DirectIO for AdvFS

Design Specification

Version 1.0

JRB

CASL

Building ZK3
110 Spit Brook Road
Nashua, NH 03062

Design Specification Revision History		
Version	**Date**	**Changes**
0.1	06/13/03	First draft for internal review
1.0	07/16/03	First document for general distribution

Table of Contents

1 Introduction 10
1.1 Abstract 10
1.2 Product Identification 10
1.3 Intended Audience 10
1.4 Related Documentation 10
1.5 Purpose of Document 10
1.6 Acknowledgments & Contacts 11
1.7 Terms and Definitions 11
2 Design Overview 12
2.1 Design Approach 12
2.2 Overview of Operation 12
2.3 Major Modules 13
2.3.1 Open/Close Paths 13
2.3.2 Read/Write Paths 14
2.3.3 Strategy Routine 14
2.3.4 fcntl()/ioctl() 14
2.3.5 I/O Start and Completion Paths 14
2.3.7 Interaction with Memory-Mapped files 15
2.4 Dependencies 15
2.4.1 File System 15
2.4.2 I/O System and Drivers 15
2.5 Major Data Structures 15
2.6 Exception Conditions 15
2.7 Design Considerations 16
3 Detailed Design 17
3.1 Data Structure Design 17
3.1.1 file structure 17
3.1.2 bfAccess structure 17
3.1.3 vnode structure 17
3.1.4 advfs_pvt_param structure 18
3.1.5 advfs_pvt_param Structure flags 18
3.1.6 ioanchor_t Structure 18
3.1.7 advfs_args_t Structure 19
3.1.8 actRangeT Structure 19
3.1.9 New Macros 20

3.2 Module Design 21
3.2.1 File Open 21
3.2.1.1 advfs_open() 21
3.2.1.2 Other Changes in this Path 22
3.2.2 File Close 23
3.2.2.1 advfs_close() 23
3.2.2.2 Other Changes in this Path 23
3.2.3 Memory-mapping a file 24
3.2.3.1 advfs_mmap() 24
3.2.4 Open/Close on a cluster 25
3.2.4.1 advfs_cfs_set_cachepolicy() 25
3.2.5 Read Path 27
3.2.5.1 advfs_fs_read() 27
3.2.5.2 advfs_fs_read_direct() 28
3.2.6 Write Path 34
3.2.6.1 advfs_fs_write() 34
3.2.6.2 advfs_fs_write_direct() 35
3.2.7 Routines common to directIO Read and Write paths 44
3.2.7.1 advfs_dio_unaligned_xfer() 44
3.2.8 I/O Completion Path 47
3.2.8.1 advfs_iodone() 47
3.2.8.2 advfs_bs_io_complete() 47
3.2.9 AIO Interface 50
3.2.9.1 advfs_strategy() 50
3.2.9.2 vn_aio_strategy() 51
3.2.10 fcntl Path 52
3.2.10.1 advfs_fcntl() 52
3.2.11 ioctl Path 52
3.2.11.1 advfs_ioctl() 52
3.2.12 Storage Allocation and Snapshotting 52
3.2.12.1 advfs_getpage() 52
3.2.13 Mount Path 54
3.2.13.1 mount() 54
3.2.13.2 advfs_mountfs() 54
3.2.13.3 advfs_vfs_is_mounted_for_dio() 55
3.2.14 Other Miscellaneous Routine Changes 56
3.2.14.1 File Truncate Path 56

3.2.14.2 migrate_normal() 56
3.2.14.3 bs_refpg_direct() 56
3.2.14.4 bs_pinpg_direct() 56
3.2.14.5 Active Range Allocation 56
3.2.14.6 Access structure construction and destruction. 56
3.2.14.7 Routines using the obsolete VDIRECTIO bit 56
3.3 Script Design 57
4 Dependencies 58
4.1 System Administration 58
4.2 Memory Management 58
4.3 ccNUMA 58
4.4 Process Management 58
4.5 File System Layout 58
4.6 File Systems 58
4.7 I/O System and Drivers 58
4.8 Security 58
4.9 Auditing 58
4.10 Multiprocessor 58
4.11 Behavior in a cluster 58
4.12 Kernel Instrumentation/Measurement Systems 59
4.13 Diagnostics 59
4.14 Panic/HPMC/TOC 59
4.15 Commands 59
4.16 Standards 59
4.17 Kernel Debugger 59
4.18 Boot Kernel 59
4.19 Install Kernel 59
4.20 Update/Rolling Upgrade 59
4.21 Support Products 59
4.22 Learning Products (Documentation) 59
5 Issues (Optional) 60
High Priority 60
1 Introduction 7
1.1 Abstract 7
1.2 Product Identification 7
1.3 Intended Audience 7
1.4 Related Documentation 7

1.5 Purpose of Document 8
1.6 Acknowledgments & Contacts 8
1.7 Terms and Definitions 8
2 Design Overview 9
2.1 Design Approach 9
2.2 Overview of Operation 9
2.3 Major Modules 10
2.3.1 Open/Close Paths 10
2.3.2 Read/Write Paths 11
2.3.3 Strategy Routine 11
2.3.4 fcntl()/ioctl() 11
2.3.5 I/O Start and Completion Paths 11
2.3.6 Filesystem Mount Path 12
2.3.7 Interaction with Memory-Mapped files 12
2.4 Dependencies 12
2.4.1 File System 12
2.4.2 I/O System and Drivers 12
2.5 Major Data Structures 13
2.6 Exception Conditions 13
2.7 Design Considerations 13
3 Detailed Design 15
3.1 Data Structure Design 15
3.1.1 file structure 15
3.1.2 bfAccess structure 15
3.1.3 vnode structure 15
3.1.4 advfs_pvt_param structure 16
3.1.5 advfs_pvt_param Structure flags 16
3.1.6 ioanchor_t Structure 16
3.1.7 advfs_args_t Structure 17
3.1.8 actRangeT Structure 17
3.1.9 New Macros 18
3.2 Module Design 19
3.2.1 File Open 19
3.2.1.1 advfs_open() 19
3.2.1.2 Other Changes in this Path 20
3.2.2 File Close 21
3.2.2.1 advfs_close() 21

3.2.2.2 Other Changes in this Path 21
3.2.3 Memory-mapping a file 22
3.2.3.1 advfs_mmap() 22
3.2.4 Open/Close on a cluster 23
3.2.4.1 advfs_cfs_set_cachepolicy() 23
3.2.5 Read Path 25
3.2.5.1 advfs_fs_read() 25
3.2.5.2 advfs_fs_read_direct() 26
3.2.6 Write Path 32
3.2.6.1 advfs_fs_write() 32
3.2.6.2 advfs_fs_write_direct() 33
3.2.7 Routines common to directIO Read and Write paths 42
3.2.7.1 advfs_dio_unaligned_xfer() 42
3.2.8 I/O Completion Path 45
3.2.8.1 advfs_iodone() 45
3.2.8.2 advfs_bs_io_complete() 45
3.2.9 AIO Interface 48
3.2.9.1 advfs_strategy() 48
3.2.9.2 vn_aio_strategy() 49
3.2.10 fcntl Path 50
3.2.10.1 advfs_fcntl() 50
3.2.11 ioctl Path 50
3.2.11.1 advfs_ioctl() 50
3.2.12 Storage Allocation and Snapshotting 50
3.2.12.1 advfs_getpage() 50
3.2.13 Mount Path 52
3.2.13.1 mount() 52
3.2.13.2 advfs_mountfs() 52
3.2.13.3 advfs_vfs_is_mounted_for_dio() 53
3.2.14 Other Miscellaneous Routine Changes 54
3.2.14.1 File Truncate Path 54
3.2.14.2 migrate_normal() 54
3.2.14.3 bs_refpg_direct() 54
3.2.14.4 bs_pinpg_direct() 54
3.2.14.5 Active Range Allocation 54
3.2.14.6 Access structure construction and destruction. 54
3.2.14.7 Routines using the obsolete VDIRECTIO bit 54

3.3 Script Design 55
4 Dependencies 56
4.1 System Administration 56
4.2 Memory Management 56
4.3 ccNUMA 56
4.4 Process Management 56
4.5 File System Layout 56
4.6 File Systems 56
4.7 I/O System and Drivers 56
4.8 Security 56
4.9 Auditing 56
4.10 Multiprocessor 56
4.11 Behavior in a cluster 56
4.12 Kernel Instrumentation/Measurement Systems 57
4.13 Diagnostics 57
4.14 Panic/HPMC/TOC 57
4.15 Commands 57
4.16 Standards 57
4.17 Kernel Debugger 57
4.18 Boot Kernel 57
4.19 Install Kernel 57
4.20 Update/Rolling Upgrade 57
4.21 Support Products 57
4.22 Learning Products (Documentation) 57
5 Issues (Optional) 58
High Priority 58

Preface

Version 1.0 of the DirectIO for AdvFS Design Specification is being made available to the reader for design review. It is also being made available to all partners in order to allow them to design and implement code meeting the specifications contained herein.

If you have any questions or comments regarding this document, please contact:

Author Name	Mailstop	Email Address
JRB		

Sign-off review

Approver Name	Approver Signature	Date

1 Introduction

1.1 Abstract

This document details how the direct I/O functionality of AdvFS on Tru64 UNIX will be ported to AdvFS on HP-UX. Direct I/O (directIO) is a technique that allows applications to read/write data directly from disk into their application buffers by-passing the file system's buffer cache. This can be thought of as a replacement for the use of raw partitions, while retaining the ease of administration (file naming, copy, backup, etc). The ultimate purpose of directIO is to improve the performance for applications such as databases that do not depend on the file system's caching of file pages.

The goal of this design is to port the existing functionality and interface from Tru64 to HP-UX by the target release, while retaining the ability, as much as possible, to add future enhancements.

1.2 Product Identification

Project Name	Project Mnemonic	Target Release Date
AdvFS/DirectIO		

1.3 Intended Audience

This document is intended for review by those interested in the AdvFS design for HP-UX, and its interactions with other kernel layers such as the UFC, VFS, and AIO processing.

1.4 Related Documentation

The following list of references was used in the preparation of this Design Specification. The reader is urged to consult them for more information.

Item	Document	URL
1		
2		
3		

1.5 Purpose of Document

This design document will detail how AdvFS will port its existing directIO functionality on Tru64 UNIX into the AdvFS code base on HP-UX. The intent is to provide sufficient design detail for engineers to begin implementing the design into code.

1.6 Acknowledgments & Contacts

The author would like to gratefully acknowledge the contributions of the following people:

DA, DB, BH, DL, TM, and AZ.

1.7 Terms and Definitions

Term	Definition
AIO	The code layer that implements the POSIX Asynchronous I/O interface. This code implements asynchronous I/O on top of essentially synchronous I/O operations to allow greater application efficiency. This is used extensively by high-performance applications such as Oracle.
Active Range	A form of synchronization used by threads operating on files opened for directIO. Each range is specified for a certain region of the file, allowing threads working on different areas of the file to operate concurrently. Operations guarded by active ranges are directIO reads/writes, migrates, and truncates. The primary interaction prevented is data corruption caused by one thread reading/writing directly to disk while another thread manipulates a cached page that represents the same portion of the file.
FOB	A File Offset Block is a 1k aligned offset in a file. This is being used as a common unit since it will allow page sizes from 1k upwards on 1k boundaries (although in practice page sizes will be >4k and powers of 2).

2 Design Overview

2.1 Design Approach

The design approach chosen for the AdvFS directIO porting project is to utilize the existing Tru64 AdvFS code base and design, while allowing for improvements that may be advantageous to introduce into the HP-UX environment.

2.2 Overview of Operation

This design document is organized principally by functional operation (file open, read, write, ioctl, etc.), with an emphasis on how the directIO operations mesh with and are differentiated from cached I/O operations. The major functional operations include file open, file close, read, write, fcntl, ioctl, and the AIO interface.

In order to understand how these modules work, some background on the algorithms used in AdvFS's directIO on Tru64 UNIX is provided. The following paragraphs describe behavior on Tru64 UNIX. Changes proposed for the HP-UX implementation will be elaborated with the description of each module in the following sections.

On Tru64 AdvFS, a file is opened for directIO using the O_DIRECTIO flag on the open() command. Once a file is opened for directIO by one process, that file is opened for directIO for all processes, and will remain so until all processes close that file. If operating on a standalone system (non-clustered), all dirty pages on the file are flushed to disk, but not invalidated (removed from the cache), when the file is first opened for directIO. If operating in a cluster, the first open causes all pages to be both flushed to disk and invalidated. An application can tell if a file is open in directIO mode by using the fcntl(F_GETCACHEPOLICY) command, but there is no corresponding 'setcachepolicy' command available.

Once a file is opened for directIO, all read() and write() commands are intercepted in the AdvFS read and write routines and dispatched into the directIO read/write paths. A 'typical' path for directIO reads and writes is considered to be one where the data is not already in a cached page, and the data is transferred directly to or from the disk via Direct Memory Access (DMA) without any complications from cached pages. If the data is already in a cached page during a read operation, the data is simply copied to the application buffer from the cached page. There is no attempt to remove pages from the cache on a directIO read operation. If a page is modified by a directIO write, however, the page is removed from the cache and the application data is written directly to the storage device. File operations such as storage allocation, migration, copy-on-write transfers of pages to cloned files, and sometimes truncation, can bring pages into the cache for a file opened for directIO.

To prevent data corruption between the operations that bring pages into the cache, and directIO operations that remove pages from the cache and then write data directly to disk, active ranges are used for synchronization. Active ranges replace a file-wide lock that would otherwise serialize these operations across the entire file. Active ranges allow one thread to operate on one region of a file while other threads operate on different regions of the file. Thus, a file can have directIO writes at the beginning of a file while pages at the end of the file are being migrated to a different disk.

AdvFS using directIO must present data to the device driver in buffers that are an exact multiple of the sector size since it will DMA this data directly to disk. Because the user can request a data transfer of any size, if the request is neither 1) aligned on a sector boundary, nor 2) an even multiple of the sector size, then AdvFS must compensate by reading the underlying sector from the disk, merging the application data, and then writing the sector back to the disk. For this reason, applications that request properly aligned data during a directIO transfer will get better performance than those that do not.

DirectIO reads and writes are inherently synchronous, meaning that the read() or write() call will not return to the application until the data has been transferred to or from the storage device. Because waiting for the I/O to complete is potentially inefficient, applications may take advantage of the POSIX AIO interface that

will allow for an asynchronous I/O. For directIO files on Tru64, the AIO layer will call AdvFS's strategy routine as a special entry point for the I/O. In this case the I/O is set up specially for the read/write paths so those paths will return to the caller before the I/O transfer has been completed.

All calls to read or write a directIO file must ultimately come through the read or write paths. There is no provision for reading files via VOP_GETPAGE, for example, once a file has been opened for directIO. Doing so will result in an error being returned on the getpage call.

Differences between the Tru64 implementation and the proposed HP-UX implementation for each major module will be provided in Section 2.3.

2.3 Major Modules

2.3.1 Open/Close Paths

The open() routine specifying the O_DIRECTIO flag is the method for enabling directIO for a file. The notion that once a file is opened for directIO by one process it is open for directIO for all processes will also be carried forward. However, this design proposes changing one aspect of the open/close semantics. It is proposed that the number of opens for directIO be tracked, and that when all of the threads that opened the file for directIO close the file, the file may remain open, but now in cached I/O mode. This is in contrast to the Tru64 implementation in which the file remains open for directIO until the file is closed by all processes. This is being done to avoid a problem such as the following: A backup utility is written to utilize directIO to read the files to be backed up. As it opens each file for directIO, the performance for existing applications that have the file opened for cached I/O is modified as the file is now open for directIO for all applications. This may be acceptable for the short time it takes the backup to read the file, but in the Tru64 world, the file stays open for directIO until all threads close the file. Under the proposed HP-UX semantics, the file would revert to cached I/O as soon as the backup closed the file.

To do this, a mechanism is needed to track the opens and closes. When the file is opened, a counter of the number of open() calls processed for directIO for this file will be incremented in the file's bfAccess structure. This design proposes that a change will be made to the FMASK definition in fcntl.h that will allow the F_DIRECT bit to remain in the file.f_flags field after the file has been opened. This set of flags gets passed through the close() call, allowing AdvFS to decrement the counter in the bfAccess structure. In this way, when the last thread that opened the file for directIO closes the file, the file will revert to cached I/O behavior. The presence of the F_DIRECT bit in the file structure can also be used by the AIO interface to detect when a file is open for directIO and to dispatch the call to AdvFS's strategy routine.

The ability to track the number of opens/closes for directIO is relatively straightforward on a non-clustered system. On a clustered system, CFS will need to track the number of times a file has been opened for directIO. To prevent the situation where CFS and AdvFS counts for the number of threads that have opened a file for directIO get out of sync, AdvFS will only track whether the file is currently opened for directIO. This will be determined by CFS calling AdvFS with a new private interface routine. Thus, on a clustered system, AdvFS will not track directIO opens and closes, but will respond to explicit requests to set the mode in which the file is opened.

On Tru64, the open mode of the file was tracked in the vnode.v_flags field by the presence of the VDIRECTIO bit. On HP-UX, this bit will not be used; instead it will be replaced with a counter in the bfAccess structure called dioCnt. If this field is 0, the file is open for cached I/O, and if greater than zero, the file is open for directIO.

Another change for the HP-UX version is that when the file is first opened for directIO, all cached pages are flushed to disk and invalidated. On Tru64, the pages were flushed to disk (but not invalidated) on a non-clustered system, and both flushed and invalidated on clustered systems. This change has been adopted primarily because the directIO code on HP-UX cannot tell if pages in the file range are in the cache or not. The simplest way to handle this situation is to eliminate caching as much as possible when the file is first opened for directIO. Then, if processes subsequently bring pages into the cache for this file, they must remove them to eliminate stale cached data.

Detailed code flow for the AdvFS open and close paths, as well as the new CFS interface routine, are given in Sections 3.2.1 through 3.2.4.

2.3.2 Read/Write Paths

Functionally, the AdvFS read and write paths remain largely the same as they are on Tru64, even though the code has been restructured to accommodate the UFC interface. One change that has been made is that directIO reads will no longer be satisfied from cached pages. All cached pages are removed when the file is opened, and no cached pages will be maintained for this file so long as it is open for directIO.

AdvFS Direct I/O and AIO Porting Investigation Report suggested handling unaligned buffers by simply passing these requests through the cached I/O path. The primary goal of this change was to reduce code complexity in the directIO path. This design does not implement that suggestion. The primary reason is that this change would allow those threads that do not 'play by the rules' and issue unaligned I/O requests to populate the cache, forcing other threads (that do play by the rules) to spend cycles flushing and invalidating those pages. Keeping the unaligned request handling in-line forces the additional cycles to be done in the context of the threads that are making the unaligned requests. Since the primary consumers of directIO, the large database vendors, use aligned requests, this prevents the actions of other thread from inhibiting their performance. A secondary reason is that storage allocation for a file opened for directIO may force the code into a very similar path as the unaligned request for zeroing newly-allocated sectors that will not be overwritten by the I/O request. In fact, both unaligned accesses and zeroing these sectors are designed to be handled by a common subroutine.

Details about the code flow for the read and write paths are given in Sections 3.2.5 and 3.2.6.

2.3.3 Strategy Routine

The AdvFS strategy routine, advfs_strategy(), is the entry point into the file system from the AIO read and write paths. On HP-UX this routine is also an interface for the VM Swap Filesystem functionality. A buf structure is passed into the strategy routine where the data is converted to a uio structure and then the AdvFS read or write routines are invoked. The AIO-handling of this routine on HP-UX will work essentially the same way as it does on Tru64, but there are a few changes in the uio and buf structures that necessitate some minor modifications. The use of this routine for the AIO interface is detailed in Section 3.2.9.

2.3.4 fcntl()/ioctl()

A management decision was made to remove the fcntl(F_GETCACHEPOLICY) call in the HP-UX version of AdvFS. This was available for an application to query whether a file was open in directIO or cached I/O mode. This will be replaced with an ioctl() call using the ADVFS_GETCACHEPOLICY command. The movement of this functionality from fcntl() to ioctl() has already been made in the current AdvFS porting baselines.

In the Tru64 code, there is an ioctl(F_SETCACHEPOLICY) call that is used as a private interface between CFS and AdvFS. On HP-UX, this interface is being replaced by a truly private interface, a new routine called advfs_cfs_set_cachepolicy(). The functionality in ioctl() has already been removed in the current AdvFS porting baselines.

2.3.5 I/O Start and Completion Paths

In Tru64, there is code to maintain a count of the number of I/Os started for each active range. This count is used primarily in the AIO path to be sure that all I/Os started for a given active range have been completed before the active range is removed during I/O completion. The ioanchor_t structure in AdvFS on HP-UX serves the same purpose, and therefore this design proposes a simplification of the I/O start and completion code (when servicing an AIO-started I/O) to remove unneeded accounting for I/Os within the active range.

There is a minor change to the I/O completion path proposed so that the status of a multi-part I/O can be determined from the data in the ioanchor structure after the I/O has completed. A multi-part I/O is one for which there are multiple buf structures generated per ioanchor structure. The ioanchor structure change is detailed in Section 3.1.6, and the code changes are listed in Section 3.2.8.

2.3.7 Interaction with Memory-Mapped files

The interaction between directIO and memory mapping will remain the same on HP-UX as it is on Tru64 UNIX. That is, these two modes are mutually exclusive. If a file is already open for directIO, a call to memory map the file will fail with an error. To determine this, the bfap->dioCnt field is checked in advfs_mmap(). Conversely, if a file is memory-mapped, an attempt to open the file for directIO will also fail with an error. This is done by checking the VMMF flag in the vnode in advfs_open(). See Sections 3.2.1 and 3.2.4.

2.4 Dependencies

2.4.1 File System

This project assumes that it is being integrated with the AdvFS code base described in the AdvFS/UBC Integration Design Specification. Use of active ranges to synchronize directIO with other AdvFS code paths are assumed to continue as in the Tru64 code base except where specific changes are specified.

2.4.2 I/O System and Drivers

It is assumed that the block size will be DEV_BSIZE (currently 1k) for disk transfers.

The advfs_strategy() routine is dependent on the buf structure being set up by the AIO calling routine. It is assumed that the yet-to-be-written routine vn_aio_strategy() will set up the data as outlined in Section 3.2.9. However, the routine can be adapted to retrieving the data from different structure fields if that code has already been written.

2.5 Major Data Structures

Data structure changes will be discussed in Section 3.1, Data Structure Design, since most of them are internal AdvFS structures. However, there is one external structure for which this design proposes a change, and that is the file structure. This design proposes adding the FDIRECT flag to the FMASK definition so that the call to close() will know if this process originally opened the file for directIO. See Section 3.1.1 for more details.

2.6 Exception Conditions

The following general classes of exception conditions have been considered for this design.

- Invalid input parameters.
  - Parameter out of range.
  - Referenced data invalid.
- Resource depletion.
  - Memory resources (free memory, physical memory, memory objects (buf structures, etc.)).
  - Hardware resources.
- Race conditions.
  - Sleeping for an event just as it occurs (or missing it completely).
  - Locking protocol deadlocks.
  - Consider cluster-wide races

- Insufficient privilege.
  - User privilege level.
  - Memory access rights.
  - Privilege instructions.
- Hardware errors.
  - Reported errors.
  - Non-responding hardware.
- Power failure.
  - System power failure.
  - Device power failure.
- Multiprocessor.
  - What the other processor is doing.
- Cluster exceptions
  - What are other cluster members doing
  - Failures during failover (i.e. multiple failures)

Specific cases of these general exception conditions that are applicable to this design are discussed in the following subsections.

2.7 Design Considerations

The AdvFS directIO design will be hardware-architecture independent so that a single version will run on both PA-RISC and Itanium systems.

The design will work in multi-CPU environments with the intention of operating in the smallest to largest hardware configurations.

The design attempts to minimize dedicated memory resources by relying on the memory arena allocator to dynamically allocate and free memory for data structures.

AdvFS Direct I/O and AIO Porting Investigation Report suggested adding functionality to the pathconf() command to allow it to return directIO information such as whether directIO is available for a given file, or the directIO read/write granularity. Because of time limitations I have not pursued this feature in this design, but I believe that there are no design issues that would prevent this from being added in the future.

3 Detailed Design

3.1 Data Structure Design

3.1.1 file structure

This design proposes to change the definition of FMASK in file.h so that the FDIRECT bit will be preserved in the file.f_flag field. The file structure is used to retain information about a per-open instance for a given file. The FMASK is used to strip away certain flags that got passed in on the open() call from the flags field before they get saved. If the FDIRECT bit were saved in the flags, then it would be passed to the close routine and used to determine if the directIO count in the bfAccess structure should be decremented. In addition, the AIO layer can use this bit, or an ioctl(ADVFS_GETCACHEPOLICY) to determine if the file was opened for directIO by this file to know that this call should be sent to the AdvFS strategy routine.

To make porting from Tru64 to HP-UX easier for applications, there will be a new #define added to fcntl.h. Currently on HP-UX FDIRECT is #defined to be the same as O_DIRECT, the value used by the application to specify directIO mode. Since Tru64 uses the value O_DIRECTIO to open the file for directIO, the application would not have to be modified if O_DIRECTIO were also #defined to be the same as O_DIRECT. Therefore, the line:

```
#define  O_DIRECTIO  O_DIRECT
```

will be added to fcntl.h. Then either value could be specified on the open() call to enable directIO.

3.1.2 bfAccess structure

In order to maintain a count of the number of times a file has been opened for directIO, a new field is being added to the bfAccess structure. It is called dioCnt, and has already been added to this structure in the AdvFS porting baseline. The addition of this field makes the need for the VDIRECTIO bit in the vnode flags obsolete.

A new read-write lock has also been added to the access structure, the cacheModeLock. This lock is used to guard the dioCnt field, synchronize threads that are changing the file's cache mode (either from cached I/O to directIO or vice versa), to synchronize opening a file for directIO with respect to a file being memory mapped, to allow the read/write paths to reliably determine whether the file is in cached I/O or directIO mode, and to allow the open/close paths to wait for in-flight I/Os when changing the cache mode.

```
typedef struct bfAccess {
    …
    int32_t     refCnt;              /* number of access structure references */
    int32_t     dioCnt;              /* threads having file open for direct I/O */
    rwlock_t    cacheModeLock;       /* guards dioCnt field */
    stateLkT    stateLk;             /* state field */
    …
} bfAccessT;
```

3.1.3 vnode structure

On Tru64, the VDIRECTIO flag in vnode.vflag is used to determine if a file is open for directIO. This flag is not going to be propagated into the HP-UX version, so there is really no change to the vnode structure. However, all places in the code that reference this bit must be examined and have the functionality corrected. These routines are listed in Section 3.2.13.6.

3.1.4 advfs_pvt_param structure

This is a structure used for passing AdvFS information transparently through UFC functions such as fcache_as_uiomove(), fcache_as_fault() and fcache_vn_flush(). The AdvFS getpage or putpage routines are typically the recipient of this information. Getpage or putpage may also return information back through this structure. The directIO write path will call fcache_as_fault() or advfs_getpage() to add storage to a file. Storage may be added to the file that is outside the write range in the application's request because of the request's alignment and the granularity of the storage allocation. If this happens, the directIO write must zero the leading and trailing regions in the newly-allocated storage. Therefore, two new fields will be added to this structure so that advfs_getpage() can return the range of newly-allocated storage to the directIO path. The new fields are shown in bold in the following structure:

```
struct advfs_pvt_param {
        struct bsBuf *app_bp;                   /* Associated bsBuf */
        off_t app_total_bytes;                  /* Total read() or write() length */
        off_t app_starting_offset;              /* Starting Offset of original request */
        uint64_t app_flags;                     /* Flags */
        uint32_t app_started_readahead;         /* TRUE if read-ahead was started */
        uint64_t app_ra_first_page;             /* Read-ahead: first page to read */
        uint64_t app_ra_num_pages;              /* Read-ahead: number of pages to read */
        bf_fob_t app_stg_start_fob;             /* First FOB with newly-allocated storage */
        bf_fob_t app_stg_end_fob;               /* Last FOB with newly-allocated storage. */
};
```

3.1.5 advfs_pvt_param Structure flags

When the directIO write path calls fcache_as_fault() or advfs_getpage() to add storage, it will pass the new APP_ADDSTG_NOCACHE flag in the app_flags field. This will be a signal to advfs_getpage() that it is being called to add storage, but not to zero-fill or bring these pages into the cache.. The use of this new flag in advfs_getpage() is discussed further in Section 3.2.12.

3.1.6 ioanchor_t Structure

DirectIO read and write requests can be broken down into a series of I/Os that make up the total request. This is done by handing a series of buf structures off to the device driver, all of which are associated with the same ioanchor structure. When the last I/O has completed, the thread waiting for I/O wakes up and does whatever processing is required to indicate to the application the status of the entire request. Since the I/Os can be completed out-of-order, and since POSIX requires that the number of bytes transferred from the logical start of the I/O be reported to the application, this design proposes adding several new fields to the ioanchor structure to handle tracking the number of bytes successfully transferred. These fields are shown in bold in the following structure. The anchr_io_status field is always set to EOK or to the return code for the 'first' error encountered. Anchr_min_req_offset should be set to the original (lowest) file offset being requested in the series of I/Os. Its value is not modified. Anchr_min_err_offset should be set to a value greater than any file offset requested in the series. It will be used internally to track the lowest file offset for which an error has occurred. On the last I/O, if an error has been encountered, the value of anchr_min_req_offset is subtracted from this value to get the number of logical bytes successfully transferred up to the first error. The number of bytes transferred is returned in anchr_min_err_offset field.

```
typedef struct  ioanchor  {
    spin_t anchr_lock                   /* Coordinate changes to anchor using lock. */
```

```
                                        /* advfs_iodone() always gets spin lock. */
    int64_t anchr_iocounter             /* Single IO request callers set to 1. Else, */
                                        /* set to number of IO's in multi-IO set */
    uint64_t anchr_flags;               /* Anchor flags for advfs_iodone to check */
    struct buf *anchr_origbuf;          /* Set to the original UFC IO buf structure*/
                                        /* for advfs_iodone to use. */
    condvar_t anchr_cvwait;             /* Optionally allows caller to sleep on this */
                                        /* condition variable until IO completes. */
                                        /* Caller can also use with */
                                        /* IOANCHORFLG_WAKEUP_ON_ALL_IO flag */
    struct ioanchor *anchr_listfwd;     /* Caller can link multiple anchors to */
    struct ioanchor *anchr_listbwd      /* take responsibility for freeing anchors. */
                                        /* Caller must set the */
                                        /* IOANCHOR_KEEP_ANCHOR flag to use the link.*/
    actRangeT *anchr_actrangep;         /*Active range pointer when using */
                                        /* active range locking Otherwise, set to 0.*/
    struct buf *anchr_aio_bp;           /* Asynchronous IO buffer for directIO only.*/
    uint32_t anchr_magicid;             /* Unique structure validation identifier */
    statusT anchr_io_status;            /* Final status of a multi-part I/O */
    off_t  anchr_min_req_offset;          /* Set to the lowest file offset requested in a multi-part
                                          I/O if the number of bytes successfully transferred is to
                                          be calculated and returned in the next field.*/
    off_t  anchr_min_err_offset;          /* In: File offset greater than any in the multi-part I/O */
                                        /* Out: returns the logical number of bytes successfully
                                          transferred for a series of I/Os associated with this
                                          ioanchor.  This value is only meaningful after I/O
                                          completion if anchr_io_status is not EOK.  This field is
                                          used internally to track the offset of the logically lowest
                                          I/O that was not successful.
} ioanchor_t;
```

3.1.7 advfs_args_t Structure

A new bit, MS_ADVFS_DIRECTIO, must be #defined for the advfs_args_t.adv_flags field in advfs_public.h and (temporarily) ms_osf.h. This bit will be set in mount() if the –o directIO option is specified to the mount command. This set of flags is then passed down to advfs_mountfs() where it is used to decide whether to set the analogous in-memory bit in the bfSetT.bfSetFlags field.

3.1.8 actRangeT Structure

The arIosOutstanding field is no longer needed and will be removed. In addition, the arStartFob and arEndFob fields will be changed from uint64_t to bf_fob_t types to clarify their usage.

3.1.9 New Macros

The following macros will be added to bs_public.h for use in converting between byte, fob, and disk block values:

```
/* These two macros give the first/last FOB within the disk block that encompasses the input file offset.
   They will compensate for changing sizes of DEV_BSIZE with respect to ADVFS_FOB_SZ so long as
   DEV_BSIZE is always >= ADVFS_FOB_SZ.*/

#define ADVFS_OFF_TO_FIRST_FOB_IN_VDBLK(offset) \

    (((offset)/DEV_BSIZE) * ADVFS_FOBS_PER_DEV_BSIZE)

#define ADVFS_OFF_TO_LAST_FOB_IN_VDBLK(offset) \

    (ADVFS_OFF_TO_FIRST_FOB_IN_VDBLK(offset) + (ADVFS_FOBS_PER_DEV_BSIZE-1))


/* This macro converts a possibly unaligned byte offset to one that is aligned to the lowest byte in its disk
    block */

#define ADVFS_ALIGN_OFFSET_TO_VDBLK_DOWN(offset) \

    (((offset) / DEV_BSIZE) * DEV_BSIZE)


/* This macro is used to determine the number of disk blocks that will span a given range of bytes. */

#define ADVFS_BYTES_TO_VDBLKS(bytes) \

    ((bytes) / DEV_BSIZE)
```

3.2 Module Design

3.2.1 File Open
3.2.1.1 advfs_open()

3.2.1.1.1 Interface

```
advfs_open(     struct vnode **vpp,     /* in - vnode pointer */
                int mode,               /* in - open mode */
                struct ucred *cred      /* in - credentials of caller */
        )
```

3.2.1.1.2 Description

This routine is called each time the file is opened. Actually, the file is already open at this point, but this routine allows setting open file attributes which we will use here.

The intent of this code path is to increment the number of times that this file has been opened for directIO if running on a non-clustered system. If running on a cluster, the count of directIO opens is maintained by CFS, and they call AdvFS using advfs_cfs_set_cachepolicy() to increment this counter so it is not done here. Also, metadata and reserved files are never opened for directIO.

There is provision for a 'hidden' *–o directio* mount option. This is used for internal testing only and is not supported in the field.

The bfAccess.cacheModeLock will be seized for shared access initially in this routine while it is determined if the cache mode needs to change. If so, the lock will be upgraded to an exclusive lock to ensure that all in-flight I/Os have ended, to prevent new I/Os from starting, and to protect the value of the bfAccess.dioCnt field.

3.2.1.1.3 Execution Flow

```
bfSetp = (bfSetT *)bfap->bfSetp

Intialize return value to EOK

extract fsContext and bfAccess structures from vnode pointer

if ((mode & FDIRECT)  || bfSetp->bfSetFlags & BFS_IM_DIRECTIO))
    if (clu_is_ready()) return EOK;  /* Do nothing if operating under CFS */
    /* If this is a reserved or metadata  file, return an error if this is an explicit request to open for DIO; if
       the metadata  file is on a directIO-mounted fileset, just return EOK without doing anything.*/
    if ( bfap->dataSafety != BFD_USERDATA )
        if (Advfs_enable_dio_mount && (bfSetp->bfSetFlags & BFS_IM_DIRECTIO) )
            return  EOK
        else
            return EINVAL
    seize the bfAccess.cacheModeLock for SHARED access
    lock vnode using VN_SPINLLOCK()
    /* Don't allow non-regular or memory-mapped files to be opened for DIO */
    if  (*vpp)->v_type != VREG)
```

```
        return value = EINVAL
        unlock vnode using VN_SPINLUNLOCK()
    else if ( ((*vpp)->v_flag & VMMF )  &&  (bfap->dioCnt == 0) )
        /* the file is already mmapped or there is an mmapper in progress and the file is not already open
           for DIO.  Let the mmapper succeed */
        return value = EINVAL
        unlock vnode using VN_SPINLUNLOCK()
    else            /* increment the DIO counter for this file */
        unlock vnode using VN_SPINLUNLOCK()
        if (bfap->dioCnt > 0)
            /* file is already open for directIO; just increment the counter */
            increment bfap->dioCnt
        else
            upgrade cacheModeLock to EXCLUSIVE access
            /* At this point there are no in-flight I/Os or mmappers */
            lock vnode using VN_SPINLLOCK()
            if ((*vpp)->v_flag & VMMF )
                /* mmapper snuck in before the lock was upgraded */
                return value = EINVAL
                unlock vnode using VN_SPINLUNLOCK()
            else
                unlock vnode using VN_SPINLUNLOCK()
                increment bfap->dioCnt
                /* Flush and invalidate any buffers for this file */
                fcache_vn_flush( &bfap->bfVnode, 0, bfap->bfaNextFob, NULL,
                                 FVF_WRITE | FVF_SYNC | FVF_INVAL );
    release the bfAccess.cacheModeLock
return (return value);
```

3.2.1.1.4 Other Items of Note for advfs_open()

All code using the O_CACHE flag in advfs_open() will be removed; it is obsolete.

3.2.1.2 Other Changes in this Path

There are several places where the VDIRECTIO bit was set or cleared in the open path on Tru64. Specifically, the paragraphs in fs_create_file() and bf_get_l() that set or clear this bit will be removed since they are not needed in this implementation.

3.2.2 File Close

3.2.2.1 advfs_close()

3.2.2.1.1 Interface

```
statusT
advfs_close (   struct vnode *vp,       /* in - vnode of file to close */
                int fflag,              /* in - flags - not used */
                struct ucred *cred      /* in - credentials of caller */
            )
```

3.2.2.1.2 Description

This is the path that closes the file. It is used here to decrement the directIO use counter if the thread that is closing the file opened it for directIO. If running under CFS, the counter is not decremented here. Rather, the cache mode of the file is modified by a call to advfs_cfs_set_cachepolicy().

The bfAccess.cacheModeLock is seized for exclusive lock to ensure that all in-flight I/Os have ended, to prevent new I/Os from starting, and to protect the value of the bfAccess.dioCnt field.

3.2.2.1.3 Execution Flow

After the NFS_FREEZE_LOCK but before the check to see if the file stats need to be flushed, the following will be added:

```
if (((fflag & FDIRECT) || (((bfSetT *)bfap->bfSetp)->bfSetFlags & BFS_IM_DIRECTIO)) &&
    !clu_is_ready() )
    /* Decrement counter if not on a cluster and the file was opened for directIO */
    seize the bfap->cacheModeLock for exclusive access
    if (bfap->dioCnt > 0)
        bfap->dioCnt--;
    release bfap->cacheModeLock
```

3.2.2.1.4 Other items of note for advfs_close()

The comment on the fflag parameter that it is not used will be removed. This implementation uses the FDIRECT bit in the file flags when closing the file.

3.2.2.2 Other Changes in this Path

In bs_close_one(), the paragraph that sets or clears the VDIRECTIO bit for regular files on the last close will be removed. This is no longer needed.

3.2.3 Memory-mapping a file
3.2.3.1 advfs_mmap()

3.2.3.1.1 Interface

The interface to this routine remains unchanged.

3.2.3.1.2 Description

To ensure unambiguous decisions between opening a file for directIO and enabling memory mapping on a file, the bfAccess.cacheModeLock must be seized for shared access in this path. .

Note that the VMMF flag in vnode.v_flag (indicating that the file has memory-mapping enabled) is conditionally set before this routine is called. If this routine returns an error, then the flag is cleared.

3.2.3.1.3 Execution Flow

This is a simplified version of the execution flow so that the differences between the cached and directIO paths are more apparent. Areas of change from the current implementation are in a bold font.

Seize the bfAccess.cacheModeLock for shared access

If (bfap->dioCnt > 1)

    /* the file is already open for directIO; return failure */

    release the bfAccess.cacheModeLock

    return failure

Release the bfAccess.cacheModeLock

Set dirty stats

Return success

3.2.4 Open/Close on a cluster
3.2.4.1 advfs_cfs_set_cachepolicy()

3.2.4.1.1 Interface

statusT

advfs_cfs_set_cachepolicy(vnode *vp, /* pointer to vnode for file */

int flag) /* flag denoting which cache mode to set */

Flag values = ADVFS_DIRECTIO or ADVFS_CACHED; these values will be #defined in advfs_ioctl.h.

3.2.4.1.2 Description

This is a new private interface routine being provided to CFS for the purpose of turning directIO on or off for a given file. CFS will track the number of times that a given file is opened for directIO, and will use this routine to tell AdvFS when to enable or disable directIO for the file. This is to eliminate the possibility of the two layers having directIO counters that get out of synchronization. On a clustered system, the directIO counter in the access structure will always be either 0 (cached mode) or 1 (directIO mode).

There is no need to wait for DIO-induced I/Os to complete when reverting from directIO to cached I/O mode here because CFS will ensure that AdvFS does not get any I/O requests when it wants to switch modes by holding a CFS lock on the file. In addition, the AIO close code waits for in-flight AIO-induced I/Os to complete.

3.2.4.1.3 Execution Flow

get fsContext and bfAccess structures from vnode pointer

If this is a metadata or reserved file

    return EACCES;

seize bfAccess.cacheModeLock for exclusive access

If (flag == ADVFS_DIRECTIO)

    If this is a memory mapped file

        release bfAccess.cacheMode lock

        return EACCES;

    If (bfap->dioCnt == 0)

        bfap->dioCnt++;

        if (contextp->dirty_alloc)

            /* flush any storage allocation transactions to disk */

            advfs_lgr_flush(bfap->dmnP->ftxLogP, nilLSN, FALSE);

        /* flush and invalidate any pages currently in the cache for this file */

        fcache_vn_flush(&bfap->bfVnode, 0, bfap->bfaNextFob, NULL,

                FVF_WRITE | FVF_SYNC | FVF_INVAL);

    MS_SMP_ASSERT(bfap->dioCnt == 1);

else /* (flag == ADVFS_CACHED) */

    MS_SMP_ASSERT(flag == ADVFS_CACHED);

    if (bfap->dioCnt > 0) bfap->dioCnt--;

```
        MS_SMP_ASSERT(bfap->dioCnt == 0);
release the bfAccess.cacheModeLock
return EOK
```

3.2.5 Read Path
3.2.5.1 advfs_fs_read()

3.2.5.1.1 Interface

```
int
advfs_fs_read(    struct vnode *vp,          /* in - vnode of file to read    */
                  struct uio *uio,           /* in - structure for uiomove    */
                  enum uio_rw rw,            /* in - flags: AIO or not        */
                  int ioflag,                /* in - includes sync i/o flags  */
                  struct ucred *cred         /* in - credentials of caller    */
                )
```

3.2.5.1.2 Description

This function is mainline code for the read() system call in AdvFS. All reads of normal files (not metadata or reserved files) for directIO must come through this routine. If a file is open for cached I/O, the UFC routines are used to retrieve the data. If the file is open for directIO, then the directIO-specific routines are used as much as possible to avoid bringing pages into the UFC.

3.2.5.1.3 Execution Flow

This is a simplified version of the execution flow so that the differences between the cached and directIO paths are more apparent. Areas of change from the current implementation are in a bold font.

seize the bfap->cacheModeLock for shared access

seize the file lock for shared access

setup read offset and # bytes to read

Determine if the file is opened for directIO (DIO) access; this is true if (bfap->dioCnt > 0)

Loop until all bytes read (bytes_left == 0):

if (!DIO)

fcache_as_map();

fcache_as_uiomove();

if bytes still left call fcache_as_unmap();

else /* directIO */

bytes_read = 0;

error = advfs_fs_read_direct(bfap, uio, &bytes_read);

if (error) break;

bytes_left -= bytes_read;

if (!DIO)

If necessary, call fcache_as_fault() to start readahead.

fcache_as_unmap()

Update st_atime in the file's stat structure

if (!DIO)

If FRSYNC & FDSYNC set call fcache_vn_flush() to flush pending writes.

If FSYNC set, call fs_update_stats() to update metadata.

release file_lock

release the bfap->cacheModeLock

return (return value)

3.2.5.2 advfs_fs_read_direct()

3.2.5.2.1 Interface

```
static statusT
advfs_fs_read_direct(   bfAccessT *bfap,        /*  in:  access structure */
                        struct uio *uio,        /*  in:  uio struct */
                        size_t *bytes_read      /*  out:  # bytes  read */
                    )
```

3.2.5.2.2 Description

This routine is responsible for reading data from disk into an application buffer when a file is open for directIO. There is no usage of data that is already in the UFC, primarily because we don't have a mechanism for knowing what data already exists in the UFC. The requested transfer may or may not be aligned on sector boundaries. If the request is aligned, then the performance will be better than if it is not aligned.

If there are multiple uio->uio_iov vectors, then each is handled on a single pass through this routine, and advfs_fs_read() will call back in multiple times until the requests in all of the vectors have been handled.

3.2.5.2.3 Execution Flow

MS_SMP_ASSERT(uio->uio_iov->iov_len != 0); /* handle multiple uio_iov vectors correctly */

*bytes_read = 0;

if (uio->uio_old_offset == UIO_AIORW) /* handle AIO requests */

    iov = &uio->uio_iov[1];

    aio_bp = (struct buf *)uio->uio_iov[1].iov_base;

/* If an application requests an extremely large directIO read, the call to vaslockpages() later to wire the application's memory buffer could hang because we try to wire too many pages. To prevent that, we artificially limit the size of a directIO transfer here to a predetermined value, initially planned to be 1 Mb. This value will be saved in a global variable that could be modified via debugger if necessary, rather than making this a #defined value. If the read size is limited here, advfs_fs_read() will detect that the number of bytes requested was not transferred, and will call back into this routine to transfer the next set of bytes. This process would continue until the original I/O request is fulfilled. This limitation will not be done for calls from the AIO layer since the application buffer will already be wired. */

If ((!AIO) && (request > artificial size limit))

    set request size to artificial size limit.

/* seize active range for sectors underlying this IO.*/

```
allocate an active range structure by calling advfs_bs_get_actRange()
actRange.arStartFob = ADVFS_OFF_TO_FIRST_FOB_IN_VDBLK (uio->uio_offset);
actRange.arEndFob   = ADVFS_OFF_TO_LAST_FOB_IN_VDBLK (uio->uio_offset + uio->uio_iov->iov_len);
insert_actRange_onto_list();
#ifdef ADVFS_DEBUG
    /* Check for any cached pages in this range if running in a debug environment.  This is a test that there
       are no pages in the UFC for the range about to be written via directIO.  Having stale cache pages that
       are being bypassed could result in data contamination.  So this is a test to be sure that there are no
       paths that allowed UFC pages to be associated with this range for a file opened for directIO.
       Normally there will be no UFC pages for this file.  This is assured by flushing and invalidating when
       the file is opened for directIO, and by having any operation (such as migrate) flush any pages to disk
       if it brings them into the cache for a directIO-enabled file.  The following call to fcache_vn_info()
       merely tells us if there are UFC pages associated with this file.  However, we do not have any way of
       telling if they are within the range we have locked, so there remains the possibility that another
       thread is migrating in another region of the file, and we will unnecessarily call the invalidate for our
       range. */
    /* Initialize a structure of fcache_vninfo_t type */
    fcache_info.fvi_pages = 0;
    if ( (fcache_vn_info( bfap->bfVp, FVINFO_PAGES, &fcache_info ) == EINVAL) ||
        fcache_info.fvi_pages != 0 )
            /* Set 2 special (new) flags in the fs_pvt_params structure that tell advfs_putpage() to
               ASSERT if either clean or dirty pages are encountered in this file range. */
            fs_pvt_param.app_flags = APP_ASSERT_NO_DIRTY | APP_ASSERT_NO_CLEAN
            fcache_vn_invalidate( vp, byte offset of start of active range, size of active range,
                 &fs_pvt_param, FVI_INVAL );
#endif ADVFS_DEBUG
release file lock
/* If this read is not invoked by the AIO layer, wire the application buffer */
if (!AIO)
    sts = bufpin( ldusid(buffer), user's buffer address, request size,  B_READ);
    if (sts != EOK)
        if ( ! luseracc( ldusid(buffer), user's buffer addr, request size, B_WRITE)
            status = EFAULT;                /* no access rights */
            goto cleanup;
        vas_read_lock(p_vas(u.u_procp));
        Loop until the total request is locked:
            vaslockpages(user's buffer address, &request size, flags, B_READ, FALSE, NULL);
        vas_unlock( p_vas(u.u_procp) );
    else
        wired_via_bufpin = TRUE;
```

```
If ( there is a leading unaligned region )
    /* Use advfs_get_blkmap_in_range() to see if this sector has storage or is in a hole. */
    sts = advfs_get_blkmap_in_range( bfap, bfap->xtnts.xtntMap, start of active range, DEV_BSIZE,
                        &storage_blkmap, &no_of_maps, RND_NONE, EXB_ONLY_STG,
                        XTNT_WAIT_OK );
    If (sts)
        MS_SMP_ASSERT( sts != E_OFFSET_NOT_BLK_ALIGNED );
        status = sts;
        goto cleanup;
    if this region is in a hole (no_of_maps == 0), bzero() the region in the application buffer
    else /* there is backing store, so read the data from disk into application buffer */
        sts = advfs_dio_unaligned_xfer(   bfap     = bfap
                                          addr     = start of application buffer
                                          offset   = (file offset % DEV_BSIZE)
                                          nbyte    = DEV_BSIZE - (file offset % DEV_BSIZE)
                                          vdIndex  = storage_blkmap->ebd_vd_index,
                                          lbn      = storage_blkmap->ebd_vd_blk;
                                          stg_fobs = 0
                                          flag     = B_READ )
        if (sts)
            status = sts;
            goto cleanup;
        else
            adjust # bytes remaining,  uio->uio_iov->iov_base and iov_len for the # bytes just read
            *bytes_read += nbyte
            advfs_free_blkmaps( &storage_blkmap );
If ( there is a trailing unaligned region )
    /* Use advfs_get_blkmap_in_range() to see if this sector has storage or is in a hole. */
    tmp_offset = ADVFS_ALIGN_OFFSET_TO_VDBLK_DOWN(uio->uio_offset, uio->uio_iov-
                        >iov_len);
    sts = advfs_get_blkmap_in_range( bfap, bfap->xtnts.xtntMap, &tmp_offset, DEV_BSIZE,
                        &storage_blkmap, &no_of_maps, RND_NONE, EXB_ONLY_STG,
                        XTNT_WAIT_OK );
    If (sts)
        MS_SMP_ASSERT( sts != E_OFFSET_NOT_BLK_ALIGNED );
        status = sts;
        goto cleanup;
    if this region is in a hole (no_of_maps == 0), bzero() the region in the application buffer
    else /* there is backing store, so read the data from disk into application buffer */
```

```
MS_SMP_ASSERT( no_of_maps == 1 );
advfs_dio_unaligned_xfer( bfap     = bfap
                          addr     = start of trailing region in application buffer
                          offset   = 0
                          nbyte    = original  request size – (# bytes read  in leading unaligned
                                     request + bytes in remaining  full sector transfers)
                          vdIndex  = storage_blkmap->ebd_vd_index,
                          lbn      = storage_blkmap->ebd_vd_blk
                          stg_fobs = 0
                          flag     = B_READ )

if (sts)
    /* If an error occurs during the trailing transfer, we need to continue on and do any aligned
    transfer, and then account for the success or failure of the trailing request at the end */
    trailing_bytes_read = 0;
else
    adjust # bytes remaining to be read for the # bytes just read
    trailing_bytes_read = nbyte
advfs_free_blkmaps( &storage_blkmap );
```

Adjust number of bytes that still need to be read, and possibly starting application buffer address based on unaligned reads, if any.

If there are no more bytes to be transferred, goto cleanup. /* entire read satisfied in unaligned transfers */

/* setup and start IO for aligned region */

```
/* This starts the 'typical' code path: for performing aligned I/O requests */
/* Allocate and initialize the ioanchor structure  */
ioap = advfs_bs_get_ioanchor(TRUE);
ioap->anchr_iocounter = 0;
ioap->anchr_flags = IOANCHORFLG_DIRECTIO;
ioap->anchr_flags |= (!AIO) ? IOANCHORFLG_KEEP_ANCHOR : NULL;
ioap->anchr_actrangep =  (AIO) ? active range pointer : NULL;
ioap->anchr_aio_bp = aio_bp;   /* AIO buf struct if using AIO */
ioap->anchr_io_status = 0;
ioap->anchr_min_req_offset = file offset of aligned read
ioap->anchr_min_err_offset = file offset of aligned read + size of aligned read
/* Get the extent maps for the aligned region of the file   Retrieve both storage and hole information. */
sts = advfs_get_blkmap_in_range( bfap, bfap->xtnts.xtntMap, start of aligned region, length of aligned
                                 region, &storage_blkmap, NULL, RND_NONE, EXB_COMPLETE,
                                 XTNT_WAIT_OK );
If (sts)
```

```
        MS_SMP_ASSERT( sts != E_OFFSET_NOT_BLK_ALIGNED );
        status = sts;
        goto cleanup;
    /* Loop through the extent map structures;  generate a single IO for each contiguous extent on a given
    disk provided that it is smaller than the preferred transfer size of the disk.  A given contiguous extent
    may be broken up into multiple I/Os if it exceeds preferred transfer size. */
    For (each extent inside the aligned I/O region ):
        if (current extent is a hole)
            bzero() the appropriate portion of the application buffer
            update user's buffer addr and block number
        else /* there is storage; generate buf struct for IO */
            /* allocate and initialize the buf structure.  The buf structures for each ioanchor  will be
            chained together so that we can start an I/O for each after they have all been generated. */
            bp  = advfs_bs_get_buf(TRUE);
             bp->b_un.b_addr = (caddr_t)(user's buffer address)
            bp->b_spaddr    = ldsid( bp->b_un.b_addr );
            bp->b_blkno     = starting logical disk block number
            bp->b_offset    = current file offset for this I/O
            bp->b_foffset    = current file offset for this I/O
            bp->b_bcount    = byte count for this transfer
            bp->b_vp        = bfap->bfVp;
            bp->b_proc      = u.u_procp;
            bp->b_flags     = B_READ | B_RAW | B_PHYS
            bp->b_flags     |= (AIO) ? B_ASYNC : B_SYNC;
            /* The next assignment temporarily saves the vdT pointer for each buf structure.  This is
               needed for passing to advfs_bs_startio(), but that routine will overwrite the buf.b_private
               field for its own purposes. */
            bp->b_private = (void *)vdp;
            If this transfer is larger than the preferred transfer size, be sure to break it down into a series
            of smaller transfers
            Link the buf structures together here to form a null-terminated forward-linked list.  This is
            used to keep track of the buf structures so we can walk through them and start an I/O for each.
            Use the buf.b_nextchain field.
            ioap-> anchr_iocounter++
            Adjust application buffer address, file offset, and disk block number based on the buf structs
            already set up.
    For each I/O (buf structure) set up in previous loop:
        vdp = (vdT *)bp->b_private;
        advfs_bs_startio( bp, ioap, NULL, bfap, vdp, 0)          /* start each I/O */
If (!AIO)
```

```
    /* wait for I/O to complete */
    spin_lock(&ioanchorp->anchr_lock);
    cv_wait(&ioanchorp->anchr_cvwait, &ioanchorp->anchr_lock, CV_SPINLOCK, CV_NO_RELOCK);
    /* Check the status of  the whole I/O */
    if (ioap->anchr_io_status != EOK )
        status = ioap->anchr_io_status;
        aligned_bytes_read  = ioap->anchr_min_err_offset;
        trailing_bytes_read = 0;
else /* AIO */
    iov->iov_len = 1;       /* records start of asynch IO for advfs_strategy */
*bytes_read += trailing_bytes_read + aligned_bytes_read;
Adjust uio->uio_offset and uio->uio_resid to reflect the # bytes successfully transferred.

cleanup:
    seize file_lock for shared access
    If (no asynchronous I/Os were started)
        /* These all get cleaned up in I/O completion path for AIO */
        release the active range by calling advfs_bs_free_actRange()
        free the active range structure by calling advfs_bs_free_actRange()
        if (ioanchor) advfs_bs_free_ioanchor( ioanchor, M_WAITOK );
    If ( we wired the application pages above )
        if (wired_via_bufpin)
            bufunpin(ldusid(buffer), user's buffer address, request size,  B_READ);
        else
            vasunlockpages(user's buffer address, request size, 0, ldusid(user's buffer), NULL,
                B_READ);
    /* Handle transfers with multiple uio->uio_iov vectors */
    if (uio->uio_iovcnt > 1 && uio->uio_iov->iov_len == 0)
        uio->uio_iov++;
        uio->uio_iovcnt--;
    if ( storage_blkmap )  advfs_free_blkmaps( &storage_blkmap );
    return status
```

3.2.6 Write Path
3.2.6.1 advfs_fs_write()

3.2.6.1.1 Interface

```
int
advfs_fs_write(   struct vnode *vp,          /* in - vnode of file to write */
                  struct uio *uio,           /* in - structure for uiomove */
                  enum uio_rw rw,            /* in - read/write flags */
                  int ioflag,                /* in - flags - append, sync, etc. */
                  struct ucred *cred         /* in - credentials of caller */
                )
```

3.2.6.1.2 Description

This function is mainline code for the write() system call in AdvFS. All writes of normal files (not metadata or reserved files) for directIO must come through this routine. If the file is open for directIO, then the directIO-specific routines are used as much as possible to avoid bringing pages into the UFC.

3.2.6.1.3 Execution Flow

This is a simplified version of the execution flow so that the differences between the cached and directIO paths are more apparent. Areas of change from the current implementation are in a bold font.

seize the bfap->cacheModeLock for shared access

seize file_lock for shared or exclusive access depending on IO_APPEND mode

advfs_fs_check_write_limits(); /* verify write request against limits */

Initialize filesytem private information for advfs_getpage().

Determine if the file is opened for directIO (DIO) access; this is true if (bfap->dioCnt > 0)

Loop while bytes left to write:

if (!DIO)

fcache_as_map()

fcache_as_uiomove()

increment cumulative_bytes_written and next offset

decrement # of bytes left

if bytes still left call fcache_as_unmap()

else /* DIO */

error = fs_write_direct(bfap, uio, &bytes_written, ioflag);

if (error) break;

increment cumulative_bytes_written and next_write_offset

decrement # of bytes left

ensure we don't loop again if we are in an AIO call.

check for errors or partial write

if (!DIO)

Check to see if we want to start a read-ahead.

fcache_as_unmap() for final segment

update bfap->bfaLastWrittenFob;

Update st_mtime and st_ctime in the file's stat structure

Update file length if needed

If FSYNC or FDSYNC is set:

If (!DIO)

fcache_vn_flush(); /* synchronously write the data */

if FSYNC or allocation has happened call fs_update_stats()

release file lock

release the bfap->cacheModeLock

return

3.2.6.2 advfs_fs_write_direct()

3.2.6.2.1 Interface

```
static statusT
advfs_fs_write_direct(      struct bfAccess bfap        /* in:  bfAccess structure for this file */
                            struct uio *uio,            /* in: uio struct describing transfer request */
                            size_t  *bytes_written,     /* out: # bytes written */
                            int ioflag                  /* in - flags – needed for IO_APPEND. */
                        )
```

3.2.6.2.2 Description

This routine is responsible for writing data to disk from an application buffer when a file is open for directIO. The UFC is bypassed on the data transfer. The requested transfer may or may not be aligned on sector boundaries. If the request is aligned, then the performance will be better than if it is not aligned.

If there are multiple uio->uio_iov vectors, then each is handled on a single pass through this routine, and advfs_fs_write() will call back multiple times until all of the bytes in all of the vectors have been written.

3.2.6.2.3 Execution Flow

When this routine is called, the file lock can be held for either shared or exclusive access. It will be held for exclusive access only if we are in IO_APPEND mode.

The following is an outline of the code flow through this routine:

```
if (uio->uio_old_offset == UIO_AIORW)           /* handle AIO requests */
    iov = &uio->uio_iov[1];
    aio_bp = (struct buf *)uio->uio_iov[1].iov_base;
MS_SMP_ASSERT(uio->uio_iov->iov_len != 0);      /* handle multiple uio_iov vectors correctly */
```

```
/* If an application requests an extremely large directIO write, the call to vaslockpages() later to wire the
application's memory buffer could hang because we try to wire too many pages.  To prevent that, we
artificially limit the size of a directIO transfer here to a predetermined value, initially planned to be 1 Mb.
This value will be saved in a global variable that could be modified via debugger if necessary, rather than
making this a #defined value.  If the write size is limited here, advfs_fs_write() will detect that the number
of bytes requested was not transferred, and will call back into this routine to transfer the next set of bytes.
This process would continue until the original I/O request is fulfilled.  This limitation will not be done for
calls from the AIO layer since the application buffer will already be wired.  */

If ( (!AIO) && (request > artificial size limit) )

    set request size to artificial size limit.

allocate an active range structure by calling advfs_bs_get_actRange()

/* If we are in IO_APPEND mode and there is an active range spanning the current EOF, then we need to
take out the active range now to protect the EOF value before we add storage. */

retry:

if (ioflag & IO_APPEND  && actRange_spans(bfap, bfap->file_size) )

    save current value of EOF (bfap->file_size)

    initialize range from current EOF to infinity

        arStartFob =ADVFS_OFF_TO_FIRST_FOB_IN_VDBLK (bfap->file_size);

        arEndFob   = ~0L;

    insert_actRange_onto_list();

    uio->uio_offset = bfap->file_size;        /* may have changed while we waited for the range */

    if (current EOF < saved EOF)

        release this active range

        go back to  'retry' and get active range using new EOF

    active_range_held = 1;                    /* remember for later */

initialize the advfs_pvt_param struct:

    bzero the pvt_param structure (declared on the stack)

    pvt_param.app_total_bytes      = uio->uio_iov->iov_len;

    pvt_param.app_starting_offset = uio->uio_offset;

    pvt_param.app_stg_start_fob   = 0;

    pvt_param.app_stg_end_fob     = 0;

    pvt_param.app_flags             = APP_ADDSTG_NOCACHE;

/* Add storage and check snapshotting requirements.  The APP_ADDSTG_NOCACHE flag is new, and
   indicates to getpage that the intent is to add storage within the range where needed, but not to bring pages
   into the cache for the regions needing new storage.  The snapshotting may bring pages into the cache;
   this is OK.  Those pages will be invalidated shortly.  The new_stg_xxx_fob fields in the
   advfs_pvt_param structure allow getpage to return the range of newly allocated storage.  */

if (racy_check_for_clones || the fileset is mounted for object safety )

    /* The snapshot check is not designed yet; will do this with the snapshotting work.  If we get here, it is
       possible that there is a need to move storage to a snapshotted file, and that will require a VM map, so
       we will go through the VM interface, even though the actual intent is to get into advfs_getpage().
```

```
    Any races with threads creating snapshots will be dealt with in advfs_getpage() and will be
    documented in the snapshot design. */

  fcmap = fcache_as_map(  bfap->dmnP->userdataVas, vp, uio->uio_offset, uio->uio_iov->iov_len, 0,
                          FAM_WRITE, &status );

  fcache_as_fault(  fcmap, address of user's buffer, uio->uio_iov->iov_len, &sts, (uintptr_t)&pvt_param,
                    FAF_GPAGE | FAF_WRITE | FAF_SYNC );

  fcache_as_unmap(     fcmap, NULL, 0, FAU_FREE );

else  /* no snapshot possible; use getpage to add storage */

  sts = advfs_getpage(  NULL, vp, uio->uio_offset, uio->uio_iov->iov_len, FCF_DFLT_WRITE,
                        (uintptr_t)&pvt_param, 0 );

  if (sts != EOK  &&  sts != ENOSPC)

      goto cleanup;
```

If we get ENOSPC back from fcache_as_fault() or advfs_getpage(), then we did not get all the storage which was requested. The range of new storage added is returned in the pvt_param.app_stg_xxx_fob fields. Adjust the range of the original request to stay within the range of allocated storage. Be sure that the requested range that lies outside the storage is reflected in the uio.uio_resid returned to the caller.

Adjust the # of bytes to be written if we didn't get as much storage as we requested. This will be reflected in the 'size' field returned from either advfs_getpage() or the fault routine.

```
/* Check if storage was added to the file before or after the actual requested IO range.  This can happen
   since storage will be allocated in chunks larger than a single fob.  If the IO request starts in the middle of
   the first newly-allocated chunk, the new storage that lies before the application data must be zero'd on
   disk.  The same is true at the end of the IO request range.  The entire newly-allocated range of storage
   must also be included in our active range.  We will also check for data alignment at this time since we
   will handle zeroing new storage outside of the original IO request and unaligned data merging at the
   same time.*/

If (pvt_param.app_stg_end_fob)

    /* We know storage was added if pvt_param.app_stg_end_fob is non-zero since the starting fob can be
       0, but we can't add storage with just one fob.  Determine the span of the active range.  It must
       include all new storage, which will be greater than or equal to the original transfer range requested.
       */

    if ( !active_range_held )

        actRange.arStartFob = ADVFS_OFF_TO_FIRST_FOB_IN_VDBLK(
                  pvt_param.app_stg_start_fob * ADVFS_FOB_SZ);

        actRange.arEndFob   =ADVFS_OFF_TO_LAST_FOB_IN_VDBLK(
                  pvt_param.app_stg_end_fob * ADVFS_FOB_SZ);

    /* First check for leading storage and alignment */

    if (!(pvt_param.app_starting_offset % DEV_BSIZE) && (pvt_param.app_stg_start_fob ==
          ADVFS_OFFSET_TO_FOB_DOWN (pvt_param.app_starting_offset)))

        /* The application data aligns with start of new storage, so no special zeroing required */

        new_stg_leading_fobs  = 0;

    else

        /* The application data does not align with the start of new storage, so calculate the # of fobs that
           must be zero-filled.  This will include the fob into which the unaligned data will be merged. */
```

```
            new_stg_leading_fobs  = (ADVFS_OFFSET_TO_FOB_DOWN(pvt_param.app_starting_offset) -
                                        pvt_param.app_stg_start_fob) + 1;

        /* Now check trailing storage and alignment */

        if (!((pvt_param.app_starting_offset +pvt_param.app_total_bytes) % DEV_BSIZE) &&
            (pvt_param.app_stg_end_fob ==ADVFS_OFFSET_TO_FOB_DOWN(
            pvt_param.app_starting_offset + pvt_param.app_total_bytes)

            /* The application data aligns with the end of new storage, so no zeroing here. */

            new_stg_trailing_fobs = 0;

        else

            /* The application data does not align with the end of new storage, so calculate the # of fobs that
               must be zero-filled.  This will include the fob into which the unaligned data will be merged. */

            new_stg_trailing_fobs = (pvt_param.app_stg_end_fob -
                        ADVFS_OFFSET_TO_FOB_DOWN(pvt_param.app_starting_offset +
                        pvt_param.app_total_bytes)) + 1;


else /* there is no new storage */

        new_stg_leading_fobs  = 0;

        new_stg_trailing_fobs = 0;

        if ( !active_range_held )

            actRange. arStartFob = ADVFS_OFF_TO_FIRST_FOB_IN_VDBLK (uio->uio_offset);

            actRange. arEndFob   =ADVFS_OFF_TO_LAST_FOB_IN_VDBLK (uio->uio_offset + uio-
                                        >uio_iov->iov_len);

/* At this point, the variables new_stg_leading_fobs and new_stg_trailiing_fobs will be zero if either no
   new storage was added or if the new storage will be completely overwritten by the application data. */


/* If we don't already have an active range, seize it now.  This is the typical situation. */

if ( ! active_range_held )

        insert_actRange_onto_list( bfap, actRange, fsContextp );

#ifdef ADVFS_DEBUG

        /* Check for any cached pages in this range if running in a debug environment.  This is a test that there
           are no pages in the UFC for the range about to be written via directIO.  Having stale cache pages that
           are being bypassed could result in data contamination.  So this is a test to be sure that there are no
           paths that allowed UFC pages to be associated with this range for a file opened for directIO.
           Normally there will be no UFC pages for this file.  This is assured by flushing and invalidating when
           the file is opened for directIO, and by having any operation (such as migrate) flush any pages to disk
           if it brings them into the cache for a directIO-enabled file.  The following call to fcache_vn_info()
           merely tells us if there are UFC pages associated with this file.  However, we do not have any way of
           telling if they are within the range we have locked, so there remains the possibility that another
           thread is migrating in another region of the file, and we will unnecessarily call the invalidate for our
           range. */

        /* Initialize a structure of fcache_vninfo_t type */

        fcache_info.fvi_pages = 0;

        if ( (fcache_vn_info( bfap->bfVp, FVINFO_PAGES, &fcache_info ) == EINVAL) ||
```

```
        fcache_info.fvi_pages != 0 )
            /* Set 2 special (new) flags in the fs_pvt_params structure that tell advfs_putpage() to
               ASSERT if either clean or dirty pages are encountered in this file range. */
            fs_pvt_param.app_flags = APP_ASSERT_NO_DIRTY | APP_ASSERT_NO_CLEAN
            fcache_vn_invalidate( vp, byte offset of start of active range, size of active range,
                &fs_pvt_param, FVI_INVAL );
#endif ADVFS_DEBUG
Check whether the file lock is held for shared or exclusive access; will need to re-seize in same mode later.
release the file lock
/* If this write is not invoked by the AIO layer, wire the application buffer,  (The AIO layer does this for
   us). */
if (!AIO)
    sts = bufpin( ldusid(buffer), user's buffer address, request size,  B_WRITE);
    if (sts != EOK)
        if ( ! luseracc( ldusid(buffer), user's buffer addr, request size, B_READ)
            status = EFAULT;                /* no access rights */
            goto cleanup;
        vas_read_lock(p_vas(u.u_procp));
        Loop until the total request is locked:
            vaslockpages(user's buffer address, &request size, flags, B_WRITE, FALSE, NULL);
        vas_unlock( p_vas(u.u_procp) );
    else
        wired_via_bufpin = TRUE;
/* Handle any zero-filling of leading newly-allocated storage or unaligned request now. */
if (new_stg_leading_fobs != 0  ||  there is a leading unaligned region)
    if (leading unaligned region)
        bytes_to_transfer = DEV_BSIZE - (file offset % DEV_BSIZE);
    else
        bytes_to_transfer = 0;
    /* Use advfs_get_blkmap_in_range() to get the extent map for this sector. */
    sts = advfs_get_blkmap_in_range( bfap, bfap->xtnts.xtntMap, file offset, DEV_BSIZE,
                        &storage_blkmap, &no_of_maps, RND_NONE, EXB_ONLY_STG,
                        XTNT_WAIT_OK );
    If (sts)
        MS_SMP_ASSERT( sts != E_OFFSET_NOT_BLK_ALIGNED );
        status = sts;
        goto cleanup;
    /* This call takes care of zero-filling newly-allocated storage that will not be overwritten as well as
       merging unaligned data with the underlying data on disk. */
```

```
    sts = advfs_dio_unaligned_xfer( bfap,
                                    start of application buffer,
                                    (file offset % DEV_BSIZE),
                                    bytes_to_transfer,
                                    storage_blkmap->ebd_vd_index,
                                    storage_blkmap->ebd_vd_blk,
                                    new_stg_leading_fobs,
                                    B_WRITE );
    if (sts)
        status = sts;
        *bytes_written = 0;
        goto cleanup;
    else
        *bytes_written += nbyte
        advfs_free_blkmaps( &storage_blkmap );
    Adjust buffer addresses and count of bytes remaining to transfer.
/* Handle any zero-filling of trailing newly-allocated storage or unaligned request now. */
if (new_stg_trailing_fobs != 0  ||  there is a trailing unaligned region)
    if (trailing unaligned region)
        bytes_to_transfer = total transfer size - (# bytes in leading unaligned request + bytes in full sector
                            transfers)
    else
        bytes_to_transfer = 0;
    /* Use advfs_get_blkmap_in_range() to get the extent map for this sector. */
    tmp_offset = ADVFS_ALIGN_OFFSET_TO_VDBLK_DOWN(req_xfer_offset +req_xfer_size);
    sts = advfs_get_blkmap_in_range( bfap, bfap->xtnts.xtntMap, &tmp_offset, DEV_BSIZE,
                        &storage_blkmap, &no_of_maps, RND_NONE, EXB_ONLY_STG,
                        XTNT_WAIT_OK );
    If (sts)
        MS_SMP_ASSERT( sts != E_OFFSET_NOT_BLK_ALIGNED );
        status = sts;
        goto cleanup;
    /* This call takes care of zero-filling newly-allocated storage that will not be overwritten as well as
       merging unaligned data with the underlying data on disk. */
    sts = advfs_dio_unaligned_xfer( bfap,
                                    start of trailing region in application buffer,
                                    0,
                                    bytes_to_transfer,
```

```
                                        storage_blkmap->ebd_vd_index,
                                        storage_blkmap->ebd_vd_blk,
                                        new_stg_trailing_fobs,
                                        B_WRITE );
    if (sts)
        /* If an error occurs during the trailing transfer, we need to continue on and do any aligned
        transfer, and then account for the success or failure of the trailing request at the end */
        trailing_bytes_written = 0;
    else
        trailing_bytes_written = nbyte
    advfs_free_blkmaps( &storage_blkmap );
    Adjust buffer addresses and count of bytes remaining to transfer.
/* Setup and start the aligned IOs, if any remain */
if ( there are data left to transfer )
    /* This starts the 'typical' code path: for performing aligned I/O requests */
    /* allocate and initialize the ioanchor structure. */
    ioap = advfs_bs_get_ioanchor(TRUE);
    ioap->anchr_iocounter = 0;
    ioap->anchr_flags = IOANCHORFLG_DIRECTIO
    ioap->anchr_flags  |= (AIO) ? 0 :| IOANCHORFLG_KEEP_ANCHOR
    ioap->anchr_actrangep =  (AIO) ? active range pointer : NULL;
    ioap->anchr_aio_bp = aio_bp;   /* AIO buf struct if using AIO */
    ioap->anchr_io_status = 0;
    ioap->anchr_min_req_offset = file offset of aligned write
    ioap->anchr_min_err_offset = file offset of aligned write + size of aligned write
    /* Get the extent maps for the aligned region of the file   Retrieve both storage and hole information. */
    sts = advfs_get_blkmap_in_range( bfap, bfap->xtnts.xtntMap, start of aligned region, length of aligned
                                        region, &storage_blkmap, NULL, RND_NONE, EXB_COMPLETE,
                                        XTNT_WAIT_OK );
    If (sts)
        MS_SMP_ASSERT( sts != E_OFFSET_NOT_BLK_ALIGNED );
        status = sts;
        goto cleanup;
    /* Loop through the extent map structures;  generate a single IO for each contiguous extent on a given
    disk provided that it is smaller than the preferred transfer size of the disk.  A given contiguous extent
    may be broken up into multiple I/Os if it exceeds preferred transfer size. */
    For (each extent inside the aligned I/O region):
        /* allocate and initialize the buf structure.  The buf structures for each ioanchor  will be chained
        together so that we can start an I/O for each after they have all been generated. */
```

bp = advfs_bs_get_buf(TRUE);

bp->b_un.b_addr = (caddr_t)(user's buffer address)

bp->b_spaddr = ldsid(bp->b_un.b_addr);

bp->b_blkno = starting logical disk block number

bp->b_offset = current file offset for this I/O

bp->b_foffset = current file offset for this I/O

bp->b_bcount = byte count for this transfer

bp->b_vp = bfap->bfVp;

bp->b_proc = process structure doing the IO

bp->b_flags = B_WRITE | B_RAW | B_PHYS;

bp->b_flags |= (AIO) ? B_ASYNC : B_SYNC;

/* The next assignment temporarily saves the vdT pointer for each buf structure. This is needed for passing to advfs_bs_startio(), but that routine will overwrite the buf.b_private field for its own purposes. */

bp->b_private = (void *)vdp;

If this transfer is larger than the preferred transfer size, be sure to break it down into a series of smaller transfers, adjusting buffer addresses and disk lbn for each transfer.

Link the buf structures together here to form a null-terminated forward-linked list. This is used to keep track of the buf structures so we can walk through them and start an I/O for each. Use the buf.b_nextchain field.

ioap-> anchr_iocounter++

Adjust application buffer address, file offset, and disk block number based on the buf structs already set up.

For each I/O (buf structure) set up in previous loop:

vdp = (vdT *)bp->b_private;

advfs_bs_startio(bp, ioap, NULL, bfap, vdp, 0) /* start each I/O */

If (!AIO)

/* For non-AIO transfers, we must wait for the I/Os to complete and then cleanup. Transfers done via AIO will be cleaned up in the IO completion path. */

/* wait for I/O */

spin_lock(&ioanchorp->anchr_lock);

cv_wait(&ioanchorp->anchr_cvwait, & ioanchorp->anchr_lock, CV_SPINLOCK, CV_NO_RELOCK);

/* Check the status of the whole I/O */

if (ioap->anchr_io_status != EOK)

status = ioap->anchr_io_status;

aligned_bytes_written = ioap->anchr_min_err_offset;

trailing_bytes_written = 0;

else

iov->iov_len = 1; /* records start of asynch IO for advfs_strategy */

```
*bytes_written += trailing_bytes_written + aligned_bytes_written;

cleanup:
    Adjust uio->uio_offset and uio->uio_resid to reflect the # bytes successfully transferred.
    seize file_lock for access type recorded earlier
    If no asynchronous I/Os were started
        release the active range
        free the active range structure
        free the ioanchor structure
    /* Handle I/O requests with multiple uio_iov vectors */
    if (uio->uio_iovcnt > 1 && uio->uio_iov->iov_len == 0)
        uio->uio_iov++;
        uio->uio_iovcnt--;
    if (app_pages_wired)            /* unwire user's pages if we wired them above */
        if (wired via bufpin)
            bufunpin(ldusid(buffer), user's buffer address, request size,  B_WRITE);
        else
            vasunlockpages(user's buffer address, request size, 0, ldusid(user's buffer), NULL,
                B_WRITE);
    if ( storage_blkmap )
        advfs_free_blkmaps( &storage_blkmap );
    return status
```

3.2.7 Routines common to directIO Read and Write paths
3.2.7.1 advfs_dio_unaligned_xfer()

3.2.7.1.1 Interface

```
static statusT
advfs_dio_unaligned_xfer(  bfAccessT     *bfap       /* in: file's access structure */
                           caddr_t       addr        /* in: address of application buffer */
                           off_t         offset      /* in: byte offset in sector to start transfer */
                           size_t        nbyte       /* in: # of bytes of data to transfer;  may be
                                                        zero if being called to zero-fill fobs on
                                                        disk in newly-allocated storage */
                           vdIndexT      vdIndex     /* in: index of disk to be read/written */
                           bf_vd_blk_t   lbn         /* in: logical disk block to read/write */
                           bf_fob_t      stg_fobs    /* in: # of fobs in newly-allocated storage
                                                        that require some amount of zeroing.  If
                                                        there is unaligned data, it will go in one of
                                                        these fobs.  This value will only be non-
                                                        zero on write operations. */
                           int           flag        /* in:  B_READ/B_WRITE flag */
                         )
```

3.2.7.1.2 Description

This routine is called to handle unaligned data transfers and zero-filling of newly allocated storage that will not be overwritten by the application data. For an unaligned read, this routine is responsible for reading the sector from disk and transferring the appropriate subset of that data into the application buffer. For an unaligned write, this routine reads the sector from disk, merges the appropriate bytes from the application buffer with that data, and then rewrites the sector back to disk synchronously. However, if the stg_fobs parameter is non-zero, this routine is being called to zero-fill and possibly merge some application data into a newly-allocated region on the disk. In this case, we do not need to read the existing (uninitialized) data from disk; we can simply overwrite the disk blocks.

3.2.7.1.3 Execution Flow

The following is an outline of the code flow through this routine:

```
MS_SMP_ASSERT( nbyte || stg_fobs );  /* one should be non-zero */

/* Allocate a zero'd temporary buffer */

bufsize = MAX(DEV_BSIZE, ((stg_fobs/ADVFS_FOBS_PER_DEV_BSIZE) * ADVFS_FOB_SZ));

tmpbuf = ms_malloc( bufsize );

/* determine if we need to read the sectors from disk */

write_only = (!B_READ && stg_fobs != 0);

ioap = advfs_bs_get_ioanchor(TRUE);              /* allocate the ioAnchor structure */

bp = advfs_bs_get_buf(TRUE);                     /* allocate the buf structure */

vdp = VD_HTOP( vdIndex, bfap->dmnP );
```

```
if (!write_only)                        /* read op or first part of read/modify/write op */
    /* Initialize the ioAnchor structure */
    ioap->anchr_iocounter = 1
    ioap->anchr_flags = IOANCHORFLG_DIRECTIO | IOANCHORFLG_KEEP_BUF |
                        IOANCHORFLG_KEEP_ANCHOR;
    /* Initialize the buf structure */
    bp->b_un.b_addr = (caddr_t)( tmpbuf );
    bp->b_spaddr    = ldsid( bp->b_un.b_addr );
    bp->b_blkno     = lbn;
    bp->b_offset   = (bufp->b_blkno << DEV_BSHIFT);
    bp->b_bcount    = bufsize;
    bp->b_vp        = bfap->bfVp;
    bp->b_proc      =  u.u_procp;
    bp->b_flags     = B_READ | B_RAW | B_PHYS | B_SYNC;
    /* read the data */
    advfs_bs_startio( bp, ioap, NULL, bfap, vdp, 0);
    /* wait for IO to complete */
    spin_lock(&ioap->anchr_lock);
    cv_wait(&ioap->anchr_cvwait, &ioap->anchr_lock, CV_SPINLOCK, CV_NO_RELOCK);
    if ( (bp->b_flags & B_ERROR) && (bp->b_error != 0) )
        /* There was an error on this IO */
        status = bp->b_error;
        goto cleanup;
    /* If this is a READ operation, copy data to application buffer */
    if ( flag & B_READ )
        bcopy( tmpbuf+offset, addr, nbyte );
        goto cleanup;


/* We are here if we must write some data to disk */
if (nbyte)                          /* there is data to merge */
    /* calculate locaton in temporary buffer to write application data, then copy it  If  (offset == 0) then
       this is a trailing write and there is no need to adjust for the # storage fobs passed in for the merge. */
    toaddr = (offset) ? (tmpbuf+((stg_fobs-1)*ADVFS_FOB_SZ)+offset) : tmpbuf;
    bcopy( addr, toaddr, nbyte );
/* (Re)initialize the ioAnchor and buf structures */
ioap->anchr_iocounter = 1
ioap->anchr_flags = IOANCHORFLG_DIRECTIO | IOANCHORFLG_KEEP_BUF |
                    IOANCHORFLG_KEEP_ANCHOR;
```

```
bp->b_un.b_addr = (caddr_t)( tmpbuf )
bp->b_spaddr  = ldsid( bp->b_un.b_addr );
bp->b_blkno   = lbn
bp->b_offset  =  (bufp->b_blkno << DEV_BSHIFT);
bp->b_bcount  = bufsize;     /* # fobs (in bytes) calc above */
bp->b_vp        = bfap->bfVp
bp->b_proc      = u.u_procp;
bp->b_flags     = B_WRITE | B_RAW | B_PHYS | B_SYNC
bp->b_error = 0;
/* Start the I/O */
advfs_bs_startio( bp, ioap, NULL, bfap, vdp, 0 );
/* wait for IO to complete */
spin_lock(&ioap->anchr_lock);
cv_wait(&ioap->anchr_cvwait, &ioap->anchr_lock, CV_SPINLOCK, CV_NO_RELOCK);
if ( (bp->b_flags & B_ERROR) && (bp->b_error != 0) )
      /* There was an error on this IO */
      status = bp->b_error;

cleanup:
      advfs_bs_free_buf(bp, M_WAITOK);
      advfs_bs_free_ioanchor(ioap, M_WAITOK);
      ms_free( tmpbuf )
      return status
```

3.2.8 I/O Completion Path
3.2.8.1 advfs_iodone()

3.2.8.1.1 Interface

The interface for this routine is not changing.

3.2.8.1.2 Description

This routine completes AdvFS I/O processing after the driver completes the I/O and calls biodone(). The primary change being proposed here is to allow this routine to update statistics in the ioanchor structure in case one of a series of I/Os per ioanchor fails. At the end of the I/O, the information about how many bytes were logically transferred will be present in the ioanchor structure so the uio->uio_resid field can be updated properly.

3.2.8.1.3 Execution Flow

The following code logic will be inserted after the ioanchor spinlock is seized and the ioanchor->anchr_iocounter is decremented:

```
/* If there was an I/O error, update the statistics in the ioanchor structure if this error is logically 'before'
     any other error that may have already been recorded.  It is assumed that the anchr_min_err_offset field
     has been initialized to an integer value larger than any I/O offset requested  if these statistics are to be
     collected.  */
if ( sts != EOK ) {
     if ( bp->bp_bfoffset < ioanchorp->anchr_min_err_offset ) {
          ioanchorp->anchr_io_status = sts;
          ioanchorp->anchr_min_err_offset = bp->bp_bfoffset;
     }
}
```

3.2.8.2 advfs_bs_io_complete()

3.2.8.2.1 Interface

The interface for this routine is not changing.

```
void
advfs_bs_io_complete(struct buf *bufp)    /* in */
```

3.2.8.2.2 Description

This routine performs the final I/O completion processing on a buf structure. In the case of multiple buf structures associated with a given I/O or ioanchor_t, this routine is called when processing the final buf structure.

Changes are required here for several different reasons. The first is to have the I/O completion code, on a directIO-requested I/O, check for successful I/O completion and calculate the number of bytes successfully transferred. The second reason is to eliminate the need for maintaining the actrange.arIosOutstanding field for I/Os started for an AIO request. This functionality is no longer required, and so is being removed here to simplify the code. The third reason for changing this code path is to remove the call to bs_aio_write_cleanup() which is not needed on HP-UX. The last reason is to simplify the code structure somewhat.

3.2.8.2.3 Execution Flow

The code currently has 2 paragraphs at the end of the routine with the following structure:

```
if (directIO-specific I/O)
    if (AIO)
        cleanup and call AIO iodone routine
if ( !ioanchor->anchr_actrangep)
    … processing if there is no active range (calls advio_save_iodone() for cached I/Os) …
else
    … processing for an active range
```

Under HP-UX, we really only need 2 sections, one for directIO cleanup, and another for cached I/O cleanup. Therefore, these paragraphs will be restructured as follows:

```
if ( !ioanchorp->anchr_flags & IOANCHORFLG_DIRECTIO ) {
    ... use existing code for calling advio_save_iodone() here. (cached I/O only)…
} else {
    /* DirectIO specific cleanup */
    if ( ioanchorp->anchr_io_status != 0 ) {
        /* Calculate the # bytes successfully transferred. This will be reported back to the application.
           The ioanchor fields used here were set in advfs_iodone() before calling this routine for the final
           I/O in this request. The ioanchor.anchr_min_err_offset field is overloaded and used to return
           the number of bytes transferred back to the in-line directIO read/write routines. */
        bytes_transferred = ioanchorp->anchr_min_err_offset - ioanchorp->anchr_min_req_offset;
        ioanchorp->anchr_min_err_offset = bytes_transferred;
    }
    aio_bp = ioanchorp->anchr_aio_bp;
    if (aio_bp) {
        /* Do this for AIO-invoked I/Os */
        MS_SMP_ASSERT( ioanchorp->anchr_dio_bufList );
        MS_SMP_ASSERT( ioanchorp-> anchr_actrangep );
        aio_bp->b_error = ioanchorp->anchr_io_status;
        aio_bp->b_resid -= bytes_transferred;
        /* Now invoke aio completion routine; may deallocate the aio_bp */
        aio_bp->b_iodone(aio_bp);
        /* Do active range cleanup. Active ranges are passed only for AIO requests */
        spin_lock(&bfap->actRangeLock);
        remove_actRange_from_list(bfap, ioanchorp->anchr_actrangep);
        spin_unlock( &bfap->actRangeLock );
        advfs_bs_free_actRange(ioanchorp->anchr_actrangep);
```

```
            ioanchorp->anchr_actrangep = NULL;
        }
    }
```

3.2.9 AIO Interface
3.2.9.1 advfs_strategy()

3.2.9.1.1 Interface

int

advfs_strategy(struct buf *bp)

The buf structure passed to the strategy must contain byte-level addressing for the data to be transferred. Some strategy routines only require block-level addressing, but we need to be able to transfer on any boundary or for any size. The actual mechanism needs to be agreed upon. I am proposing putting this byte-level addressing directly into the struct buf fields: b_bcount and b_offset. The fields in the buf structure that must be passed in are:

bp.b_flags	in: set to B_READ or B_WRITE for operation type
bp.b_vp	in: vnode for file
bp.b_rp	in: set to (struct file *) for this file
bp.b_un.b_addr	in: address of application buffer
bp.b_bcount	in: size (in bytes) of transfer requested
bp.b_resid	out: returns # bytes not transferred on error or if IO is done synchronously
bp.b_offset	in: byte offset in file for start of transfer
bp.b_iodone	in: AIO iodone routine

Returned values will be zero for success, or an error returned from the AdvFS read/write path. This error will also be in the bp->b_error field.

3.2.9.1.2 Description

This routine is the AdvFS interface with the AIO layer when a file is opened for directIO. Thus, the AIO routine vn_aio_strategy() will set up the struct buf and call our strategy routine for reads or writes if the file is open for directIO. (Non-directIO reads and writes are handled by a threaded implementation).

The strategy routine takes the data from the buf structure, builds a uio structure, and then invokes the advfs_rdwr() routine so that this call will end up in the mainline AdvFS read/write path as described in Sections 3.2.4 and 3.2.5

This code has only minor modifications from the Tru64 version.

On HP-UX this routine also provides an interface for the VM Swap Filesystem functionality, but that is not detailed here. This routine will likely distinguish between AIO and VM Swap calls by the non-zero value of the bp.b_rp field in the AIO interface, as well as the fact that the file must be opened for directIO for the AIO call.

3.2.9.1.3 Execution Flow

The following is the code flow for this routine after determining that the call has been made on behalf of the AIO subsystem:

Declare a uio structure on the stack

Declare a 2-entry struct iovec vector on the stack

If (bp->b_vp == NULL) /* This file must have a vnode */

```
    bp.b_iodone();
    return EINVAL
Extract the struct file *fp = bp->b_rp;
/* setup the iovec vector and the uio structure  */
vec[0].iov_base = bp.b_un.b_addr;                /* application buffer */
vec[0].iov_len  = bp.b_bcount;                   /* transfer size */
vec[1].iov_base = bp;                            /* original buf struct; gets passed to iodone */
vec[1].iov_len  = 0;                             /* will be set to 1 if async I/O is started */
uio.uio_iov     = vec;                           /* iovec vector address */
uio.uio_iovcnt  = 1;                             /* second entry is 'hidden' */
uio.uio_old_offset = UIO_AIORW;                  /* for want of a better way to signal an AIO call */
uio.uio_seg     = UIOSEG_USER;                   /* user space transfer */
uio.uio_resid   = bp.b_bcount                    /* # of bytes to transfer */
uio.uio_fpflags = fp.f_flag;                     /* file flags: FREAD/FWRITE */
uio.uio_offset  = bp.b_offset;                   /* file offset */
bp.b_resid = bp.b_bcount;
sts = advfs_rdwr( bp->b_vp, &uio, (bp->b_flags & B_READ) ? UIO_READ : UIO_WRITE,
                    (fp->f_flag & FAPPEND) ?  IO_APPEND : 0, fp->f_cred  );
/* Check for error or no asynchronous IO started */
if ( sts != EOK  ||  (vec[1].iov_len == 0) )
    bp.b_error =sts;                             /* return error value */
    bp.b_resid = uio.uio_resid;                  /* update resid from uio structure */
    bp.b_iodone(bp);                             /* call AIO completion routine */
else
    bp.b_error = 0;
return ( sts );
```

3.2.9.1.4 Other Items of Note

The call to bs_aio_write_cleanup() has been removed from advfs_strategy() since it is not needed under the HP-UX AIO architecture. The call to it will also be removed from advfs_bs_io_complete(), and the routine itself will be removed from bs_qio.c.

3.2.9.2 vn_aio_strategy()

This is the routine that I believe will be responsible for calling the AdvFS strategy routine for AdvFS files that are open for directIO. To determine whether the file is open for directIO, this routine should probably first check the FDIRECT bit in the fp->f_flags field. If it is set, then this thread has the file opened for directIO. This should be a fairly quick check. If this bit is not set, then this routine can call the ioctl() routine passing ADVFS_GETCACHEPOLICY command. This will return either ADVFS_DIRECTIO or ADVFS_CACHED.

3.2.10 fcntl Path
3.2.10.1 advfs_fcntl()

A management decision was made to remove the fcntl(F_GETCACHEPOLICY) call in the HP-UX version of AdvFS. This was available for an application to query whether a file was open in directIO or cacheIO mode. This will be replaced with an ioctl() call with the ADVFS_GETCACHEPOLICY command. The removal of this code from fcntl() has already been made in the current AdvFS porting baselines.

3.2.11 ioctl Path
3.2.11.1 advfs_ioctl()

In the Tru64 code, there is an ioctl(F_SETCACHEPOLICY) call that is used as a private interface between CFS and AdvFS. On HP-UX, this interface is being replaced by a truly private interface, a new routine called advfs_cfs_set_cachepolicy() (see Section 3.2.4.1). The fcntl(F_GETCACHEPOLICY) functionality on Tru64 is being put into advfs_ioctl() This functionality has already been moved into ioctl() in the current AdvFS porting baselines. However, the use of the VDIRECTIO bit in the vnode needs to be changed to use bfap->dioCnt to determine if the file is opened for directIO.

3.2.12 Storage Allocation and Snapshotting
3.2.12.1 advfs_getpage()

3.2.12.1.1 Interface

```
int
advfs_getpage(    fcache_vminfo_t *fc_vminfo,              /* An opaque pointer to a vm data struct*/
                  struct vnode * vp,                       /* The vnode pointer */
                  off_t *off,                              /* The offset in the file of the fault */
                  size_t *size,                            /* The size in bytes to fault in */
                  fcache_ftype_t ftype,                    /* The fault type */
                  struct advfs_pvt_param *fs_priv_param,   /* File system private parameter */
                  fcache_pflags_t pflags                   /* Options or modifiers to the function */
              )
```

3.2.12.1.2 Description

The private parameters passed to getpage will now accept a new flag, APP_ADDSTG_NOCACHE. This will indicate to getpage that the request should not bring pages into the cache, but should allocate storage if any holes are present in the passed in range.

The private parameters will require that the app_starting_offset and app_total_bytes be filled in.

The calculations for the amount of pages that need to be zeroed will remain the same. This means that the request may end up allocating storage below the passed in offset and above the total request size. This will be communicated back to the caller via the *off and *size arguments. NOTE: adjusting the *off will cause the *size to be larger even if the end of the request was not adjusted. The caller will need to distinguish this. Since *off is not passed back to the caller via the fault path, the private parameters fields app_stg_start_fob and app_stg_end_fob will also reflect the range of new storage added.

Getpage will upgrade the passed in file lock to write if storage needs to be allocated (unless the lock is already taken for write).

Getpage will allocate as much storage as it can before returning an ENOSPACE error. The *off and *size arguments will reflect the amount of the request that was processed up to the first hole that getpage was unable to allocated storage for. NOTE: the *off could have been adjusted and the size and ENOSPACE returned as well.

Getpage may seize and drop the extent map lock.

3.2.12.1.3 Execution Flow

The following changes must be made to the existing advfs_getpage() logic:

- The initial calculations round the offset and length to a VM page boundary. This will need to either change for APP_ADDSTG_NOCACHE or we will need to restore the original offset and length (and the private params new_stg_xxx_fob fields) to be returned if they did not need to be modified for storage reasons.
- After calling to determine if we need to allocate any storage, return to the caller if this is APP_ADDSTG_NOCACHE and no storage needs to be allocated for this requested range.
- If storage does need to be obtained and it modified the starting offset, update the offset and size of the request, and the private params new_stg_xxx_fob fields.
- Wrap the call to lock the extent map (not for snapshots) and the call to fcache_page_alloc/UFC processing with a conditional that this is not APP_ADDSTG_NOCACHE. Set up the alloc_size and alloc_offset size to match the offset and size of the request.
- The call to advfs_bs_add_stg should stay as is. If there is an ENOSPACE error skip invalidating pages if APP_ADDSTG_NOCACHE.
- Allow the code to fall into the object safety case for zeroing pages if this is APP_ADDSTG_NOCACHE.
- Do not zero pages if APP_ADDSTG_NOCACHE.
- Do not unprotect pages if APP_ADDSTG_NOCACHE.

3.2.13 Mount Path

3.2.13.1 mount()

3.2.13.1.1 Interface

This routine will be changed to allow the '-o directIO' option to be specified.

3.2.13.1.2 Description

This routine will be changed to allow the '-o directIO' option to be specified, but this option will not be shown on any list of option or on the man pages. It is used for internal testing only. The changes noted below will pass advfs_args_t.adv_flags with the MS_ADVFS_DIRECTIO bit set through advfs_mount() and on to advfs_mountfs().

3.2.13.1.3 Execution Flow

The following changes need to be made to mount.c:

- Add string "directio" to end of myopts[] vector (before NULL terminator).
- Add: #define OPT_DIRECTIO 10
- In parse_std_options(), add:

```
case OPT_DIRECTIO:
    adv_mntflg |= MS_ADVFS_DIRECTIO;
    break;
```

- In print_flags() add:

  "directIO" as penultimate member of adv_opt_str[]

  MS_ADVFS_DIRECTIO as penultimate member of adv_opt[]

3.2.13.2 advfs_mountfs()

3.2.13.2.1 Interface

The interface for this routine will not change. The flags set in advfs_args_t.adv_flags in mount() get passed down to this routine.

3.2.13.2.2 Description

This routine initializes a new fileset structure for a given mount point.

3.2.13.2.3 Execution Flow

If the MS_ADVFS_DIRECTIO bit is set in the advfs_args_t.adv_flags field and the global variable Advfs-enable_dio_mount is non-zero, then this routine will set the BFS_IM_DIRECTIO bit in the bfSetT.bfSetFlags field to enable directIO for all files on this fileset.

This routine already does the work necessary. Only changing the name of the passed-in flag from M_DIRECTIO to MS_ADVFS_DIRECTIO is needed.

3.2.13.3 advfs_vfs_is_mounted_for_dio()

3.2.13.3.1 Interface

boolean

advfs_vfs_is_mounted_for_dio(struct vfs *vfsp) /* input: vfs struct for mount point */

3.2.13.3.2 Description

This routine is provided primarily for use by CFS to determine if a fileset has been mounted with the –o directIO option. It is passed a pointer to the vfs structure, and returns TRUE if the fileset has directIO enabled, and FALSE otherwise:

3.2.13.3.3 Execution Flow

The following is the necessary code flow for this routine:

```
struct bfSet        *bfSetp = ADVGETBFSETP(vfsp);


if ( BFSET_VALID( bfSetp ) {
      if ( bfSetp->bfSetFlags & BFS_IM_DIRECTIO) {
            return TRUE;
      }
}
return FALSE;
```

3.2.14 Other Miscellaneous Routine Changes

3.2.14.1 File Truncate Path

The code in the truncate path should continue to seize an active range while truncating the file. One change to the functionality in fs_setattr() is necessitated by the directIO design. Before the active range is released, if the file is open for directIO, any cached pages within the active range must be flushed and invalidated.

3.2.14.2 migrate_normal()

The code in the migrate path should continue to seize an active range while migrating a region of the file. One change to the functionality in migrate_normal() is necessitated by the directIO design. Before the active range is released, if the file is open for directIO, any cached pages within the active range must be flushed and invalidated. On Tru64, this was done if operating on a cluster, but not on a standalone system. On HP-UX this will be required for all directIO files whether on a cluster or not.

3.2.14.3 bs_refpg_direct()

This routine can be removed from bs_buffer2.c; it is used in the Tru64 baseline, but is not needed for the HP-UX implementation.

3.2.14.4 bs_pinpg_direct()

This routine can be removed from bs_buffer2.c; it is used in the Tru64 baseline, but is not needed for the HP-UX implementation.

3.2.14.5 Active Range Allocation

In Tru64 UNIX, the active range structures are allocated at the start of every directIO read or write request, and deallocated when the I/O has completed. Because there could be a fairly high turnover of these structures, we will create a fixed-size memory arena from which these structures will be allocated. This will involve creating constructor, destructor, allocate, and free routines for the active ranges.

3.2.14.6 Access structure construction and destruction.

The access structure constructor and destructor routines (advfs_access_constructor() and advfs_access_destructor()) must be modified to initialize and destroy the new read-write lock bfAccessT.cacheModeLock.

3.2.14.7 Routines using the obsolete VDIRECTIO bit

The VDIRECTIO flag in vnode.vflag is not being propagated into the HP-UX version. Therefore, all places in the code that reference this bit must be examined and have the functionality removed or replaced by correctly testing the bfaccess.dioCnt field. This includes:

- removing the definition of VDIRECTIO from ms_privates.h,
- changing the checks of this bit to a check of bfap->dioCnt in advfs_mmap(), advfs_ioctl(), migrate_normal(), ss_chk_fragratio(), ss_snd_hot(), and ss_vd_migrate().
- remove the paragraphs in fs_create_file(), bf_get_l(), and bs_close_one() that set or clear this bit.

3.3 Script Design

No scripts are being designed or implemented.

4 Dependencies

4.1 System Administration

No dependencies.

4.2 Memory Management

Review of calls to wire application buffer in read/write paths by someone on VM team would be appreciated.

4.3 ccNUMA

No dependencies.

4.4 Process Management

No dependencies.

4.5 File System Layout

No changes to the AdvFS layout are assumed.

4.6 File Systems

This project assumes that it is being designed to be integrated with the AdvFS code base described in the AdvFS/UFC Integration Design Specification.

The changes proposed to the FMASK are dependent on approval from the group responsible for the file descriptor structure.

4.7 I/O System and Drivers

It is assumed that the block size will be DEV_BSIZE (currently 1k) for disk transfers.

The advfs_strategy() routine is dependent on the buf structure being set up by the AIO calling routine. It is assumed that the yet-to-be-written routine vn_aio_strategy() will set up the data as outlined in Section 3.2.9.

4.8 Security

No dependencies.

4.9 Auditing

No dependencies.

4.10 Multiprocessor

No dependencies.

4.11 Behavior in a cluster

No dependencies.

4.12 Kernel Instrumentation/Measurement Systems

No dependencies.

4.13 Diagnostics

No dependencies.

4.14 Panic/HPMC/TOC

No dependencies.

4.15 Commands

No dependencies.

4.16 Standards

No dependencies.

4.17 Kernel Debugger

No dependencies.

4.18 Boot Kernel

No dependencies.

4.19 Install Kernel

No dependencies.

4.20 Update/Rolling Upgrade

No dependencies.

4.21 Support Products

No dependencies.

4.22 Learning Products (Documentation)

There will be documentation impacts, and these are being addressed elsewhere..

5 Issues (Optional)

High Priority

- Issue: Adding the FDIRECT bit in the FMASK definition remains to be agreed upon. This is discussed in Section 3.1.1, and this should be agreed to by the group responsible for the file structure, since this change will result in the FDIRECT bit being preserved in the file.f_flag field..
  - Status: Open.

- Issue: The details regarding the AIO/AdvFS interface remain to be agreed upon. One proposal has been detailed in Section 3.2.9, but this should be agreed upon by the group responsible for the AIO interface
  - Status: Open.