerks that they use to read data from locations in the file address space, to write new data to the file address space, and to specify which previously written data is no longer required. The interface also allows clerks to group sets of dependent writes into units that succeed or fail as if they were a single write. In this section, we introduce the file address space and describe the data structure used to map it. Then we explain the method used to handle clerk requests to modify the address space.

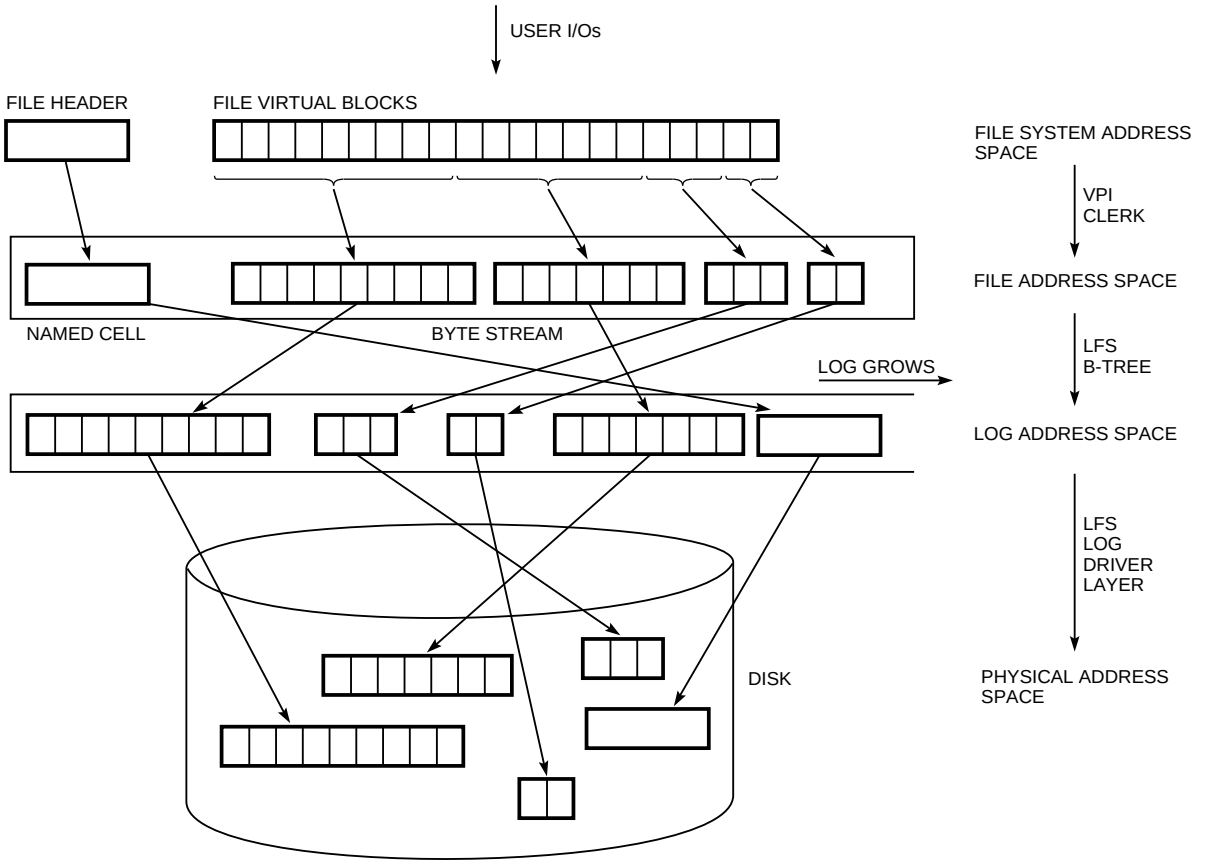

Figure 2
Address Translation

File Address Space

The file address space is a structured address space. At its highest level it is divided into objects, each of which has a numeric object identifier (OID). An object may have any number of named cells associated with it and up to $2^{16}-1$ streams. A named cell may contain a variable amount of data, but it is read and written as a single unit. A stream is a sequence of bytes that are addressed by their offset from the start of the stream, up to a maximum of $2^{64}-1$. Fundamentally, there are two forms of addresses defined by the file address space: Named addresses of the form

```
<OID, name>
```

specify an individual named cell within an object, and numeric addresses of the form

```
<OID, stream-id, stream-offset, length>
```

specify a sequence of *length* contiguous bytes in an individual stream belonging to an object.

The clerks use named cells and streams to build files and directories. In the Spiralog file system version 1.0, a file is represented by an object, a named cell containing its attributes, and a single stream that is used to store the file's data. A directory is represented by an object that contains a number of named cells. Each named cell represents a link in that directory and contains what a traditional file system refers to as a directory entry. Figure 3 shows how data files and directories are built from named cells and streams.

The mapping layer provides three principal operations for manipulating the file address space: read, write, and clear. The read operation allows a clerk to read the contents of a named cell, a contiguous range of bytes from a stream, or all the named cells for a particular object that fall into a specified search range. The write operation allows a clerk to write to a contiguous range of bytes in a stream or an individual named cell. The clear operation allows a clerk to remove a named cell or a number of bytes from an object.

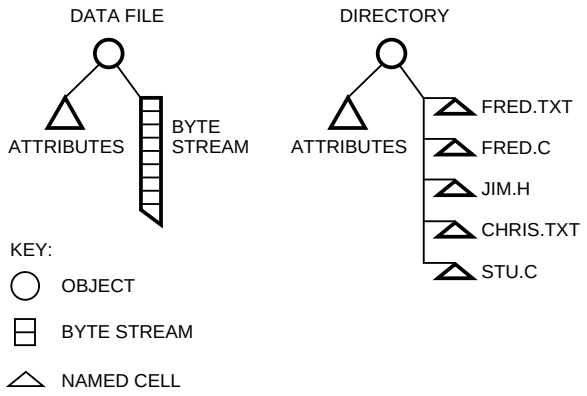

Figure 3
File System

Mapping the File Address Space

We looked at a variety of indexing structures for mapping the file address space onto the log address space.[1,12] We chose a derivative of the B-tree for the following reasons. For a uniform address space, B-trees provide predictable worst-case access times because the tree is balanced across all the keys it maps. A B-tree scales well as the number of keys mapped increases. In other words, as more keys are added, the B-tree grows in width and in depth. Deep B-trees carry an obvious performance penalty, particularly when the B-tree grows too large to be held in memory. As described above, directory entries, file attributes, and file data are all addresses, or keys, in the file address space. Treating these keys as equals and balancing the mapping B-tree across all these keys introduces the possibility that a single directory with many entries, or a file with many extents, may have an impact on the access times for all the files stored in the log.

To solve this problem, we limited the keys for an object to a single B-tree leaf node. With this restriction, several small files can be accommodated in a single leaf node. Files with a large number of extents (or large directories) are supported by allowing individual streams to be spawned into subtrees. The subtrees are balanced across the keys within the subtree. An object can never span more than a single leaf node of the main B-tree; therefore, nonleaf nodes of the main B-tree only need to contain OIDs. This allows the main B-tree to be very compact. Figure 4 shows the relationship between the main B-tree and its subtrees.

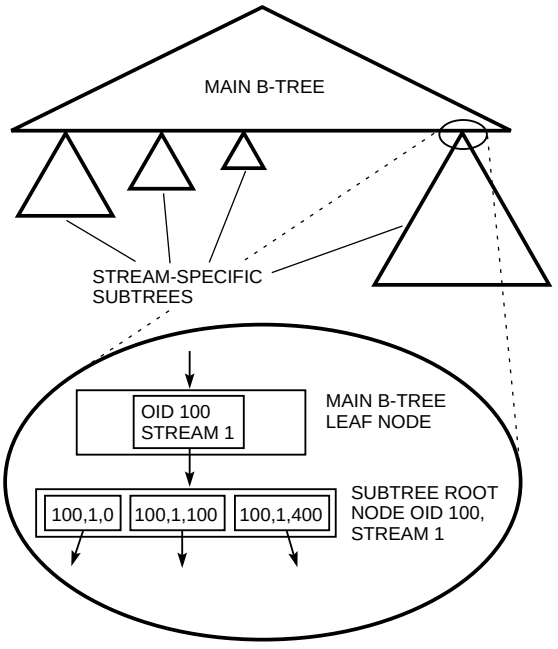

Figure 4
Mapping B-tree Structure

To reduce the time required to open a file, data for small extents and small named cells are stored directly in the leaf node that maps them. For larger extents (greater than one disk block in size in the current implementation), the data item is written into the log and a pointer to it is stored in the node. This pointer is an address in the log address space. Figure 5 illustrates how the B-tree maps a small file and a file with several large extents.

Processing Read Requests

The clerks submit read requests that may be for a sequence of bytes from a stream (reading a data from a file), a single named cell (reading a file's attributes), or a set of named cells (reading directory contents). To fulfill a given read request, the server must consult the B-tree to translate from the address in the file address space supplied by the clerk to the position in the log address space where the data is stored. The extents making up a stream are created when the file data is written. If an application writes 8 kilobytes (KB) of data in 1-KB chunks, the B-tree would contain 8 extents, one for each 1-KB write. The server may need to collect data from several different parts of the log address space to fulfill a single read request.

Read requests share access to the B-tree in much the same way as processes share access to the CPU of a multiprocessing computer system. Read requests arriving from clerks are placed in a first in first out (FIFO) work queue and are started in order of their arrival. All operations on the B-tree are performed by a single worker thread in each volume. This avoids the need for heavyweight locking on individual nodes in the B-tree, which significantly reduces the complexity of the tree manipulation algorithms and removes the potential for deadlocks on tree nodes. This reduction in complexity comes at the cost of the design not scaling with the number of processors in a symmetric multiprocessing (SMP) system. So far we have no evidence to show that this design decision represents a major performance limitation on the server.

The worker thread takes a request from the head of the work queue and traverses the B-tree until it reaches a leaf node that maps the address range of the read request. Upon reaching a leaf node, it may discover that the node contains

- Records that map part or all of the address of the read request to locations in the log, and/or
- Records that map part or all of the address of the read request to data stored directly in the node, and/or
- No records mapping part or all of the address of the read request

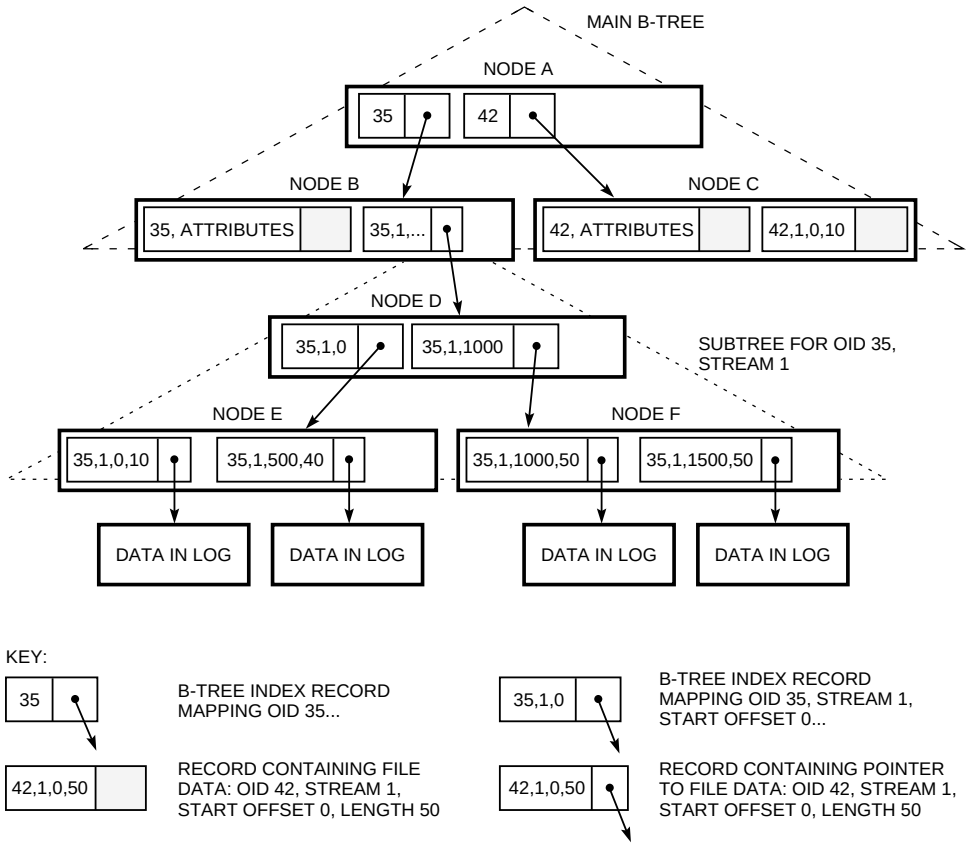

Figure 5
Mapping B-tree Detail

Data that is stored in the node is simply copied to the output buffer. When data is stored in the log, the worker thread issues requests to the LD to read the data into the output buffer. Once all the reads have been issued, the original request is placed on a pending queue until they complete; then the results are returned to the clerk. When no data is stored for all or part of the read request, the server zero-fills the corresponding part of the output buffer.

The process described above is complicated by the fact that the B-tree is itself stored in the log. The mapping layer contains a node cache that ensures that commonly referenced nodes are normally found in memory. When the worker thread needs to traverse through a tree node that is not in memory, it must arrange for the node to be read into the cache. The address of the node in the log is the value of the pointer to it from its parent node. The worker thread uses this to issue a request to the LD to read the node into a cache buffer. While the node read request is in progress, the original clerk operation is placed on a pending queue and the worker thread proceeds to the next request on the work queue. When the node is resident in memory, the pending read request is placed back on the work queue to be restarted. In this way, multiple read requests can be in progress at any given time.

Processing Write Requests

Write requests received by the server arrive in groups consisting of a number of data items corresponding to updates to noncontiguous addresses in the file address space. Each group must be written as a single failure atomic unit, which means that all the parts of the write request must be made stable or none of them must become stable. Such groups of writes are called wunners and are used by the clerk to encapsulate complex file system operations.[11]

Before the server can complete a wunner, that is, before an acknowledgment can be sent back to the clerk indicating that the wunner was successful, the server must make two guarantees:

1. All parts of the wunner are stably stored in the log so that the entire wunner is persistent in the event of a system failure.
2. All data items described by the wunner are visible to subsequent read requests.

The wunner is made persistent by writing each data item to the log. Each data item is tagged with a log record that identifies its corresponding file space address. This allows the data to be recovered in the event of a system failure. All individual writes are made as part of a single compound atomic operation (CAO). This method is provided by the LD layer to bracket a set of writes that must be recovered as an atomic unit. Once all the writes for the wunner have been issued to the log, the mapping layer instructs the LD layer to end (or commit) the CAO.

The wunner can be made visible to subsequent read operations by updating the B-tree to reflect the location of the new data. Unfortunately, this would cause writes to incur a significant latency since updating the B-tree involves traversing the B-tree and potentially reading B-tree nodes into memory from the log. Instead, the server completes a write operation before the B-tree is updated. By doing this, however, it must take additional steps to ensure that the data is visible to subsequent read requests.

Before completing the wunner, the mapping layer queues the B-tree updates resulting from writing the wunner to the same FIFO work queue as read requests. All items are queued atomically, that is, no other read or write operation can be interleaved with the individual wunner updates. In this way, the ordering between the writes making up the wunner and subsequent read or write operations is maintained. Work cannot begin on a subsequent read request until work has started on the B-tree updates ahead of it in the queue.

Once the B-tree updates have been queued to the server work queue and the mapping layer has been notified that the CAO for the writes has committed, both of the guarantees that the server gives on write completion hold. The data is persistent, and the writes are visible to subsequent operations; therefore, the server can send an acknowledgment back to the clerk.

Updating the B-tree

The worker thread processes a B-tree update request in much the same way as a read request. The update request traverses the B-tree until either it reaches the node that maps the appropriate part of the file address space, or it fails to find a node in memory.

Once the leaf node is reached, it is updated to point at the location of the data in the log (if the data is to be stored directly in the node, the data is copied into the node). The node is now dirty in memory and must be written to the log at some point. Rather than writing the node immediately, the mapping layer writes a log record describing the change, locks the node into the cache, and places a flush operation for the node to the mapping layer's flush queue. The flush operation describes the location of the node in the tree and records the need to write it to the log at some point in the future.

If, on its way to the leaf node, the write operation reaches a node that is not in memory, the worker thread arranges for it to be read from the log and the write operation is placed on a pending queue as with a read operation. Because the write has been acknowledged to the clerk, the new data must be visible to subsequent read operations even though the B-tree has not been updated fully. This is achieved by attaching an in-memory record of the update to the node that is

being read. If a read operation reaches the node with records of stalled updates, it must check whether any of these records contains data that should be returned. The record contains either a pointer to the data in the log or the actual data itself. If a read operation finds a record that can satisfy all or part of the request, the read request uses the information in the record to fetch the data. This preserves the guarantee that the clerk must see all data for which the write request has been acknowledged.

Once the node is read in from the log, the stalled updates are restarted. Each update removes its log record from the node and recommences traversing the B-tree from that point.

Writing B-tree Nodes to the Log

Writing nodes consumes bandwidth to the disk that might otherwise be used for writing or reading user data, so the server tries to avoid doing so until absolutely necessary. Two conditions make it necessary to begin writing nodes:

1. There are a large number of dirty nodes in the cache.
2. A checkpoint is in progress.

In the first condition, most of the memory available to the server has been given over to nodes that are locked in memory and waiting to be written to the log. Read and update operations begin to back up, waiting for available memory to store nodes. In the second condition, the LD has requested a checkpoint in order to bound recovery time (see the section Checkpointing later in this paper).

When either of these conditions occurs, the mapping layer switches into flush mode, during which it only writes nodes, until the condition is changed. In flush mode, the worker thread processes flush operations from the mapping layer's flush queue in depth order, that is, starting with the nodes furthest from the root of the B-tree. For each flush operation, it traverses the B-tree until it finds the target node and its parent. The target node is identified by the keys it maps and its level. The level of a node is its distance from the leaf of the B-tree (or subtree). Unlike its depth, which is its distance from the root of the B-tree, a node's level does not change as the B-tree grows and shrinks.

Once it has reached its destination, the flush operation writes out the target node and updates the parent with the new log address. The modifications made to the parent node by the flush operation are analogous to those made to a leaf node by an update operation. In this way, a modification to a leaf node eventually works its way to the root of the B-tree, causing each node in its path to be rewritten to the log over time. Writing dirty nodes only when necessary and then in deepest first order minimizes the number of nodes written to the log and increases the average number of changes that are reflected in each node written.

Log Driver

The log driver is responsible for creating the illusion of a semi-infinite sequential log on top of a physical disk. The entire history of the file system is recorded in the updates made to the log, but only those parts of the log that describe its current or live state need to be persistently stored on the disk. As files are overwritten or deleted, the parts of the log that contain the previous contents become obsolete.

Segments and the Segment Array

To make the management of free space more straightforward, the log is divided into sections called segments. In the Spiralog file system, segments are 256 KB. Segments in the log are identified by their segment identifier (SEGID). SEGIDs increase monotonically and are never reused. Segments in the log that contain live data are mapped to physical, segment-sized locations or slots on the disk that are identified by their segment number (SEGNUM) as shown in Figure 6. The mapping between SEGID and SEGNUM is maintained by the segment array. The segment array also tracks which parts of each mapped segment contain live data. This information is used by the cleaner.

The LD interface layer contains a segment switch that allows segments to be fetched from a location other than the disk.[13] The backup function on the Spiralog file system uses this mechanism to restore files contained in segments held on backup media. Figure 7 shows the LD layer.

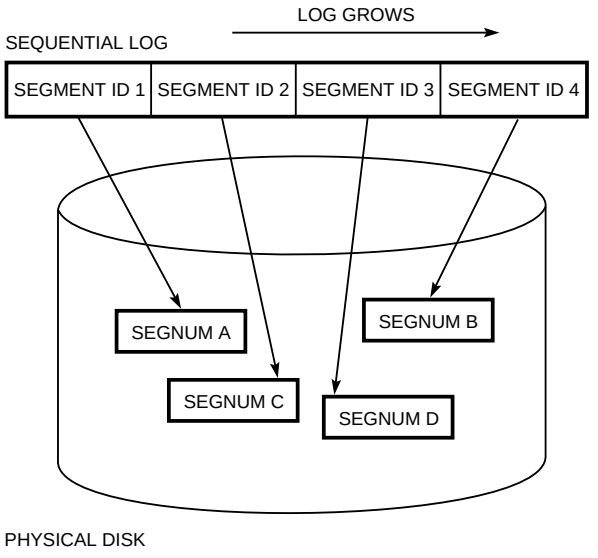

Figure 6
Mapping the Log onto the Disk

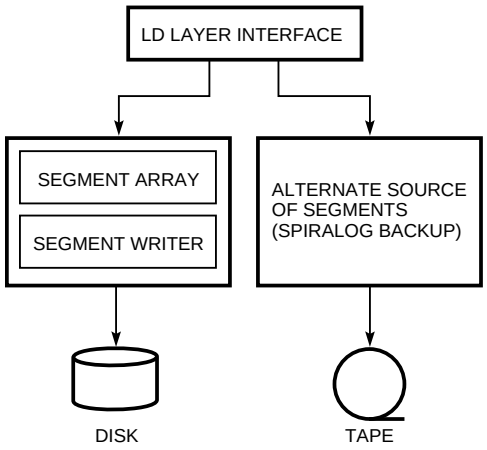

Figure 7
Subcomponents of the LD Layer

The Segment Writer

The segment writer is responsible for all I/Os to the log. It groups together writes it receives from the mapping layer into large, sequential I/Os where possible. This increases write throughput, but at the potential cost of increasing the latency of individual operations when the disk is lightly loaded.

As shown in Figure 8, the segment writer is responsible for the internal organization of segments written to the disk. Segments are divided into two sections, a data area and a much smaller commit record area. Writing a piece of data requires two operations to the segment at the tail of the log. First the data item is written to the data area of the segment. Once this I/O has completed successfully, a record describing that data is written to the commit record area. Only when the write to the commit record area is complete can the original request be considered stable.

The need for two writes to disk (potentially, with a rotational delay between) to commit a single data write is clearly a disadvantage. Normally, however, the segment writer receives a set of related writes from the mapping layer which are tagged as part of a single CAO. Since the mapping layer is interested in the completion of the whole CAO and not the writes within it, the segment writer is able to buffer additions to the commit records area in memory and then write them with a single I/O. Under a normal write load, this reduces the number of I/Os for a single data write to very close to one.

The boundary between the commit record area and the data area is fixed. Inevitably, this wastes space in either the commit record area or data area when the other fills. Choosing a size for the commit record area that minimizes this waste requires some care. After analysis of segments that had been subjected to a typical OpenVMS load, we chose 24 KB as the value for the commit record area.

This segment organization permits the segment writer to have complete control over the contents of the commit record area, which allows the segment writer to accomplish two important recovery tasks:

- Detect the end of the log
- Detect multiblock write failure

When physical segments are reused to extend the log, they are not scrubbed and their commit record areas contain stale (but comprehensible) records. The recovery manager must distinguish between records belonging to the current and the previous incarnation of the physical slot. To achieve this, the segment writer writes a sequence number into a specific byte in every block written to the commit record area. The original contents of the "stolen" bytes are stored within the record being written. The sequence number used for

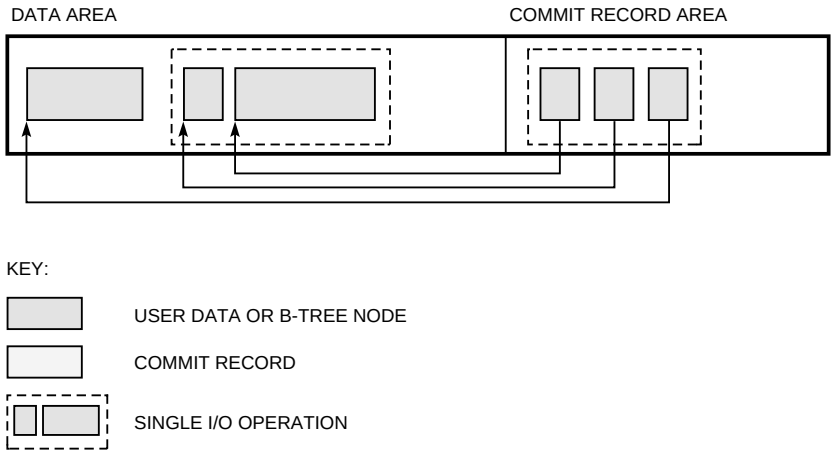

Figure 8
Organization of a Segment

a segment is an attribute of the physical slot that is assigned to it. The sequence number for a physical slot is incremented each time the slot is reused, allowing the recovery manager to detect blocks that do not belong to the segment stored in the physical slot. The cost of resubstituting the stolen bytes is incurred only during recovery and cleaning, because this is the only time that the commit record area is read.

In hindsight, the partitioning of segments into data and commit areas was probably a mistake. A layout that intermingles the data and commit records and that allows them to be written in one I/O would offer better latency at low throughput. If combined with careful writing, command tag queuing, and other optimizations becoming more prevalent in disk hardware and controllers, such an on-disk structure could offer significant improvements in latency and throughput.

Cleaner

The cleaner's job is to turn free space in segments in the log into empty, unassigned physical slots that can be used to extend the log. Areas of free space appear in segments when the corresponding data decays; that is, it is either deleted or replaced.

The cleaner rewrites the live data contained in partially full segments. Essentially, the cleaner forces the segments to decay completely. If the rate at which data is written to the log matches the rate at which it is deleted, segments eventually become empty of their own accord. When the log is full (fullness depends on the distribution of file longevity), it is necessary to proactively clean segments. As the cleaner continues to consume more of the disk bandwidth, performance can be expected to decline. Our design goal was that performance should be maintained up to a point at which the log is 85 percent full. Beyond this, it was acceptable for performance to degrade significantly.

Bytes Die Young

Recently written data is more likely to decay than old data.[14,15] Segments that were written a short time ago are likely to decay further, after which the cost of cleaning them will be less. In our design, the cleaner selects candidate segments that were written some time ago and are more likely to have undergone this initial decay.

Mixing data cleaned from older segments with data from the current stream of new writes is likely to produce a segment that will need to be cleaned again once the new data has undergone its initial decay. To avoid mixing cleaned data and data from the current write stream, the cleaner builds its output segments separately and then passes them to the LD to be threaded in at the tail of the log. This has two important benefits:

- The recovery information in the output segment is minimal, consisting only of the self-describing tags on the data. As a result, the cleaner is unlikely to waste space in the data area by virtue of having filled the commit record area.
- By constructing the output segment off-line, the cleaner has as much time as it needs to look for data chunks that best fill the segment.

Remapping the Output Segment

The data items contained in the cleaner's output segment receive new addresses. The cleaner informs the mapping layer of the change of location by submitting B-tree update operation for each piece of data it copied. The mapping layer handles this update operation in much the same way as it would a normal overwrite. This update does have one special property: the cleaner writes are conditional. In other words, the mapping layer will update the B-tree to point to the copy created by the cleaner as long as no change has been made to the data since the cleaner took its copy. This allows the cleaner to work asynchronously to file system activity and avoids any locking protocol between the cleaner and any other part of the Spiralog file system.

To avoid modifying the mapping layer directly, the cleaner does not copy B-tree nodes to its output segment. Instead, it requests the mapping layer to flush the nodes that occur in its input segments (i.e., rewrite them to the tail of the log). This also avoids wasting space in the cleaner output segment on nodes that map data in the cleaner's input segments. These nodes are guaranteed to decay as soon as the cleaner's B-tree updates are processed.

Figure 9 shows how the cleaner constructs an output segment from a number of input segments. The cleaner keeps selecting input segments until either the output segment is full, or there are no more input segments. Figure 9 also shows the set of operations that are generated by the cleaner. In this example, the output segment is filled with the contents of two full segments and part of a third segment. This will cause the third input segment to decay still further, and the remaining data and B-tree nodes will be cleaned when that segment is selected to create another output segment.

Cleaner Policies

A set of heuristics governs the cleaner's operation. One of our fundamental design decisions was to separate the cleaner policies from the mechanisms that implement them.

When to clean?

Our design explicitly avoids cleaning until it is required. This design appears to be a good match for

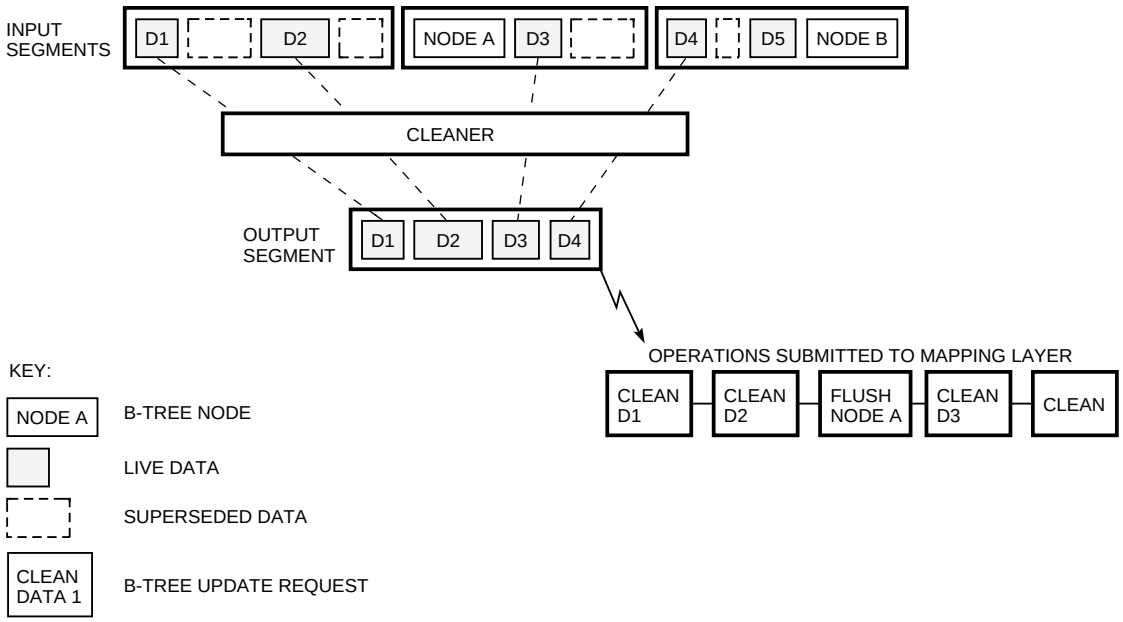

Figure 9
Cleaner Operation

a workload on the OpenVMS system. On our time-sharing system, the cleaner was entirely inactive for the first three months of 1996; although segments were used and reused repeatedly, they always decayed entirely to empty of their own accord. The trade-off in avoiding cleaning is that although performance is improved (no cleaner activity), the size of the full savesnaps created by backup is increased. This is because backup copies whole segments, regardless of how much live data they contain.

When the cleaner is not running, the live data in the volume tends to be distributed across a large number of partially full segments. To avoid this problem, we have added a control to allow the system manager to manually start and stop the cleaner. Forcing the cleaner to run before performing a full backup compacts the live data in the log and reduces the size of the savesnap.

In normal operation, the cleaner will start cleaning when the number of free segments available to extend the log falls below a fixed threshold (300 in the current implementation). In making this calculation, the cleaner takes into account the amount of space in the log that will be consumed by writing data currently held in the clerks' write-behind caches. Thus, accepting data into the cache causes the cleaner to "clear the way" for the subsequent write request from the clerk.

When the cleaner starts, it is possible that the amount of live data in the log is approaching the capacity of the underlying disk, so the cleaner may find nothing to do. It is more likely, however, that there will be free space it can reclaim. Because the cleaner works by forcing the data in its input segments to decay by rewriting, it is important that it begins work while free segments are available. Delaying the decision to start cleaning could result in the cleaner being unable to proceed.

A fixed number was chosen for the cleaning threshold rather than one based on the size of the disk. The size of the disk does not affect the urgency of cleaning at any particular point in time. A more effective indicator of urgency is the time taken for the disk to fill at the maximum rate of writing. Writing to the log at 10 MB per second will use 300 segments in about 8 seconds. With hindsight, we realize that a threshold based on a measurement of the speed of the disk might have been a more appropriate choice.

Input Segment Selection

The cleaner divides segments into four distinct groups:

1. Empty. These segments contain no live data and are available to the LD to extend the log.
2. Noncleanable. These segments are not candidates for cleaning for one of two reasons:
   - The segment contains information that would be required by the recovery manager in the event of a system failure. Segments in this group are always close to the tail of the log and therefore likely to undergo further decay, making them poor candidates for cleaning.
   - The segment is part of a snapshot.[5] The snapshot represents a reference to the segment, so it cannot be reused even though it may no longer contain live data.

3. Preferred noncleanable. These segments have recently experienced some natural decay. The supposition is that they may decay further in the near future and so are not good candidates for cleaning.
4. Cleanable. These segments have not decayed for some time. Their stability makes them good candidates for cleaning.

The transitions between the groups are illustrated in Figure 10. It should be noted that the cleaner itself does not have to execute to transfer segments into the empty state.

The cleaner's job is to fill output segments, not to empty input segments. Once it has been started, the cleaner works to entirely fill one segment. When that segment has been filled, it is threaded into the log; if appropriate, the cleaner will then repeat the process with a new output segment and a new set of input segments. The cleaner will commit a partially full output segment only under circumstances of extreme resource depletion.

The cleaner fills the output segment by copying chunks of data forward from segments taken from the cleanable group. The members of this group are held on a list sorted in order of emptiness. Thus, the first cleaner cycle will always cause the greatest number of segments to decay. As the output segment fills, the smallest chunk of data in the segment at the head of the cleanable list may be larger than the space left in the output segment. In this case, the cleaner performs a limited search down the cleanable list for segments containing a suitable chunk. The required information is kept in memory, so this is a reasonably cheap operation. As each input segment is processed, the cleaner temporarily removes it from the cleanable list. This allows the mapping layer to process the operations the cleaner submitted to it and thereby cause decay to occur before the cleaner again considers the segment as a candidate for cleaning. As the volume fills, the ratio between the number of segments in the cleanable and preferred noncleanable groups is adjusted so that the size of the preferred noncleanable group is reduced and segments are inserted into the cleanable list. If appropriate, a segment in the cleanable list that experiences decay will be moved to the preferred noncleanable list. The preferred noncleanable list is kept in order of least recently decayed. Hence, as it is emptied, the segments that are least likely to experience further decay are moved to the cleanable group.

Recovery

The goal of recovery of any file system is to rebuild the file system state after a system failure. This section describes how the server reconstructs state, both in memory and in the log. It then describes checkpointing, the mechanism by which the server bounds the amount of time it takes to recover the file system state.

Recovery Process

In normal operation, a single update to the server can be viewed as several stages:

1. The user data is written to the log. It is tagged with a self-identifying record that describes its position in the file address space. A B-tree update operation is generated that drives stage 2 of the update process.

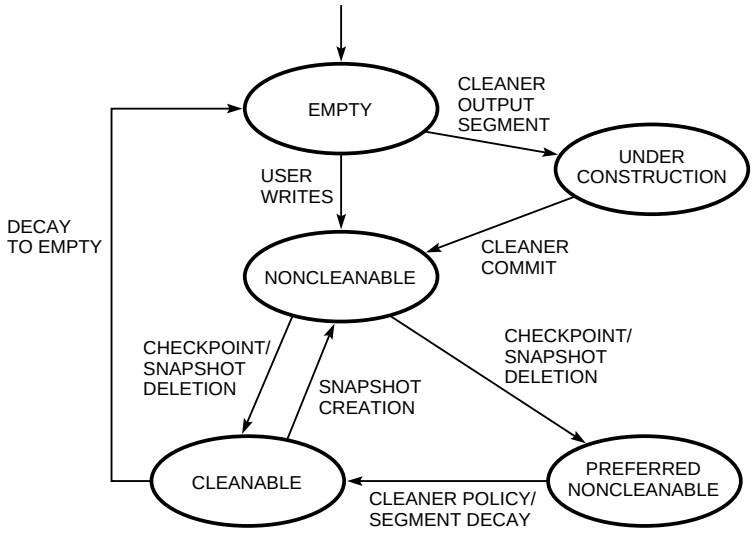

Figure 10
Segment States

2. The leaf nodes of the B-tree are modified in memory, and corresponding change records are written to the log to reflect the position of the new data. A flush operation is generated and queued and then starts stage 3.
3. The B-tree is written out level by level until the root node has been rewritten. As one node is written to the log, the parent of that node must be modified, and a corresponding change record is written to the log. As a parent node is changed, a further flush operation is generated for the parent node and so on up to the root node.

Stage 2 of this process, logging changes to the leaf nodes of the B-tree, is actually redundant. The self-identifying tags that are written with the user data are sufficient to act as change records for the leaf nodes of the B-tree. When we started to design the server, we chose a simple implementation based on physiological write-ahead logging.[9] As time progressed, we moved more toward operational logging.[9] The records written in stage 2 are a holdover from the earlier implementation, which we may remove in a future release of the Spiralog file system.

At each stage of the process, a change record is written to the log and an in-memory operation is generated to drive the update through the next stage. In effect, the change record describes the set of changes made to an in-memory copy of a node and an in-memory operation associated with that change.

Figure 11 shows the log and the in-memory work queue at each stage of a write request. The B-tree describing the file system state consists of three nodes: A, B, and C. A wunner, consisting of a single data write is accepted by the server. The write request requires that both leaf nodes A and B are modified. Stage 1 starts with an empty log and a write request for Data 1.

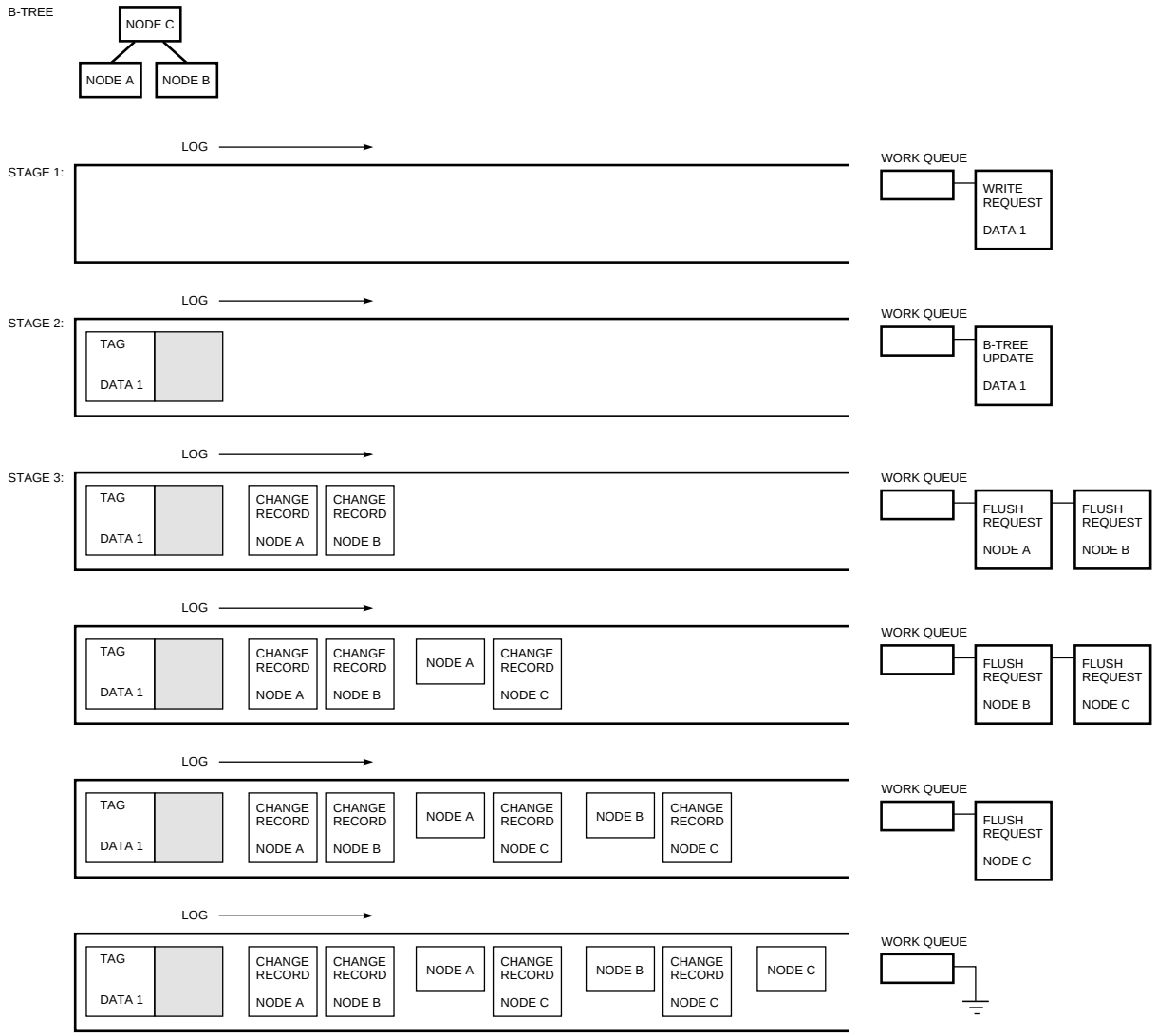

Figure 11
Stages of a Write Request

After a system failure, the server's goal is to reconstruct the file system state to the point of the last write that was written to the log at the time of the crash. This recovery process involves rebuilding, in memory, those B-tree nodes that were dirty and generating any operations that were outstanding when the system failed. The outstanding operations can be scheduled in the normal way to make the changes that they represent permanent, thus avoiding the need to recover them in the event of a future system failure. The recovery process itself does not write to the log.

The mapping layer work queue and the flush lists are rebuilt, and the nodes are fetched into memory by reading the sequential log from the recovery start position (see the section Checkpointing) to the end of the log in a single pass.

The B-tree update operations are regenerated using the self-identifying tag that was written with each piece of data. When the recovery process finds a node, a copy of the node is stored in memory. As log records for node changes are read, they are attached to the nodes in memory and a flush operation is generated for the node. If a log record is read for a node that has not yet been seen, the log record is attached to a placeholder node that is marked as not-yet-seen. The recovery process does not perform reads to fetch in nodes that are not part of the recovery scan. Changes to B-tree nodes are a consequence of operations that happened earlier in the log; therefore, a B-tree node log record has the effect of committing a prior modification. Recovery uses this fact to throw away update operations that have been committed; they no longer need to be applied.

Figure 12 shows a log with change records and B-tree nodes along with the in-memory state of the B-tree node cache and the operations that are regenerated. In this example, change record 1 for node A is superseded or committed by the new version of node A (node A′). The new copy of node C (node C′) supersedes change records 3 and 5. This example also shows the effect of finding a log record without seeing a copy of the node during recovery. The log record for node B is attached to an in-memory version of the node that is marked as not-yet-seen. The data record with self-identifying tag Data 1 generates a B-tree update record that is placed on the work queue for processing. As a final pass, the recovery process generates the set of flush operations that was outstanding when the system failed. The set of flush requests is defined as the set of nodes in the B-tree node cache that has log records attached when the recovery scan is complete. In this case, flush operations for nodes A′ and B are generated.

The server guarantees that a node is never written to the log with uncommitted changes, which means that we only need to log redo records.[9,16] In addition, when we see a node during the recovery scan, any log records that are attached to the previous version of the node in memory can be discarded.

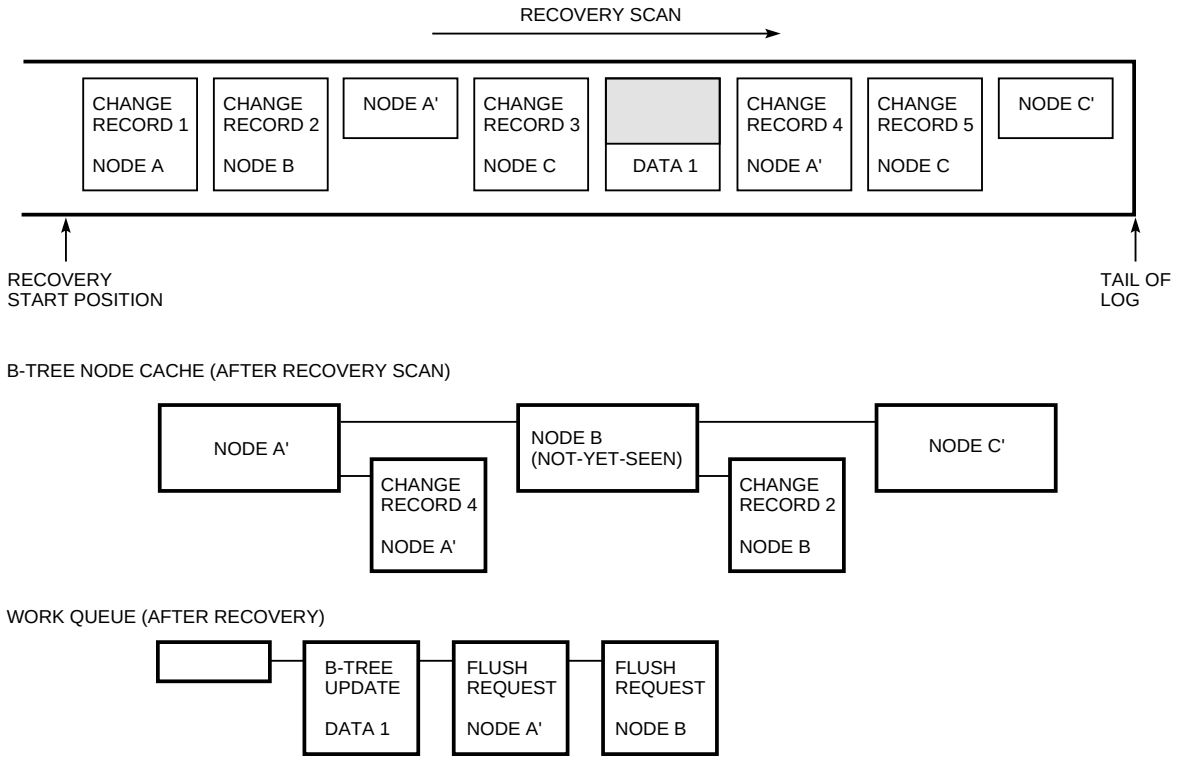

Figure 12
Recovering a Log

Operations generated during recovery are posted to the work queues as they would be in normal running. Normal operation is not allowed to begin until the recovery pass has completed; however, when recovery reaches the end of the log, the server is able to service operations from clerks. Thus new requests from the clerk can be serviced, potentially in parallel with the operations that were generated by the recovery process.

Log records are not applied to nodes during recovery for a number of reasons:

- Less processing time is needed to scan the log and therefore the server can start servicing new user requests sooner.
- Recovery will not have seen copies of all nodes for which it has log records. To apply the log records, the B-tree node must be read from the log. This would result in random read requests during the sequential scan of the log, and again would result in a longer period before user requests could be serviced.
- There may be a copy of the node later in the recovery scan. This would make the additional I/O operation redundant.

Checkpointing

As we have shown, recovering an LFS log is implemented by a single-pass sequential scan of all records in the log from the recovery start position to the tail of the log. This section defines a recovery start position and describes how it can be moved forward to reduce the amount of log that has to be scanned to recover the file system state.

To reconstruct the in-memory state when a system crashed, recovery must see something in the log that represents each operation or change of state that was represented in memory but not yet made stable. This means that at time *t,* the recovery start position is defined as a point in the log after which all operations that are not stably stored have a log record associated with them. Operations obtain the association by scanning the log sequentially from the beginning to the end. The recovery position then becomes the start of the log, which has two important problems:

1. In the worst case, it would be necessary to sequentially scan the entire log to perform recovery. For large disks, a sequential read of the entire log consumes a great deal of time.
2. Recovery must process every log record written between the recovery start position and the end of the log. As a consequence, segments between the start of recovery and the end of the log cannot be cleaned and reused.

To restrict the amount of time to recover the log and to allow segments to be released by cleaning, the recovery position must be moved forward from time to time, so that it is always close to the tail of the log.

Under any workload, a number of outstanding operations are at various stages of completion. In other words, there is no point in the log when all activity has ceased. To overcome this problem, we use a fuzzy checkpoint scheme.[9] In version 1.0 of the Spiralog file system, the server initiates a new checkpoint when 20 MB of data has been written since the previous checkpoint started. The process cannot yet move the recovery position forward in the log to the start of the new checkpoint, because some outstanding operations may have priority. The mapping layer keeps track of the operations that were started before the checkpoint was initiated. When the last of these operations has moved to the next stage (as defined by the recovery process), the mapping layer declares that the checkpoint is complete. Only then can the recovery position be moved forward to the point in the log where the checkpoint was started.

With this scheme, the server does not need to write all the nodes in all paths in the B-tree between a dirty node and the root node. All that is required in practice is to write those nodes that have flush operations queued for them at the time that the checkpoint is started. Flushing these nodes causes change records to be written for their parent nodes after the start of the checkpoint. As the recovery scan proceeds from the start of the last completed checkpoint, it is able to regenerate the flush operation on the parent nodes from these change records.

We chose to base the checkpoint interval on the amount of data written to the log rather than on the amount of time to recover the log. We felt that this would be an accurate measure of how long it would take to recover a particular log. In operation, we find this works well on logs that experience a reasonable write load; however, for logs that predominantly service read requests, the recovery time tends toward the limit. In these cases, it may be more appropriate to add timer-based checkpoints.

Managing Free Space

A traditional, update-in-place file system overwrites superseded data by writing to the same physical location on disk. If, for example, a single block is continually overwritten by a file system client, no extra disk space is required to store the block. In contrast, a log-structured file system appends all modifications to the file system to the tail of the log. Every update to a single block requires log space, not only for the data, but also for the log records and B-tree nodes required to make the B-tree consistent. Although old copies of the data and B-tree nodes are marked as no longer live, this free space is not immediately available for reuse; it must be reclaimed by the cleaner. The goal is to ensure that there is sufficient space in the log to write the

parts of the B-tree that are needed to make the file system structures consistent. This means that we can never have dirty B-tree nodes in memory that cannot be flushed to the log.

The server must carefully manage the amount of free space in the log. It must provide two guarantees:

1. A write will be accepted by the server only if there is sufficient free space in the log to hold the data and rewrite the mapping B-tree to describe it. This guarantee must hold regardless of how much space the cleaner may subsequently reclaim.
2. At the higher levels of the file system, if an I/O operation is accepted, even if that operation is stored in the write-behind cache, the data will be written to the log. This guarantee holds except in the event of a system failure.

The server provides these guarantees using the same mechanism. As shown in Figure 13, the free space and the reserved space in the log are modeled using an escrow function.[17]

The total number of blocks that contain live, valid data is maintained as the used space. When a write operation is received, the server calculates the amount of space in the log that is required to complete the write and update the B-tree, based on the size of the write and the current topology of the B-tree. The calculation is generous because the B-tree is a dynamic structure and the outcome of a single update has unpredictable effects on it. Each clerk reserves space for dirty data that it has stored in the write-behind cache using the same mechanism.

To accept an operation and provide the required guarantees, the server checks the current state of the escrow function. If the guaranteed free space is sufficient, the server accepts the operation. As operations proceed, reserved space is converted to used space as writes are performed. A single write operation may affect several leaf nodes. As it becomes clear how the B-tree is changing, we can convert any unrequired reserved space back to guaranteed free space.

If the cost of an operation exceeds the free space irrespective of how the reserved space is converted, the operation cannot be guaranteed to complete; therefore it is rejected. On the other hand, if the cost of the operation is greater than the guaranteed free space (yet it may fit in the log, depending on the outcome of the outstanding operations), the server enters a "maybe" state. For the server to leave the maybe state and return definitive results, the escrow cost function must be collapsed. This removes any uncertainty by decreasing the reserved space figure, potentially to zero. Operations and unused clerk reservations are drained so that reserved space is converted to either used space or guaranteed free space.

This mechanism provides a fuzzy measure of how much space is available in the log. When it is clear that operations can succeed, they are allowed to continue. If success is doubtful, the operation is held until a definitive yes or no result can be determined. This scheme of free space management is similar to the method described in reference 7.

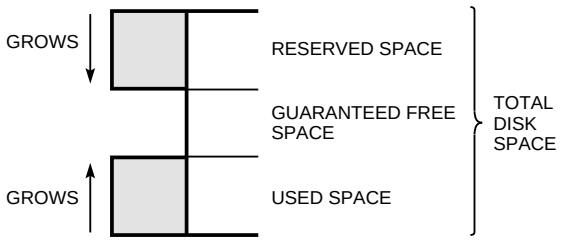

Figure 13
Modeling Free Space

Future Directions

This section outlines some of the possibilities for future implementations of the Spiralog file system.

Hierarchical Storage Management

The Spiralog server distinguishes between the logical position of a segment in the log and its physical location on the media by means of the segment array. This mapping can be extended to cover a hierarchy of devices with differing access characteristics, opening up the possibility of transparent data shelving. Since the unit of migration is the segment, even large, sparsely used files can benefit. Segments containing sections of the file not held on the primary media can be retrieved from slower storage as required. This is identical to the virtual memory paging concept.

For applications that require a complete history of the file system, segments can be saved to archive media before being recycled by the cleaner. In principle, this would make it possible to reconstruct the state of the file system at any time.

Disk Mirroring (RAID 1) Improvements

When a mirrored set of disks is forcefully dismounted with outstanding updates, such as when a system crashes, rebuilding a consistent disk state can be an expensive operation. A complete scan of the members may be necessary because I/Os may have been outstanding to any part of the mirrored set.

Because the data on an LFS disk is temporally ordered, making the members consistent following a failure is much more straightforward. In effect, an LFS allows the equivalent of the minimerge functionality provided by Volume Shadowing for OpenVMS, without the need for hardware support such as I/O controller logging of operations.[18]

Compression

Adding file compression to an update-in-place file system presents a particular problem: what to do when a data item is overwritten with a new version that does not compress to the same size. Since all updates take place at the tail of the log, an LFS avoids this problem entirely. In addition, the amount of space consumed by a data item is determined by its size and is not influenced by the cluster size of the disk. For this reason, an LFS does not need to employ file compaction to make efficient use of large disks or RAID sets.[19]

Future Improvements

The existing implementation can be improved in a number of areas, many of which involve resource consumption. The B-tree mapping mechanism, although general and flexible, has high CPU overheads and requires complex recovery algorithms. The segment layout needs to be revisited to remove the need for serialized I/Os when committing write operations and thus further reduce the write latency.

For the Spiralog file system version 1.0, we chose to keep complete information about live data and data that was no longer valid for every segment in the log. This mechanism allows us to reduce the overhead of the cleaner; however, it does so at the expense of memory and disk space and consequently does not scale well to multi-terabyte disks.

A Final Word

Log structuring is a relatively new and exciting technology. Building Digital's first product using this technology has been both a considerable challenge and a great deal of fun. Our experience during the construction of the Spiralog product has led us to believe that LFS technology has an important role to play in the future of file systems and storage management.

Acknowledgments

We would like to take this opportunity to acknowledge the contributions of the many individuals who helped during the design of the Spiralog server. Alan Paxton was responsible for initial investigations into LFS technology and laid the foundation for our understanding. Mike Johnson made a significant contribution to the cleaner design and was a key member of the team that built the final server. We are very grateful to colleagues who reviewed the design at various stages, in particular, Bill Laing, Dave Thiel, Andy Goldstein, and Dave Lomet. Finally, we would like to thank Jim Johnson and Cathy Foley for their continued loyalty, enthusiasm, and direction during what has been a long and sometimes hard journey.

References

1. D. Gifford, R. Needham, and M. Schroeder, "The Cedar File System," *Communications of the ACM,* vol. 31, no. 3 (March 1988).
2. S. Chutanai, O. Anderson, M. Kazar, and B. Leverett, "The Episode File System," *Proceedings of the Winter 1992 USENIX Technical Conference* (January 1992).
3. M. Rosenblum, "The Design and Implementation of a Log-Structured File System," Report No. UCB/CSD 92/696, University of California, Berkeley (June 1992).
4. J. Ousterhout and F. Douglis, "Beating the I/O Bottleneck: The Case for Log-Structured File Systems," *Operating Systems Review* (January 1989).
5. R. Green, A. Baird, and J. Davies, "Designing a Fast, On-line Backup System for a Log-structured File System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 32–45.
6. J. Ousterhout et al., "A Comparison of Logging and Clustering," Computer Science Department, University of California, Berkeley (March 1994).
7. M. Seltzer, K. Bostic, M. McKusick, and C. Staelin, "An Implementation of a Log-Structured File System for UNIX," *Proceedings of the Winter 1993 USENIX Technical Conference* (January 1993).
8. M. Wiebren de Jounge, F. Kaashoek, and W.-C. Hsieh, "The Logical Disk: A New Approach to Improving File Systems," *ACM SIGOPS '93* (December 1993).
9. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques* (San Mateo, Calif.: Morgan Kaufman Publishers, 1993), ISBN 1-55860-190-2.
10. A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart, "The Echo Distributed File System," Digital Systems Research Center, Research Report 111 (September 1993).
11. J. Johnson and W. Laing, "Overview of the Spiralog File System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 5–14.
12. A. Sweeney et al., "Scalability in the XFS File System," *Proceedings of the Winter 1996 USENIX Technical Conference* (January 1996).
13. J. Kohl, "Highlight: Using a Log-structured File System for Tertiary Storage Management," USENIX Association Conference Proceedings (January 1993).
14. M. Baker et al., "Measurements of a Distributed File System," Symposium on Operating System Principles (SOSP) 13 (October 1991).
15. J. Ousterhout et al., "A Trace-driven Analysis of the UNIX 4.2 BSD File System," Symposium on Operating System Principles (SOSP) 10 (December 1985).

16. D. Lomet and B. Salzberg, "Concurrency and Recovery for Index Trees," Digital Cambridge Research Laboratory, Technical Report (August 1991).

17. P. O'Neil, "The Escrow Transactional Model," *ACM Transactions on Distributed Systems,* vol. 11 (December 1986).

18. *Volume Shadowing for OpenVMS AXP Version 6.1* (Maynard, Mass.: Digital Equipment Corp., 1994).

19. M. Burrows et al., "On-line Data Compression in a Log-structured File System," Digital Systems Research Center, Research Report 85 (April 1992).

Biographies

Christopher Whitaker
Chris Whitaker joined Digital in 1988 after receiving a B.Sc. Eng. (honours, 1st-class) in computer science from the Imperial College of Science and Technology, University of London. He is a principal software engineer with the OpenVMS File System Development Group located near Edinburgh, Scotland. Chris was the team leader for the LFS server component of the Spiralog file system. Prior to this, Chris worked on the distributed transaction management services (DECdtm) for OpenVMS and the port of the OpenVMS record management services (RMS and RMS journaling) to Alpha.

J. Stuart Bayley
Stuart Bayley is a member of the OpenVMS File System Development Group, located near Edinburgh, Scotland. He joined Digital in 1990 and prior to becoming a member of the Spiralog LFS server team, worked on OpenVMS DECdtm services and the OpenVMS XQP file system. Stuart graduated from King's College, University of London, with a B.Sc. (honours) in physics in 1986.

Rod D. W. Widdowson
Rod Widdowson received a B.Sc. (1984) and a Ph.D. (1987) in computer science from Edinburgh University. He joined Digital in 1990 and is a principal software engineer with the OpenVMS File System Development Group located near Edinburgh, Scotland. Rod worked on the implementation of LFS and cluster distribution components of the Spiralog file system. Prior to this, Rod worked on the port of the OpenVMS XQP file system to Alpha. Rod is a charter member of the British Computer Society.