UNIVERSITY OF OSLO
Department of Informatics

File System Supporting Arbitrarily Sized Allocation

Master Thesis

Bjørn Erik Lømo

August 15, 2008

The majority of today's filesystems use a fixed block size, defined when the filesystem is created. If an allocation request is not a multiple of the block size, space is wasted when parts of blocks are left unused. This thesis looks into the viability of a filesystem with arbitrarily sized blocks, called grains. The file system uses an existing allocator, implemented by Stanislav Sokolov, based on the Quick Fit main memory allocation algorithm. To limit the scope of the thesis, the file system is implemented using FUSE, and lacks support for growing and fragmented files, directory hierarchies and crash recovery. However, even without these featurees, the file sytem is still useful for a few scenarios, where file sizes are known in advance and the size distribution is favorable for the allocator. Examples of this are web- and mail servers. The thesis also runs some simulations based on a recorded mailserver workload, and compares results to the same workload on Ext2.

Acknowledgements

I would like to thank my primary supervisor Željko Vrba for his guidance and patience during the writing, and for the implementation of the ext2frag program. I would also like to thank my secondary supervisors, Carsten Griwodz and Pål Halvorsen, for taking me in and offering helpful advice.

Contents

1 Introduction **9**

1.1 Block Devices . . . 9
1.2 File Systems . . . 10
1.3 Fragmentation . . . 10
1.4 Extents . . . 12
1.5 Blockless Filesystem . . . 13
1.6 Overview . . . 13

2 Design **15**

2.1 Related Work . . . 15
2.2 File System Overview . . . 16
2.3 Data Structures . . . 19
2.3.1 Skip list . . . 19
2.3.2 Superblock . . . 24
2.3.3 Extents . . . 26
2.3.4 Metadata Extent Headers . . . 27
2.3.5 Inodes . . . 29
2.4 Design Alternatives & Possible Extensions . . . 31
2.4.1 Fragmented Files . . . 31
2.4.2 Directory Hierarchy, Ordering . . . 36
2.4.3 Reducing Memory Footprint . . . 38
2.4.4 Predictive Preallocation . . . 40

2.4.5 Crash Recovery & Fault tolerance 43

3 Implementation **45**

3.1 Usage Restrictions 45

3.2 FUSE - file system in Userspace 46

3.3 Architecture 49

3.3.1 Media I/O Layer 49

3.3.2 Allocation Layer 50

3.3.3 File Link Layer 51

3.3.4 Metadata Manager 58

3.3.5 Sokolov's Implementation 59

3.4 Mounting & Unmounting 60

3.5 Formatting the File System 61

4 Evaluation **63**

4.1 Simulated Workload 63

4.2 Test Cases 64

4.3 Test Results 65

5 Conclusion **75**

5.1 Summary 75

5.2 Future Work 75

Bibliography **77**

1 Introduction

1.1 Block Devices

The vast majority of todays storage media are block based. The storage space on these *block devices* is subdivided into blocks, or sectors, of uniform size. These sectors usually have a payload of 512 bytes, and some devices, like compact discs, have 2048 bytes. In addition to the data payload, each sector also contains a checksum field used by the controlling hardware to detect, and in some cases correct, data errors that crop up due to imperfections in the storage medium. All block devices have the property that all reading and writing requires entire blocks to be read or written to, even to only read or write a small part of a block. As a natural extension of this, most file systems are block based as well.

Traditionally, hard disk blocks used an addressing scheme that exposed the device's physical details. Blocks were commonly addressed by their cylinder, head and sector, referring to platter number, platter side and angle on the platter, respectively. Today, the most common scheme for specifying the locations of blocks on block devices is Logical Block Addressing, or LBA. Sectors are located by an index, where the first sector has address 0, the second has address 1, and so on.

Recently, the International Disk Drive Equipment and Materials Association (IDEMA) have defined and approved the Long Block Data (LBD) standard. This standard increases the default block size to 4096 bytes. As modern hard drives get higher data densities, The LBD standard ensures better error detection, and correction [1], as well as increased throughput for large data streams, since more data can be read per block

operation.

1.2 File Systems

A file system is responsible for storing and retrieving files, used by applications for persistent, high capacity storage as well as communication between running processes.

The definition of files and how they are represented depends on the file system. Some file systems offer record- or object oriented I/O, where the filesystem is aware, to some extent, of a file's internal structure, like the MVS file system. Most file systems perceive a file as an unstructured stream of bytes, without interpreting the contents in any way. Applications understand the meaning and internal layout of the files and can access and modify them according to defined file formats.

A file system's *allocator* services requests for space to store files. It needs to keep track of free space and find appropriate free blocks in which files are then stored. A *file link* component is responsible for tracking the storage areas associated with files and their metadata and handles reading from and writing to these areas.

The performance demands placed on a file system are efficient use of storage space, and high throughput on storage and retrieval. In most cases, a file system also needs substantial fault tolerance; It must be possible to locate file data, even after a system failure, or in the face of corruption in the underlying storage medium.

1.3 Fragmentation

There are three types of fragmentation, internal, external and data fragmentation.

Internal fragmentation occurs when a file does not fill up the space allocated to it. Classical file systems allocate an exact multiple of the block size to each file. Every file has a trailing block that is allocated but only partially used. On average, only half of this block is used, and the other half is wasted. For 512 byte blocks, this

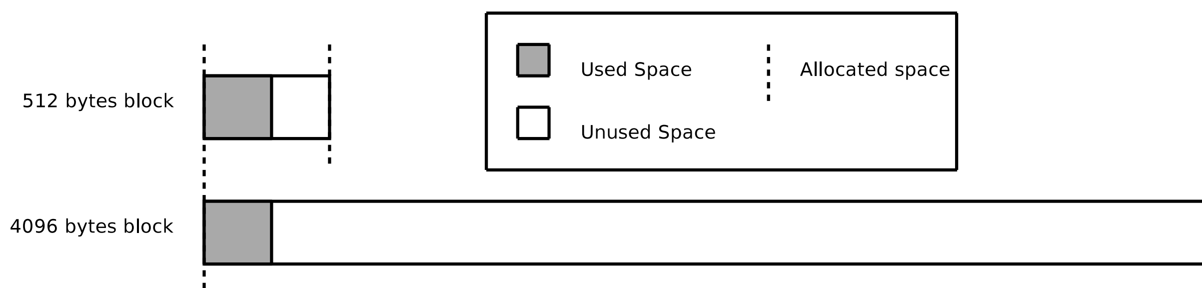


Figure 1.1: Internal fragmentation. Large blocks potentially waste more space.

means a waste of 256 bytes per file. With 4096 byte blocks however, on average 2048 bytes are wasted per file. This is an eightfold increase in the wasted space per file. The total wasted space on a file system depends on the file size distribution, but if a file system contains mostly small files, this can be significant. A study done by Tanenbaum et al. [2] shows that most files are relatively small. This is especially true for webservers, where as much as 70% of the files are smaller than 4096 bytes, and the median file size is 1180 bytes. This would cause significant internal fragmentation on file systems with 4096 byte blocks.

External fragmentation is when free space becomes divided into many small pieces over time. It happens when applications allocate and deallocate storage regions of varying sizes, and the free space gets split into many small pieces over time. [3]

Data fragmentation occurs when a file system has free space available, but no regions of contiguous blocks are large enough to satisfy a request. To use this fragmented free space, an allocation request has to be split into multiple pieces small enough to fit into the gaps of free space. Fragmenting a file degrades performance when reading and writing the file sequentially. Extra space is also needed, because of the extra bookkeeping required to keep track of the different fragments. Additionally, data fragmentation has been shown to be self reinforcing. [3] The reason for this is that when a fragmented file is deleted, the blocks surrounding the file fragments are not likely to be freed up as well, preventing a subsequent contiguous allocation.

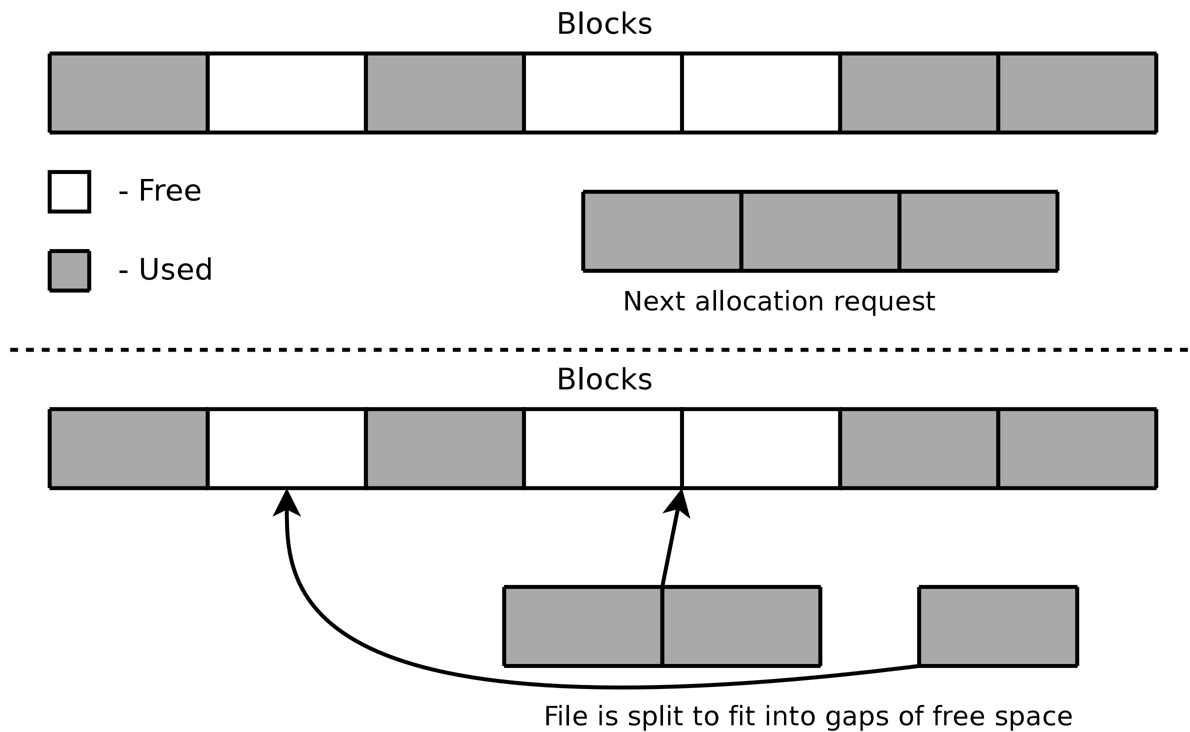


Figure 1.2: External Fragmentation and how it leads to data fragmentation

1.4 Extents

Most traditional filesystems use individual blocks as their unit of allocation. Free space is tracked by individual block numbers and allocated space is tracked by block lists associated with each file. Blocks are also allocated one at a time, potentially leading to data fragmentation.

Another way of keeping track of blocks is to use *extents*. An extent is a contiguous sequence of blocks, defined by a starting block and a range of consecutive blocks. This cuts down on the space needed for metadata and the number of writes needed for bookkeeping. [4] Using extents, it is possible to allocate blocks in groups, forcing sequential allocation. A large extent can be allocated in one operation. Subsequent writes can cover a large number of blocks without the need for further allocation and metadata updates.

Some file systems offer optional support for extents, and a few, like VxFS [5] are us-

ing extents exclusively. The block based filesystems usually have blocks whose size are a multiple of the hard drive's sector size. Extents provide the ability to further group these together in a more flexible fashion, but traditionally the smallest subdivision is still one block.

1.5 Blockless Filesystem

Block devices are transitioning to 4096 byte blocks and it has been observed that files are still relatively small, especially on e.g. web servers [2], leading to significantly increased internal fragmentation. Also, if files are significantly smaller than the block size, read performance suffers, because large amounts of unused space are read along with the files themselves.

This thesis explores the design and implementation of a file system supporting arbitrarily sized allocations, called AAFS. AAFS divides the storage space into units, called grains to distinguish then from traditional blocks. The grain size is set when the file system is created, and can be set to any size, including 1 byte. The grains are addressed in the same fashion as LBA, with the first grain having address 0, the second grain having address 1, and so on. Extents are used to group grains together into manageable units.

Using a small subdivision, internal fragmentation is reduced, and it is possible to pack multiple files into one hard disk block, as illustrated by figure 1.3, potentially improving read speeds for small files. The grain size can be chosen by the user to improve performance for a certain file size distribution.

1.6 Overview

This study looks into the viability of a blockless filesystem, using an existing allocator. To limit the scope of the thesis, the file system is implemented using FUSE, and lacks support for growing files, directory hierachies and crash recovery. However, even

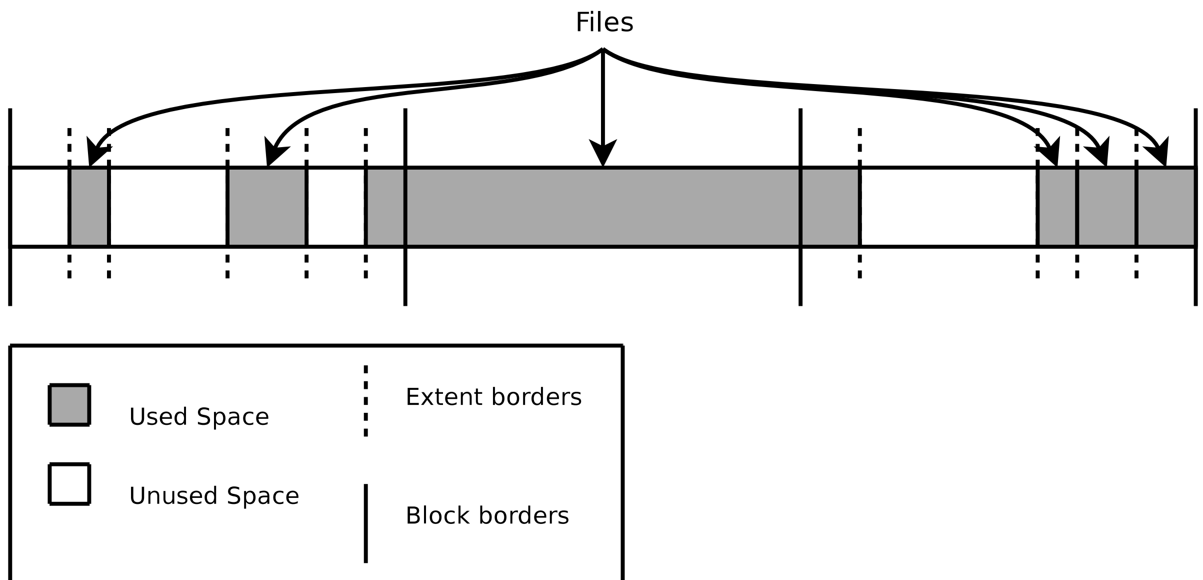


Figure 1.3: Multiple extents can be packed into one block, or an extent can span multiple blocks

without these features, the file system is still useful for a few scenarios, where file sizes are known in advance and the size distribution is favorable for the allocator. Examples of this are proxy- and mail servers. The rest of the thesis describes design and implementation, as well as testing methodology and results.

2 Design

2.1 Related Work

Some file systems treat file *tails* differently from regular file data to address the problem of internal fragmentation. A tail is a file that is smaller than the file system block size, or the trailing portion of a file that causes internal fragmentation. Some file systems address the problem of internal fragmentation by treating file tails differently from other file data. This is called *block suballocation* or *tail packing*. UFS [6] subdivides some blocks called “fragment blocks” and stores tails from multiple files packed together in such blocks.ReiserFS stores file tails packed together with file metadata.

AAFS uses an existing allocator based on the Quick Fit allocation algorithm, originally found in [7]. The algorithm uses segregated lists to store references to free extents. The extents are assigned to different lists according to their size. Quickfit uses two types of lists - *quick lists*(also called exact lists) and *misc lists*. The quick lists each store extents of the same size, whereas the misc lists store a range of sizes. This design is meant to cope with a skewed file size distribution, where the most frequently used sizes get stored in quick lists. Less frequent sizes get stored in misc lists to save space (size overhead per list).

Quick Fit was designed for main memory allocation. Iyengar et. al. designed a similar allocation algorithm called Persistent Multiple Free List Fit Allocation, or PMFLF, for use in file systems. PMFLF was intended for web servers requiring fast response times, and improved file system performance by requiring only a single disk seek for most allocations and deallocations.

The allocator around which this filesystem is built comes from the master thesis done by Stanislav Sokolov [8]. In it, he creates an allocator simulator, implementing a version of the PMFLF and Quick Fit allocators. Sokolov replaced the singly linked quick lists and misc list of Quick Fit with skip lists, and removed Quick Fit's special treatment of the free storage at the end of the address space, without adversely impacting performance, thus simplifying the implementation (for closer details, see [8]. The allocator is optimized for space efficiency and small allocations, using extents for space allocation. His approach attempts to minimize internal fragmentation as well as reduce bookkeeping overhead. Sokolov's allocator simulation consists of a core allocation algorithm, instrumented by functions for gathering statistics about the allocator's performance. Extensive testing was performed to evaluate the allocator's performance, with encouraging results. The allocator had a small memory footprint and computational overhead. It had very low external fragmentation and data fragmentation, performing better than PMFLF in the tests that were carried out.

The grain size in Sokolov's allocator is set to 512 bytes, to reflect the input/output operations of the underlying hardware. The allocator uses the skip list datastructure to store extent data, since skip lists offer simplicity and good performance. Details about the datastructure are covered in 2.3.1.

2.2 File System Overview

Figure 2.1 is meant to give a rough idea of the file system layout, and is simplified to show how the data structures are connected. As with all file systems, a superblock is placed at the very beginning of the filesystem. This data structure is the first to be read when a file system is mounted, as it contains vital information about the file system, which tells the file system driver how to read the file system. The superblock also contains general info like the size of the file system.

Some file systems store a free space map or the inodes in a designated space, the size and/or location of which can be calculated as a function of the file system's size.

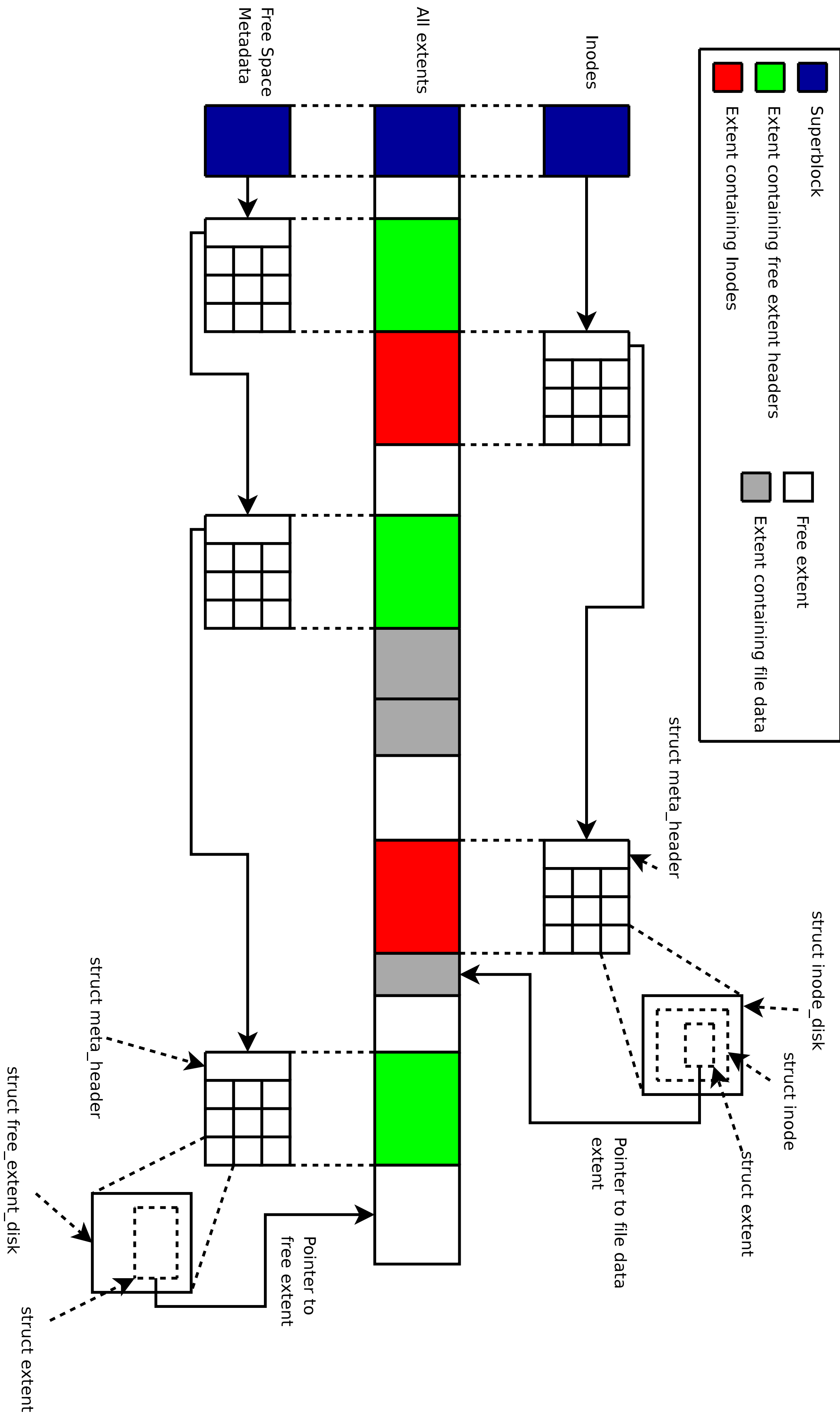


Figure 2.1: Layout of data structures on disk

Other file systems may dynamically allocate and deallocate space for this metadata, and must have a reference to the metadata location in the superblock. AAFS falls into the latter category.

A file system needs a set of datastructures to keep track of which regions are occupied, which regions are free, and which file an occupied region belongs to. AAFS organizes this information in skip lists, described in section 2.3.1. One skip list contains all the headers of the free extents, sorted by sizes, another contains all inodes, sorted by file names.

The extent headers for the free extents contain the size and location of the extent body. An inode contains all the information the file system holds about a specific file. This includes the id of the file's owner and timestamps saying when the file was created and modified, and an extent header which references the extent containing the file's data. Unlike most file systems, which store file names as entries in special directory structures with a reference to an associated inode, AAFS stores file names in the inodes themselves.

Both these lists are kept in RAM when the file system is mounted, and all modifications to the file system are committed to these skip lists. When the file system is unmounted, both skip lists need be to serialized and stored in the file system until it is mounted again. The inodes and free extent headers are wrapped in structures containing the necessary information to reconstruct the skip list, and stored in locations referenced by the superblock. Each list is stored in a linked list of extents, and the elements within each extent are stored contiguously, like an array.

The free extents reference points to a skip list, marshalled to disk, containing all the headers [1] for the free extents. The Quick Fit Allocator uses these headers distributed into different lists sorted by their size, but it is up to the mounting code to read the extents and sort them into these lists.

The inodes reference points to another marshalled skip list, containing all the inodes. This list is sorted alphabetically by the filenames.

[1] The term is used loosely, as they are not physically located with the extent body.

As mentioned in section 3.1, the file system does not support growing and fragmented files, but it needs to support growing metadata. The allocation and deallocation of metadata space are handled by ordinary file requests to the allocator. To simplify the code needed to read the metadata at mount time, it is allocated in extents of a fixed size as the used extents fill up. Each of these metadata extents have a header containing the extent's size and a reference to a next extent, forming a linked list.

The following section looks more closely at the different data structures and how they are stored on the disk.

2.3 Data Structures

All addresses and sizes are counted in grains, unless stated otherwise.

All the listed structs include the `__attribute__ ((__packed__))` compiler directive to prevent padding when the structs are written to disk.

2.3.1 Skip list

The skip list datastructure is central to the file system implementation, and is used to store and organise most internal data structures, both on disk and in memory at run-time. The skip list is stored in a marshalled state on disk when the file system is in an unmounted state.

As seen in figure 2.2, the skip list works like a linked list with added layers of pointers for skipping forward among the nodes when searching the data structure. As stated in [9],

> "Skip list algorithms are very easy to implement, extend and modify. Skip lists are about as fast as highly optimized balanced tree algorithms and are substantially faster than casually implemented balanced tree algorithms."

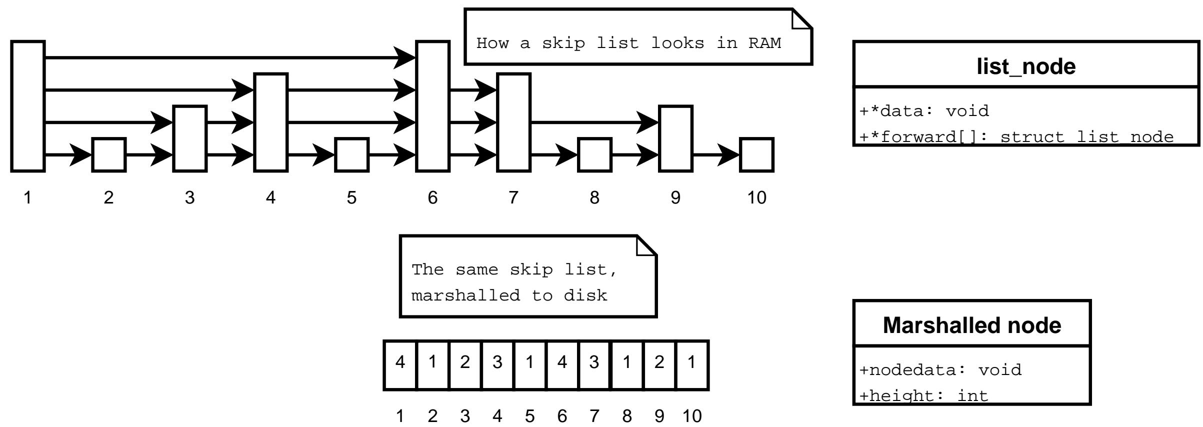


Figure 2.2: Skip list, in normal and marshalled form. Node data not included.

```
struct list_node {
    void                *data;
    height_t            height;
    struct list_node    *forward[1];
};
```

data

Pointer to data item contained by skip list node.

height

Height of the node in the skip list.

forward

Array of skip pointers, one for each level the node is in.

The `*forward[1]` member is an `ISO C90` version of a variable-length array, and is used to only allocate the array length needed.

```
struct list_node *node= malloc(sizeof(struct list_node) +
    nodeheight * sizeof(struct list_node *));
```

The code allocates room for the struct plus an array of length `nodeheight`.

For the purposes of this file system, the skip list is only searched and modified after it is loaded into RAM and unmarshalled at mount time. To ensure persistence between two mounts, the skip list needs to be marshalled and saved to disk. The marshalling entails creating nodes of a uniform size by discarding all the layers except the bottom one and only storing the node's height. These new nodes are then written sequentially to disk. Writing the elements sequentially means their ordering is not disturbed, and the unmarshalling code can initialize a new skip list in RAM and simply append the elements to the list.

Algorithm 2.3 details how the skip list is reconstructed when read into RAM. First, an array of pointers, `previtem[]`, is initialized to point at the list's header, which is a sentinel node. During unmarshalling, `previtem[i]` points to the previous node having a height of at least `i`.

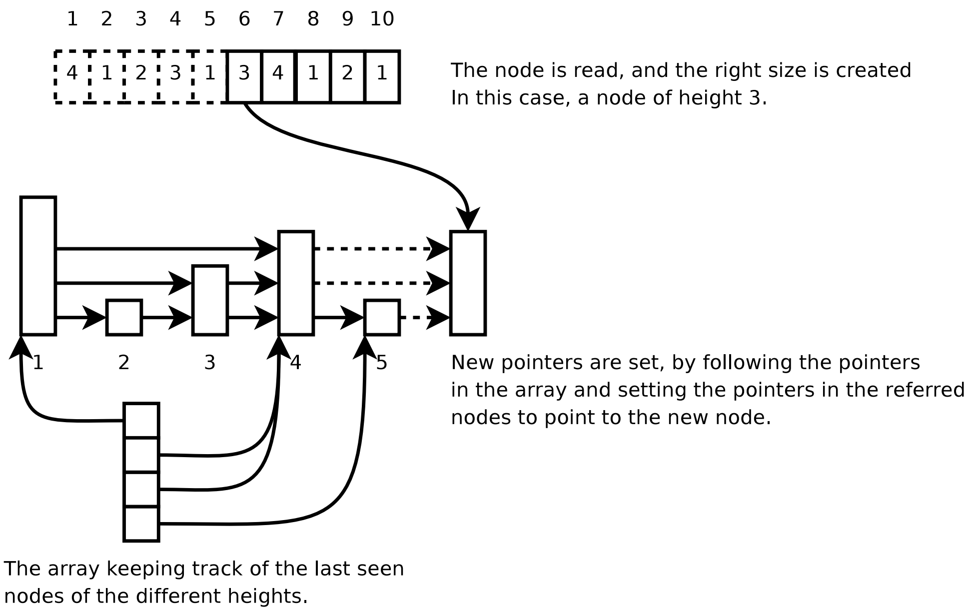
The main loop of the algorithm then reads in the items sequentially, and appends them to the end of the list (see Figure 2.4). The `previtem` array is used to locate the previous node at each level the new item exists in. The previous node's next pointer is then pointed at the new node. Lastly, the `previtem` array is updated to point at the new node. When the main loop is finished, the previtem array is traversed. For each level, the last seen node having a height of at least that level is set to point at the list tail. This way of appending the items to the skip list ensures a linear $O(n)$ unmarshalling time, compared to normal skip list item insertion, which would have a much higher running time of $O(n \log n)$.

```
for(i= 1; i<= list_height; i++)
    previtem[i]= list_header

while(more_nodes) {
    node= read_node();
    for(i= 1; i<= node_height; i++) {
        previtem[i].next[i]= node;
        previtem[i]= node;
    }
}

for(i= 1; i<= list_height; i++)
    previtem[i].next[i] = list_tail;
```

Figure 2.3: Unmarshalling a skip list



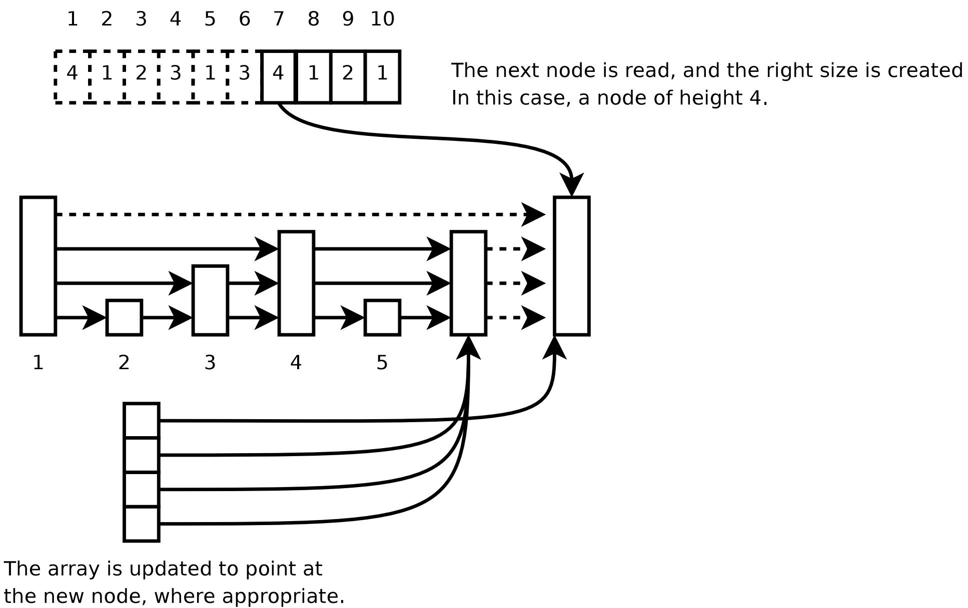


Figure 2.4: Skip list, in the process of being unmarshalled. Node data and pointers to the list tail not included.

2.3.2 Superblock

The superblock is placed at the very beginning of the partition and contains general information about the file system and the data necessary to mount it.

```
struct __attribute__ ((__packed__)) superblock {
    uint64_t    inodes_loc;
    uint64_t    inodes_num;
    uint64_t    inodes_space;
    uint64_t    freext_loc;
    uint64_t    freext_num;
    uint64_t    freext_space;
    uint32_t    AWP;
    uint8_t     dirty;
    uint64_t    size_in_grains;
    uint64_t    grains_free;
};
```

inodes_loc

Address of the first extent containing inodes.

inodes_num

Number of inodes (files), present in the file system. Since we do not support hard links, there is a 1 to 1 correspondence between files and inodes.

inodes_space

Amount of allocated space for storing inodes.

freext_loc

Address of the first extent containing free space metadata.

freext_num

The number of free extents.

freext_space

Allocated space for storing free extents.

AWP

The AWP, or Acceptable Wastage Parameter, indicates how many grains may be wasted to satisfy an allocation without splitting an oversized extent.

dirty

Checked on mount. Indicates wether or not the file system had a clean unmount. If not, cleanup measures are necessary. Functionality for crash recovery is not implemented, so this flag is currently not in use.

size_in_grains

The size of the file system.

grains_free

Amount of free space on the file system.

The `inodes_loc` and `freext_loc` members refer to the first extent in the linked list of extents containing inodes and free extent headers, respectively. The `inodes_space` and `freext_space` are the sums of the sizes of extents in each linked list. They are used to determine if the file system needs to allocate more space for metadata. The numbers are stored here so the file system will not have to read all the extent headers each time it checks to see if more space is needed.

The `inodes_num` and `freext_num` members refer to the number of structs in their respective list.

The AWP allows the allocator to widen the search parameters to satisfy a request. For example, if the AWP is set to 4, an allocation request of 32 grains can be satisfied by any extent of size 32 to 36. If no extents in this size range is found, a larger extent will be split to obtain an extent of size 32. Sokolov's allocator stored extent headers with the extent bodies, and splitting extents was an expensive operation, requiring the on-disk headers to be updated. The AWP would help the allocator avoid

splitting operations, improving allocation speed at the cost of space efficiency. Since AAFS stores extent headers separately from extent bodies, this parameter is not as important. The AWP is set when the file system is created.

2.3.3 Extents

The extent is the basic allocation unit in AAFS. It's size is a multiple of the grain size, which can be set to any size (as opposed to a block size, which is a multiple of the storage medium's sector size, usually 4 kilobytes).

The allocator keeps a series of skip lists, sorted by size, holding all the `extent` structs referring to free extents. Each inode also contains this struct, referring to the extent associated with a file.

The `free_extent_disk` struct is used to wrap the `extent` struct when the free extent lists are written to disk. The lists are merged to a single list. The list is serialized, and each `extent` struct is wrapped in a `free_extent_disk` struct. These structs are written sequentially to disk inside a metadata extent. When written to disk, the extent headers are sorted by size. The skip lists used by the allocator have a mixed ordering; the misc list is sorted by size, while the quick lists are sorted by address.

```
struct __attribute__ ((__packed__)) extent {
    uint64_t          address;
    uint64_t          size;
};

struct __attribute__ ((__packed__)) free_extent_disk {
    struct extent     extent;
    height_t          height;
};
```

address

Refers to the location of the extent body.

size

The size of the extent.

height

The height of the skip list node, used for marshalling and unmarshalling.

2.3.4 Metadata Extent Headers

As mentioned in section 2.2, AAFS keeps metadata in skip lists that are marshalled to disk, stored in linked lists of extents. The following struct is placed in the beginning of each such extent.

```
struct __attribute__ ((__packed__)) meta_header {
    uint64_t        next_ptr;
    uint64_t        size;
};
```

next_ptr

Address of the next extent in the linked list. Zero if this is the last extent in the list.

size

The size of the extent. The `meta_header` struct is included in this size.

The necessity of the `size` member is not obvious, since all the metadata extents are supposed to be the same size. However, because of the Acceptable Wastage Parameter, the extent sizes may vary slightly. For simplicity, the extra few grains added by the AWP are ignored.

AAFS keeps track of needed space for metadata by comparing how much space the inodes or free extent headers need and comparing that to how much space has been allocated for them. When more space is needed, a single extent the same size

as the other metadata extents, is allocated and appended to the end of the linked list. To prevent allocation and deallocation of such an extent from happening too often in cases where the skip list is growing and shrinking rapidly, two and a half empty extents are necessary before the last one is deallocated.

When a skip list is marshalled to disk, the elements are written in the order they are kept in the skip list, filling each metadata extent sequentially. When one extent is full, the next extent is written to and so on. Because of the way deallocation of these extents is handled, the last extent in the linked list may or may not be empty.

There is no explicit information about how many entries are in each extent. Instead, this has to be calculated using code similar to figure 2.5, which is used when a skip list is unmarshalled. The `entry_size` argument is used to calculate how many entries can fit in the extent. When the first extent in the list is read, the `entries_left` argument will be equal to the number of elements in the entire list. If the extent in question contains all the entries, the unmarshalling is done after that extent. If instead `entries_left` is bigger than `ext_contains`, the extent contains `ext_contains` entries. This number is then subtracted from the number of entries left to read before the next extent is processed.

```
entriesInExtent(int extent_size, int entries_left, int entry_size)
{
    int ext_contains = (extent_size * grain_size -
                        sizeof(struct meta_header))
                       / entry_size;
    if(entries_left > ext_contains) {
        return ext_room;
    } else {
        return entries_left;
    }
}
```

Figure 2.5: Calculating the number of metada entries in an extent.

2.3.5 Inodes

All the filesystem's inodes are kept in a single skip list, held in RAM while the file system is mounted. This list is sorted alphabetically by filename, both in the in-RAM skip list and when in a marshalled state on disk.

```
struct __attribute__ ((__packed__)) inode {
   struct extent     extent;
   uint8_t           name[MAX_FILENAME];
   uint64_t          file_fragments;
   uint64_t          bytes_size;
   uint64_t          ctime;
   uint64_t          atime;
   uint64_t          mtime;
   uint16_t          uid;
   uint16_t          gid;
```

```
    uint16_t          mode;
    uint32_t          nlink;
};

struct __attribute__ ((__packed__)) inode_disk {
    struct            inode inode;
    height_t          height;
};
```

extent

The first[2] extent associated with the file.

name

The file's name.

file_fragments

Pointer to the first extent a file has grown into, if any. This member is zero if the file consists of only one extent. This member is currently not in use and is discussed further in section 2.4.1

bytes_size

The file's size, in bytes. Note that the file's size on disk is the sum of the sizes of the extents it is composed of.

ctime

Time of last status change.

atime

Time of last access.

mtime

Time of last file modification.

[2]And currently the only, since file growth is not implemented.

uid

User ID of file owner.

gid

Group ID of file owner.

mode

The file's permission mask.

nlink

The number of hard links to this file. This member is not in use in this implementation, as links are not supported.

height

The inode's height in the skip list. The struct which this member is a part of is used in the skip list containing inodes, when it is marshalled to disk.

Since AAFS has no directory support, it is natural to store the file names inside each inode. The name string has a static size to ensure uniform size, making the code for marshalling and unmarshalling the inode list simpler. Each inode has a filename of a static length, for simplicity. `MAX_FILENAME` is defined as 16. The filename is a simple C-style nullterminated string.

2.4 Design Alternatives & Possible Extensions

This section details parts of the design that are either unnecessary to test the file system's performance, outside the scope of a master thesis, or ideas for future extensions.

2.4.1 Fragmented Files

The support for growing and fragmented files is left for future extension of the file system. However, this thesis would not be complete without a suggestion for how it could be done.

The most direct approach to growing a file is to grow the extent containing the file, which is possible only if the following extent is free. This approach is simple in appearance, but requires the filesystem to locate the neighbouring extent by its address. Currently, the free extents are sorted by size, so the free extent lists would have to be searched exhaustively. Introducing an allocation status flag inside the extents themselves allows them to be easily checked for availability, but the exhaustive search would still be necessary to locate the extent header in the in-RAM list to perform the allocation operation. One way of avoiding the exhaustive search is to simply introduce another data structure that keeps references to free extents ordered by their address, or to make the free extent list multidimensional, but it would be relatively big modification of the file system.

A more flexible (but not mutually exclusive) way of supporting file growth is to simply allocate another extent and associate it with the file. At the moment, the inode can only hold a reference to a single extent, and a way of associating multiple extents with an inode would be necessary. In the suggested solution, we use a single fragment table to store the extra extent headers associated with a file. The table is stored in a linked list of extents in the same fashion as the inode and free extent lists.

Inside the linked list extents, the file table entries are stored contiguously, like an array of `fragment_node` structs, each containing a pointer to the relevant file extent and the index of the file's next entry in the table. Each is then organized as a linked list of pointers to extents, as illustrated by figure 2.6. At the end of each file's chain, a `next_ptr` set to zero denotes no more extents for that particular file.

```
struct __attribute__ ((__packed__)) fragment_node {
   struct extent    extent;
   uint64_t         next_ptr;
};
```

extent

Extent header, referencing an associated extent.

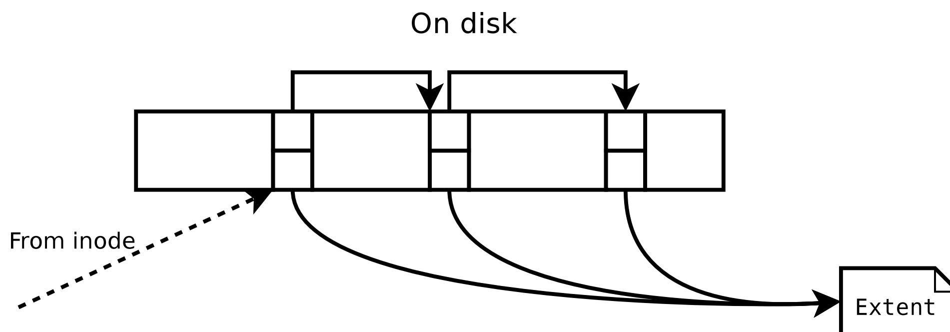


Figure 2.6: File Fragment Table.

next_ptr

Index of another `fragment_node` struct in the array.

To cope with deleted files creating holes of unused entries in the array, the array would be supported by a simple list of available indexes. When a new table entry is requested, the supporting list is checked for available indexes. Only when this list is empty would entries be added to the tail of the file fragment array. The supporting list should be relatively small, except in cases of mass file deletion.

This data structure is simple to implement and is easy to keep partially in memory, and swapped to disk on demand. Each extent contains a set number of entries, which are easily stored in an array in RAM, and the list will only shrink or grow from the tail. Available slots in the list are represented by an extent pointer pointing to zero.

Since AAFS allocates extents to exactly fit requests, it is a relatively simple extension to support deletion and injection of data in the middle of a file, at the cost of fragmenting the file. Figure 2.7 shows how a file occupying a single extent could have an area deleted. The file extent is split into three parts, and the extent containing the deleted area is returned to the free list. The extent after the deleted area gets an entry in the file fragment table and the inode `bytes_size` field is updated to reflect the new file size. After this operation, the inode's file size is shrunk in proportion to the deleted area.

These operations are useful for tasks such as video editing, where large parts of files are frequently deleted or inserted.

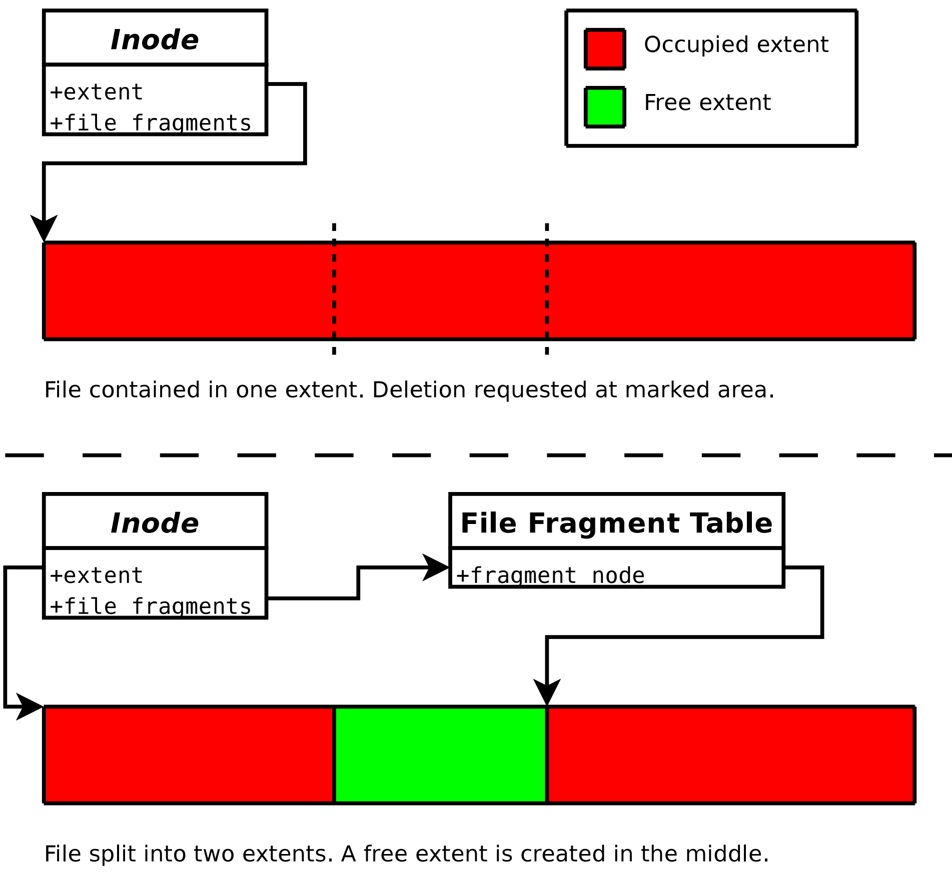


Figure 2.7: Deleting an area in the middle of a file

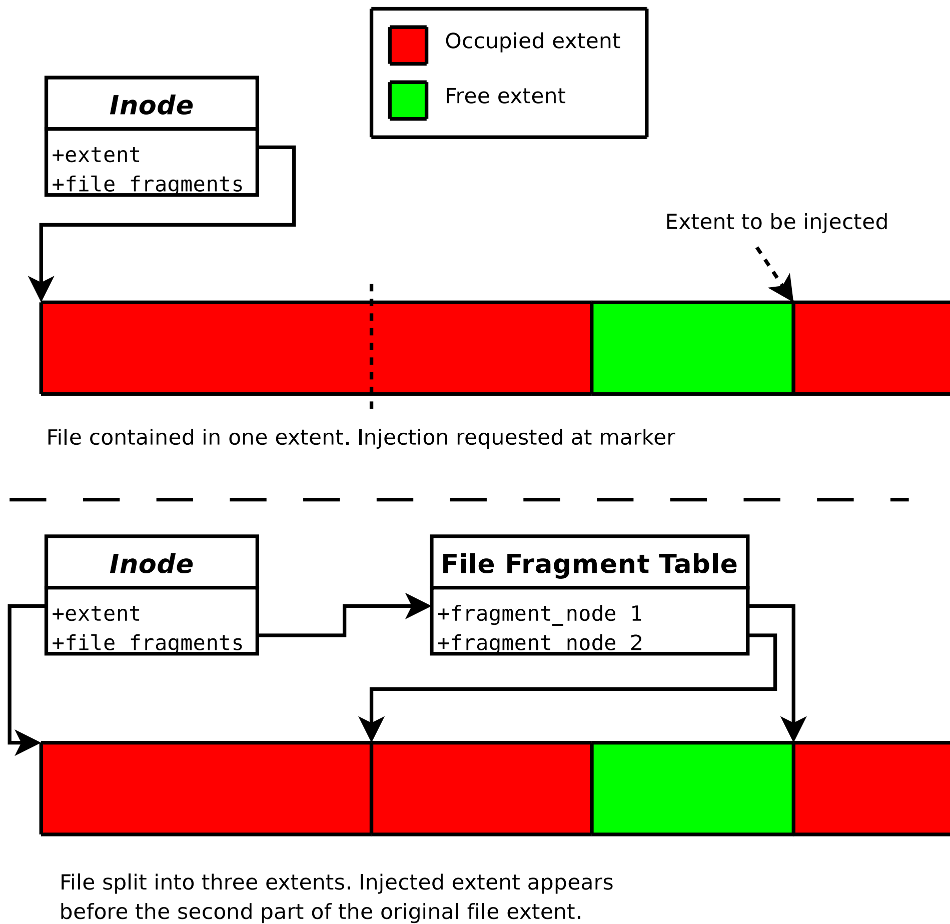


Figure 2.8: Injecting an extent into the middle of a file

Figure 2.8 illustrates injection. The original file extent is split into two parts. The injected extent and the second part of the original extent get entered into the fragment table, with the injected part appearing first.

Because of how the Acceptable Wastage Parameter can cause the allocator to return slightly oversized extents, support for this kind of data injection and removal requires an additional `payload` field in the extent headers, so the garbage at the last few bytes of an extent is not included as part of the file. Another option is to introduce a semantic into the allocator, allowing the AWP to be ignored, in order to force the allocation of an exact size.

A sparse file is a file that is mostly empty, and the file system can store them more

efficiently by only storing representations of the empty spaces instead of reserving the actual empty space in the file. When reading a sparse file, the file system returns zero bytes when the empty spaces are accessed. Sparse files can be supported by introducing a `file_offset` parameter into the extent header. This would allow each extent in a file to be placed at virtual offsets from the file's beginning. The injection and removal operations can be useful with sparse files, allowing the empty areas to be filled or the parts of a file containing data to be deleted.

2.4.2 Directory Hierarchy, Ordering

This section looks at a way to adapt the global inode skip list to support a directory hierarchy. In the suggested solution, the inode would need three additional fields:

type
A field to identify the inode as a regular file, directory or a symbolic link.

parent
A reference to the inode representing the parent folder.

child
Only used by directory inodes. this pointer points at the first file in the directory.

Figure 2.9 shows what the directory hierarchy would look like. All directories are here present as another inode, containing a single reference to the first inode in that directory. All inodes also have a pointer to its parent, with the single exception of the root inode.

To list the contents of a directory, one must first find the directory inode. From there, the `parent` pointer is followed to find the first inode in the directory. A skip list is also a linked list at the bottom layer, and this layer is now followed sequentially, checking and listing the inodes until an inode with a `parent` pointer referring to a different directory inode; That inode is the first in another directory.

The nodes are sorted in a special order. When two inodes are compared, the inodes' entire paths are assembled before comparing the strings.

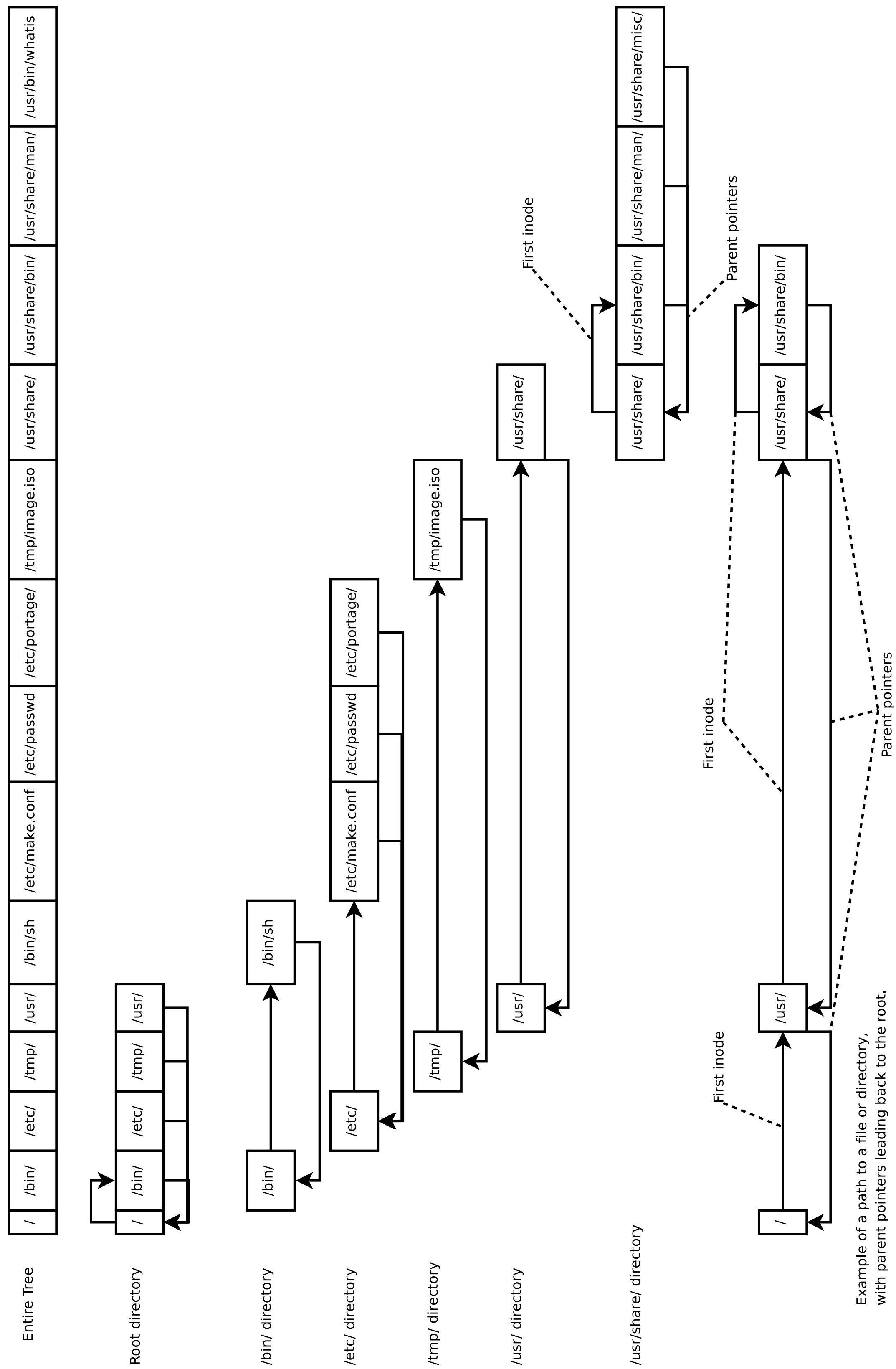


Figure 2.9: Ordering of file inodes in a directory hierarcy

The inodes are sorted after two criteria:

1. `parent` pointers are used to recursively determine the directory nesting, and the inode with the shallowest nesting comes before the other.

2. If the level of nesting is equal, the parent inodes along the nesting path are sorted lexicographically, one level at a time.

The pseudocode in figure 2.10 shows how this is done recursively. If only `node2` has a parent, the function returns 1. If only `node1` has a parent, the function returns -1. These tests can return because they know one node has a deeper nesting than the other. If both nodes have a parent, both parents are checked recursively. If the parent paths turn out to be identical, the current level is compared lexicographically and the result is returned. If none of the nodes have parents, they are both of equal depth, and the levels have to be distinguished by their names.

With this ordering, the skip list can still be used as a global search tree. Finding a file inside a directory is a matter of finding the first inode in the directory, either from the parent directory or through a global search, and then using the skip list structure for further searching.

The data structure performs a bit slower than a regular skip list, due to the extra work required for assembling and comparing strings. File names are also more complex, requiring a search through all parent directories to find a file's absolute path.

2.4.3 Reducing Memory Footprint

AAFS keeps all metadata in RAM when the file system is mounted. For relatively small file systems, this is not a problem, but on partitions with several terabytes of data, the memory requirements could become unmanageable. In such situations, it is necessary to keep only parts of the metadata skip lists in RAM, and retrieve parts of the skip list on demand.

To achieve this, the unmarshalling function reads all skip list nodes as usual during mounting, but only unmarshals the ones above a certain height. The nodes below

```
int nodecompare(struct inode1, struct inode2) {
    //determine depth, recursively
    int res;
    if(inode1.parent != NULL
        && inode2.parent == NULL) //inode1 is deeper
        return 1;
    if(inode1.parent == NULL
        && inode2.parent != NULL) //inode2 is deeper
        return -1;
    if(inode1.parent != NULL
        && inode2.parent != NULL) { //both have parents

        res = nodecompare(inode1.parent, inode2.parent);
        if(res == 0)
            return strcmp(inode1.name, inode2.name);
        else
            return res;
    }
    //none have parents
    return strcmp(node1.name, node2.name);
}
```

Figure 2.10: Compare function for directory hierarchy

this height are represented by a placeholder node containing the address of the extent where a group of nodes are located, and boundary tags specifying which nodes are contained in this extent. If a search encounters one of these placeholder nodes, the represented extent is unmarshalled and inserted into the list at the placeholder's position. Each area of the skip list belongs to a specific metadata extent, and when a part of the skip list needs to be evicted from RAM, nodes belonging to one extent are evicted as a whole, and written to disk. If the area belonging to an extent grows so the extent can not contain it, a new extent is allocated and the area is split in half, one half belonging to the old extent, the other to the new. If all the inodes belonging to an extent are removed from the list, the extent is returned to the free space pool.

2.4.4 Predictive Preallocation

If a file's size is not known on creation, it is necessary to grow the file as it is written to. A naive allocator might allocate a new extent or block each time the file grows, possibly causing data fragmentation. Some file systems handle this problem by holding file data in memory buffers and delaying the actual allocation operation until the application is done writing and the final size is known, or when the buffers must be flushed to disk to make room for other things. Some files may be too large to fit in such allocation buffers, causing file fragmentation as the file is written to disk in many operations.

Another possible method of preventing this kind of file fragmentation is to try to predict the file's final size using heuristics based on file size statistics. A similar idea was explored in [10], where Markov Models were used for predictive allocation of RAM, giving "performance gains in some classes of application". With e.g. a cumulative distribution function, it is possible to predict a minimum file size with a desired certainty. However, a certainty of e.g. 90% would also mean one is very likely to overshoot. Also, a global cumulative distribution function for all files would not be particularly accurate, since e.g. plain text files are typically a lot smaller than video files. Text files are also more likely to grow after their initial write, so wasting

some space by allocating a larger extent may be useful to give files some leeway for growth. Finding the most efficient certainty is a subject that bears closer study.

To make predictions more accurate, some way of differentiating files, or at least classes of files is be necessary. A simple criterion for differentiating file types is the filename extension, and a file system could store the size distribution of all file types it comes across. However, a file system storing many different file types, having mostly few files of each type can end up storing a lot of information, with some file types having too small datasets to give useful predictions. One way to combat this is to give a file system user the option to intervene, using e.g. structured files stored in a special location, to tell the file system which files to keep statistical data on. This also makes it possible for a user, who might have detailed knowledge about the files being stored, to give data for more accurate predictions.

Figure 2.11 is an example of what a structured file used for size predictions could look like. The first `FILE` entry lists an mp3 file and some reasonable characteristics.

It is an audio file, which means that unless the user does a lot of audio editing or creates music, the file is likely to be a finished product, and growth is unlikely. This is a characteristic an mp3 file is likely to share with other audio files, and it has been put in a separate `CATEGORY` entry.

The size of a typical mp3 file may drift over time as general storage capacity grows and higher bitrate files become commonplace. To prevent stale data from influencing new predictions, some form of decay function is necessary. An easy way to achieve this is to store the statistical data as a ringbuffer. The `last N seen sizes` entry refers to such a buffer, and the listed `buffer cursor` points to the next entry to be overwritten in a particular ringbuffer.

The ringbuffer offers another trade-off in that a larger buffer uses more space, but can store more statistical data, possibly giving more precise predictions. In the case of mp3 files, however, a lot of music is engineered to fit a certain timespan, and file sizes can reasonably be expected to be relatively uniform. Consequently, the `buffer size` can be set quite low, since a smaller dataset is necessary for reasonable predictions.

```
--FILE
type: mp3
category: media/audio
buffer size: 10 entries
last N seen sizes: <list of sizes>
buffer cursor: 53

--FILE
type: tex
category: text/plain
buffer size: 100
last N seen sizes: <list of sizes>
buffer cursor: 34

--CATEGORY
category: media/audio
likelyhood of growth: 0.1
growth factor: 1.2

--CATEGORY
category: text/plain
likelyhood of growth: 0.8
growth factor: 2
```

Figure 2.11: Data format for file size prediction

Also, some file types may have sizes that drift over time, potentially making a shorter buffer more suited to predict their sizes.

The second `FILE` entry lists a plain text tex file. The `likelyhood of growth` and `growth factor` are set high, since text documents are likely to start small and grow several times it's own size as it's author develops it. Also, the final size is harder to predict than an mp3 file, leading to a larger buffer of file sizes.

A system like this could also dynamically tune the Acceptable Wastage Parameter for better allocator performance in terms of fragmentation. This approach will not necessarily perform better (or worse) than delayed allocation, but it is an alternative that can reduce RAM usage since less buffering is necessary, and may perform better in some cases, especially if a user has detailed knowledge about the size distributions of his files.

2.4.5 Crash Recovery & Fault tolerance

AAFS currently has no capabilities for detecting or correcting errors. Error protection is not necessarily desirable in all usage scenarios, since it has some additional overhead. For example, a proxy server might be better served with the increased speed from having no protection, as data loss from e.g. a crash or power outage may not be a problem. Depending on the downtime, some of the data might be stale, and the rest of it can be downloaded again.

In most cases, however, data protection is of vital importance. This section presents some ways of adding protection to AAFS.

A popular way of ensuring data consistency is through journalling. Any changes to metadata, and in some cases file data, are written to an on-disk journal before they are properly commited. If the system should fail, the journal can guarantee that a write operation is either performed properly, or not at all.

Minor corruptions can destroy a file system's metadata, leaving the file system inoperable. A simple and reasonable safeguard against this is to keep backups of metadata at different places on the disk. Especially the superblock should be backed up

in this way. Also, metadata extents should contain checksums in order to detect corruption. Having backups of a metadata item (superblock, inode list, free space list) containing checksums can enable the file system to detect an error and retrieve a good copy from one of the backups, provided the backup passes the integrity check.

3 Implementation

3.1 Usage Restrictions

The goal of this thesis is to implement a blockless file system and to measure it's external fragmentation. Since creating a full-featured file system is outside the scope of a master thesis, AAFS has a few limitations to limit its complexity. As such, features have been removed on the basis of not significantly impacting performance and not being strictly necessary for the operation of a basic file system.

To simplify the handling of inodes, all the inodes have the same size. Filenames have a maximum length of 16 characters, stored in a static array within each inode. As a consequence, the file system does not support directory hierarchies. Hard and soft links are also not supported.

A file's owner, group id and permissions mask are all recorded and stored in the inodes, but this particular implementation does not enforce permissions checks.

AAFS has no data structures in place to handle multiple extents associated with each file. Instead, each inode can hold a reference to a single extent. Other file systems have data structures for each inode to keep track of multiple blocks or extents, and file growth is handled seamlessly. With AAFS, the user is required to explicitly allocate space for a file using `truncate()` or `ftruncate()`. Also, once a file's size has been set, it is not possible to change.

The file system lacks features for detecting and correcting errors in the storage medium, and cannot recover from crashes. Hence, if the file system is not unmounted cleanly, the data it contains is lost.

3.2 FUSE - file system in Userspace

FUSE consists of a kernel module and a user space library that allow a non-privileged user to create a file system without modifying the operating system kernel. The module provides a relatively simple API, and acts as a translator to the actual kernel interface. The API uses function callbacks for most common file system functions (`open()`, `creat()`, `unlink()` etc.). A file system developer implements his or her own versions of these functions and registers them with the FUSE module. A major benefit of using FUSE is the API simplicity, which makes file system development quick and relatively easy, and the fact that the file system runs in userspace, which makes it possible to use the usual debugging tools.

A file system running on FUSE populates a struct with pointers to functions adhering to the FUSE API. This struct is passed to the `fuse_main` function and used to generate calls to the file system. Figure 3.1 shows the path of a call to a FUSE file system. The FUSE kernel module receives a request to a file system, and passes the request on to libfuse which generates a call to the file system. This file system is then free to do whatever any normal userspace program can do, e.g. use SSH to interface with a file system at a remote location.

The file system in this thesis, AAFS, resides in an image file, stored inside another file system, so all file system calls to our file system are routed back through VFS.

An important data structure in FUSE is the `fuse_file_info` struct.

```
struct fuse_file_info {
    int flags;
    unsigned long fh_old;
    int writepage;
    unsigned int direct_io : 1;
    unsigned int keep_cache : 1;
    unsigned int flush : 1;
    unsigned int padding : 29;
```

```
    uint64_t fh;
    uint64_t lock_owner;
};
```

It is provided to every callback involving *already open* files, and provides information about the file in question. Only a few entries are used by AAFS:

flags This member contains the mode bits for the open file. It is provided by FUSE and is used by the file system to perform permissions checking.

fh This member is not touched by FUSE and is available for the file system to store private data between operations on a file. Any information in this field is put there by the file system, not FUSE. In AAFS, the entry is used as a reference to the file system's internal representation of an open file.

If a callback needs to know the user id, group id or process id of the calling process, FUSE makes that information available through the `fuse_get_context()` function. This function returns the `fuse_context` struct.

```
struct fuse_context {
    /** Pointer to the fuse object */
    struct fuse *fuse;

    uid_t uid;
    gid_t gid;
    pid_t pid;

    /** Private file system data */
    void *private_data;
};
```

AAFS uses this struct when assigning e.g. owner id to a file.

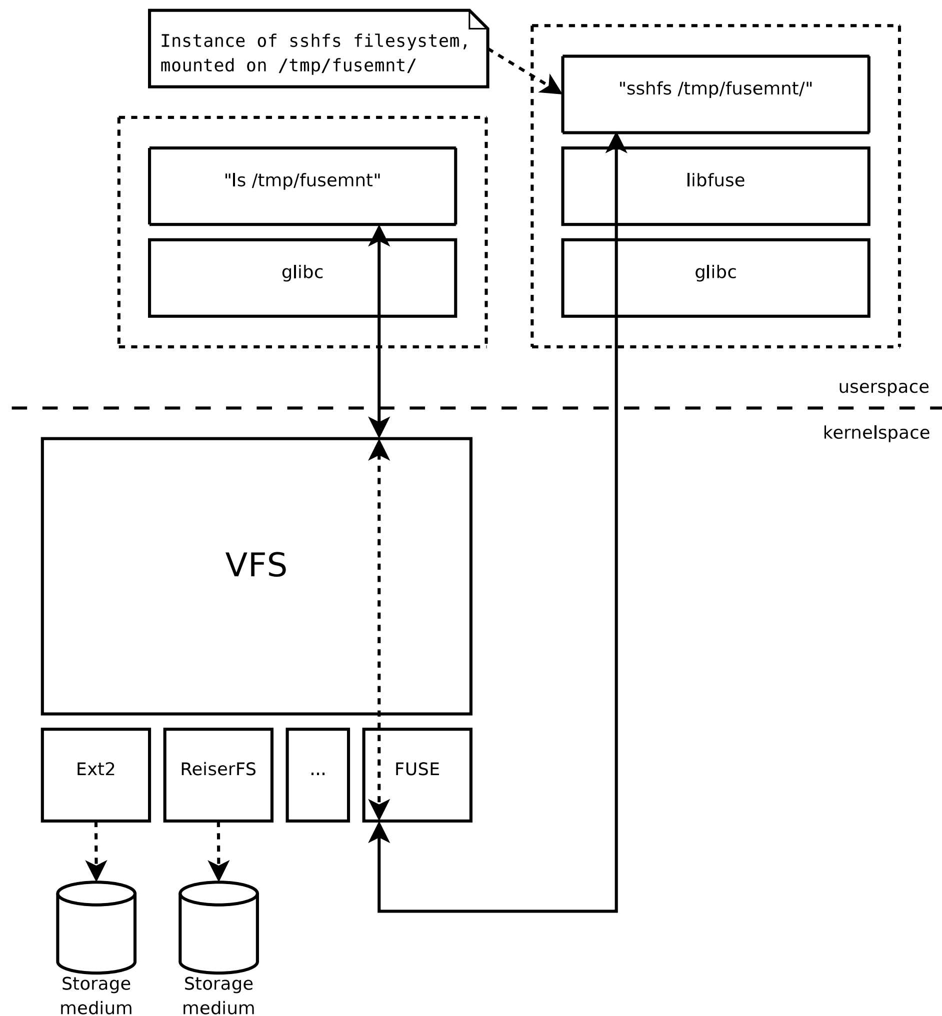


Figure 3.1: The path of a file system call, and the FUSE modules. (The figure is taken from [11], with some additions.)

Also of interest is the way FUSE handles error codes. The regular POSIX file system calls, like `open()`, `close()`, `creat()`, `unlink()` etc. assign error codes to a global variable called `errno`. These error codes are defined in `/usr/include/errno.h`. FUSE wants negated error values returned directly from the file system callbacks. For example, an `open()` system call assigns ENOENT to `errno` if a requested file does not exist, but a FUSE callback would return -ENOENT directly from the callback instead.

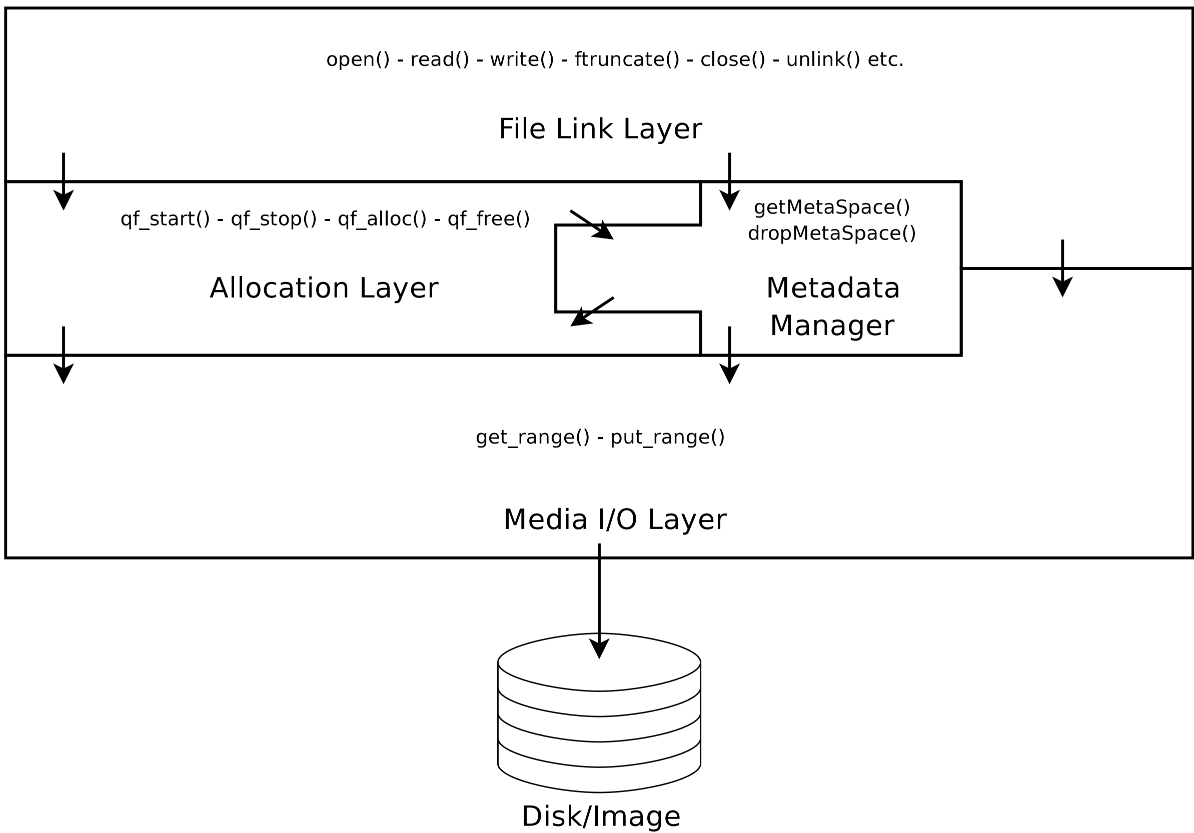


Figure 3.2: The layered hierarchy of Sokolov's allocator (The figure is taken from Sokolov's text, with some additions [8])

3.3 Architecture

Sokolov's allocator uses a layered model, shown in figure 3.2. This section gives an overview of what each layer is supposed to do, as well as what is implemented in each layer of his simulator. For more details on the allocator, consult [8].

3.3.1 Media I/O Layer

The Media I/O Layer handles all reading and writing, on requests from the upper layers, to the storage medium.

In AAFS, extents can be smaller than a hard disk sector. However, hard disks can not read and write units smaller than a disk sector at a time, so when a read is re-

quested, the I/O Layer has to read the entire sector the extent resides in. Only the requested part of the sector is then returned to the layer from which the request originated.

When writing, the file system needs to preserve the data surrounding an extent inside a sector. To do this, the entire sector needs to be read, and the modifications to the extent is committed before the sector is written back to disk.

The following functions perform writing and reading, respectively. The arguments are the address of an extent counted in grains, the offset into it, the length of the data to be read or written, and a pointer to the data buffer.

```
ssize_t put_range(uint64_t address, uint64_t offset,
          uint64_t length, void *data);

ssize_t get_range(uint64_t address, uint64_t offset,
          uint64_t length, void *data);
```

In a full-fledged file system, this layer would communicate with a file system cache, but due to time constraints, the layer only uses `pread()` and `pwrite()` calls to acces the file system image. Since this is the case, the kernel handles caching of the underlying image file. However, since we use `pread()` and `pwrite()`, the kernel handles caching and the above mentioned issues for us.

3.3.2 Allocation Layer

The Allocation Layer is responsible for keeping track of free extents and performing all the operations on them. Requests from the File Link Layer and Metadata Manager are satisfied, and free extents are allocated and returned to those layers. Additionally, this layer contains functions for mounting and unmounting the file system.

```
struct skip_list *qf_alloc(asize_t byte_size,
                           bool_t cont_hint,
```

```
                              asize_t loc_hint);

void qf_free(struct skip_list *ext);
```

`qf_alloc()` satisfies allocation requests for free space. A request is made for a number of bytes. The function calculates how many grains are needed to satisfy the request and uses a variation of the Quick Fit algorithm to search the free space for a suitable extent. After making sure there is enough free space on the device, the function searches for an extent of the requested size or larger. If the smallest extent of the requested size or larger is larger than the size plus the Acceptable Wastage Parameter (see section 2.3.2), the extent is split to meet the requested size.

If the file system has enough free space, but no extents are large enough to meet the request, the function tries to coalesce extents. The coalesce operation merges all adjacent free extents, in an attempt to produce an extent large enough to accomodate the request. `qf_alloc()` may subsequently split the resulting extents, if all of them are too big for the size requested.

If coalescing fails to produce extents of sufficient size, Stanislav's `qf_alloc()` returns a list of smaller extents which meet the request. However, since AAFS lacks support for growing and fragmented files, our version of `qf_alloc()` returns a null pointer instead.

The continuity hint (`cont_hint`) triggers the coalescing subroutine before searcing for extents. The locality hint (`loc_hint`) makes the function search for an extent of a suitable size, with a starting address larger, but as close as possible to the locality hint.

`qf_free()` simply receives a list of extents of extents that are to be freed and returns them to the free list.

3.3.3 File Link Layer

The File Link Layer is responsible for maintaining the datastructures containing file metadata, including the locations of files' extents. This layer acts as the interface

towards FUSE, implementing function callbacks for the usual file system operations. `read()` and `write()` functions communicate directly with the Media I/O Layer. `create()` and `unlink()` functions call the Allocation Layer as well as the Metadata Manager. `truncate()` and `ftruncate()` functions are responsible for resizing and setting initial file size, and call `qf_alloc()`, which returns an extent of the requested size if successful. The functions present in the File Link Layer are a subset of the functions supported by FUSE, and their actions and calling orders are described here.

```
int getattr(const char *path, struct stat *stbuf);
int fgetattr(const char *path, struct stat *stbuf,
             struct fuse_file_info *fi);
```

The `getattr()` and `fgetattr()` callbacks retrieve metadata about a file. These functions fill the same purpose, with the difference being that `fgetattr()` is used on open files. `getattr()` will search the file system for the requested file or directory. If the file is not found, `getattr()` returns a search failure.

The `getattr()` callback is also how FUSE always checks if a file exists, and the function is called once for every filename in any file system callback, except `getattr()`, `read()` and `write()`. The `read()` and `write()` calls don't require a prior callback, since they require the file in question to be opened first. If the `getattr()` callback returns `-ENOENT`, FUSE does not perform the requested operation, but returns to the calling process. Because of this, any function involving files that have not been opened can safely assume that a file exists.

In the case of `getattr()`, if the file exists, the corresponding inode is retrieved, and a `stat` struct is populated with file metadata and returned. These metadata include timestamps, file size, allocated space and the file type and permissions. The same is true for directories, but directories are not supported, and so is not applicable. If the file does not exist, `-ENOENT` is returned.

Since `fgetattr()` is called for open files, it receives a `fuse_file_info` struct containing a pointer to the opened inode. This inode is used to populate the aforementioned `stat` struct with file metadata.

```
int open(const char *path, struct fuse_file_info *fi);
int release(const char *path, struct fuse_file_info *fi);
```

The open() callback implements the file system specific part of the corresponding POSIX function. FUSE provides a fuse_file_info struct (fi). The fh member of this struct is used to point to the file's inode. This same struct instance is then provided to other callbacks involving that particular open file, where the fh member is used to locate the correct inode.

The POSIX counterpart to open() can be told to create a file, if it does not exist, if the O_CREAT flag is set. However, if FUSE receives a POSIX open() call, with O_CREAT set, FUSE first generates a getattr() callback. If getattr() returns -ENOENT, FUSE calls create() rather than open(). If the file exists, the open() callback is used as normal.

The release() callback simply removes the pointer to the inode in the fuse_file_info struct. This necessity is not apparent at first glance, since AAFS does no buffering, and commits every change directly, but it is necessary to prevent FUSE from deleting the file when the file system is unmounted. The reason for this is unclear, as documentation for FUSE is lacking.

```
int create(const char *path, mode_t mode, struct fuse_file_info *fi);
```

The create() callback handles all file creation, unlike the POSIX creat() function which is only called explicitly. create() does not have to check for the existence of the file, as getattr() has already been called at this point and FUSE does not generate a create() callback if the file exists. The getMetaSpace() function is called, to preallocate space for storing the inode on disk, if needed. If this space is needed, and the allocation fails, create() fails as well, returning -ENOSPC. This is a failsafe mechanism to ensure that the file system has enough space to save the new inode.

If the allocation succeeds, or no space is needed, a new inode is created and inserted into the skip list that holds all the file system's inodes. The inode is populated with

the provided file name and file mode, as well as timestamps and the file owner's id and group.

The private data pointer in the provided `fuse_file_info` struct is assigned to the new inode, before the `create()` function returns.

```
int unlink(const char *path);
```

`unlink()` deletes a file. Checking for file existence is not necessary, because `getattr()` has already been called at this point. If a user tries to delete a file that does not exist, FUSE does not generate the corresponding `unlink()` call.

The path function argument is used to locate the file's inode. If the inode is associated with any extents, the extents are deallocated by calling `qf_free()`. Subsequently, a call to `dropMetaSpace()` will deallocate an extent reserved for storing inodes if possible. The inode is then deleted from the file system's data structures.

```
int rename(const char *from, const char *to);
int chmod(const char *path, mode_t mode);
int chown(const char *path, uid_t uid, gid_t gid);
```

`rename()` takes two path arguments, both of which are checked by `getattr()` prior to the `rename()` call. All three of these functions locate the requested inode and change the corresponding metadata. The `rename()` function assigns the new file name if it is shorter than the global maximum file length. `chmod()` assigns new permissions to the file. The `chown()` function assigns new user and group ownership. The necessary information is retrieved by calling `fuse_get_context()` and using the `fuse_context` struct it returns. The file ownership and mode data is stored in each inode, but access restrictions based on this information are not enforced, since it is not strictly necessary for the file system's operation.

```
int truncate(const char *path, off_t size);
int ftruncate(const char *path, off_t size, struct fuse_file_info *fi);
```

These functions explicitly change a file's size. They fill the same purpose, with the difference being that `ftruncate()` is used on files that are already open.

In AAFS, they fill a central role. Since growing files are not supported[1], all files have to be truncated to an appropriate size, before any `write()` calls will succeed.

The file's inode is located by searching the skip list containing the inodes, in the case of `truncate()`, or by following the private data pointer in the provided `fuse_file_info` struct, in the case of `ftruncate()`.

A file can be truncated only once, due to the lack of support for growing files. Consequently it is also not possible to shrink a file.

If the file has not been previously truncated, the function calls `qf_alloc()` to allocate an extent of the requested size. This extent is then associated with the file's inode.

```
int read(const char *path, char *buf, size_t size,
         off_t offset, struct fuse_file_info *fi);
```

As one of the functions that does not mandate a prior call to `getattr()`, `read()` gets the file's inode from the provided `fuse_file_info` struct. If the inode is associated with an extent, `read()` checks that the sum of `size` and `offset` is inside the bounds of the file, before calling `get_range()`.

FUSE does not require the file system to implement a seek function, usually responsible for moving a file pointer to a requested offset, which is used for subsequent `read()` and `write()` calls. Instead, any offsets from the beginning of a file are provided as arguments to the `read()` and `write()` functions.

The length read by `get_range()` has to match the length requested, or FUSE will pad the read buffer with null bytes up to the requested length.

```
int write(const char *path, const char *buf, size_t size,
          off_t offset, struct fuse_file_info *fi);
```

[1] see section 2.4

write() gets the required inode reference from the provided fuse_file_info struct. If the inode is not associated with an extent, write() returns -ENOSPC and refuses to write any data before the calling process has called ftruncate().

If the inode is associated with an extent, write() checks that the sum of size and offset is inside the bounds of the file, before calling put_range(). FUSE mandates that write() should return exactly the number of bytes requested, except on error. As such, if the allocated space is not large enough to contain the requested write, a partial write is *not* carried out, and -ENOSPC is returned.

```
int statfs(const char *path, struct statvfs *stbuf);
```

statfs() retrieves file system statistics and populates the statvfs struct. The values returned in this manner are the file system's block size, in this case the grain size, the size of the file system counted in grains, the number of free grains, the number of files in the file system and the maximum file name length. There are a few more values that can be populated in the statvfs struct, but they are either ignored by FUSE or not in use in AAFS.

```
int readdir(const char *path, void *buf,
            fuse_fill_dir_t filler, off_t offset,
            struct fuse_file_info *fi);


typedef int (*fuse_fill_dir_t) (void *buf, const char *name,
             const struct stat *stbuf, off_t off);
```

readdir() lists the files and directories in a single directory, specified by the path parameter. The buf buffer is populated with entries using the provided filler function pointer. readdir() runs a loop, iterating over all the directory entries.

The filler() function takes the buf pointer as a parameter along with the name of the directory entry and a pointer to a stat struct, which is populated by the loop in readdir(). This struct is specified in the POSIX standard and contains timestamps, size-, ownership- and mode information.

Since AAFS lacks support for a directory hierarchy, `readdir()` is used to list the entire file system. Because of this, the `path` argument is ignored, and the callback always lists the file system.

```
int utimens(const char *path, const struct timespec tv[2]);
```

`utimens()` checks for the existence of the file specified by the path argument. If the file exists, the `timespec` struct is used to assign new access and modification times to the file. The `timespec` struct supports timestamps with nanosecond resolution as well as second resolution, but AAFS only uses second resolution.

Return Values

All the functions in the File Link Layer return zero on success. Any functions that deal with files that have not already been opened perform a check on the path name to make sure the path is shorter than the file system's maximum filename length. If this test fails, the function returns `-ENAMETOOLONG`. The `-ENOSPC` error code is used whenever the file system lacks the necessary space to perform a particular operation. However, since the file system lacks support for growing files, it is also used to force calling processes to specify the size of a file before `write()` operations can be performed. Any file without a size reserved with `truncate()` or `ftruncate()` will cause `write()` to return `-ENOSPC`. This is important to note, and is not to be confused with a lack of space on the file system. This use of the `-ENOSPC` error code can be verified by a `statfs()` operation and making sure the file system is not out of space. `-ENOENT` is returned whenever a requested file is not found.

Unsupported Functions

FUSE offers functionality for a number of functions in addition to the ones listed here. Examples include functions for extended attributes, symbolic links, file locks. However, these functions are not strictly necessary for a working file system, and are not supported by AAFS due to time constraints.

3.3.4 Metadata Manager

The Metadata Manager is responsible for reserving space the Allocation and File Link layers need to store the necessary data when the file system is unmounted. The two functions call the `qf_alloc()` and `qf_free()` functions in the Allocation Layer to reserve and release storage space for metadata. The functions also call the Media I/O Layer to commit the changes immediately.

AAFS does not have designated space for storing metadata. Instead, space for the different metadata types is allocated and deallocated in the same way as regular files. Since the file system lacks support for dynamic files, the Metadata Manager module offers rudimentary support for growing and shrinking space for metadata.

The allocator has a coalesce function used to merge adjacent free extents, to reduce external fragmentation and bookkeeping overhead. For details on the coalescing algorithm, see [8]. In Sokolov's design, the coalesce function requires updating on-disk headers, and was only used when there was no available extent large enough to fill a request. Since AAFS keeps all metadata in RAM, coalescing is a lot faster. AAFS needs additional disk accesses when metadata extents are allocated and freed. As such, coalescing is performed whenever it seems necessary to allocate additional metadata extents. If the coalescing reduces the size of the free list sufficiently, the allocation can be avoided. This coalescing scheme is referred to as "lazy" coalescing in the remainder of this thesis. For testing purposes, AAFS also supports continuous colaescing, where a coalescing is performed every time an extent is freed.

```
int getMetaSpace(int (*needspace)(), uint64_t *loc, uint64_t *space);
inline int needFreextSpace();
inline int needInodeSpace();
```

`getMetaSpace()` is called at the end of `qf_free()` and the beginning of `create()`. The `needspace()` function pointer refers to a check for space for storing either free extent information or inodes. If more space is needed, `getMetaSpace()` requests a free extent from the allocator. The reserved metadata space is organised like a linked list, and the

new extent is added to the list's tail.

```
void dropMetaSpace(int (*aboundspace)(), uint64_t *loc, uint64_t *space);
inline int aboundFreextSpace();
inline int aboundInodeSpace();
```

`dropMetaSpace()` is called by `qf_alloc()` and `unlink()`. The `aboundspace()` function pointer checks if the file system has more space than it needs for free extents or inodes. If that is the case, the extent at the tail of the linked list is freed.

3.3.5 Sokolov's Implementation

As stated earlier, AAFS uses the same architecture as the one Sokolov developed, and for comparison, this section lists what was implemented in his simulator.

Sokolov's Media I/O Layer is minimal, only implementing counters that get incremented whenever one of the functions are called.

The `put_range()` function in his Media I/O Layer had a `destructive` argument, used when only a portion of an extent is modified. This argument could be flagged when the rest of the extent could be safely overwritten, so the extent wouldn't need to be read first. This argument has been removed in the AAFS implementation. With extents smaller than a sector, this argument is not needed since the sector containing the extent has to be read regardless. Bounds checking the extent can remove the need to read a sector if the entire sector is covered by one extent.

The `qf_alloc()` and `qf_free()` functions in the Allocation Layer are complete as of Sokolov's implementation, with some modifications necessary to accomodate the additional data structures needed by the file system. The functions that handle mounting and unmounting, `qf_start()` and `qf_stop()` are designed and implemented from scratch in this thesis. These functions are covered in section 3.4

The File Link Layer as implemented by Sokolov is minimal. Functions for manipulating file contents are not implemented. Functions for creating and deleting files are implemented in a simulator driver that generates calls to `qf_alloc` and `qf_free`

according to trace data. These functions don't implement any of the bookkeeping associated with the file link layer, but simply implement counters for the purpose of gathering statistics. The simulator has no data structures for storing file inodes or otherwise track individual files.

The Metadata Manager module is not featured in Sokolov's design and is added in this thesis.

3.4 Mounting & Unmounting

Building a FUSE file system creates an executable used to mount the file system in question. This is different from how the linux `mount` utility is used to mount file systems.

The AAFS executable is started with the desired mountpoint path and an image file path as command line arguments. A `fuse_operations` struct is first populated with pointers to all the functions listed in AAFS' File Link Layer. The `main()` function then initializes the file system by calling `qf_start()` and calls `fuse_main()`, providing the mount point path and the `fuse_operations` struct as a parameter. `fuse_main` registers the file system with VFS, and at this point the file system is mounted. FUSE performs operations on the file system by generating callbacks to the functions listed in the `fuse_operations` struct.

To unmount a FUSE file system, the `fusermount` utility is used. The utility is invoked with a `-u` switch and the mount point path as arguments. The `fusermount` utility causes the `fuse_main()` function to return, giving control back to AAFS' `main()` function. After that, shuts down the file system by calling `qf_stop()`.

```
void qf_start(char *filename);
void qf_stop(void);
```

AAFS initializes and shuts down the file system through the `qf_start()` and `qf_stop()` functions.

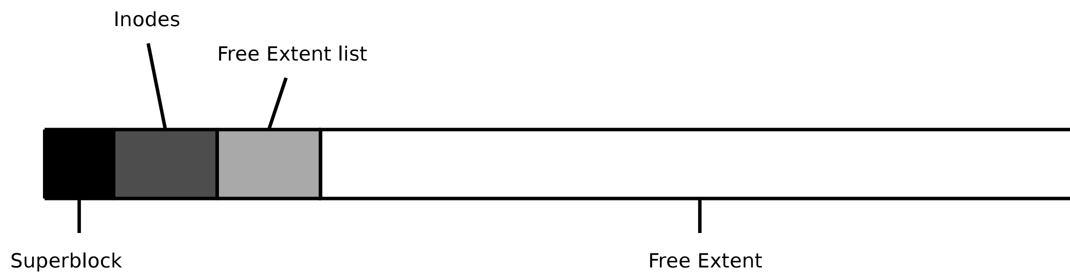


Figure 3.3: Disk layout on an empty file system

`qf_start()` uses the filename string to locate and open the file system image. It then reads the superblock, located at the beginning of the file. The size of the superblock is hard coded. Subsequently, the inodes are read from the image file and the inode skip list is constructed using the method described in section 2.3.1. After that, `qf_start()` reads the free extent data from the image file, and constructs a temporary skip list containing the free extent data in the same fashion as with the inodes. Finally, the extent headers in this temporary list are sorted into the quick lists and misc list used by the allocator implemented by Sokolov.

`qf_stop()` does the opposite, and in reversed order. The quick lists and misc list are merged into a temporary skip list and marshalled and written to the image file. Next, the inode skip list is marshalled and written. The superblock is then written to the image file, and the file is closed.

3.5 Formatting the File System

A separate utility called `initfs` has been implemented to create and initialize an image file so it can be mounted by AAFS. This utility is invoked with an image file path and a desired size, counted in bytes. `initfs` opens the image file and truncates it to the requested length.

Figure 3.3 shows how a new file system looks. A `superblock` struct is created and populated with the correct information. The metadata extents have a hardcoded size, and `inodes_loc` is aligned to the first grain after the superblock. `freext_loc` is set to

inodes_loc plus the metadata extent size. There are no files present, so inodes_num is set to zero. One big free extent spans the space after the metadata extents to the end of the file system, so freext_num is set to 1. inodes_space and freext_space are set to the hardcoded metadata extent size. grains_size is set to the image's requested size divided by the hardcoded grain size, and grains_free is set to grains_size minus the total size of the superblock and the metadata extents.

After populating the superblock struct, initfs writes it to the start of the image. meta_header structs with next_ptr set to zero and size set to the hardcoded metadata extent are written to the positions entered into inodes_loc and freext_loc in the superblock. Since no files, and consequently no inodes, are present when the image is created, the inode extent is finished. The extent containing free extent headers is populated with an extent struct with it's address field set to freext_loc plus the metadata extent size, and a size field set the same as grains_free in the superblock.

4 Evaluation

AAFS is a relatively minimal implementation of a file system. A lot of features, such as error correction and resilience, are missing. Also, since AAFS stores all metadata in RAM when the file system is mounted, it is hard to make a fair performance comparison with other file systems. AAFS uses FUSE, whose overhead is uncertain, which further clouds the performance issue. Instead, the performance testing focuses fragmentation characteristics, rather than speed of operation.

4.1 Simulated Workload

A log was acquired from a mailpipe server run by the University of Oslo IT department. The server handles incoming emails which are to be forwarded to programs such as list archives, mail2news and other services. The emails are retained for a short while until the correct destination can be determined. The emails are then forwarded and deleted from the server's file system. This server does not carry out any direct deliveries to these programs because of high I/O workloads.

This log was used to run simulations on AAFS and Ext2, while recording statistics on the two file systems.

```
2008-06-27T00:00:47 1KBzWJ-0000TO-2m 1637
2008-06-27T00:00:48 1KBzWJ-0000TO-2m
```

The log consists of lines of plain text listing a timestamp, a hash representing a file name, and a size in bytes. Any names or other sensitive information was removed before the log was released. A line containing a size represents an email being stored,

and a line with the size missing represents the mail being deleted from the file system. The log was of a 30 day period, and contains some lines for deleting non-existant files, as they were created before the log starts. Similarly, some files are retained in the file system at the end of the log.

The log was processed by a simple program that delayed all delete lines by a random, uniformly distributed, number of seconds up to one hour. This was done in an attempt to simulate a more problematic workload, with more files in the file system simultaneously and worse fragmentation.

AAFS is implemented with the intention of running as a normal file system on FUSE. However, to speed up the simulation, a the file system was modified, adding a function that parses the log file and calls the File Link Layer functions directly. On create, a file is created and truncated to the specified size. Truncate allocates the required space, and filling the file with data is not necessary.

The Ext2 simulation, similarly, is a program that parses the log file, and executes a program developed by Željko Vrba to gather detailed statistics about the Ext2 file system. Since Ext2 does not support persistent preallocation, the simulation writes zero bytes to the file to force allocation for a file of the specified size. To speed up the Ext2 simulation, the file system was running in an image in RAM.

The AAFS and Ext2 simulations run through the same log, recording statistics each time the log transitions from create operations to delete operations, delete operations to create operations. The AAFS simulation records statistics on free space, external fragmentation, number of files present, number of free extents and internal fragmentation. The Ext2 simulation additionally records data fragmentation. Since AAFS does not support fragmented file, data fragmentation is not applicable.

4.2 Test Cases

AAFS has a few adjustable attributes that lead to different performance characteristics. Primarily, different grain sizes are tried and their respective performances com-

pared. Also, the effect the two coalescing schemes (lazy and continuous) have on fragmentation and necessary image size. The Acceptable Wastage Parameter is kept at 0 for these tests, but some indirect observations are made. The base case for comparison with AAFS was chosen to be 16 byte grains and lazy coalescing[1].

File systems tend to fragment data more as they fill up, and it becomes necessary to use fragmented free space for storing files. AAFS does not support data fragmentation, but external fragmentation can be used for cautious comparisons. The tests were run at the smallest image size possible. A binary search was conducted on both ext2 and AAFS to find the smallest image that does not run out of space when running the log. The largest of these two sizes was chosen as a base for comparison between the two file systems. AAFS requires a larger image, since it does not support fragmented files.

For comparison with Ext2, AAFS was tested at the required image size of 996.5MB with both lazy and continuous coalescing on 16 byte grains. Tests were also run at the same image size with both coalescing schemes on 1 byte grains. Ext2 was tested on an image size of 996.5MB with a 1024 byte block size and a 4096 byte block size.

AAFS was also tested on a minimal image size of 1501.5MB with 4096 byte grains with lazy coalescing, and compared to continuous coalescing withh the same image and grain size. Additionally, since continuous coalescing improves external fragmentation and makes it possible to run on a smaller image, the smallest possible image size was found and tested for 16 byte grains and 4096 byte grains at 900MB and 886.5MB respectively.

4.3 Test Results

Figure 4.1 shows the external fragmentation statistics for the ext2 tests in comparison with the AAFS base case. External fragmentation is calculated as the size of the biggest free extent divided by total free space. On ext2, weekends are easily dis-

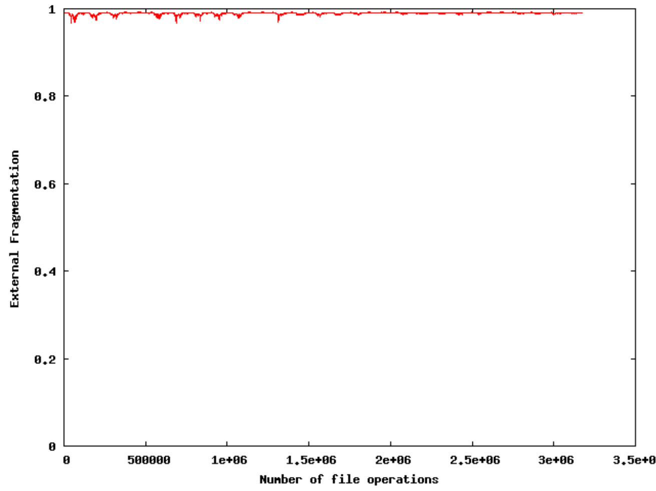
[1]see section 3.3.4

tinguishable from weekdays, with less movement in the graph, as the file system is mostly empty. On AAFS, plateaus are visible followed by sharp drops indicating a coalescing has taken place.

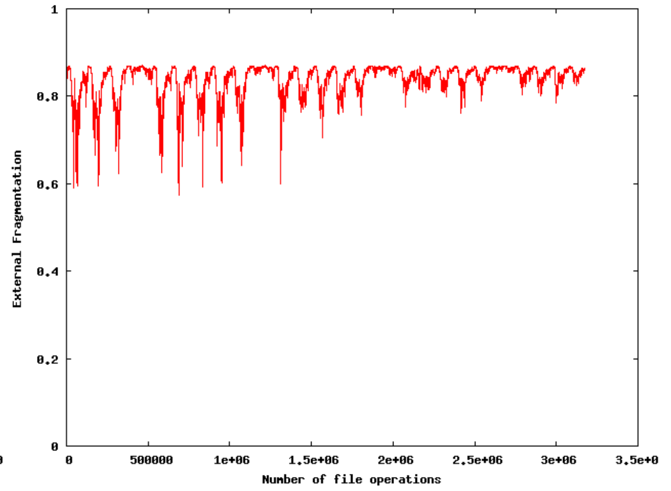
Ext2 is divided into block groups, each group starting with a superblock, free space bitmap and inode list for the blocks in that group. As a consequence, free extents can never be bigger than the size of a block group. The size of the block groups is restricted by the fact that the free space bitmap is limited to a single block. This means that the maximum size of a block group is 8 times the size of a block. So with 1024B blocks, each block group is 8192 blocks, and with 4096B blocks, each group is 32768 blocks. This leads to external fragmentation measurements that are upside down from what one might expect; fragmentation decreases as the file system fills up, since amount of free space is decreasing relative to the size of the biggest free extent. External fragmentation on ext2 with 1024B blocks is quite severe, since each block group is only 8MB. On ext2 with 4096B blocks, the block groups are significantly larger, causing overall lower external fragmentation. Also more space is wasted on internal fragmentation, meaning the amount of free space is further reduced, somewhat reducing external fragmentation.

The number of files present in the file systems show a clear pattern of weekdays and weekends, which is reflected in the fragmentation statistics on the ext2 file system. The fact that ext2 gets better external fragmentation when more files are present indicates that a smaller image size should have been tested to get a clear picture of external fragmentation on this file system. However, due to time constraints and problems during testing, this was not possible.

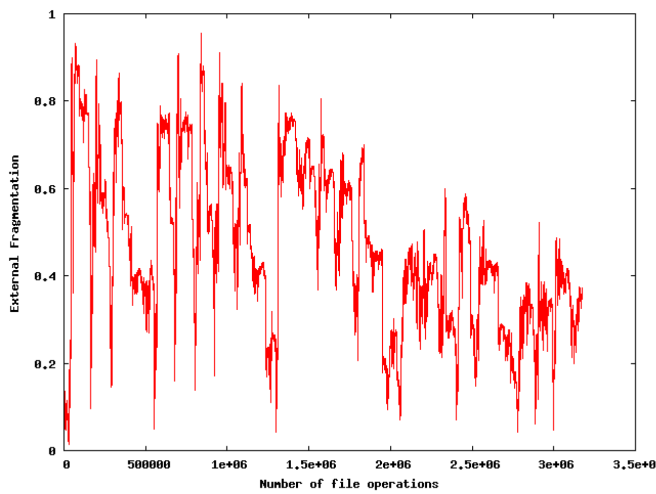
Figure 4.2 shows data fragmentation on ext2. Data fragmentation is measured as the number of extents per file on average. The small block groups of ext2 with 1024B blocks leads to significantly increased data fragmentation over ext2 with 4096B blocks. Data fragmentation is lower with 4096B blocks, since fewer different extent sizes are possible, but still quite pronounced, even though a block group on a 4096B block size can contain 128MB. Ext2 tries to place blocks belonging to a file in the same



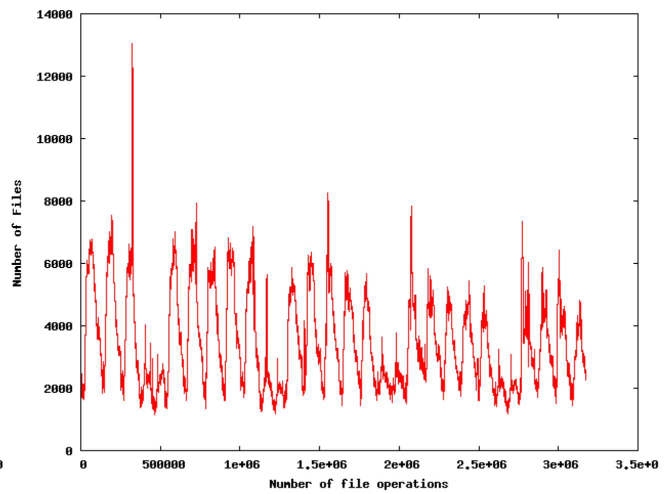
(a) Ext2 with 1024B blocks



(b) Ext2 with 4096B blocks



(c) AAFS with 16B grains and lazy coalescing



(d) Number of files

Figure 4.1: Comparison of External Fragmentation

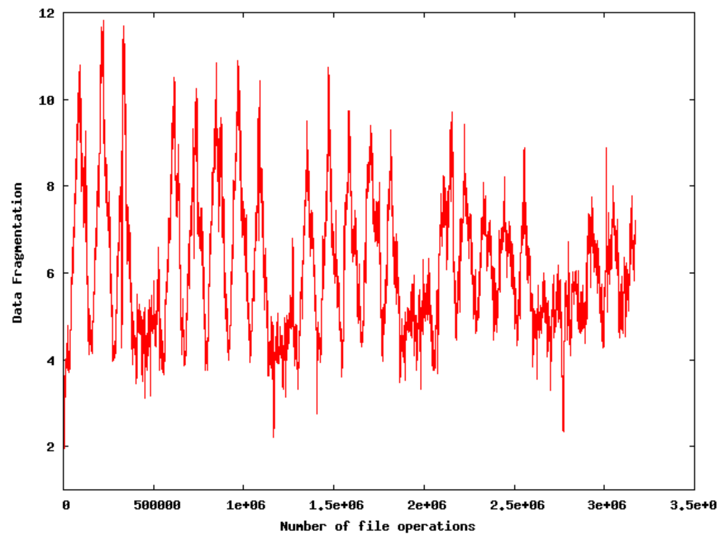
block group, but does not seem to care if the blocks are contiguous and in order.

Figure 4.3 shows internal fragmentation on different grain sizes. For identical grain and block sizes, internal fragmentation is the same between file systems. 4096B grains shows internal fragmentation peaking at almost 40% and routinely going above 15%. Interestingly, in this particular log, the internal fragmentation is bigger in the weekends, which indicates smaller files. The weekends are also when the fewest files are present, on average, so the internal fragmentation is not problematic in this case, as there is a lot of free space available. This is not the case for all workloads, however. 1024B block sizes bring internal fragmentation down to around 5%, peaking at almost 10% in the weekends, which is a significant improvement. 16B grains almost eliminate internal fragmentation entirely, indicating that a smaller grain size will not bring significant improvements in internal fragmentation. 1B grains were also tested, but has no internal fragmentation and is not shown in this comparison. Note the difference in scale between the 16B graph and the other two graphs.

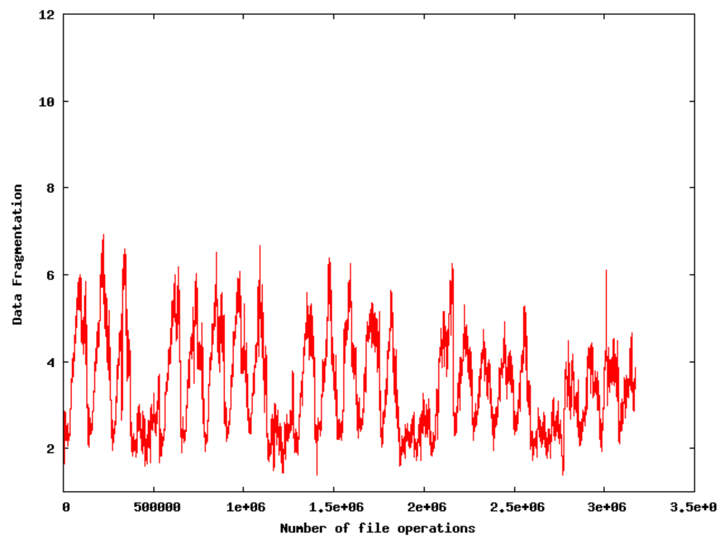
Figure 4.4 shows the significant improvements in external fragmentation continuous coalescing gives over lazy coalescing. External fragmentation is lower overall, often more than halved, and more stable. With the current design, coalescing is a relatively expensive operation, coalescing the entire free space list each time, so it could be desirable to introduce a function that coalesces single extents when they are freed, as coalescing on this file system does not necessitate disk access. Performance will be more predictable in these cases, as no long-running coalescings can disrupt response times. External fragmentation would also be lower, which is beneficial in itself.

Figure 4.5 shows external fragmentation on AAFS with a 1501.5MB grain size. Because of internal fragmentation, the image has to be significantly larger than the 16B grain size at 996.5MB. The 4096B grain size with lazy coalescing has similar or worse external fragmentation than the 16B grain size counterpart. Continuous coalescing offers significant improvement in external fragmentation but this is not a net gain, since it requires a substantially larger image.

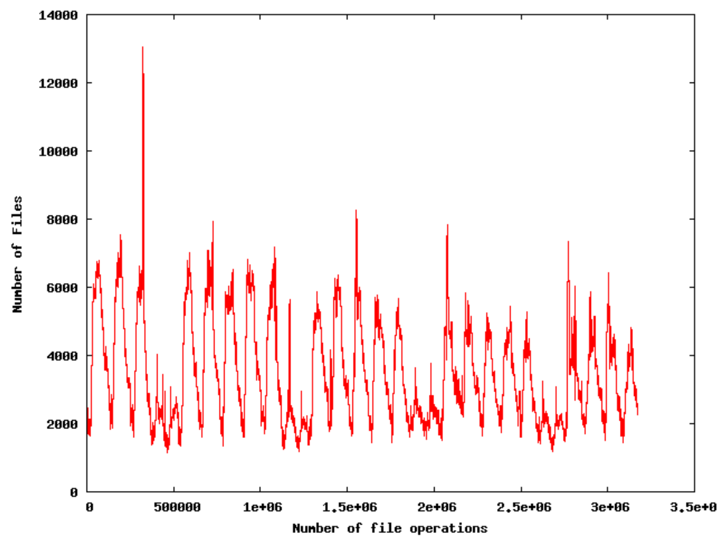
Figure 4.6 shows external fragmentation for the smallest possible image size with



(a) Ext2 with 1024B blocks

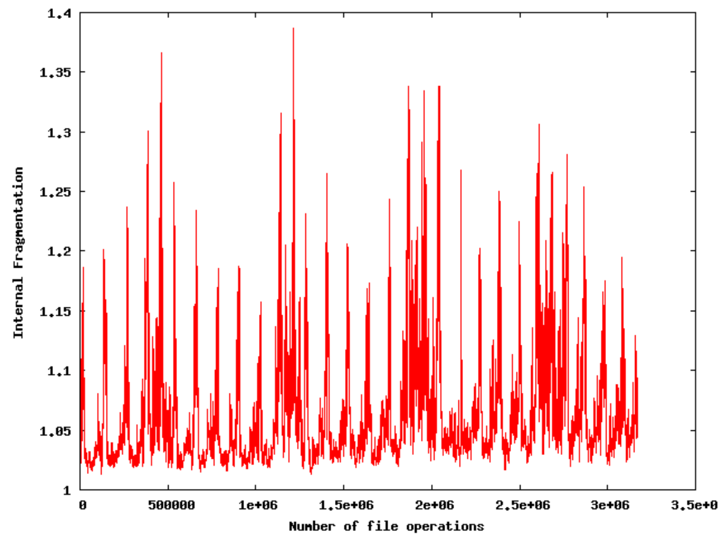


(b) Ext2 with 4096B blocks

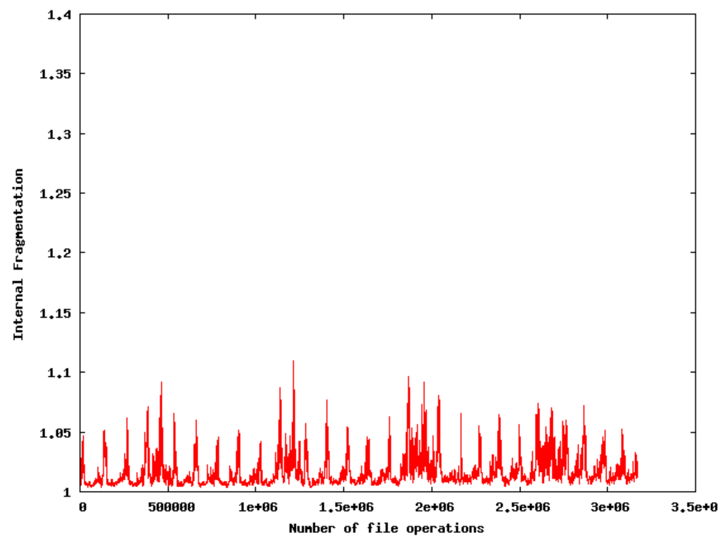


(c) Number of files

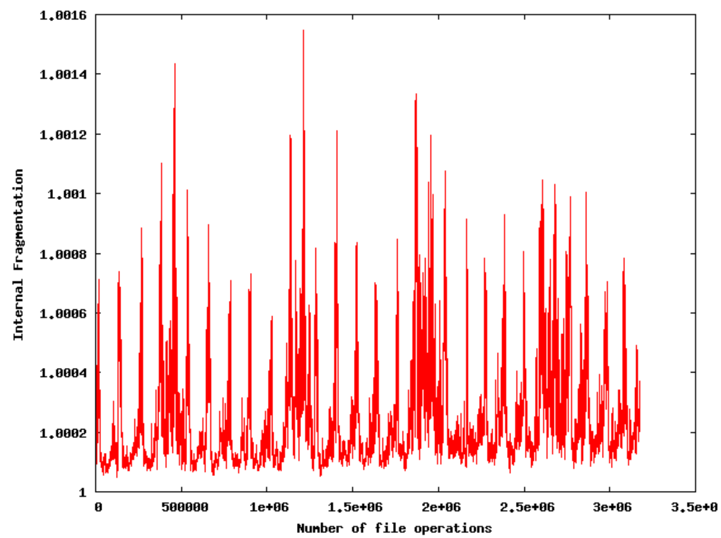
Figure 4.2: Comparison of Data Fragmentation



(a) Internal fragmentation with 4096B grains

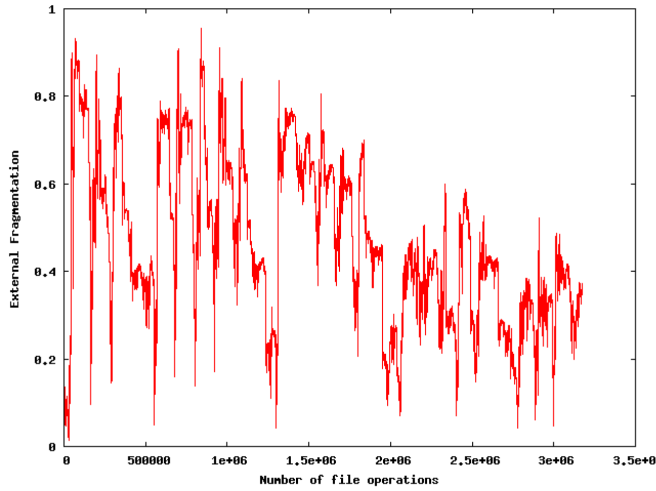


(b) Internal fragmentation with 1024B grains

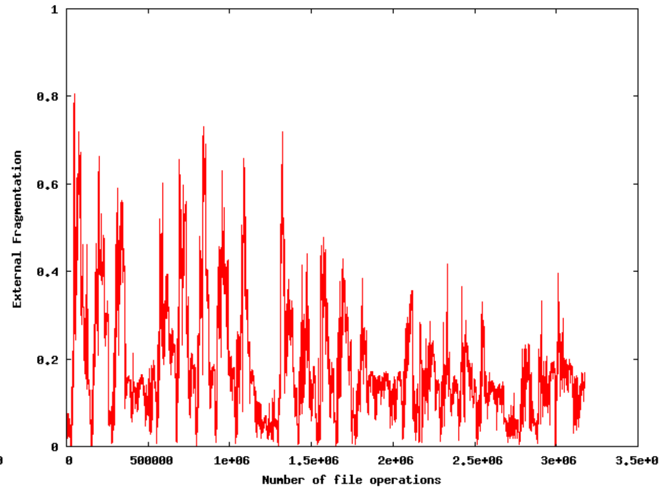


(c) Internal fragmentation with 16B grains (different scale)

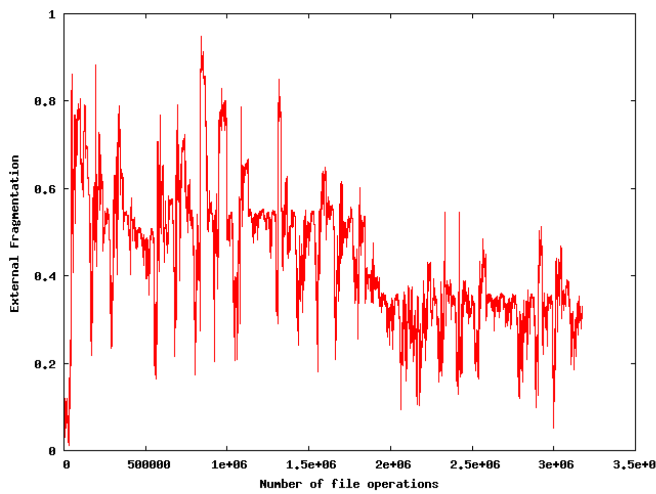
Figure 4.3: Comparison of Internal Fragmentation



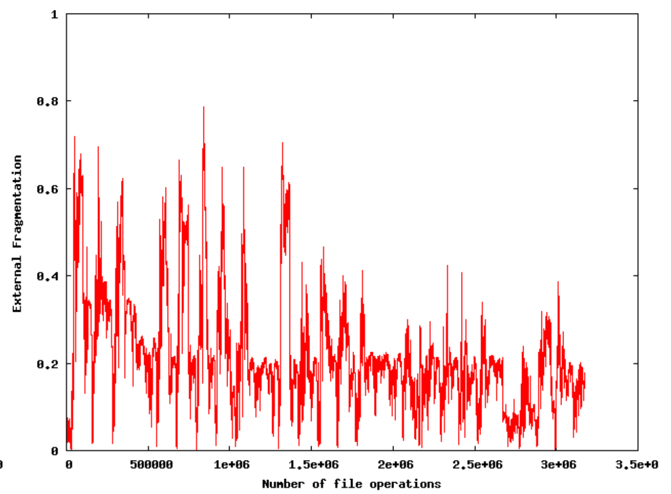
(a) 16B grains, lazy coalescing



(b) 16B grains, continuous coalescing

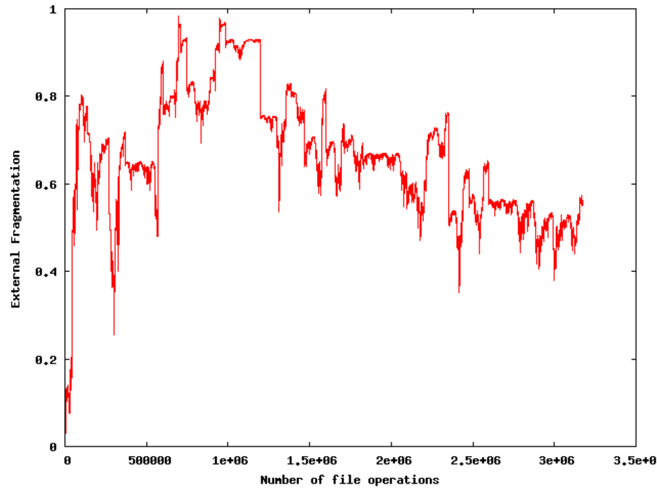


(c) 1B grains, lazy coalescing

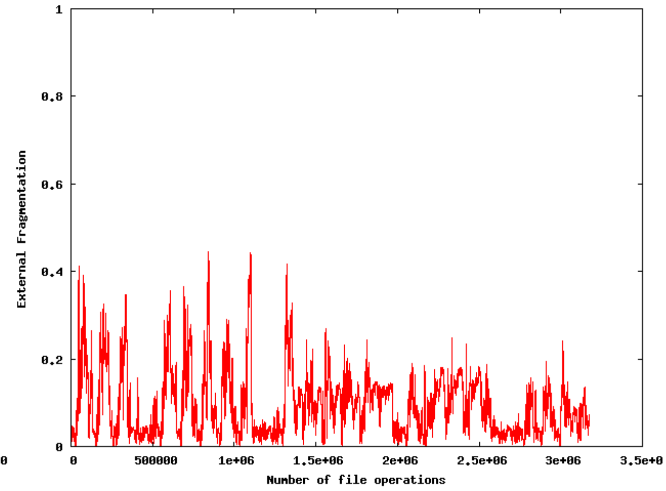


(d) 1B grains, continuous coalescing

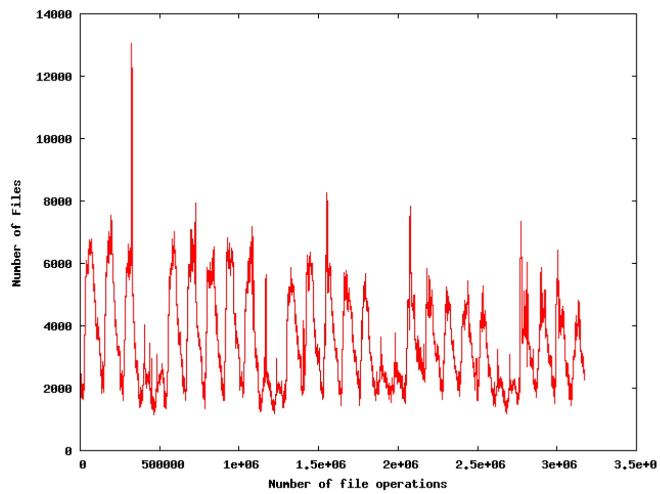
Figure 4.4: Comparison of External Fragmentation, different AAFS settings for an image size of 996.5MB



(a) lazy coalescing



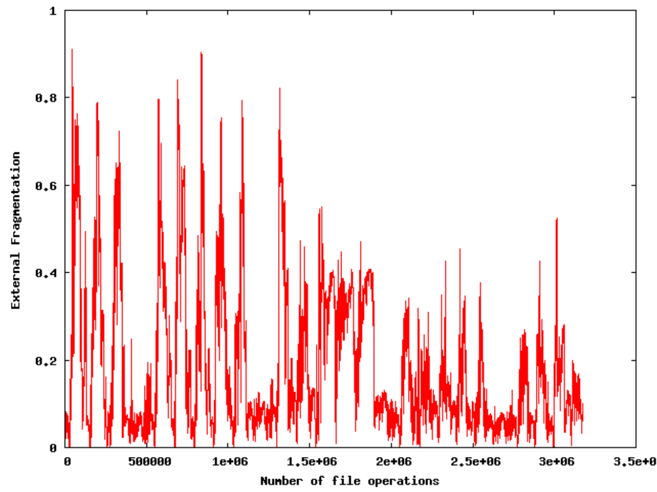
(b) continuous coalescing



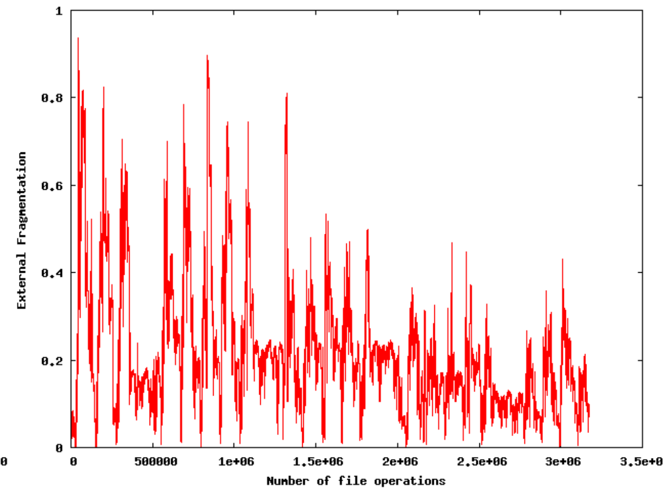
(c) Number of files

Figure 4.5: Comparison of External Fragmentation, AAFS with an image size of 1501.5MB, 4096B grains

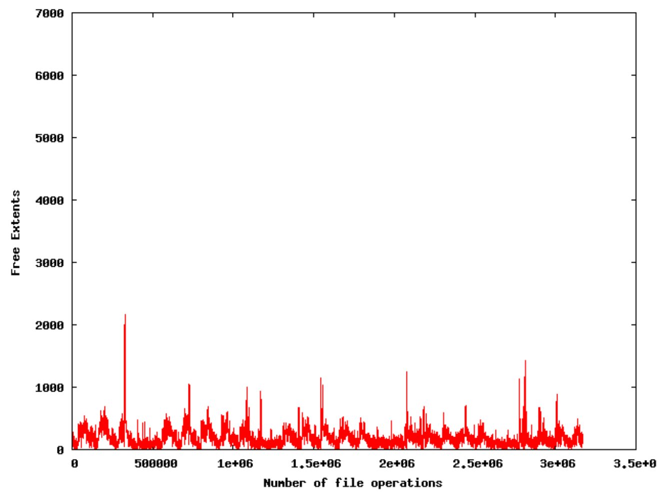
continuous coalescing with 16B and 4096B grain sizes. Both configurations peak at approximately the same external fragmentation, with the 16B grains having slightly worse fragmentation overall. Interestingly, the 4096B grain size fits in a slightly smaller image than 16B. The reason for this is the more numerous free extents, giving a bigger metadata overhead with 16B grains. During preliminary experiments that were not recorded, a large number of the free extents were discovered to be only 1 grain long. Adjusting the AWP would eliminate any extents smaller than the AWP. Adjusting the AWP can bring down the required image size and external fragmentation at the cost of some internal fragmentation. Speeds could also be improved, as there are fewer extents for the allocator to search through. However, testing the Acceptable Wastage Parameter in detail is a big experiment by itself and is outside the scope of this thesis.



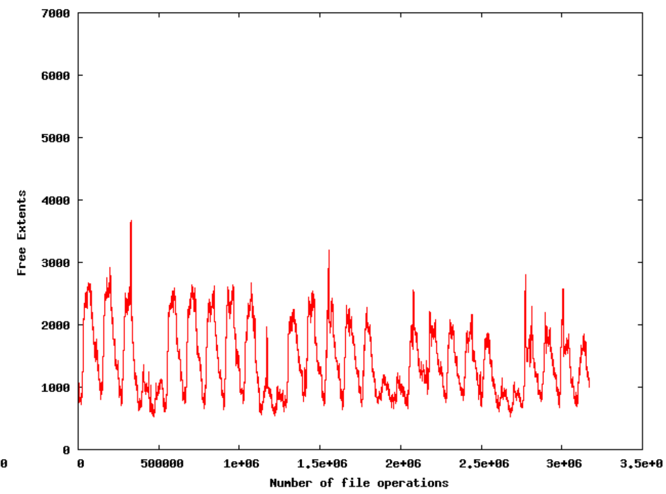
(a) 4096B grains, continuous coalescing, 886.5MB image size



(b) 16B grains, continuous coalescing, 900MB image size



(c) Number of free extents, 4096B grains



(d) Number of free extents, 16B grains

Figure 4.6: Comparison of External Fragmentation, AAFS with continuous coalescing on minimal image sizes

5 Conclusion

5.1 Summary

In this thesis, the design, implemented and evaluation of a file system supporting extents and arbitrarily sized grains has been presented. Tests were carried out to compare AAFS' fragmentation statistics with Ext2. Ext2 was chosen because it is well documented and open source, making the benchmarking process easier. The tests results are encouraging but not conclusive. The file system is quite fast, as it only needs one disk seek for most disk operations, and some times none, but lacks some features like data fragmentation support which makes direct comparisons difficult. Additionally, the Acceptable Wastage Parameter has not been extensively tested, and bear closer study.

As it stands, being a file system implementation of the allocator implemented by Sokolov, AAFS is well suited for applications with small file size distributions, like web- and mail servers.

5.2 Future Work

This thesis implements a minimal file system, and as such has many possibilities for extensions. Many of them are explored in detail in section 2.4, and are considered standard file system features, like directory hierarchies and proper access control. Crash recovery and general fault resilience is currently not implemented, and is a desirable feature in any file system. Most important is perhaps support for fragmented

files, for further testing. File data injection and removal is a small extension which could make AAFS useful for tasks like video editing.

Adapting the coalescing algorithm will offer direct improvement without significant computational overhead, possibly even reducing it compared to the current lazy coalescing, and give the file system more predictable response times and improved external fragmentation.

Since AAFS hopes to improve read speeds by placing multiple files in each block, some mechanism for file placement should be considered, to ensure related files are located next to each other.

Bibliography

[1] IDEMA P. Chicoine, M. Hassner, M. Noblitt, G. Silvus, B. Weber, and E. Grochowski. Idema(r) hard disk drive long data sector white paper. White Paper, 2007. Available from http://www.idema.org/ - Document Library.

[2] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. File size distribution on unix systems - then and now. *ACM SIGOPS Operating Systems Review*, 40:100–4, 2006.

[3] Leland L. Beck. A dynamic storage allocation technique based on memory residence time. *Communications of the ACM*, 25:714–724, 1982.

[4] Richard McDougall. Getting to know the solaris filesystem, part 1: Allocation and storage strategy. Article, 1999. Available from http://www.solarisinternals.com/si/reading/sunworldonline/swol-05-1999/swol-05-filesystem.html.

[5] Hp-ux system administration tasks: Hp 9000, chapter 4, section 4: An introduction to vxfs. Manual, 1999. Available from http://docs.hp.com/en/B2355-90672/ch04s04.html.

[6] Marshall K. McKusick and William N. Joy. A fast file system for unix. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1984.

[7] Charles B. Weinstock. Dynamic storage allocation techniques. *PhD thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania*, 1976.

[8] Stanislav Sokolov. Design and analysis of a dynamic extent-based allocaiont technique for multimedia file systems used in cdn proxies. *Master thesis, Department of Informatics, University of Oslo, Norway*, 2006.

[9] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Workshop on Algorithms and Data Structures*, pages 437–49, 1989.

[10] Mike Jurka Adin Scannell. Markov models for predictive memory management. Available from http://adin.scannell.ca/w3/doc/mmmm.pdf.

[11] Fuse: Filesystem in userspace, webpage. Available from http://fuse.sourceforge.net/.