# Implementing Write Support in dtfs

Christian Czezatke

May 12, 1998

**Abstract**

This document gives a short overview of the support for read/write operations in dtfs. It shows how the dtfs core can be implemented without making any assumptions about the traditional file system's indirect addressing scheme. It is recommended to have a look at chapter 1 of "Implementing dtfs", altough this document is not completely up-to-date anymore.

## Contents

**1 Integrating dtfs Writes in Linux** **1**

**2 dtfs and Traditional Filesystem's Indirect Data** **2**

**3 Integrating Reads and Writes** **3**

**4 Delayed Indirect Information Update And Read Accesses** **4**

**5 Advantages Of This Approach** **5**

**A Changes** **6**

## 1 Integrating dtfs Writes in Linux

dtfs reads blocks from the device by means of the traditional Linux buffer cache. However, no write access is made by using the traditional Linux buffer cache. dtfs implements its own write cache, the segment manager. This is necessary because of two reasons:

1. It would be very hard to implement the special write semantics that are required for a log-structured file system by using the Linux buffer cache for writes.

   The reason for that is that the Linux buffer cache is implemented in a way that makes it impossible for the file system to determine the actual

sequence of writes that is performed on the device. However, such a control over the actual write taking place is required because

- a log-structured file system has to guarantee a transaction-like semantic when writing out a partial segment.
- a log-structured file system has to write out data in large chunks (and not just block by block) in order to achieve good performance.

2. When using the traditional Linux buffer cache, a block must be addressed by the device it belongs to and its *physical* address on that device. However, this addressing scheme causes several problems when used with dtfs that will be discussed in this article.

In the long run the standard Linux buffer cache and dtfs's writte cache should be merged again.

# 2 dtfs and Traditional Filesystem's Indirect Data

## Dealing With Different Kinds Of Indirect Data

A filesystem must use some kind of indirect information to refer to the actual data blocks in a file. The problem is that different kinds of traditional file systems use different ways to represent this information. Traditional Unix file systems use an addressing scheme that involves indirect blocks containing pointers to the actual data blocks, while other file systems use different structures to represent this information.

Since the dtfs core should be able to support different traditional file systems it would be beneficial if the dtfs core could be implemented in a way that avoids making any assumption about the structure of this indirect information.

Furthermore, the Linux VFS layer does not make any assumption about this indirect information too. Blocks within a file are addressed by the inode they belong to and their logical block number within the file.

The BSD LFS implementation uses a special technique to map the indirect blocks used by the FFS into the 4.4BSD cache that accesses blocks by the vnode number of their owner and their logical offset within the file (which is essentially almost the same view that is used by the Linux VFS interface).

Since the Linux VFS layer does not deal with indirect information directly, there should be no need to deal with the indirect information within the generic dtfs code, too. So the handling of indirect information should be left to the filesystem-specific part only.

### The Role Of the Segment Manager

Dirty file blocks are accumulated in the segment manager. The segment manager must be able to provide the following services to other layers of dtfs:

- Fast access to a (modified) block of a file indexed by the inode number it belongs to and the logical block number within the file.
- Access blocks in a file-by-file way: It must be possible to retrieve all the dirty blocks of a file ordered by their logical block number. This is required for writing out the blocks to a partial segment.

## 3 Integrating Reads and Writes

### Accessing blocks through the Address Translation Layer (ATL)

Every read operation performed by a tradfs using the dtfs layer must be performed by calling either

```
atl_bread(uint inode_num, uint32 log_blocknum, int create);
```

or

```
atl_getblk(uint inode_num, uint32 log_blocknum);
```

The difference between these two calls is that a call to `atl_getblk` indicates that the caller is not really interested in the actual content of the file's block (because he is going to overwrite the block anyway, for example...) but needs a handle for the block.[1]

Locking blocks is not necessary since the kernel serializes write access to an inode and all the data structures associated with it in a higher layer (have a look at fs/read_write.c) by using semaphores.[2]

### Writing Out A Segment – An Outline

Every write operation to a file will finally happen to a block that has been accessed by a call to `atl_bread` with the"want_modify" flag set or by a call to `atl_getblk`. So all dirty blocks are aggregated within the segment manager.

The segment manager groups dirty blocks by the file they belong to. Since inodes are mapped into files, a modification to an inode will finally boil down to a simple file operation too. When a segment gets written out, the dirty blocks get written on a per-file basis. The segment manager writes out the data dirty blocks and adds their new physical location to a database within the address translation layer.

[1] Check with getblk implementation of the kernel.

[2] However, in on an SMP system, writes to different files can happen synchronously, so care should be taken when updating the segment manager's data structures.

When the writing of dirty data blocks for a certain file is finished, a function in the filesystem abstraction layer (FAL) will be called that rebuilds the indirect data of that file and writes it out. The physical address of these indirect data blocks is added to a special address space in the fixup database of the address translation layer.

After that the next dirty file is written out.

### Delayed Indirect Information Update

When dtfs runs out of free blocks in the partial segment before all dirty files are written out, it may decide to continue writing out the data in another partial segment or may decide to delay the pending write operations further.

No special measures must be taken in the first case when dtfs continues to write out dirty blocks into a new partial segment.

However, the second case is a bit more complicated to deal with: In this case a situation may arise in which some data blocks of a dirty file are already written out (so that they are not in the segment manager any more), but not the updated indirect data.

In order to be able to locate these new data blocks while the metadata information is not written out, the entries for these blocks in the address translation layer must remain there until all the metadata information is finally written out.[3]

## 4 Delayed Indirect Information Update And Read Accesses

Every read access to any block of a file will finally be performed by using a call to atl_bread. From the write algorithm outlined in the section above, the following pseudo-code implementation of atl_bread can be derived:

```
atl_bread(inode_num, log_blocknum, create, want_modify)
{
     block = find_block_in_segmgr(inode, log_blocknum, create);
     if (found(block))
        return block;

     phys_blockaddr = lookup_phys_blockaddress_in_atl(inode,
                         log_blocknum);

     if (found(phys_blockaddr)) {
```

[3] However, all this could be avoided if we decide to write out *all* the available dirty blocks once we have started writing. This has the drawback that many partial segments will be written that are smaller than a logical segment. However, maybe we can simply live with it since this means only one additional block to be written out.

```
                block = bread(inode->device, phys_blockaddr);
                if (want_modify)
                        transfer_block_to_segmgr(inode, log_blocknum,
                           block);
                return block;
        }

        block = fal_readblock(inode, log_blocknum);
        if (!found(block)) {
           if(!create)
                   block = NO_BLOCK;
           else
                   block = get_new_block_from_segmgr(inode,
                              log_blocknum);
        }

        return block;
    }
```

So a block is first looked for in the segment manager. If it cannot be found there, we try to look up its current physical block address in the address translation layer. This takes care of situations in which only a part of the dirty blocks of a file have already been written out and the on-disk indirect information has not been updated.

If we can determine the physical address of the requested block on the device by that means, the respective block is read in directly by calling the device read function "bread".[4]

If the block could not be found in neither the segment manager nor the address translation layer, the traditional file system is asked to read in the block by calling `fal_readblock`.

If this fails too, then the block is not a part of the file already. This might happen when new data is appended at the end of a file and a new disk block must be allocated to hold the new data.

By using this scheme it is not necessary to update the indirect information of the traditional file system at the same time the data is written by the application program:

# 5 Advantages Of This Approach

When the application program writes data to a file, the blocks affected by the write will be found in the segment manager without using the traditional filesystem's indirect information. So it does not matter that this information may not be up-to-date. Whenever data is accessed that has not been modified,

[4] maybe use ll_rw_block.

it will not be found in the segment manager and `fal_readblock` might use the indirect information of the traditional file system to located the data requested.

However, this does not cause a problem since the information needed from the indirect information for locating a block that has not been modified is still accurate.[5]

This also means that it is not necessary to map the indirect blocks somehow into the logical block addressing scheme (as it is done by the BSD LFS implementation) that is used by dtfs since it is not necessary to access them from dtfs code.

Another benefit of this approach is that it avoids incorporating a certain addressing scheme used by a traditional file system into dtfs-specific code thus easing the future porting of other traditional file systems that might use a different scheme of indirection.

Furthermore, this strategy avoids any consistency problems in indirect block information: The Linux VFS layer serializes write accesses to a file, but it does not hinder any read accesses to that file while a write to this file is in progress. Since the indirect information of the traditional file system is not used to locate modified blocks of a file, no problems can arise from inconsistencies in these data structures as they could happen for a read operation that is performed while a write is in progress.

# A Changes

$Id: writesupport.lyx,v 1.2 1998/05/12 09:24:52 ceci Exp $ - $Log: writesupport.lyx,v $$Id: writesupport.lyx,v 1.2 1998/05/12 09:24:52 ceci Exp $ - Revision 1.1 1998/05/12 09:14:06 ceci$Id: writesupport.lyx,v 1.2 1998/05/12 09:24:52 ceci Exp $ - Initial revision$Id$ -

[5] This holds true as long as the writing of block A in a file does not affect the on-disk location of another, unmodified block B in the same file, which should not be too hard a limitation.