

The Effects of Filesystem Fragmentation

Giel de Nijs

Philips Research

giel.de.nijs@philips.com

Ad Denissen

Philips Research

ad.denissen@philips.com

Ard Biesheuvel

Philips Research

ard.biesheuvel@philips.com

Niek Lambert

Philips Research

niek.lambert@philips.com

Abstract

To measure the actual effects of the fragmentation level of a filesystem, we simulate heavy usage over a longer period of time on a constantly nearly full filesystem. We compare various Linux filesystems with respect to the level of fragmentation, and the effects thereof on the data throughput. Our simulated load is comparable to prolonged use of a Personal Video Recorder (PVR) application.

1 Introduction

For the correct reading and writing of files stored on a block device (*i.e.*, hard drive, optical disc, etc.) the actual location of the file on the platters of the disk is of little importance; the job of the filesystem is to transparently present files to higher layers in the operating system. For *efficient* reading and writing, however, the actual location does matter. As blocks of a file typically need to be presented in sequence, they are mostly also read in sequence. If the blocks of a file are scattered all over a hard drive, the head of the drive needs to seek to subsequent blocks very often, instead of just reading those

blocks in one go. This takes time and energy, and so the effective transfer speed of the hard drive is lower and the energy spent per bit read is higher. Obviously, one wants to avoid this.

In the early days of the PC, the filesystem of choice for many was Microsoft's FAT [1], later followed by FAT32. The allocation strategy, the strategy that determines which blocks of a file go where on the disk, was very simple: write every block of the file to the first free block found. On an empty hard drive, the blocks will be contiguous on the disk and reading will not involve many seeks. As the filesystem ages and files are created and deleted, the free blocks will be scattered over the drive, as will newly created files. The files on the hard drive become fragmented and this affects the overall drive performance. To solve this, a process called *defragmentation* can re-order blocks on the drive to ensure the blocks of each file and of the remaining free space will be contiguous on the disk. This is a time consuming activity, during which the system is heavily loaded.

More sophisticated filesystems like Linux's ext2 [2] incorporate smarter allocation strategies, which eliminate the need for defragmentation for everyday use. Specific usage scenarios might exist where these, and other, filesystems

perform significantly worse or better than average. We have explored such a scenario and derive a theoretical background that can predict, up to a point, the stabilisation of the fragmentation level of a filesystem.

In this paper we study the level of fragmentation of various filesystems throughout their lifetime, dealing with a specific usage scenario. This scenario is described in section 2 and allows us to derive formulae for the theoretical fragmentation level in section 3. We elaborate on our simulation set-up in section 4, followed by our results in section 5. We end with some thoughts on future work in section 6 and conclude in section 7.

2 The scenario

Off-the-shelf computers are used more and more for storing, retrieving and processing very large video files as they assume the role of a more advanced and digital version of the classic VCR. These so-called Personal Video Recorders (PVR) are handling all the television needs of the home by recording broadcasts and playing them back at a time convenient for the user, either on the device itself or by streaming the video over a home network to a separate rendering device. Add multiple tuners and an ever increasing offer of broadcast programs to the mix and you have an application that demands an I/O subsystem that is able to handle the simultaneous reading and writing of a number of fairly high bandwidth streams. Both home-built systems as well as Consumer Electronics (CE) grade products with this functionality exist, running software like MythTV [3] or Microsoft Windows Media Center [4] on top of a standard operating system.

As these systems are meant to be always running, power consumption of the various components becomes an issue. The costs of the

components is of course an important factor as well, especially for CE devices. Furthermore, the performance of the system should not deteriorate over time to such a level that it becomes unusable, as a PVR should be low maintenance and should just work. Clearly, overdimensioning the system to overcome performance issues is not the preferred solution. A better way would be to design the subsystems in such a way that they are able to deliver the required performance efficiently and predictably. As stated above, the hard-disk drive (HDD) will be one of the most stressed components, so it is interesting to see if current solutions are fulfilling our demands.

2.1 Usage pattern

The task of a PVR is mainly to automatically record broadcast television programs, based on a personal preference, manual instruction or a recommendation system. As most popular shows are broadcast around the same time of day and PVRs are often equipped with more than one tuner, it is not uncommon that more than one program is being recorded simultaneously. As digital broadcast quality results in video streams of about 4 to 8 megabit/s, the size of a typical recording is in the range of 500 MB to 5 GB.

As the hard drive of a PVR fills up, older recordings are deleted to make room for newer ones. The decision which recording to delete is based on age and popularity, *e.g.*, the news of last week can safely be deleted, but a user might want to keep a good movie for a longer period of time.

The result of this is that the system might be writing two 8 megabit/s streams to a nearly full filesystem, sometimes even while playing back one or more streams. For 5 to 10 recordings per day, totalling 3 to 5 hours of content, this results

in about 10 GB of video data written to the disk. Will the filesystem hold up if this is done daily for two years? Will the average amount of fragmentation keep increasing or will it stabilise at some point? Will the effective data rate when reading the recorded files from a fresh filesystem differ from the data rate when reading from one that has been used extensively? We hope to answer these questions with our experiments.

Although the described scenario is fairly specific, it is one that is expected to be increasingly important. The usage and size of media files are both steadily increasing and general Personal Computer (PC) hardware is finding its way into CE devices, for which cost and stability are main issues. The characterised usage pattern is a general filesystem stress test for situations involving large media files.

As an interesting side-note, our scenario describes a pattern that had hardly ever been encountered before. Normal usage of a computer system slowly fills the hard drive while reading, writing and deleting. If the hard drive is full, it is often replaced by a bigger one or used for read-only storage. Our PVR scenario, however, describes a full filesystem that remains in use for reading and writing large files over a prolonged period of time.

2.2 Performance vs. power

A filesystem would ideally be able to perform equally well during the lifetime of the system it is part of, without significant performance loss due to fragmentation of the files. This is not only useful for shorter processing times of non-real-time tasks (*e.g.*, the detection of commercial blocks in a recorded television broadcast), but it also influences the power consumption of the system [5].

If a real-time task with a predefined streaming I/O behaviour is running, such as the recording

of a television program, power can be saved if the average bit rate of the stream is lower than the maximum throughput of the hard drive. If a memory buffer is assigned for low power purposes, it can be filled as fast as possible by reading the stream from the hard drive and powering down the drive while serving the application from the memory buffer. This also holds for writing: the application can write into the memory buffer, which can be flushed to disk when it is full, allowing us to power off the drive between bursts. The higher the effective read or write data rate is, the more effective this approach will be. If the fragmentation of the filesystem is such that it influences the effective data rate, it directly influences the power consumption. A system that provides buffering capabilities for streaming I/O while providing latency guarantees is ABISS [6].

3 The theory

To derive a theory dealing with the fragmentation level of a filesystem, we first need some background information on filesystem allocation. This allocation can (and will) lead to fragmentation, as we will describe below. We determine what level of fragmentation is acceptable and as a result we can derive the fragmentation equilibrium formulae of section 3.3.

3.1 Block allocation in filesystems

Each filesystem has some sort of rationale that governs which of the available blocks it will use next when more space needs to be allocated. This rationale is what we call the allocation strategy of a filesystem. Some allocation strategies are more sophisticated than others. Also, the allocation strategy of a particular filesystem can differ between implementations without sacrificing interoperability, provided that

every implementation meets the specifications of how the data structures are represented on disk.

3.1.1 The FAT filesystem

The *File Allocation Table File System* (FATFS) [1] is a filesystem developed by Microsoft in the late seventies for its MS-DOS operating system. It is still in wide use today, mainly for USB flash drives, portable music players and digital cameras. While recent versions of Windows still support FATFS, it is no longer the default filesystem on this platform.

A FATFS volume consists of a boot sector, two file allocation tables (FATs), a root directory and a collection of files and subdirectories spread out across the disk. Each entry of the FAT maps to a cluster in the data space of the disk, and contains the index number of the next cluster of the file (or subdirectory) it belongs to. An index of zero in the FAT means the corresponding cluster on the disk is free, other magic numbers exist that denote that a cluster is the last cluster of a file or that the cluster is damaged or reserved. The second FAT is a backup copy of the first one, in case the first one gets corrupted.

An instance of FATFS is characterised by three parameters: the number of disk sectors in a cluster (2^i for $0 \leq i \leq 7$), the number of clusters on the volume and the number of bits used for each FAT entry (12, 16 or 32 bits), which at least equals the base-2 logarithm of the number of clusters.

The allocation strategy employed by FATFS is fairly straight-forward. It scans the FAT linearly, and uses the first free cluster found. Each scan starts from the position in the FAT where the previous scan ended, which results in all of the disk being used eventually, even if the

filesystem is never full. However, this strategy turns out to be too naive for our purpose: if several files are allocated concurrently, the files end up interleaved on the disk, resulting in high fragmentation levels even on a near-empty filesystem.

3.1.2 The LIMEFS filesystem

The *Large-file metadata-In-Memory Extent-based File System* (LIMEFS) was developed as a research venture within Philips Research. [7]

LIMEFS is extent-based, which means it keeps track of used and free blocks in the filesystem by maintaining lists of (*index, count*) pairs. Each pair is called an *extent*, and describes a set of *count* contiguous blocks starting at position *index* on the disk.

The allocation strategy LIMEFS uses is slightly more sophisticated than the strategy FAT incorporates, and turns out to be very effective in avoiding fragmentation when dealing with large files. When space needs to be allocated, LIMEFS first tries to allocate blocks in the free extent after the last written block of the current file. If there is no such extent, the list of free extents is scanned and an extent is chosen that is not preceded by an extent that contains the last block of another file that is currently open for writing. If it finds such an extent, it will start allocating blocks from the beginning of the extent. If it cannot find such an extent, it will pick an extent that *is* preceded by a file that is open for writing. In this case however, it will split the free extent in half and will only allocate from the second half. When the selected extent runs out of free blocks, another extent is selected using the approach just described, with the added notion that extents close to the original one are preferred over more distant ones.

3.2 How to count fragments

A hard drive reading data sequentially is able to transfer, on average, the amount of data present on one track in the time it takes the platter to rotate once. Reading more data than one track involves moving the head to the next track, which is time lost for reading. The layout of the data on the tracks is corrected for the track-to-track seek time of the hard drive, *i.e.*, the data is laid out in such a way that the first block on the next track is just passing underneath the head the moment it has moved from the previous track and is ready for reading. This way no additional time is left waiting for the right block. This is known as *track skew*, shown in figure 1. Hence, the effective transfer rate for sequential reading is the amount of data present on one track, divided by the rotation time increased with the track skew.

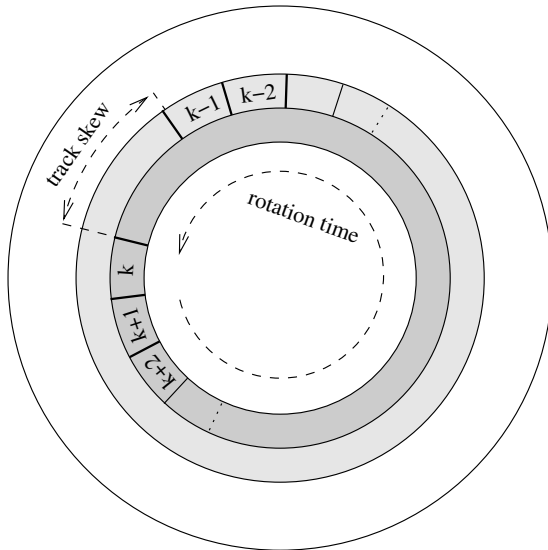


Figure 1: In the time it takes the head of the hard drive to move to the next track, the drive continues to rotate. The lay-out of blocks compensates for this. After reading block k , the head moves to the next track and arrives just in time to continue reading from block $k + 1$ onwards. This is called *track skew*.

When a request is issued for a block that is not on the current track, the head has to *seek* to the correct track. Subsequently, after the seek the drive has to wait until the correct block passes underneath the head, which takes on average half the rotation time of the disk. The time such a seek takes is time not spent reading, thus seeking lowers the data throughput of the drive. Seeks occur for example when blocks of a single non-contiguous file are read in sequence, *i.e.*, when moving from fragment to fragment. Some non-sequential blocks do not induce seeks however, and should not be counted as fragments.

Two consecutive blocks in a file that are not contiguously placed on the disk only lead to a seek if the seek time is lower than the time it would take to bridge the gap between the two blocks by just waiting for the block to turn up beneath the head (maybe after a track-to-track seek).

If t_s is the average time to do a full seek, and t_r is the rotation time of the disk, we can derive T_s , the access time resulting from a seek:

$$T_s = t_s + \frac{1}{2}t_r \quad (1)$$

The access time resulting from waiting, T_w , can be expressed in terms of track size s_t , gap size s_g and track skew t_{ts} :

$$T_w = \frac{s_g}{s_t}(t_r + t_{ts}) \quad (2)$$

So the maximum gap size S_g that does not induce a seek is the largest gap size s_g for which $T_w \leq T_s$ still holds, and therefore

$$S_g = s_t \frac{\frac{t_s}{t_r} + \frac{1}{2}}{1 + \frac{t_{ts}}{t_r}} \quad (3)$$

The relevance of the maximum gap size S_g is that it allows us to determine how many rotations and how many seeks it takes to read a particular file, given the layout of its blocks on the disk.

3.3 Fragmentation equilibrium

A small amount of fragmentation is not bad per se, if it is small enough to not significantly reduce the transfer speed of a hard drive. For instance, the ext3 filesystem [8] by default inserts one metadata block on every 1024 data blocks. Although, strictly speaking, this leads to non-sequential data and thus fragmentation, this kind of fragmentation does not impact the data rate significantly. If anything, it increases performance because the relevant metadata is always nearby.

A more important factor is the predictability and the stabilisation of the fragmentation level. Dimensioning the I/O subsystem for a certain application is only possible if the effective data transfer rate of the hard drive is known a priori, *i.e.*, predictable. As one should dimension a system for the worst case, it is also helpful if the fragmentation level has an upper bound, *i.e.*, it reaches a maximum at some point in time after which it does not deteriorate any further. With the help of some simple mathematics, we can estimate the theoretical prerequisites for such stabilisation.

3.3.1 Neighbouring blocks

Suppose we have a disk with a size of N allocation blocks and from these blocks, we make a random selection of M blocks. The first selected block will of course never be a neighbour of any previously selected blocks. The probability that the second block is a neighbour

of the first is $\frac{2}{N-1}$, because 2 of the remaining $N-1$ blocks are adjacent to the first block. The probability of the third block being a neighbour of one of the previous two is then $\frac{4}{N-2}$, or, more general, the probability of a block i being a neighbour of one of the previously selected blocks is:

$$P(i) = \frac{2i}{N-i} \quad (1 \leq i \ll N) \quad (4)$$

The average number of neighbouring blocks when randomly selecting M blocks, the expected value, can be determined by the summation of the probability of a selected block being a neighbour of a previously selected block over all blocks, although with two errors: we are not correcting for blocks at the beginning or end of the disk with only one neighbour and we are conveniently forgetting that two previously selected blocks could already be neighbours. As long as $N \gg 1$ and $M \ll N$ this will not influence the outcome very much and simplifies the formulae. Furthermore, if $M \ll N$ we can approximate (4) by:

$$P(i) \approx \frac{2i}{N} \quad (5)$$

The expected value of the number of neighbouring blocks is then, according to (5):

$$\begin{aligned} E &= \sum_{i=1}^{M-1} \frac{2i}{N-i} \\ &\approx \sum_{i=1}^{M-1} \frac{2i}{N} \\ &= \frac{(M-1)M}{N} \end{aligned} \quad (6)$$

3.3.2 Neighbouring fragments

The above holds for randomly allocated blocks, which is something not many filesystems do. The blocks in the above argumentation however can also be seen as fragments of both files as well as free space. This only changes the meaning of N and M and does not change the argumentation. If we now look at a moderately fragmented drive, the location of the fragments will be more or less random and our estimation of the expected value of the number of neighbouring fragments will be closer to the truth.

Furthermore, to have a stable amount of fragments, an equilibrium should exist between the number of new fragments created during allocation and the number of fragments eliminated during deletion (by deleting data next to a free fragment and thus 'glueing' two fragments).

Let us interpret N as the total number of fragments on a drive, both occupied as well as free and M as the number of free fragments. Also, assume that we are dealing with a drive that is already in use for some time and is almost full, according to our PVR scenario as described in section 2.1.

The fore mentioned equilibrium can be obtained with the simple allocation strategy of LIMEFS (described in section 3.1.2) used in our PVR scenario, by eliminating one fragment when deleting a file, as on average one fragment is created during the allocation of a file (as shown in figure 2). We assume that, as the files are on average of the same size, for each file created a file is deleted; when writing a file of L fragments (increasing the fragment count with one), a file of L fragments is deleted. This should eliminate one fragment to create a balance in the amount of fragments.

By deleting a file of L fragments on a drive with N total fragments, we increase the number of

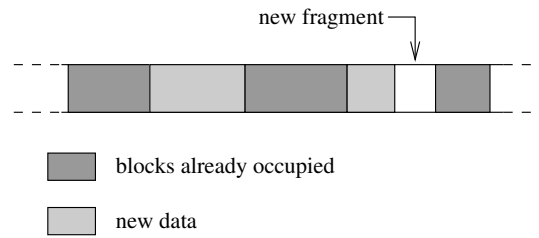


Figure 2: Empty fragments are filled (and thus not creating more fragmentation), until the end of the file. As an empty fragment is now divided into a fragment of the new file and a smaller empty fragment, one new fragment is created.

free fragments from $(M - L)$ to M . The number of neighbouring fragments n before deleting and n_d after deleting the file amongst the free fragments is, according to (6):

$$n = \frac{(M - L - 1)(M - L)}{N} \quad (7)$$

$$n_d = \frac{(M - 1)M}{N} \quad (8)$$

Combining (7) and (8) gets us the increase of neighbouring free fragments and thus the decrease of free fragments f (two neighbouring free fragments are one fragment):

$$\begin{aligned} f &= n_d - n \\ &= \frac{(M - 1)M}{N} - \frac{(M - L - 1)(M - L)}{N} \\ &= \frac{(2M - L - 1)L}{N} \end{aligned} \quad (9)$$

3.3.3 Balancing fragmentation

Now we define two practical, understandable parameters m for the fraction of the disk that is

free and s for the fraction of the disk occupied by a file:

$$m = \frac{M}{N} \quad (10)$$

$$s = \frac{L}{N} \quad (11)$$

This is mathematically not correct, as we defined N , M and L as distributions of an amount of fragments, but in the experiments of section 4 we will see that it works well enough for us, together with the numerous assumptions we already made.

To fulfil the demand that the number of eliminated fragments f when deleting a file of L fragments should equal the number of created fragments by allocating space for a file of an average equal size, $f = 1$ in (9). We combine this statement with (10) and (11):

$$L = \frac{1 + s}{2m - s} \quad (12)$$

This can be simplified even more by assuming $s \ll 1$, which is true if $L \ll N$. This is a realistic assumption, as the size of a file is most often a lot smaller than the size of the disk. We get:

$$L = \frac{1}{2m - s} \quad (13)$$

A remarkable result of this equation is that the average number of fragments in each of our files does not depend in the allocation unit size. Of course, the above only holds if files consist of more than one fragment, and a fragment consists of a fairly large number of allocation units. This fits our PVR scenario, but the deduced formulae do not apply to general workloads.

Another interesting result of (13) is the insight that a disk without free space besides the

amount needed to write the next file will result in a situation where $M = L$ and thus $m = s$. The average number of fragments in a file becomes $L = \frac{1}{s}$, meaning the number of fragments in a file equals the number of files on the disk.

4 The simulation

To test the validity of our theory described in section 3 on the one hand and to be able to assess if fragmentation is an issue for PVR systems on the other hand, we have conducted a number of simulations. First, we have done in-memory experiments with the simple allocation strategy of LIMEFS [7]. Next, we have tried simulating a real PVR system working on a filesystem as closely as possible with our program `pvrsim`. This was combined with our `hddfrgchk` to analyse the output of the simulation. Finally, we have done throughput and actual seek measurements by observing the block I/O layer of the Linux kernel with Jens Axboe's `blktrace` [9].

All experiments were performed on a Linux 2.6.15.3 kernel, only modified with the `blktrace` patches. The PVR simulations on actual filesystems were running on fairly recent Pentium IV machines with a 250 GB Western Digital hard drive. The actual speed of the machine and hard drive should not influence the results of the fragmentation experiments, and the performance measurements were only compared with those conducted on the same machine.

4.1 LIMEFS

We isolated the simple allocation strategy of LIMEFS described in section 3.1.2 from the rest of the filesystem and implemented it in a

simulation in user space. This way, the creation and deletion of files was performed by only modifying the in-memory meta data of the filesystem, and we were quickly able to investigate if our theory has practical value. The results of this experiment can be found in section 5.2.

4.2 `pvrsim`

As we clearly do not want to use a PVR for two years to see what the effects are of doing so, we have written a simulation program called `pvrsim`. This multi-threaded application writes and deletes files, similar in size to recorded broadcasts, as fast as possible. It is able to do so with a number of concurrent threads defined at runtime, to simulate the simultaneous recording of multiple streams.

The size of the generated file is uniformly distributed in the range from 500 MB to 5 GB. Besides that, every file is assigned a Gaussian distributed popularity score, ranging roughly from 10 to 100. This score is used to determine which file should be deleted to free up disk space for a new file.

When writing new files, `pvrsim` always keeps a minimum amount of free disk space to prevent excessive fragmentation. This dependency between fragmentation and free disk space was shown in (13) in section 3.3.3. If writing a new file would exceed this limit, older files are deleted until enough space has been freed. The file with the lowest weighed popularity is deleted first. The weighed popularity is determined by dividing the popularity by the logarithm of the age, where the age is expressed as the number of files created after the file at hand.

Blocks of 32 KB filled with zeroes are written to each file until it reaches the previously determined size. Next, the location of each block of

the file is looked up and the extents of the file are determined. The extents are written to a log file for further processing by `hddfrgchk`.

4.3 `hddfrgchk`

Our main simulation tool `pvrsim` outputs the extents of each created file, which need further processing to be able to analyse the results properly. This processing is done by `hddfrgchk`, which provides two separate functions, described below.

4.3.1 Fragmentation measures

`hddfrgchk` is able to calculate a number of variables from the extent lists. First, it determines the number of fragments of which the file at hand consists. As explained in section 3.2, this number is often higher than the actual seeks the hard drive has to do when reading the file. We therefore calculate the theoretical number of seeks, based on the characteristics of a typical hard drive, with (3) from section 3.2. The exact parameters in this calculation were determined from a typical hard drive by measurements, as described in [10].

Furthermore, we have defined a measure for the relative effective data transfer speed. The minimum number of rotations the drive has to make to transfer all data of a file if all its blocks were contiguous can be calculated by dividing the number of blocks of the file by the average number of blocks on a track. The actual number of rotations the drive theoretically has to make to transfer the data can also be calculated. This is done by adding the blocks in the gaps that were not counted as fragments in our earlier calculations to the total amount of blocks in the file. Furthermore, the number of rotations that took place in the time the hard drive

was seeking (the number of theoretical seeks as calculated earlier is used for this) is also added to the estimation of the number of rotations the drive has to make to transfer the file. Dividing the minimum number of rotations by the estimation of the actual number of rotations gives us the relative transfer speed.

4.3.2 Filesystem lay-out

Besides these variables, `hddfrgchk` also generates a graphical representation of the simulation over time. The filesystem is depicted by an image, each pixel representing a number of blocks. With each file written by the simulator, the blocks belonging to that file are given a separate colour. When the file is deleted, this is also updated in the image of the filesystem.

A new picture of the state of the filesystem is generated for every file, and the separate pictures are combined into an animation. The animation gives a visualisation of the locations of the files on the drive, and gives an insight on how the filesystem evolves.

4.4 Performance measurements

To verify the theoretical calculations of `hddfrgchk`, we also have done some measurements. We have looked at the requests issued to the block I/O device (the hard drive in this case) by the block I/O layer. The `blktrace` [9] patch by Jens Axboe provides useful instrumentation for this purpose in the kernel.

4.4.1 `blktrace`

The kernel-side mechanism collects request queue operations. The user space utility

`blktrace` extracts those event traces via the Relay filesystem (RelayFS) [11]. The event traces are stored in a raw data format, to ensure fast processing. The `blkparse` utility produces formatted output of these traces afterwards, and generates statistics.

The events that are collected originate either from the file system or are SCSI commands. The filesystem block layer requests consist of the read or write actions of the filesystem. These actions are queued and inserted in the internal I/O scheduler queue. The requests might be merged with other items in the queue, at the discretion of the I/O scheduler. Subsequently, they are issued to the block device driver, which finally signals when a specific request is completed.

4.4.2 Deriving the number of seeks

All requests also include the Logical Block Address (LBA) of the starting sector of the request, as well as the size (in sectors) of the request. As we are interested in the exact actions the hard drive performs, we only look at the requests that are reported to be completed by the block device driver, along with their location and size. With this information, we can count the number of seeks the hard drive has made: if the starting location of a request is equal to the ending location of the previous request, no seek will take place. This does not yet account for the fact that small gaps might not induce seeks, but do lower the transfer rate.

4.4.3 Determining the data transfer rate

The effective data transfer rate can be derived by the information provided by `blktrace`, but can also be calculated just by the wall clock time needed to read a file, divided by the file

size. Comparing transfer rates of various files should be done with caution: the physical location on the drive significantly influences this. To have a fair comparison, the average transfer rate over the whole drive should be compared at various stages in the simulation.

5 The results

We have conducted a number of variations of the simulations described in section 4. The variables under consideration were the filesystem on which the simulations were taking place, the size of the filesystem, the minimum amount of free space on the filesystem, the length of the simulation (i.e., the number of files created) and the number of concurrent files being written.

5.1 Simulation parameters

With exploratory simulations we discovered that the size of the filesystem is not of significant influence on the outcome of the experiments, as long as the filesystem is sufficiently large compared to both the block size and the average file size. As typical PVR systems typically offer storage space ranging from 100 GB to 300 GB, we decided on a filesystem size of 138 GB. Due to circumstances, however, some of the experiments were conducted on a filesystem of 100 GB.

The minimum amount of space that is always kept free is in the simulations with `pvrsim` fixed at 5% of the capacity of the drive. According to the preliminary in-memory LIMEFS experiments this is a reasonable value. The results of these experiments are elaborated in section 5.2.

The size of the created files is chosen randomly between 500 MB and 5 GB, uniformly distributed. As explained in section 2.1, these are typical file sizes for a PVR recording MPEG2 streams in Standard Definition (SD) resolution.

The length of the experiments, expressed in number of files created, was initially set at 10,000. The results from these runs showed that after about 2,500 files the behaviour stabilised. Therefore, the length of the simulations was set at 2,500 files.

We have done simulations with up to four simultaneous threads, so several files were written to the disk concurrently. This was done to observe the behaviour when recording multiple simultaneous broadcasts.

The filesystems we have covered in the experiments described in this paper are FAT (both Linux and Microsoft Windows), ext3 [2] [8], ReiserFS [12], LIMEFS and NTFS [13] (Microsoft Windows). We plan to cover more filesystems.

5.2 LIMEFS in-memory simulation

The results of the simulation of LIMEFS as described in section 4.1 are shown in figure 3. We have run our in-memory simulation on an imaginary 250 GB hard drive, with the minimum amount of free space as a variable parameter.

As can be seen, in the runs with 5%, 10% and 20% free space the fragmentation stabilises quickly. In longer runs we observed that the 0%, 1% and 2% options also stabilise, but the final fragmentation count is much higher and the stabilisation takes longer. The sweet spot appears to be 5% minimum free space, as this gives a good balance between fragmentation and hard drive space usage.

A nice result from this experiment is that the observed fragmentation counts fit our formula.

We have taken a filesystem of 250 GB and an average file size of 2750 MB. For the 5% free space run, this makes the fraction of free space $m = 0.05$ (see section 3.3.3). The fraction of the disk occupied by a file is $s = \frac{L}{N} = \frac{2.75}{250} = 0.01$. So, the number of fragments in a file on average, according to the formula, is:

$$L = \frac{1}{2m - s} = \frac{1}{2 \cdot 0.05 - 0.01} \approx 11$$

From the plot in figure 3, we can see the calculation matches the outcome of the simulation. The same holds for the other values for the minimum amount of free space.

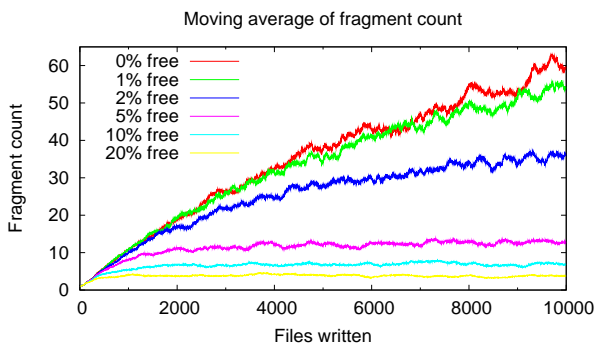


Figure 3: The average fragment count during a simulation run of 10,000 files, with a variable percentage of space that was kept free.

5.3 Fragmentation simulation

While the name `pvrsim` might suggest otherwise, we must stress that *all* results obtained by `pvrsim` were obtained by writing *real* files to *real* filesystems. The filesystems were running on their native platforms (*i.e.*, FAT and NTFS on Windows XP SP2, the others on Linux 2.6.15). For an interesting comparison however, we have also tested how the Linux

version of FAT performs in a number of situations.

These experiments resulted in the plots in figure 4, where the number of seeks according to our theory (see section 3.2) and the relative speed are shown for single- and multi-threaded situations.

5.3.1 Single-threaded performance

All single-threaded simulations show similar results on all filesystems: the effective read speed is not severely impacted by writing many large files in succession. Some filesystems handle the fragmentation more gracefully than others, but the effects on system performance are negligible in all cases, as can be seen in the top two plots of figure 4. Although ext3 seems to have quite some fragmentation, the relative speed does not suffer: 98% of the raw data throughput is a hardly noticeable slowdown.

5.3.2 Multi-threaded performance

The multi-threaded simulations show that a file allocation strategy that tries to cluster files that are created concurrently performs considerably better compared to one that does not. The performance of NTFS deteriorates very quickly (after having written only a couple of files) to a relative speed of around 0.6, while the relative speed of ReiserFS and ext3 do not drop below 0.8. Linux FAT is doing slightly worse, while LIMEFS is not impacted at all.

5.3.3 LIMEFS

According to our results, the area of PVR applications is one where LIMEFS really shines. LIMEFS never produces more than around ten

fragments, even with four threads. We do admit that LIMEFS is the only filesystem used in this experiment that was designed specifically for the purpose of storing PVR recordings, and that it might perform horribly or even not at all in other areas. However, the results are encouraging, and will hopefully serve as an inspiration for other filesystems.

5.3.4 FAT and NTFS

One interesting result of running the simulation on FAT and NTFS on Windows is that Windows appears to allocate 1 megabyte chunks regardless of the block size (extents are typically 16 clusters of 64k, or 32 clusters of 32k). As our simulation does not produce files smaller than 1 megabyte, we have no way of determining the effect of small files on the fragmentation levels of the filesystems. However, the chunked allocation seems to alleviate the negative effects of the otherwise quite naive allocation strategies of FAT and NTFS.

5.4 Seeks and throughput

We have measured the the number of the average data rate of files on a newly created filesystem and compared that with the data rate of files after `pvrsim` simulated a PVR workload, to confirm that our relative speed calculations are representative for the actual situation. Furthermore, we have analysed the activity in the block I/O layer with `blktrace` to see if the number of seeks derived from the placement of the blocks on the drive can be used to estimate the real activity.

The raw data throughput of the drive on which we have executed our single-threaded `ext3` run was 60,590 KB/s. After writing 10,000 files with `pvrsim`, the data throughput while reading those files was on average 58,409 KB/s.

This results in a relative speed of 0.96, which is close to the 0.95 we have estimated with our calculations. The figures of the two-threaded `ext3` run on a different machine (52,520 KB/s raw data throughput, 40,270 KB/s while reading the final files present and thus a relative speed of 0.77, the same as calculated) confirm this.

With the use of `blktrace` we counted 10,267 seeks when reading the final files present on the disk after the single-threaded `ext3` run. This is an average of 366 fragments per file. If we take into account that small fragments do not cause the drive to seek, as explained in section 3.2, the number of seeks caused by fragments after a gap of more than 676KB was, again according to the `blktrace` observations, 5692 or an average of 203 seeks per file. This is for all practical purposes close enough to the 198 seeks we derive from the location of the fragments on the disk.

6 Future work

The experiments and results presented in this paper are a starting point to improve the fragmentation robustness of filesystem for PVR-like scenarios. To obtain a good overview of the current state-of-the-art, we are planning to run our simulations on other filesystems, *e.g.*, on XFS [14]. We intend to cover more combinations of the parameters of our simulations as well, *e.g.*, different distributions for file size and popularity, different filesystem sizes, and different filesystem options.

Following this route, experimental measurements of different workloads and scenarios might provide interesting insights as well. We feel our tools could easily be modified to incorporate other synthetic workloads, and therefore be of great help for further experiments.

A more practical matter is improving the allocation strategy of the FAT filesystem. Making the allocation extent-based and multi-stream aware like LIMEFS will greatly improve the fragmentation behaviour, while the resulting filesystem will remain backwards compatible. On one hand, this proves our LIMEFS strategies in real-life applications, while on the other hand this will be useful for incorporation in mobile digital television devices, which might also act as a mass storage device and should therefore use a compatible filesystem. Unfortunately, the usefulness of FAT in a PVR context remains limited due to its file size limit of 4 gigabytes.

7 Conclusion

The formulae derived in 3 give an indication of the average fragmentation level for simple allocation strategies and large files. Although we made quite some assumptions and took some shortcuts in the mathematical justification, the results of the LIMEFS in-memory experiments of section 5.2 support the theory. Another useful outcome of the formulae is that, at least with large files, the fragmentation level stabilises, which seems to be true as well for filesystems with more sophisticated allocation strategies.

When dealing only with large files, a simple allocation strategy seems very efficient in terms of fragmentation prevention. Especially if only one stream is written, even FAT performs very well. Writing multiple streams simultaneously requires some precautions, but a strategy as implemented in LIMEFS suffices and outperforms all more complicated strategies with respect to the fragmentation level.

The ext3 and ReiserFS filesystems have a relatively high fragmentation in our scenario. However, the fragmentation stabilises and the im-

pact is therefore predictable, and is no real issue with large files. A file of 2 GB consisting of 500 fragments will result in 4.5 seconds of seeking (with an average seek time of 9 ms). This is not significant for a movie of two hours, if the seeks are not clustered.

The relative speeds measured with ReiserFS and ext3 are not as good as the ones of LIMEFS, but still acceptable: 80% of the performance after prolonged use. NTFS however perform horribly when using multiple simultaneous streams. The Linux version of FAT is doing surprisingly well with two concurrent streams, much better than the Microsoft Windows implementation. We have still to investigate why this is.

An interesting observation is the fact that ext3 keeps about 2% of unused free space at the end of the drive, independent of the "reserved space" options (used to prevent the filesystem from being filled up by a normal user). If this free space is kept clustered at the end instead of being used throughout the simulation, this is inefficient in terms of fragmentation, as our formulae tell us.

In general, a PVR-like device is able to provide sustainable I/O performance over time if a filesystem like ext3 or ReiserFS is used. This does not assert anything about scenarios where file sizes are in the order of magnitude of the size of an allocation unit. However, the ratio between rotation time and seek time in modern hard drives is such that seeks are not something to avoid at all costs anymore. For optimal usage of the hard drive under a load of a number of concurrent streams, an allocation strategy that is aware of such a scenario is needed.

References

- [1] Microsoft Corporation. *Microsoft Extensible Firmware Initiative FAT32*

- File System Specification*. Whitepaper, December 2000.
<http://www.microsoft.com/whdc/system/platform/firmware/fatgendown.msp?>
- [2] Card, Rémy; Ts'o, Theodore; Tweedie, Stephen. *Design and Implementation of the Second Extended Filesystem*. Proceedings of the First Dutch International Symposium on Linux, 1994.
<http://web.mit.edu/tytso/www/linux/ext2intro.html>
- [3] MythTV. <http://www.mythtv.org>
- [4] Microsoft Windows XP Media Center. <http://www.microsoft.com/windowsxp/mediacenter/default.msp?>
- [5] Mesut, Özcan; Brink, Benno van den; Blijlevens, Jennifer; Bos, Eric; Nijs, Giel de. *Hard Disk Drive Power Management for Multi-stream Applications*. Proceedings of the International Workshop on Software Support for Portable Storage, March 2005.
- [6] Nijs, Giel de; Almesberger, Werner; Brink, Benno van den. *Active Block I/O Scheduling System (ABISS)*. Proceedings of the Linux Symposium, vol. 1, pp. 109–126, Ottawa, July 2005.
http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf
- [7] Springer, Rink. *Time is of the Essence: Implementation of the LimeFS Realtime Linux Filesystem*. Graduation Report, Fontys University of Applied Sciences, Eindhoven, 2005.
- [8] Johnson, Michael K. *Red Hat's New Journaling File System: ext3*. Whitepaper, 2001. <http://www.redhat.com/support/wpapers/redhat/ext3/>
- [9] Axboe, Jens; Brunelle, Alan D. *blktrace User Guide*. <http://www.kernel.org/pub/linux/kernel/people/axboe/blktrace/>
- [10] Mesut, Özcan; Lambert, Niek. *HDD Characterization for A/V Streaming Applications*. IEEE Transactions on Consumer Electronics, Vol. 48, No. 3, 802–807, August 2002.
- [11] Dagenais, Michel; Moore, Richard; Wisniewski, Bob; Yaghmour, Karim; Zanussi, Tom. *RelayFS - A High-Speed Data Relay Filesystem*. <http://relayfs.sourceforge.net/relayfs.txt>
- [12] Reiser, Hans. *ReiserFS v.3 Whitepaper*. Whitepaper, 2003.
- [13] Microsoft Corporation. *Local File Systems for Windows*. WinHEC, May 2004. <http://www.microsoft.com/whdc/device/storage/LocFileSys.msp?>
- [14] Hellwig, Chrisoph. *XFS for Linux*. UKUUG, July 2003. <http://oss.sgi.com/projects/xfs/papers/ukuug2003.pdf>

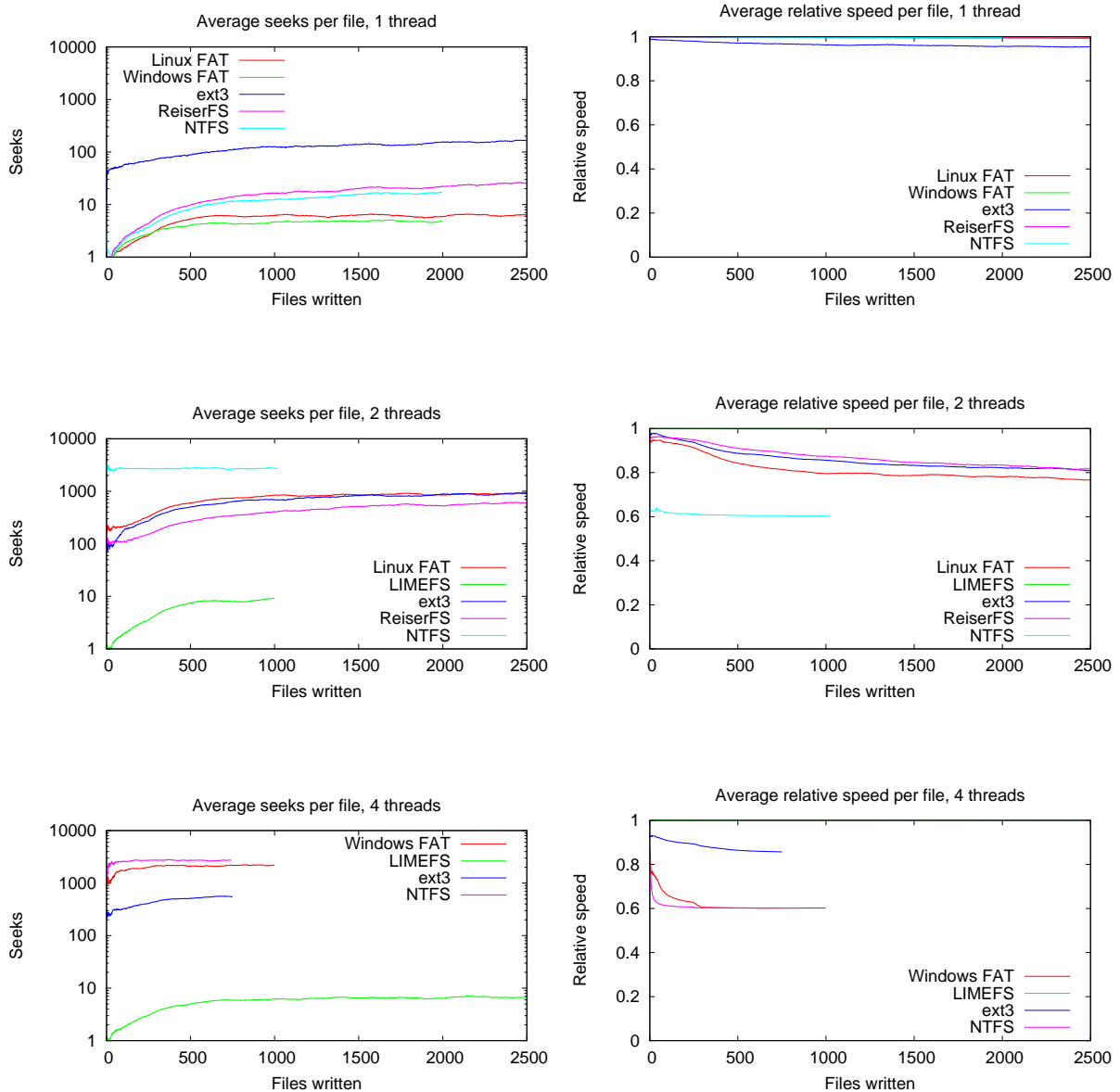


Figure 4: Results of `pvrsim` run on various filesystems. From top to bottom the number of concurrent threads was respectively one, two and four. The plots on the left side are the average amount of fragments over time, corrected to exclude small fragments as described in section 3.2. On the right side the relative speed (see section 4.3) is shown.

Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.