AdvFS Space-Efficient Small File Support

Design Specification

Version 2.0

TM

CASL

Building ZK3
110 Spit Brook Road
Nashua, NH 03062

Design Specification Revision History		
Version	**Date**	**Changes**
1.0	03/09/04	First draft for internal review.
2.0	03/30/04	Incorporate changes from formal review.

Table of Contents

Preface

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

Author Name	Mailstop	Email Address
TM		

Sign-off review

Approver Name	Approver Signature	Date

1 Introduction

1.1 Abstract

This design describes an implementation for supporting space-efficient small files in AdvFS on HP-UX 11.31. A small file in this context is defined as a file that uses less than 4k of disk space.

1.2 Product Identification

Project Name	Project Mnemonic	Target Release Date
AdvFS Space-Efficient Small File Support	Small Files	

1.3 Intended Audience

This design assumes a good deal of familiarity with AdvFS kernel internals and with the mechanisms that AdvFS uses to interface with the UFC. As a result, the design is intended to be read and reviewed by AdvFS kernel engineers. But since it also describes functional aspects of the small file support, other readers might include technical writers, GUI designers, and other individuals interested in how AdvFS will behave on HP-UX.

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 presents a description of the support for space-efficient small files to be implemented in AdvFS for HP-UX. The design will address the kernel modifications required to implement support for space-efficient small files as well as the minimal user space changes required. Further work such as storage allocation improvements and a conversion utility for changing a filesystem from one block size to another will not be presented in this design but neither will the design preclude such work.

1.6 Acknowledgments & Contacts

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

DA, FG, MD.

1.7 Terms and Definitions

Term	Definition
Small Files	Files using less than 4k of disk space

2 Functional Description

The functionality provided in this design is described below:

1. Users may select 1k or 2k block sizes at mkfs time. These will be referred to as "small block sizes" in the remainder of this functional description.
2. If a small block size is used, storage will be added to a file in allocations the size of the chosen block size if the following two conditions are met:
   - The file is not sparse.
   - The file is less than 4k (VM page size) in length.

   If a small block size is used, then once files grow beyond the 4k offset, storage will be allocated to the files in contiguous 4k chunks. By switching over to 4k allocations, AdvFS will attempt to provide better performance for larger files in the small file environment.
3. If a small block size is used, the smallest hole in a file will be 4k. This is to prevent mmappers from writing to holes within a single VM page. If a small block size is used and a non-sparse file which is less than 4k in size (and therefore, has 1k, 2k, or 3k of storage) is then extended writing to or beyond the 4k offset, AdvFS will at that time "fill in" the rest of the first 4k page with storage. This is to prevent an mmapper from modifying the new hole at the end of the first 4k page in the file. Similarly, if a file with 1k of storage at offset 0 is then extended by writing 1k at offset 2k, AdvFS would fill in the intermediate 1k hole at offset 1k to avoid having a hole smaller than 4k in the file.
4. If a small block size is used and a file that is larger than 4k is then truncated down to a size less than 4k, AdvFS will again allow the file to use less than 4k of storage.

Functionality specifically not provided by this design includes:

1. Capability to convert a filesystem from one block size to another block size.
2. Changes to the general storage allocation algorithms being addressed in a separate project.

3 Design Description

3.1 Design Approach

This design focuses on the requirement that small files are to be stored in a space-efficient manner. They should also perform as closely to other file sizes as possible. In a filesystem optimized for small files, larger files may perform worse than they would in a filesystem optimized for larger files.

3.2 Overview of Design

On Tru64 UNIX, small files in AdvFS were implemented using a filesystem-wide frag file. This approach was attractive in that it allowed a filesystem to contain large and small files in a space-efficient way. This design originated around 1990, when storage was more expensive than it is today. Tru64 had a fixed 8k allocation unit, so the minimal file size was 8k if frags were not being used. The frag was minimally 1k in size and had a maximum of 7k. A frag would contain the whole file, for files less than 7k, or would contain the non-8k-aligned end of a larger file. This allowed files to consume any 1k increment by combining the 8k allocations units with a single frag at the end of the file. This was an elegent approach that provided a space-efficient way to store small files and the ends of larger files. It also had the advantage of not requiring system administrators to choose a block size at filesystem creation time. But it also had an associated cost. A frag would be created on the last close of a file. This meant that the file would initially consume 8K of storage and then at close time AdvFS would have to copy the data from the 8K allocation into a smaller fragment. Similarly at open time if the file was being written the frag would need to be copied into an 8K allocation unit. This double copying was expensive. A further drawback of this approach was realized with the introduction of the cluster filesystem over AdvFS. CFS reads files directly from the client node bypassing the advfs filesystem. This is known as cached concurrent reads. The client has a copy of where the storage resides on disk and has direct access to the disk. AdvFS would revoke these storage maps whenever the location of storage changed on disk for a file. As long as the storage did not change these cached reads could take place without having to transfer data over the interconnect. Frags presented a problem since the location of the frag was unknown to the client and the data always needed to be transferred over the interconnect. Also, the truncation of storage when creating a frag caused AdvFS to revoke the storage maps too frequently. The final disadvantage was that the frags themselves belonged to a single bitfile. There was a single frag file per filesystem. This file and its associated locking became a bottleneck when lots of small file activity was taking place in the domain. In addition, if the frags file itself became fragmented, with many extents, access to small files could be slow as the code had to walk the in-memory extent maps.

The approach to be taken in HP-UX aims to alleviate these drawbacks. First, storage is now much cheaper than it was when AdvFS on Tru64 was designed and the need for space-efficient small files is not as universal as it was when the Tru64 solution was designed. This need is now mainly constrained to a special class of users who primarily have very large numbers of very small files. Since space-efficient small files have become more of a specialized requirement, this design will target that requirement as opposed to attempting a general solution as Tru64 did.

This design will provide support for filesystem block sizes of 1k and 2k, to complement the existing 4k and 8k block sizes already supported by AdvFS on HP-UX. These fixed block sizes, while lacking the flexibility of the Tru64 design, will eliminate the double copying, the CFS issues, and the one-file bottleneck of the Tru64 design. Performance of files that are smaller than or equal to the new smaller block sizes should be comparable to the performance of such files in filesystems created with larger block sizes. One unfavorable side effect of allowing a block size that is different from the 8k block size used by metadata (and this applies to the existing 4k block size already supported in AdvFS, as well) is that applications and commands may get ENOSPC errors when trying to allocate metadata storage. For example, a “mkdir” might fail with ENOSPCE even though “df” reports that the filesystem is only 60% full. Intuitively, it feels like the smaller the block size, the more likely this kind of behavior could occur.

When a file reaches offset 4k, AdvFS will switch over to allocating storage in 4k chunks. This is an effort to adapt to a usage pattern that doesn’t match the filesystem block size specified at mkfs time. In addition, to avoid situations where a VM page fully contained within the file but only partially backed by storage is touched by an mmapper, AdvFS will prevent holes in sparse files that have size less than 4k. So a page will either be fully backed by storage or not backed by storage at all. The exception (which, we hope, will be the general rule!) is when a file has no holes and is less than 4k in size. In that case, the VM page for the file may, indeed, be partially unbacked but POSIX semantics will prevent any data that an mmapper modifies beyond EOF from being written to disk.

3.3 Module Design

3.3.1 SBM

Currently, a bit in the SBM represents 4 DEV_BSIZE units of disk space:

```
#define ADVFS_BS_CLUSTSIZE    4
```

```
vdattr->sbmBblksBit = ADVFS_BS_CLUSTSIZE;
```

Since we will need to be able to allocate storage in units of 1k and 2k, a bit in the SBM will now represent 1 DEV_BSIZE unit of disk space:

```
#define ADVFS_BS_CLUSTSIZE    1
```

This is very likely a temporary change as the storage allocation project being done independently from this project will most likely remove the SBM entirely.

3.3.2 advfs_fs_write()

If we are writing to a file and the page to which we want to write is in cache and has translations, fcache_as_uiomove() will not call advfs_getpage(). If that page is the one page encompassing an entire small file and we are writing beyond EOF and beyond the last current allocation unit, we need to force the writer into advfs_getpage() so new storage can be allocated. So we will add new logic to advfs_fs_write() just before the existing call to fcache_as_uimove():

- If the file's allocation unit is less than `VM_PAGE_SIZE` and the `bfap->file_size` is less than `VM_PAGE_SIZE` and we are going to write beyond `bfap->file_size` rounded up to the allocation unit but still within the first VM_PAGE_SIZE bytes of the file, force a call into advfs_getpage() by calling fcache_as_fault(FAF_GPAGE). This will give advfs_getpage() the opportunity to allocate more storage. Then perform the uiomove().
- Otherwise, issue the current fcache_as_uiomove().

3.3.3 fs_setattr()

Because we are not going to allow holes smaller than VM_PAGE_SZ, if an application calls truncate() to increase the size of a file which is a small file and if the new file size causes the file to end in a new allocation unit, we must allocate storage and zero-fill it. If the file is, say, extended from 1k to 3k this way, we will have to allocate 2k more of storage, leaving the file as a 3k file. On the other hand, if the file is 1k and the truncate() call extends it to 100k, we will add zero-filled storage to fill out the first VM_PAGE_SZ byte of the file but leave the rest as a hole:

- If the block size of the filesystem is less than `VM_PAGE_SZ` and the `bfap->file_size` rounded up to an allocation unit boundary is less than `VM_PAGE_SZ` (this is a small file), and the user is truncating the file to a larger file size and the new file size ends in a new allocation unit:
    - Set up private parameters describing the amount of storage to be added.
    - While more storage to add and zero-fill:

    - Call fcache_as_map().
    - If this is the first time through the loop, call fcache_as_fault(FAF_GPAGE) to force us into advfs_getpage() so storage is added.
    - Call bzero() on the mapping's virtual address to zero-fill the range.
    - Call fcache_as_unmap().

If the file is being truncated smaller, and it will become a small file, make the following adjustment:

- Modify the code that today zero-fills from the new EOF to the end of the allocation unit so that it zero-fills from the new EOF to the end of the allocation unit or the end of the VM page, whichever is greater. So, for example, a 12k file being truncated to 1.5 k will have the range from 1.5k to 4k zero-filled here. This is because we are not going to invalidate this first page later in the storage removal path. The storage removal path will, in fact, truncate the storage down to the small allocation unit size (less than 4k).

If the file is being truncated smaller and it will not become a small file, we need to treat it as if it had a 4k allocation unit. For example, a 12k file being truncated to 10k should not have any of its storage removed.

- Modify the code that today zero-fills from the new EOF to the end of the allocation unit so that it zero-fills from the new EOF to the end of the allocation unit or the end of the VM page, whichever is greater.

3.3.4 fs_trunc_test()

This function decides whether there are allocation units beyond the end of the file that could be truncated. It needs to be modified to recognize that if a small allocation unit (1k or 2k) is in use and the file size is 4k or greater, the decision to truncate should be made as if the allocation unit were 4k. So a file with 12k of storage allocated and a file size of 10k should cause fs_trunc_test() to set `bfap->trunc` to zero.

3.3.5 stg_remove_stg_start()

In this function, we first remove the requested storage range from the file and then invalidate the corresponding pages. The logic here will be adjusted so that if the storage removal request starts in the middle of a page (say, a truncation from 12k to 2k), the call to fcache_vn_invalidate() will be issued on page boundaries. Specifically, the offset of the request will be rounded up to page size and the length adjusted down accordingly.

3.3.6 advfs_getpage()

3.3.6.1 Small file flags

At the beginning of advfs_getpage(), a local flag variable, `is_a_small_file`, will be set which will indicate that the current file is a small file. This flag will be set to TRUE if the block size of the filesystem is less than `VM_PAGE_SZ` and the `bfap->file_size` rounded up to an allocation unit boundary is less than `VM_PAGE_SZ`. Otherwise, it will be set to FALSE. A second local variable, `will_remain_a_small_file`, will be set to indicate whether the current file will still be a small file after the current request. It will be set to TRUE if `is_a_small_file` is TRUE and if the size of the file after the request, rounded up to an allocation unit boundary, is still less than 4k. Otherwise, it will be FALSE.

3.3.6.2 Faulting for a read

The following changes will be made in the read fault path for small files:

- If the call to advfs_getpage() is for read intent and `is_a_small_file` is TRUE, assert that fcache_page_alloc() did not return a large page.
- If the page is not in cache, then when calling advfs_get_blkmap_in_range(), if `is_a_small_file` is TRUE, do not pass the `alloc_size` as the length parameter. Rather, pass `bfap->file_size` rounded up to an allocation unit.
- If the page was not found in cache, then before calling advfs_start_blkmap_io() to read the page from disk, if `is_a_small_file` is TRUE, zero-fill the portion of the VM page that will not be read from disk by calling advfs_bs_zero_fill_pages(). Do not protect the page. Then, call advfs_start_blkmap_io() but passing in the entire size of the `storage_blkmap` rather than just the size of the first entry for the length parameter.

3.3.6.3 Faulting for a write

The following changes will be made in the write fault path for small files:

- If the page is in cache, we perform some checks under ADVFS_SMP_ASSERT to make sure that pages with holes to be filled are read-only. The assertion condition will need to be changed to exclude the case where `is_a_small_file` is TRUE and the current pfdat has `pfs_off` == 0.
- If the page is not in cache, then when calling advfs_get_blkmap_in_range(), if `is_a_small_file` is TRUE, do not pass the `alloc_size` as the length parameter. Rather, pass `bfap->file_size` rounded up to an allocation unit. If this calculation yields zero (file is currently empty), then round the length parameter up to VM_PAGE_SZ. This will allow us to get a valid blockmap (of a 4k hole) back from advfs_get_blkmap_in_range(), which we can then process as usual.
- If we did not find the page in cache, follow the non-`FP_CREATE` path if `will_remain_a_small_file` is TRUE. This is because we will not be overwriting any entire VM pages.
- If we did not find the page in cache and `FP_CREATE` is not set or `will_remain_a_small_file` is TRUE (the current non-`FP_CREATE` path), if `is_a_small_file` is TRUE, then before calling advfs_start_blkmap_io() to read the page in from disk, zero-fill the portion of the VM page that will not be read from disk by calling advfs_bs_zero_fill_pages(). Do not protect the page. An exception to this rule will be when the file is currently zero bytes and the offset of the write is byte zero. In that case, only zero-fill the bytes that are not covered by a write. This is safe because no mmapper could see that page until the file size is bumped in advfs_fs_write(). After zeroing, call advfs_start_blkmap_io() but passing in the entire size of the `storage_blkmap` rather than just the size of the first entry for the length parameter. This is because the blockmap may have several entries.
- In the `FP_CREATE` path where we have a blockmap entry that represents a hole, we currently zero and protect pages that are not protected by `FP_CREATE`. Change this slightly so that if `is_a_small_file` is TRUE and the blockmap entry offset is zero, we don't protect the page.
- In the `FP_CREATE` path where we have a blockmap entry that represents storage, we currently follow a three-step program to deal with fobs (friends of Bill) that fall outside of the range that will be completely overwritten. We will add a new condition to get into this logic: `is_a_small_file` is TRUE but `will_remain_a_small_file` is FALSE. In that case, we will add new logic for Step 1. This logic will see if the blockmap starts at an offset in the first 4k of the file and, if so, it will read from disk the portion of the first page that is backed by storage

and zero-fill the rest, not protecting the page, as described in the preceding bullet item. We will then go on to Step 2 as in the current code.

- In the "`if(request_needs_storage)`" logic, before servicing the current request's need for storage, add new logic to see if `is_a_small_file` is TRUE and if `will_remain_a_small_file` is FALSE. If so, and if the current request offset is 4k or higher, we need to go back and fill in the rest of the first 4k with storage to prevent a hole smaller than 4k. To do this, call advfs_get_blkmap_in_range() on the first 4k, asking for holes only. Then call advfs_bs_add_stg() to fill in the rest of the first 4k of the file. Then call advfs_bs_zero_fill_pages() to zero-fill the area that just had storage added. Do not protect the page. Finally, call advfs_free_blkmaps() to free the blockmap.
- In the "`if(request_needs_storage)`" logic, when calling advfs_get_blkmap_in_range() to find the holes that need to be filled, if `will_remain_a_small_file` is TRUE, do not pass `alloc_size` as the length parameter. Rather, pass the end of the write request, rounded up to an allocation unit.
- Modify the logic to add storage so that if this will remain a small file, we allocate in allocation units. If the file is currently a small file but will not remain a small file, we will add storage in two calls:

  1. Call advfs_bs_add_stg() with a `minimum_contiguous_fobs` argument of allocation unit size to get storage up to the 4k mark.
  2. Call advfs_bs_add_stg() with a `minimum_contiguous_fobs` argument of 4k for the remainder of the storage added.

  Extending a previously small file in this way allows us to hide the small file/non-small file allocation policy difference from the storage stack. If the storage allocation algorithm is working well, we'll actually get one piece of contiguous storage from these two calls.
- After successfully adding storage, if `is_a_small_file` is TRUE and `will_remain_a_small_file` is FALSE, set `is_a_small_file` to FALSE, since the file uses less than 4k of storage.
- Modify the code that calls advfs_unprotect_range() so that it will not attempt to unprotect the first page of a file if `is_a_small_file` is TRUE. There are four calls to advfs_unprotect_range(). The conditions to call the first one do not need to change. For the second one, which unprotects all pages in the sparseness map in the non-`FP_CREATE` or in-cache case, we will now check to see if `is_a_small_file` is TRUE and, if so, only unprotect the range beyond 4k, if any. For the third and fourth calls to advfs_unprotect_range(), which deal with pages that were not in cache for the `FP_CREATE` case and were on either side of the request, again check if `is_a_small_file` is TRUE and, if so, adjust the range to be unprotected to avoid the first 4k of the file.

3.3.7 advfs_putpage()

The following changes will be made to handle the new possibility that a single VM page may be backed by multiple storage locations. This might happen, for example, if a 3k file had three non-contiguous 1k allocations of storage. We also need to handle the possibility of doing a 1k or 2k I/O from the middle of a VM page.

- Change the rounding logic done at the beginning of the function so that we round to allocation unit, not VM page size, for small files.
- If fcache_page_scan() returns an offset and size outside of the request, do not bump the large page statistics in the case where this is a small file.

- When calling advfs_get_blkmap_in_range(), it is possible that it will now return a multiple-entry extent map ending with a hole entry. Today, if we encounter a hole in such an extent map on a dirty page, we panic. This will now be tolerated if the following conditions are met:

  1. The allocation unit for the file is less than VM_PAGE_SZ.
  2. The hole does not start at the beginning of the page.
  3. The hole does end at the end of the page.

  If all of these conditions are met, simply go to the end of the loop to get the next blockmap entry (which should be NULL; assert this).

- Modify the conditions under which we release pages at the beginning of the plist so that we don't accidentally release the one page in small file to which we will write.
- When calling advfs_start_blkmap_io(), pass in the entire size of the `storage_blkmap` rather than just the size of the first entry for the length parameter.

3.3.8 advfs_kickoff_readahead()

It is possible that a thread reading a small file sequentially may decide that it wants to do read-ahead and read in page 0 of the file. That page is most likely cached but it may not be since advfs_getpage() releases pages before calling advfs_kickoff_readahead(). So having advfs_kickoff_readahead() re-issue the read for this page that was in cache but now is not does make some sort of logical sense. But the complexity of zeroing part of the page and handling small file semantics in this function is not worth it because it is so unlikely to be used. Instead, at the beginning of this function, if this is a small file, return. If the caller wanted to issue a read-ahead on page 0 and it is now not in cache, the system is probably under very high contention and even if the read-ahead code were to bring page 0 back in, it may be gone again by the time the application did its next read.

3.3.9 advfs_bs_zero_fill_pages()

Currently, advfs_bs_zero_fill_pages() only zero-fills entire pages. It will be modified to zero-fill only the exact byte range passed in by the caller. This will allow advfs_getpage() to only zero-fill a part of the page that is not covered by file data. Do not release the page if this is a small file and we are zeroing only part of the one page in the file.

3.3.10 advfs_start_blkmap_io()

Currently, advfs_start_blkmap_io() assumes that the primary_blkmap has only one extent_blk_desc_t, which encompasses the entire I/O to be done. Now, however, advfs_getpage(), when working with small files, will call advfs_start_blkmap_io() with, for example, a primary_blkmap comprised of three 1k extent_blk_desc_t entries chained together. This will require the following changes to be made:

- When assigning the value of the original buf's `b_bcount` field, we will now use `MIN(io_size, primary_blkmap->ebd_byte_cnt).`
- After setting up the buf struct for the primary I/O but before processing the secondary extent map, add a new logic loop which will process any further primary_blkmap extents. This logic will be similar to that which processes the secondary extent map:

  While (primary_blkmap->ebd_next_desc) {

- Current descriptor = primary_blkmap->ebd_next_desc.

- If the current descriptor describes a hole (only should happen when called from advfs_putpage), go to the bottom of the loop to remove the descriptor from the chain and free it.
- Lock I/O anchor, bump anchr_iocounter, unlock I/O anchor.
- Call advfs_bs_get_buf() to get a buf structure.
- Copy the original buf structure into the new buf structure.
- Update the b_foffset, b_un.b_addr, b_blkno, and b_bcount fields.
- Call advfs_bs_startio().
- Remove the current descriptor from the chain and call advfs_free_blkmaps() to free it.

3.3.11 Add storage functions get a new argument

The following functions will get a new argument, `bf_fob_t minimum_contiguous_fobs`:

- advfs_bs_add_stg()
- stg_add_stg()
- add_stg()
- alloc_stg()
- alloc_append_stg()
- stg_alloc_from_svc_class()
- stg_alloc_from_one_disk()
- alloc_from_one_disk()
- stg_alloc_one_xtnt()

This will be set to `bfap->bfPageSz`, the file's allocation unit, in general. The exception (initially) will be when advfs_bs_add_stg() is called from advfs_getpage() to allocate at least 4k of storage for a file is in a 1k or 2k blocksize filesystem but which is not a small file.

3.3.12 stg_alloc_one_xtnt()

This low-level storage allocation function calls the SBM code to get one contiguous chunk of free space. Specifically, it calls sbm_find_space(), passing `bfap->bfPageSz` as the minimum number of contiguous blocks that it needs. It will now pass `minimum_contiguous_fobs`.

3.3.13 Commands

Two commands are affected by this project:

- mkfs – Will need to be modified to allow for 1k and 2k block size selection.
- fsck – Will need to be modified to recognize as incorrect any holes in a file less than 4k in size.

4 Open Issues

4.1 1k/2k block size vs. small-file behavior with 4k block size

In the formal review of version 1.0 of this design, the idea was raised that perhaps instead of implementing a 1k block size and a 2k block size, AdvFS should behave as described in this design when a 4k block size was requested and the files are smaller than 4k. When files grew to 4k or larger, the existing 4k block size behavior would occur, with storage being allocated to the file in contiguous 4k chunks. This approach has the following advantages:

- Customers get space-efficient small files for “free”. They don’t have to ask for them.
- Some code changes are simplified.
- AdvFS can claim that it adapts storage allocation policy to system behavior.

This approach also has some disadvantages:

- Performance may be worse for 4k filesystems with the automatic small-file behavior.
- Customers may find the behavior confusing. This could result in support calls and the resulting cost. This is similar to the many calls we got about “huge” quota files on Tru64 when we increased the number of user IDs available. True, it could all be explained and we could point users to documentation. But the cost of the support might be considerable.
- Backups of a 4k AdvFS filesystem that are restored to an equivalent sized VxFS filesystem may get ENOSPC errors at the time of the restore because VxFS is truly using 4k for small files and AdvFS may use less.
- Veritas marketing may claim that AdvFS doesn’t support 1k and 2k block sizes.

The decision was made to compare the performance of a 4k filesystem with the small file behavior against the performance of a 4k filesystem without the small file behavior. If the small file support degrades performance, the issue will be closed and 1k and 2k block sizes will be supported. If the performance does not degrade, this issue will be revisited.