Mark A. Howell
Julian M. Palmer

Integrating the Spiralog File System into the OpenVMS Operating System

Digital's Spiralog file system is a log-structured file system that makes extensive use of write-back caching. Its technology is substantially different from that of the traditional OpenVMS file system, known as Files-11. The integration of the Spiralog file system into the OpenVMS environment had to ensure that existing applications ran unchanged and at the same time had to expose the benefits of the new file system. Application compatibility was attained through an emulation of the existing Files-11 file system interface. The Spiralog file system provides an ordered write-behind cache that allows applications to control write order through the barrier primitive. This form of caching gives the benefits of write-back caching and protects data integrity.

The Spiralog file system is based on a log-structuring method that offers fast writes and a fast, on-line backup capability.[1–3] The integration of the Spiralog file system into the OpenVMS operating system presented many challenges. Its programming interface and its extensive use of write-back caching were substantially different from those of the existing OpenVMS file system, known as Files-11.

To encourage use of the Spiralog file system, we had to ensure that existing applications ran unchanged in the OpenVMS environment. A file system emulation layer provided the necessary compatibility by mapping the Files-11 file system interface onto the Spiralog file system. Before we could build the emulation layer, we needed to understand how these applications used the file system interface. The approach taken to understanding application requirements led to a file system emulation layer that exceeded the original compatibility expectations.

The first part of this paper deals with the approach to integrating a new file system into the OpenVMS environment and preserving application compatibility. It describes the various levels at which the file system could have been integrated and the decision to emulate the low-level file system interface. Techniques such as tracing, source code scanning, and functional analysis of the Files-11 file system helped determine which features should be supported by the emulation.

The Spiralog file system uses extensive write-back caching to gain performance over the write-through cache on the Files-11 file system. Applications have relied on the ordering of writes implied by write-through caching to maintain on-disk consistency in the event of system failures. The lack of ordering guarantees prevented the implementation of such careful write policies in write-back environments. The Spiralog file system uses a write-behind cache (introduced in the Echo file system) to allow applications to take advantage of write-back caching performance while preserving careful write policies.[4] This feature is unique in a commercial file system. The second part of this paper describes the difficulties of integrating write-back caching into a write-through environment and how a write-behind cache addressed these problems.

Providing a Compatible File System Interface

Application compatibility can be described in two ways: compatibility at the file system interface and compatibility of the on-disk structure. Since only specialized applications use knowledge of the on-disk structure and maintaining compatibility at the interface level is a feature of the OpenVMS system, the Spiralog file system preserves compatibility at the file system interface level only. In the section Files-11 and the Spiralog File System On-disk Structures, we give an overview of the major on-disk differences between the two file systems.

The level of interface compatibility would have a large impact on how well users adopted the Spiralog file system. If data and applications could be moved to a Spiralog volume and run unchanged, the file system would be better accepted. The goal for the Spiralog file system was to achieve 100 percent interface compatibility for the majority of existing applications. The implementation of a log-structured file system, however, meant that certain features and operations of the Files-11 file system could not be supported.

The OpenVMS operating system provides a number of file system interfaces that are called by applications. This section describes how we chose the most compatible file system interface. The OpenVMS operating system directly supports a system-level call interface (QIO) to the file system, which is an extremely complex interface.[5] The QIO interface is very specific to the OpenVMS system and is difficult to map directly onto a modern file system interface. This interface is used infrequently by applications but is used extensively by OpenVMS utilities.

OpenVMS File System Environment

This section gives an overview of the general OpenVMS file system environment, and the existing OpenVMS and the new Spiralog file system interfaces. To emulate the Files-11 file system, it was important to understand the way it is used by applications in the OpenVMS environment. A brief description of the Files-11 and the Spiralog file system interfaces gives an indication of the problems in mapping one interface onto the other. These problems are discussed later in the section Compatibility Problems.

In the OpenVMS environment, applications interact with the file system through various interfaces, ranging from high-level language interfaces to direct file system calls. Figure 1 shows the organization of interfaces within the OpenVMS environment, including both the Spiralog and the Files-11 file systems.

The following briefly describes the levels of interface to the file system.

- High-level language (HLL) libraries. HLL libraries provide file system functions for high-level languages such as the Standard C library and FORTRAN I/O functions.
- OpenVMS language-specific libraries. These libraries offer OpenVMS-specific file system functions at a high level. For example, lib$create_dir( ) creates a new directory with specific OpenVMS security attributes such as ownership.
- Record Management Services. The OpenVMS Record Management Services (RMS) are a set of complex routines that form part of the OpenVMS kernel. These routines are primarily used to access structured data within a file. However, there are also routines at the file level, for example, open, close, delete, and rename. The RMS parsing routines for file search and open give the OpenVMS operating system a consistent syntax for file names. These routines also provide file name parsing operations for higher level libraries. RMS calls to the file system are treated in the same way as direct application calls to the file system.

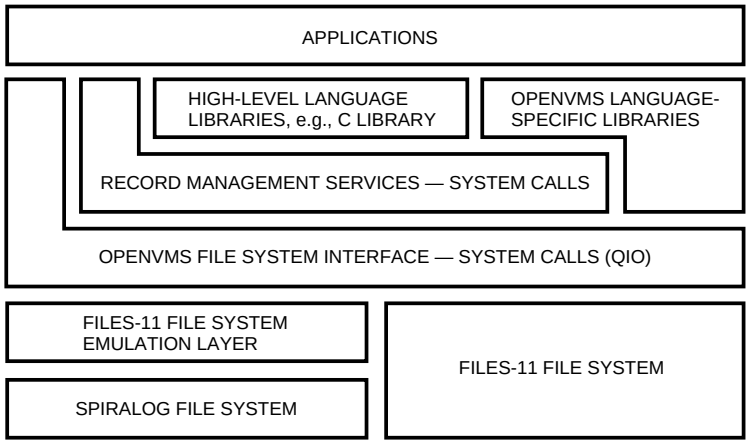

Figure 1
The OpenVMS File System Environment

- Files-11 file system interface. The OpenVMS operating system has traditionally provided the Files-11 file system for applications. It provides a low-level file system interface so that applications can request file system operations from the kernel.

  Each file system call can be composed of multiple subcalls. These subcalls can be combined in numerous permutations to form a complex file system operation. The number of permutations of calls and subcalls makes the file system interface extremely difficult to understand and use.
- File system emulation layer. This layer provides a compatible interface between the Spiralog file system and existing applications. Calls to export the new features available in the Spiralog file system are also included in this layer. An important new feature, the write-behind cache, is described in the section Overview of Caching.
- The Spiralog file system interface. The Spiralog file system provides a generic file system interface. This interface was designed to provide a superset of the features that are typically available in file systems used in the UNIX operating system. File system emulation layers, such as the one written for Files-11, could also be written for many different file systems.[6] Features that could not be provided generically, for example, the implementation of security policies, are implemented in the file system emulation layer.

  The Spiralog file system's interface is based on the Virtual File System (VFS), which provides a file system interface similar to those found on UNIX systems.[7] Functions available are at a higher level than the Files-11 file system interface. For example, an atomic rename function is provided.

Files-11 and the Spiralog File System

On-disk Structures

A major difference between the Files-11 and the Spiralog file systems is the way data is laid out on the disk. The Files-11 system is a conventional, update-in-place file system.[8] Here, space is reserved for file data, and updates to that data are written back to the same location on the disk. Given this knowledge, applications could place data on Files-11 volumes to take advantage of the disk's geometry. For example, the Files-11 file system allows applications to place files on cylinder boundaries to reduce seek times.

The Spiralog file system is a log-structured file system (LFS). The entire volume is treated as a continuous log with updates to files being appended to the tail of the log. In effect, files do not have a fixed home location on a volume. Updates to files, or cleaner activity, will change the location of data on a volume. Applications do not have to be concerned where their data is placed on the disk; LFS provides this mapping.

With the advent of modern disks in the last decade, the exact placement of data has become much less critical. Modern disks frequently return geometry information that does not reflect the exact geometry of the disk. This nullifies any advantage that exact placement on the disk offers to applications. Fortunately, with the Files-11 file system, the use of exact file placement is considered a hint to the file system and can be safely ignored.

Interface Decision

Many features of the Spiralog file system and the Files-11 file system are not directly compatible. To enable existing applications to use the Spiralog file system, a suitable file system interface had to be selected and emulated. The file system emulation layer would need to hook into an existing kernel-level file system interface to provide existing applications with access to the Spiralog file system.

Analysis of existing applications showed that the majority of file system calls came through the RMS interface. This provides a functionally simpler interface onto the lower level Files-11 interface. Most applications on the OpenVMS operating system use the RMS interface, either directly or through HLL libraries, to access the file system.

Few applications make direct calls to the low-level Files-11 interface. Calls to this interface are typically made by RMS and OpenVMS utilities that provide a simplified interface to the file system. RMS supports file access routines, and OpenVMS utilities support modification of file metadata, for example, security information. Although few in number, those applications that do call the Files-11 file system directly are significant ones. If the only interface supported was RMS, then these utilities, such as SET FILE and OpenVMS Backup, would need significant modification. This class of utilities represents a large number of the OpenVMS utilities that maintain the file system.

To provide support for the widest range of applications, we selected the low-level Files-11 interface for use by the file system emulation layer. By selecting this interface, we decreased the amount of work needed for its emulation. However, this gain was offset by the increased complexity in the interface emulation.

Problems caused by this interface selection are described in the next section.

Interface Compatibility

Once the file system interface was selected, choices had to be made about the level of support provided by the emulation layer. Due to the nature of the log-structured file system, described in the section Files-11 and the Spiralog File System On-disk Structures, full compatibility of all features in the emulation layer was not possible. This section discusses some of the decisions made concerning interface compatibility.

An initial decision was made to support documented low-level Files-11 calls through the emulation layer as often as possible. This would enable all well-behaved applications to run unchanged on the Spiralog file system. Examples of well-behaved applications are those that make use of HLL library calls. The following categories of access to the file system would not be supported:

- Those directly accessing the disk without going through the file system
- Those making use of specific on-disk structure information
- Those making use of undocumented file system features

A very small number of applications fell into these categories. Examples of applications that make use of on-disk structure knowledge are the OpenVMS boot code, disk structure analyzers, and disk defragmenters.

The majority of OpenVMS applications make file system calls through the RMS interface. Using file system call-tracing techniques, described in the section Investigation Techniques, a full set of file system calls made by RMS could be constructed. After analysis of this trace data, it was clear that RMS used a small set of well-structured calls to the low-level file system interface. Further, detailed analysis of these calls showed that all RMS operations could be fully emulated on the Spiralog file system.

The support of OpenVMS file system utilities that made direct calls to the low-level Files-11 interface was important if we were to minimize the amount of code change required in the OpenVMS code base. Analysis of these utilities showed that the majority of them could be supported through the emulation layer.

Very few applications made use of features of the Files-11 file system that could not be emulated. This enabled a high number of applications to run unchanged on the Spiralog file system.

Compatibility Problems

This section describes some of the compatibility problems that we encountered in developing the emulation layer and how we resolved them.

When considering the compatibility of the Spiralog file system with the Files-11 file system, we placed the features of the file system into three categories: supported, ignored, and not supported. Table 1 gives examples and descriptions of these categories. A feature was recategorized only if it could be supported but was not used, or if it could not be easily supported but was used by a wide range of applications.

The majority of OpenVMS applications make supported file system calls. These applications will run as intended on the Spiralog file system. Few applications make calls that could be safely ignored. These applications would run successfully but could not make use of these features. Very few applications made calls that were not supported. Unfortunately, some of these applications were very important to the success of the Spiralog file system, for example, system management utilities that were optimized for the Files-11 system.

Analysis of applications that made unsupported calls showed the following categories of use:

- Those that accessed the file header—a structure used to store a file's attributes. This method was used to return multiple file attributes in one call. The supported mechanism involved an individual call for each attribute.

  This was solved by returning an emulated file header to applications that contained the majority of information interesting to applications.
- Those reading directory files. This method was used to perform fast directory scans. The supported mechanism involved a file system call for each name.

  This was solved by providing a bulk directory reading interface call. This call was similar to the getdirentries( ) call on the UNIX system and was

Table 1
Categorization of File System Features

Category	Examples	Notes
Supported. The operation requested was completed, and a success status was returned.	Requests to create a file or open a file.	Most calls made by applications belong in the supported category.
Ignored. The operation requested was ignored, and a success status was returned.	A request to place a file in a specific position on the disk to improve performance.	This type of feature is incompatible with a log-structured file system. It is very infrequently used and not available through HLL libraries. It could be safely ignored.
Not supported. The operation requested was ignored, and a failure status was returned.	A request to directly read the on-disk structure.	This type of request is specific to the Files-11 file system and could be allowed to fail because the application would not work on the Spiralog file system. It is used only by a few specialized applications.

straightforward to replace in applications that directly read directories.

The OpenVMS Backup utility was an example of a system management utility that directly read directory files. The backup utility was changed to use the directory reading call on Spiralog volumes.

- Those accessing reserved files. The existing file system stores all its metadata in normal files that can be read by applications. These files are called reserved files and are created when a volume is initialized.

  No reserved files are created on a Spiralog volume, with the exception of the master file directory (MFD). Applications that read reserved files make specific use of on-disk structure information and are not supported with the Spiralog file system. The MFD is used as the root directory and performs directory traversals. This file was virtually emulated. It appears in directory listings of a Spiralog volume and can be used to start a directory traversal, but it does not exist on the volume as a real file.

Investigation Techniques

This section describes the approach taken to investigate the interface and compatibility problems described above. Results from these investigations were used to determine which features of the Files-11 file system needed to be provided to produce a high level of compatibility.

The investigation focused on understanding how applications called the file system and the semantics of the calls. A number of techniques were used in lieu of design documentation for applications and the Files-11 file system. These techniques were also used to avoid the direct examination of source code.

The following techniques were used to understand application calls to the file system:

- Tracing file system operations

  Tracing file system operations provided a large amount of data for applications. A modified Files-11 file system was constructed that logged all file operations on a volume. A full set of regression tests were then run for the 25 Digital and third-party products most often layered on the Files-11 file system. The data was then reduced to determine the type of file system calls made by the layered products. Analysis of log data showed that most layered products made file system calls through HLL libraries or the RMS interface. This technique is useful where source code is not available, but full code path coverage is available to construct a full picture of calls and arguments.

- Surveying application maintainers on file system use

  Surveying application maintainers was a potentially useful technique for alerting the other maintainers about the impact of the Spiralog file system. More than 2,000 surveys were sent out, but fewer than 25 useful results were returned. Sadly, most application maintainers were not aware of how their product used the file system.

- Automated application source code searching

  Automated source code searching quickly checks a large amount of source code. This technique was most useful when analyzing file system calls made by the OpenVMS operating system or utilities. However, this does not work well when applications make dynamic calls to the file system at run time.

The following techniques were used to understand the semantics of file system calls:

- Functional analysis of the Files-11 file system

  Functional analysis of the Files-11 file system was one of the most useful techniques adopted. It avoided the need to reverse-engineer the Files-11 file system. Whenever possible, the Files-11 file system was treated as a black box, and its function was inferred from interface documentation and application calls. This technique avoided duplicating defects in the interface and enabled the design of the emulation layer to be derived from function, rather than the existing implementation of the Files-11 system.

- Test programs to determine call semantics

  Test programs were used extensively to isolate specific application calls to the file system. Individual calls could be analyzed to determine how they worked with the Files-11 file system and with the emulation layer. This technique formed the basis for an extensive file system interface regression test suite without requiring the complete application.

Level of Compatibility Achieved

The level of file system compatibility with applications far exceeded our initial expectations. Table 2 summarizes the results of the regression tests used to verify compatibility.

Table 2 illustrates that applications that use the C or the FORTRAN language or the RMS interface to access the file system can be expected to work unchanged. Verification with the top 25 Digital layered products and third-party products shows that all products that do not make specific use of Files-11 on-disk features run with the Spiralog file system. With the version 1.0 release of the Spiralog file system, there are no known compatibility issues.

Providing New Caching Features

The Spiralog file system uses ordered write-back caching to provide performance benefits for applications.

Table 2
Verification of Compatibility

Test Suite	Number of Tests	Result
RMS regression tests	~500	All passed.
OpenVMS regression tests	~100	All passed.
Files-11 compatibility tests	~100	All passed.
C2 security test suite	~50 discrete tests	All passed, giving the Spiralog file system the same potential security rating as the Files-11 system.
C language tests	~2,000	All passed.
FORTRAN language tests	~100	All passed.

Write-back caching provides very different semantics to the model of write-through caching used on the Files-11 file system. The goal of the Spiralog project members was to provide write-back caching in a way that was compatible with existing OpenVMS applications.

This section compares write-through and write-back caching and shows how some important OpenVMS applications rely on write-through semantics to protect data from system failure. It describes the ordered write-back cache as introduced in the Echo file system and explains how this model of caching (known as write-behind caching) is particularly suited to the environment of OpenVMS Cluster systems and the Spiralog log-structured file system.

Overview of Caching

During the last few years, CPU performance improvements have continued to outpace performance improvements for disks. As a result, the I/O bottleneck has worsened rather than improved. One of the most successful techniques used to alleviate this problem is caching. Caching means holding a copy of data that has been recently read from, or written to, the disk in memory, giving applications access to that data at memory speeds rather than at disk speeds.

Write-through and write-back caching are two different models frequently used in file systems.

- Write-through caching. In a write-through cache, data read from the disk is stored in the in-memory cache. When data is written, a copy is placed in the cache, but the write request does not return until the data is on the disk. Write-through caches improve the performance of read requests but not write requests.
- Write-back caching. A write-back cache improves the performance of both read and write requests. Reads are handled exactly as in a write-through cache. This time though, a write request returns as soon as the data has been copied to the cache; some time later, the data is written to the disk. This method allows both read and write requests to operate at main memory speeds. The cache can also amalgamate write requests that supersede one another. By deferring and amalgamating write requests, a write-back cache can issue many fewer write requests to the disk, using less disk bandwidth and smoothing the write pattern over time.

Figure 2 shows the write-through and write-back caching models. The Spiralog file system makes extensive use of caching, providing both write-through and write-back models. The use of write-back caching allows the Spiralog file system to amalgamate writes, thus conserving disk bandwidth. This is especially important in an OpenVMS Cluster system where disk bandwidth is shared by several computers. The Spiralog file system attempts to amalgamate not just data writes but also file system operations. For example, many compilers create temporary files that are deleted at the end of the compilation. With write-back caching, it is possible that this type of file may be created and deleted without ever being written to the disk.

There are two disadvantages of write-back caching: (1) if the system fails, any write requests that have not been written to the disk are lost, and (2) once in the cache, any ordering of the write requests is lost. The data may be written from the cache to the disk in a completely different order than the order in which the application issued the write requests. To preserve data integrity, some applications rely on write ordering and the use of careful write techniques. (Careful writing is discussed further in the section below.) The Spiralog file system preserves data integrity by providing an ordered write-back cache known as a write-behind cache.

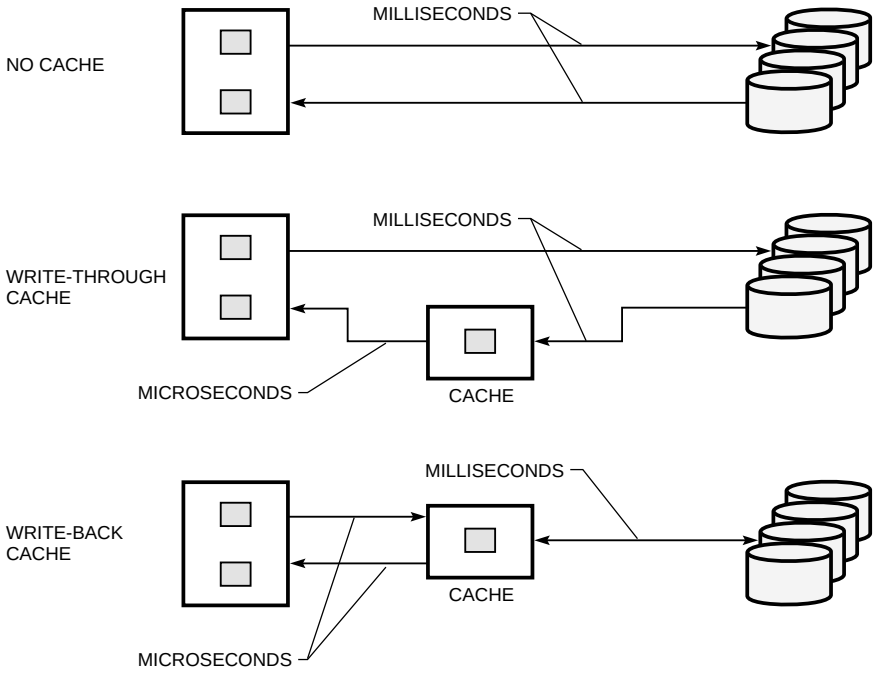

Figure 2
Caching Models

Caching is more important to the Spiralog file system than it is to conventional file systems. Log-structured file systems have inherently worse read performance than conventional, update-in-place file systems, due to the need to locate the data in the log. As described in another paper in this *Journal,* locating data in the log requires more disk I/Os than an update-in-place file system.[2] The Spiralog file system uses large read caches to offset this extra read cost.

Careful Writing

The Files-11 file system provides write-through semantics. Key OpenVMS applications such as transaction processing and the OpenVMS Record Management Services (RMS) have come to rely on the implicit ordering of write-through. They use a technique known as careful writing to prevent data corruption following a system failure.

Careful writing allows an application to ensure that the data on the disk is never in an inconsistent or invalid state. This guarantee avoids situations in which an application has to scan and possibly rebuild the data on the disk after a system failure. Recovery to a consistent state after a system failure is often a very complex and time-consuming task. By ensuring that the disk can never be inconsistent, careful writing removes the need for this form of recovery.

Careful writing is used in situations in which an update requires several blocks on the disk to be written. Most disks guarantee atomic update of only a single disk block. The occurrence of a system failure while several blocks are being updated could leave the blocks partially updated and inconsistent. Careful writing avoids this risk by defining the order in which the blocks should be updated on the disk. If the blocks are written in this order, the data will always be consistent.

For example, the file shown in Figure 3 represents a persistent data structure. At the start of the file is an index block, I, that points to two data blocks within the file, A and B. The application wishes to update the data (A, B) to the new data (A′, B′). For the file to be valid, the index must point to a consistent set of data blocks. So, the index must point either to (A, B) or to (A′, B′). It cannot point to a mixture such as (A′, B). Since the disk can guarantee to write only a single block atomically, the application cannot simply write (A′, B′) on top of (A, B) because that involves writing two blocks. Should the system fail during the updates, doing so could leave the data in an invalid state.

To solve this problem, the application writes the new data to the file in a specific order. First, it writes the new data (A′, B′) to a new section of the file, waiting until the data is written to the disk. Once (A′, B′) are known to be on the disk, it atomically updates the index block to point to the new data. The old blocks (A, B) are now obsolete, and the space they consume can be reused. During the update, the file is never in an inconsistent state.

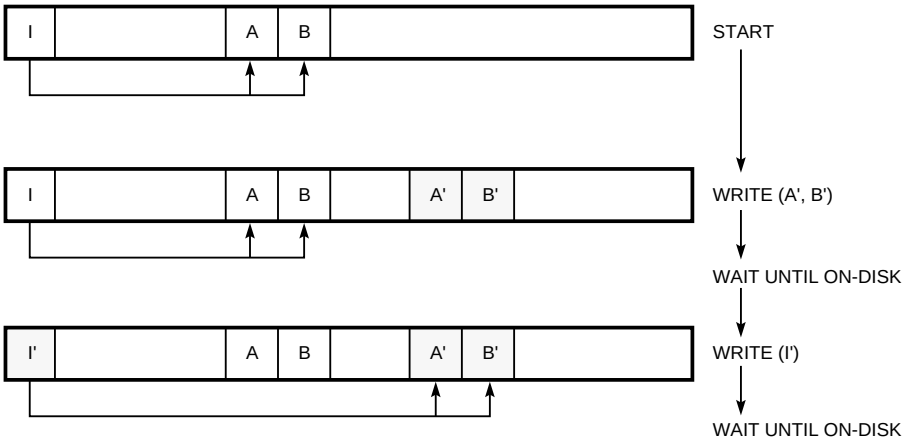

Figure 3
Example of a Careful Write

Write-behind Caching

A careful write policy relies totally on being able to control the order of writes to the disk. This cannot be achieved on a write-back cache because the write-back method does not preserve the order of write requests. Reordering writes in a write-back cache would risk corrupting the data that applications using careful writing were seeking to protect. This is unfortunate because the performance benefits of deferring the write to the disk are compatible with a careful write policy. Careful writing does not need to know when the data is written to the disk, only the order it is written.

To allow these applications to gain the performance of the write-back cache but still protect their data on disk, the Spiralog file system uses a variation on write-back caching known as write-behind caching. Introduced in the Echo file system, write-behind caching is essentially write-back caching with ordering guarantees.[4] The cache allows the application to specify which writes must be ordered and the order in which they must be written to the disk.

This is achieved by providing the barrier primitive to applications. Barrier defines an order or dependency between write operations. For example, consider the diagram in Figure 4: Here, writes are represented as a time-ordered queue, with later writes being added to the tail. In the example, the application issues the writes in the order 1,2,3,4. Without a barrier, the cache could write the data to the disk in any order (for example, 1,3,4,2). If a barrier is placed in the write queue, it specifies to the cache that all writes prior to the barrier must be written to the disk before (or atomically with) any write requests after it. In the example, if a barrier is placed after the second write, the cache file system guarantees that writes 1 and 2 will be written to the disk before writes 3 and 4. Writes 1 and 2 may still be written in any order, as could writes 3 and 4, but 3 and 4 will be written after 1 and 2.

A careful write policy can easily be implemented on a write-behind cache. As shown in Figure 5, the application would use barriers to control the write ordering. Two barriers are required. The first (B1) comes after the writes of the new data (A′, B′). The second (B2) is placed after the index update I′. B1 is required to ensure that the new data is on the disk before the index block is updated. B2 ensures that the index block is updated before any subsequent write requests.

The use of barriers avoids the need to wait for I/Os to reach the disk, improving CPU utilization. In addition, the Spiralog file system allows amalgamation of superseding writes between barriers, reducing the number of requests being written to the disk.

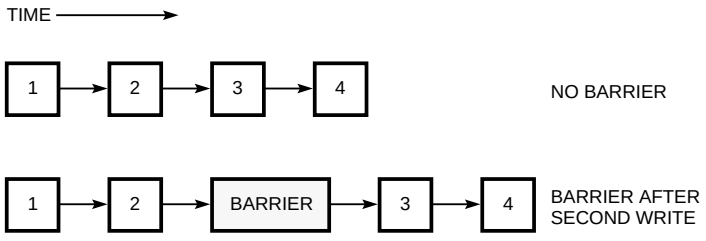

Figure 4
Barrier Insertion in Write Queue

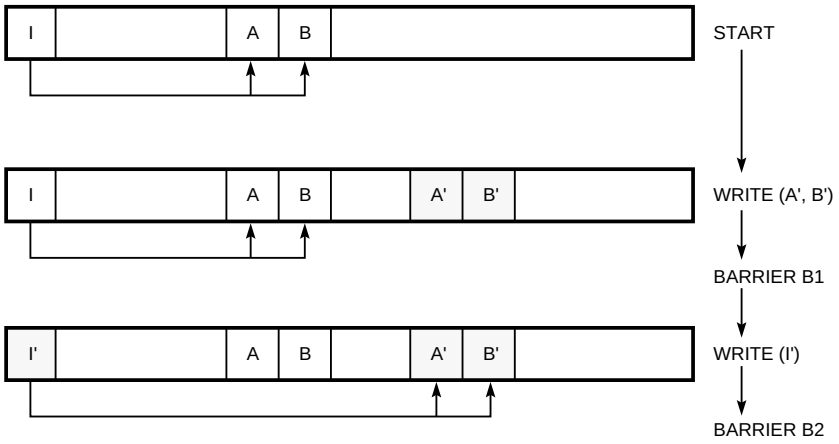

Figure 5
Example of a Careful Write Using Barrier

Internally, the Spiralog file system allows barriers to be placed between any two write operations, even if they are to different files. The Spiralog file system uses this to build its own careful write policy for all changes to files, including metadata changes. This guarantees that the file system is always consistent and gives write-back performance on changes to file metadata as well as data. One major advantage is that the Spiralog file system does not require a disk repair utility such as the UNIX system's fsck to rebuild the file system following a system failure.

Barriers are used internally in several places to preserve the order of updates to the file system metadata. For example, when a file is extended, the allocation of new blocks must be written to the disk before any subsequent data writes to the newly allocated region. A barrier is placed immediately after the write request to update the file length.

Barriers are also used during complex file operations such as a file create. These complex operations frequently update shared resources such as parent directories. The barriers prevent updates to these shared objects, avoiding the risk of corruption due to the updates being reordered by the cache.

At the application level, the Spiralog file system provides the barrier function only within a file. It is not possible to order writes between files. This was sufficient to allow RMS (described in the section OpenVMS File System Environment) to exploit the performance of write-behind caching on most of its file organizations. RMS was enhanced to use barriers in its own careful write policy, which ensures the consistency of complex file organizations, such as indexed files, even when they are subject to write-behind caching. Since the majority of OpenVMS applications access the file system through RMS, gaining write-behind caching on all RMS file organizations provides a significant performance benefit to applications.

Internally, the Spiralog file system supports barriers between files. The decision to support barriers within a file was made to limit the complexity of interface changes, in the belief that a cross-file barrier was of little use to RMS. In retrospect, this proved to be wrong. Some key RMS file organizations use secondary files to hold journal records for the main application file. These file organizations cannot express the order in which updates to the two files should reach the disk, and so are precluded from using write-behind caching.

Application-level Caching Policies

The main problem with the barrier primitive is its requirement that the application express the dependencies to the file system. Although this is unavoidable, it means that the application has to change if it wishes to safely exploit write-behind caching. Clearly, many applications were not going to make these changes. In addition, some applications have on-disk consistency requirements that tie them to a write-through environment.

The file system emulation layer provides additional support for these types of applications by exposing three caching policies to applications. The policies are stored as permanent attributes of the file. By default, when the file is opened by the file system, the permanent caching policy is used on all write requests.

The three policies are described as follows:

1. Write-through caching policy. This policy provides applications with the standard write-through behavior provided by the Files-11 file system. Each write request is flushed to the disk before the application request returns. If an application needs to know what data is on the disk at all times, it should use write-through caching.
2. Write-behind caching policy. A pure write-behind cache provides the highest level of performance. Dirty data is not flushed to the disk when the file is

closed. The semantics of full write-behind caching are best suited to applications that can easily regenerate lost data at any time. Temporary files from a compiler are a good example. Should the system fail, the compilation can be restarted without any loss of data.

3. Flush-on-close caching policy. The flush-on-close policy provides a restricted level of write-behind caching for applications. Here, all updates to the file are treated as write behind, but when the file is closed, all changes are forced to the disk. This gives the performance of write-behind but, in addition, provides a known point when the data is on the disk. This form of caching is particularly suitable for applications that can easily re-create data in the event of a system crash but need to know that data is on the disk at a specific time. For example, a mail store-and-forward system receiving an incoming message must know the data is on the disk when it acknowledges receipt of the message to the forwarder. Once the acknowledgment is sent, the message has been formally passed on, and the forwarder may delete its copy. In this example, the data need not be on the disk until that acknowledgment is sent, because that is the point at which the message receipt is committed. Should the system fail before the acknowledgment is sent, all dirty data in the cache would be lost. In that event, the sender can easily re-create the data by sending the message again.

Figure 6 shows the results of a performance comparison of the three caching policies. The test was run on a dual-CPU DEC 7000 Alpha system with 384 megabytes of memory on a RAID-5 disk. The test repeated the following sequence for the different file sizes.

1. Create and open a file of the required size and set its caching policy.
2. Write data to the whole file in 1,024-byte I/Os.
3. Close the file.
4. Delete the file.

With small files, the number of file operations (create, close, delete) dominates. The leftmost side of the graph therefore shows the time per operation for file operations. With time, the files increase in size, and the data I/Os become prevalent. Hence, the rightmost side of Figure 6 is displaying the time per operation for data I/Os.

Figure 6 clearly shows that an ordered write-behind cache provides the highest performance of the three caching models. For file operations, the write-behind cache is almost 30 percent faster than the write-through cache. Data operations are approximately three times faster than the corresponding operation with write-through caching.

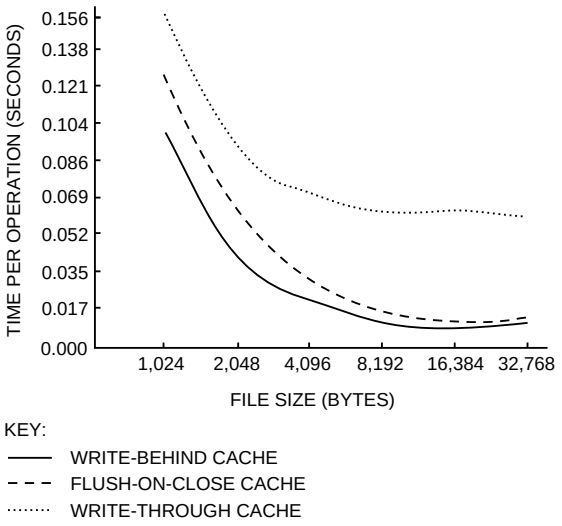

Figure 6
Performance Comparison of Caching Policies

Summary and Conclusions

The task of integrating a log-structured file system into the OpenVMS environment was a significant challenge for the Spiralog project members. Our approach of carefully determining the interface to emulate and the level of compatibility was important to ensure that the majority of applications worked unchanged.

We have shown that an existing update-in-place file system can be replaced by a log-structured file system. Initial effort in the analysis of application usage furnished information on interface compatibility. Most file system operations can be provided through a file system emulation layer. Where necessary, new interfaces were provided for applications to replace their direct knowledge of the Files-11 file system.

File system operation tracing and functional analysis of the Files-11 file system proved to be the most useful techniques to establish interface compatibility. Application compatibility far exceeds the level expected when the project was started. A majority of people use the Spiralog file system volumes without noticing any change in their application's behavior.

Careful write policies rely on the order of updates to the disk. Since write-back caches reorder write requests, applications using careful writing have been unable to take advantage of the significant improvements in write performance given by write-back caching. The Spiralog file system solves this problem by providing ordered write-back caching, known as write-behind. The write-behind cache allows applications to control the order of writes to the disk through a primitive called barrier.

Using barriers, applications can build careful write policies on top of a write-behind cache, gaining all the performance of write-back caching without risking

data integrity. A write-behind cache also allows the file system itself to gain write-back performance on all file system operations. Since many file system operations are themselves quickly superseded, using write-behind caching prevents many file system operations from ever reaching the disk. Barriers also allow the file system to protect the on-disk file system consistency by implementing its own careful write policy, avoiding the need for disk repair utilities.

The barrier primitive provided a way to get write-through semantics within a file for those applications relying on careful write policies. Changing RMS to use the barrier primitive allowed the Spiralog file system to support write-behind caching as the default policy on all file types in the OpenVMS environment.

Acknowledgments

The development of the Spiralog file system involved the help and support of many individuals. We would like to acknowledge Ian Pattison, in particular, who developed the Spiralog cache. We also want to thank Cathy Foley and Jim Johnson for their help throughout the project, and Karen Howell, Morag Currie, and all those who helped with this paper. Finally, we are very grateful to Andy Goldstein, Stu Davidson, and Tom Speer for their help and advice with the Spiralog integration work.

References

1. J. Johnson and W. Laing, "Overview of the Spiralog File System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 5–14.
2. C. Whitaker, S. Bayley, and R. Widdowson, "Design of the Server for the Spiralog File System," *Digital Technical Journal,* vol. 8, no. 2 (1996, this issue): 15–31.
3. 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.
4. 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).
5. *OpenVMS I/O User's Reference Manual* (Maynard, Mass.: Digital Equipment Corporation, 1988).
6. R. Goldenberg and S. Saravanan, *OpenVMS AXP Internals and Data Structures* (Newton, Mass.: Digital Press, 1994).
7. S. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of Summer USENIX Conference,* Atlanta, Ga. (1986): 238–247.
8. K. McCoy, *VMS File System Internals* (Burlington, Mass.: Digital Press, 1990).

Biographies

Mark A. Howell
Mark Howell is an engineering manager in the OpenVMS Engineering Group in Livingston, U.K. Mark was the project leader for Spiralog and wrote some of the product code. He is now managing the follow-on releases to Spiralog version 1.0. In previous projects, Mark contributed to Digital's DECdtm distributed transaction manager, DECdfs distributed file system, and the Alpha port of OpenVMS. Prior to joining Digital, Mark worked on flight simulators and flight software for British Aerospace. Mark received a B.Sc. (honours) in marine biology and biochemistry from Bangor University, Wales. He is one of the rare people who still like interactive fiction (the stuff you have to type, instead of the stuff you point a mouse at.)

Julian M. Palmer
A senior software engineer, Julian Palmer is a member of the OpenVMS Engineering Group in Livingston, Scotland. He is currently working on file system caching for OpenVMS. Prior to his work in file systems, Julian contributed to OpenVMS interprocess communication. Julian joined Digital in 1989 after completing his B.Sc. (honours) in computer science from Edinburgh University.