The following paper was originally published in the
Proceedings of the USENIX 1996 Annual Technical Conference
San Diego, California, January 1996

A Comparison of FFS Disk Allocation Policies

Keith A. Smith and Margo Seltzer
Harvard University

A Comparison of FFS Disk Allocation Policies

Keith A. Smith and Margo Seltzer
Harvard University

Abstract

The 4.4BSD file system includes a new algorithm for allocating disk blocks to files. The goal of this algorithm is to improve file clustering, increasing the amount of sequential I/O when reading or writing files, thereby improving file system performance. In this paper we study the effectiveness of this algorithm at reducing file system fragmentation. We have created a program that artificially ages a file system by replaying a workload similar to that experienced by a real file system. We used this program to evaluate the effectiveness of the new disk allocation algorithm by replaying ten months of activity on two file systems that differed only in the disk allocation algorithms that they used. At the end of the ten month simulation, the file system using the new allocation algorithm had approximately half the fragmentation of a similarly aged file system that used the traditional disk allocation algorithm. Measuring the performance difference between the two file systems by reading and writing the same set of files on the two systems showed that this decrease in fragmentation improved file write throughput by 20% and read throughput by 32%. In certain test cases, the new allocation algorithm provided a performance improvement of greater than 50%.

1 Introduction

Recent file systems [Peacock88][McVoy90] have used *clustering* to improve performance. These systems attempt to place logically sequential file data on physically contiguous disk blocks. When such layout is achieved, large multiple block transfers can be used to read/write files at close to the disk system's maximum bandwidth. Measurements have shown that these clustering enhancements can improve performance by a factor of two or three [McVoy90] [Seltzer93] over file systems that perform I/O one block at a time. These performance measurements were made on empty file systems and represent the best case behavior for clustering file systems. Over time, clustering is less successful, because free space becomes fragmented and the disk allocation algorithms fail to fully exploit existing free clusters. A recent study showed that UNIX file systems that are more than two years old perform as much as 15% worse than comparable empty file systems [Seltzer95]. This decline in performance correlates closely with increased fragmentation of newly created files on these file systems. Although the maximum transfer size on these file systems was seven file system blocks, the average cluster size for mid-sized files (32–128KB) was only three blocks.

As file systems age, clustered file allocation becomes less successful because the file system is unable to find clusters of free space from which to make allocations. This may occur either because free space becomes too fragmented to support clustering, or because the file system does not fully exploit existing clusters of free space when allocating space for new files. An examination of the file systems on several file servers at Harvard showed that there are many large clusters of free space on UNIX file systems that are two to three years old [Smith94]. We conclude from this that the file fragmentation observed on real files systems is the result of a disk allocation algorithm that is unable to find and exploit existing clusters of free space.

In the hopes of providing better long term clustering, Kirk McKusick modified the disk allocation algorithm used by the 4.4BSD-Lite Fast File System (FFS) to better exploit existing clusters of free space [CSRG94].

In order to understand the long term effectiveness of this new allocation algorithm, we have developed a tool that simulates a ten month work load in order to artificially age a file system. We used this tool to age two different file systems, one that used the original disk allocation algorithm and one that used the new allocation algorithm. By tracking the amount of file fragmentation during the course of the simulation, we compared the effectiveness of the two disk algorithms. We also compared the performance of the resulting aged file systems to understand the impact of the resulting differences in fragmentation.

This research was supported by Sun Microsystems Laboratories and by the National Science Foundation under grant CCR-9502156.

In Section 2, we describe the disk allocation algorithm that has traditionally been used by the UNIX Fast File System and explain the improvements offered by the new algorithm. Section 3 explains the file system aging process, including the methodology used to generate the aging workload and a validation of the aging program by comparing its results to a real file system. In Section 4, we compare the file fragmentation that results from the original FFS allocation algorithm and the new allocation algorithm. Section 5 provides a performance comparison of the aged file systems. Section 6 discusses some future research directions based on the results of this work. Section 7 summarizes this study.

2 FFS Disk Allocation Algorithms

A simplified explanation of the original FFS disk allocation algorithm is presented here. A more detailed explanation may be found in *The Design and Implementation of the 4.3BSD UNIX Operating System* [Leffler89].

FFS divides the disk into *cylinder groups,* each of which is a set of consecutive cylinders. Cylinder groups are used to exploit locality; related data are co-located in the same cylinder group. Thus FFS allocates logically sequential blocks of a file in the same cylinder group, and likewise allocates all of the files in a directory to the same cylinder group as the directory.

The FFS disk allocation policy is divided into two steps. When a new block is allocated to a file, FFS first determines the cylinder group from which the block will be allocated. FFS then selects a free block from that cylinder group and allocates it. Selecting a cylinder group is a simple task; FFS uses the cylinder group where the previous block(s) of the file are located[1]. In this paper, we focus on the second part of the allocation—selecting a block from within a cylinder group.

The original FFS disk allocation algorithm allocates one block at a time to a file, attempting to allocate contiguous blocks where possible. When a new block is allocated, FFS determines the location of the previous block of the file and attempts to allocate the next disk block. If this block is not available, FFS allocates a different block from the cylinder group, attempting to find one that minimizes the seek time from the previous block of the file. The selection of this alternate block does *not* consider the amount of free space where the new block is located. Thus if there is just one free block in a good location and a cluster of ten free blocks in a slightly worse location, FFS will allocate the single free block, making it impossible to perform contiguous allocation after that block.

McKusick's new allocation algorithm adds a reallocation step to the original FFS disk allocation algorithm. For this reason it is referred to as the *realloc* algorithm. FFS initially allocates blocks in the manner described above. Before the blocks are written to disk, however, the reallocation code gathers clusters of logically sequential blocks and tries to relocate them to free clusters of the appropriate size. The maximum size cluster produced by the realloc code is determined by a file system parameter, and is usually configured to be the maximum I/O transfer size of the underlying disk system.

3 File System Aging

To understand the effectiveness of a disk allocation algorithm, the long term effect of the algorithm on file system layout must be examined. To do this in a laboratory setting, we generated an artificial load intended to simulate the pattern of file operations that a file system sees over an extended period of time. This process is called *file system aging.* After aging a file system, the layout of its files can be analyzed and compared with similarly aged file systems that used different allocation algorithms.

3.1 Generating a Workload

The central problem in aging a file system is generating a realistic workload. Because a test system is likely to start with an empty disk, this workload should start with an empty file system and simulate the load on a new file system over many months or years. The ideal method for generating this workload would be to collect extended file system traces and to age a test file system by replaying the exact set of file operations seen in the trace. The duration of the required traces makes this strategy impractical. Instead, we generated a workload from two sets of file system data that were readily available. Fortunately, an exact reproduction of the load on a real file system is not required.

We used a set of file system *snapshots* collected from a file system on a local file server to simulate the day-to-day changes on a file system. These snapshots, which were originally collected for a study of file system fragmentation [Smith94], were collected

[1] To prevent one large file from filling an entire cylinder group, each time an indirect block is allocated to a file, allocation changes to a different cylinder group.

nightly over a period of one year. Each snapshot describes all of the files on a file system at the time of the snapshot. For each file the snapshot includes the file's inode number, inode change time, file type, file size, and a list of the disk blocks allocated to the file.

By comparing successive snapshots of one file system, we generated a list of the files that were created, deleted, or modified on each day. In order to simulate the activity on an empty file system, we chose a file system that was nearly empty at one point in the snapshot collection period, using the point of lowest utilization (9% full) as the starting time.

The major obstacle to accurately reproducing the original workload from the file system snapshots was interpolating the file system activity that occurred between successive snapshots. By comparing the list of allocated inodes in two snapshots, it was easy to determine which files were created, deleted, or modified during the intervening interval. Unfortunately, the snapshots did not provide sufficient information to determine the exact time at which these operations took place.

We used several heuristics to assign creation and deletion times to these file operations. Previous studies have shown that files are seldom modified after they are first written [Ousterhout85]. Therefore, when a new file was created, we used its inode change time as the time the file was created. Similarly, if a file was modified, we assumed that it had been removed (or truncated to zero length) and then rewritten. The most difficult operations to which to assign times were deletes. When a file was deleted between two snapshots, there was no information that provided hints about the time it was deleted. We randomly assigned times to file deletions, making sure that they fell during the range of times that other operations were occurring on the file system.

Another difficulty in recreating the file system's workload from the daily snapshots was accounting for files that were created and then deleted on the same day. Trace-based file system studies [Ousterhout85] [Baker91] have shown that most files live for less than the twenty-four hours between successive snapshots. These files, which did not show up in the snapshots, can affect the fragmentation of the longer-lived files on the file system. To approximate the additional file creations and deletions generated by these short-lived files, we used multiple-day traces of NFS requests to Network Appliance file servers [Hitz94]. This data, which was originally used in a study of cleaning algorithms for log-structured file systems [Blackwell95], includes all of the create, delete, and write requests issued to the servers during the trace periods.

We generated a list of all of the files created and deleted during each 24-hour period in the NFS traces. These files were sorted by the day they were created and the directory in which they were created (the directory information is available in the create requests). The result was a trace log describing all of the files that were created and then deleted on the same day.

The next step was to integrate these short-lived files into the workload generated from the file system snapshots. For each day in the snapshot period, we randomly selected one day from the NFS traces, and integrated that day's short-lived file activity into the aging workload. The file operations from the NFS traces were placed in the directories that had the most changes between snapshots. To ensure that the NFS operations overlapped with the file operations generated from the snapshots, all of the NFS operations in each directory were time-shifted to coincide with the peak period of activity in the directory to which they were added. The end result was that the operations on short-lived files (generated from the NFS traces) were interleaved with the creations and deletions of longer-lived files (generated from the snapshots).

The resulting workload simulates ten months of activity (from April, 1994 through February, 1995) on a 502 megabyte file system. The source file system is used for the home directories of one professor and three students in a networking research group. At the beginning of the ten month period, the file system was 9% utilized, and for most of the ten month period utilization was greater than 70%, reaching a high of 90%[2]. The workload contains approximately 800,000 file operations that write 48.6 gigabytes of data to the disk and take fourteen hours to replay on our test machine.

3.2 Replaying the Workload

To age a file system, we applied the workload described above to an empty file system. The aging program reads records from the workload file, performing the specified file operations. This task was complicated by the fact that complete pathnames for the created files were not available in the snapshots used to generate the workload. Because FFS assigns files to cylinder groups based on the cylinder group of the file's directory, the algorithm used by the aging

[2] These utilization numbers treat FFS's free space reserve (10% of the disk) as free space.

CPU Parameters		Disk Parameters		File System Parameters	
CPU	Intel Pentium	Disk Controller	Bustek 946C (SCSI)	Size	502 MB
Clock Speed	120 MHz	Disk Type	Seagate 32430N	Fragment Size	1 KB
Memory	64 MB	Total Disk Space	2.1 GB	Block Size	8 KB
Bus Type	PCI	Rotational Speed	5411 RPM	Max. Cluster Size	56 KB
		Sector Size	512 Bytes	Rotational Gap	0
		Cylinders	3992	Cylinder Groups	27
		Heads	9	*Heads*	*22*
		Average Sectors per Track	116	*Sectors per Track*	*118*
		Track Buffer	512 KB		
		Average Seek	11 ms		

Table 1: Benchmark Configuration. This table describes the hardware configuration used for benchmarking and for verifying the file system aging workload. The file system parameters shown in italics were set to match the file system from which the aging workload was generated despite the fact that they do not match the underlying hardware.

program to assign files to directories can have a major impact on the accuracy of the aging simulation.

In the absence of the original pathnames in the file system snapshots, we decided to simply create the files in the correct cylinder groups. Cylinder groups represent the pools from which disk blocks are allocated. By creating files in the same cylinder group on the simulated file system as on the original file system, we ensured that each cylinder group on the simulated file system saw the same set of allocation and deallocation requests that were presented to the corresponding cylinder group on the original file system. We used each file's inode number to compute the cylinder group to which it was allocated on the original file system. To force the files into the same cylinder groups on the aged file system, we exploited several details of the FFS implementation.

We started the aging process with an empty file system. The first step was to create one directory for each cylinder group on the file system. The algorithm used by FFS to assign directories to cylinder groups ensures that each directory was placed in a different cylinder group. For each file in the aging workload, we used its inode number to compute the cylinder group to which it was allocated on the original file system, and placed the file in the corresponding directory on the aged file system. Because FFS places all files in the same cylinder group as their directory, this guaranteed that all of the files that were in the same cylinder group on the original file system were also in the same cylinder group on the aged file system. Thus, the sequence of block allocation and freeing operations in each cylinder group was the same as on the original file system.

This approach does have one minor disadvantage. By creating an extra directory for each cylinder group, we are introducing one file per cylinder group that did not exist in any of the data sets used to generate the aging workload (i.e., the directory). The effect of these extra directories is negligable, however, since the space they occupy (approximately 300 kilobytes) represents less then 0.1% of the total disk utilization during the aging simulation.

3.3 Verifying the Aging Process

To verify the accuracy of the aging process, we compared the file fragmentation on an artificially aged file system with the fragmentation on the original file system that was used to generate the aging workload. We define a *layout score* to quantify the amount of file fragmentation in a file or file system. The layout score for an individual file is the fraction of that file's blocks that are optimally allocated. An optimally allocated block is one that is physically contiguous with the previous block of the same file. The first block of a file is not included in this calculation, since it is impossible for it to have a "previous block." Similarly, layout score is undefined for one block files, since they cannot be fragmented. A file with a layout score 1.00 is perfectly allocated; all of its blocks are contiguously allocated. A file with a layout score of 0.00 has no contiguously allocated blocks.

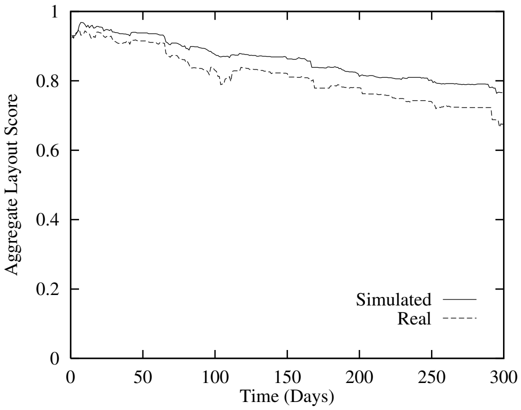

Figure 1: Aggregate Layout Score Over Time: Real vs. Simulated File Systems. This chart plots the aggregate layout score for each day in the ten month simulation period. The "Simulated" line shows the fragmentation on the artificially aged file system. The "Real" line shows the fragmentation on the original file system from which the aging workload was generated.

To evaluate the fragmentation of all of the files on a file system, we compute the file system's *aggregate layout score*. This metric is the fraction of the file system's allocated blocks that are optimally allocated (again ignoring the first block of each file and one block files).

To verify the accuracy of the aging process, we constructed a file system with the same parameters as the file system from which the aging workload was generated. These parameters, along with our hardware configuration, are summarized in Table 1. We used BSD/OS Version 2.0.1 for these and all of the other measurements in this paper. We then ran the aging workload on this file system, computing the file system's aggregate layout score at the end of each simulated day in the workload. For comparison, we also computed the aggregate layout score on the original file system for each day during the period from which the aging workload was generated. The resulting layout scores for the two file systems are plotted in Figure 1.

The simulated file system has higher layout scores than the original file system, indicating that the aging process does not cause as much file fragmentation as the original file system experienced. At the end of the ten month period, the simulated file system's aggregate layout score was 0.77, compared to the 0.68 aggregate layout score of the original file system. Despite the greater fragmentation on the original file system, the two file systems exhibit comparable behavior, as can be seen by the similar contours of the two curves in Figure 1; the simulated file system has many of the same drops and jumps as the original, although they are of smaller magnitude. There are also some areas where the simulated file system has failed to capture changes that occurred on the original. The clearest example of this is in days 90–110 of the simulation, where the aggregate layout score of the original file system changes from day to day while the layout score of the simulated file system remains relatively constant.

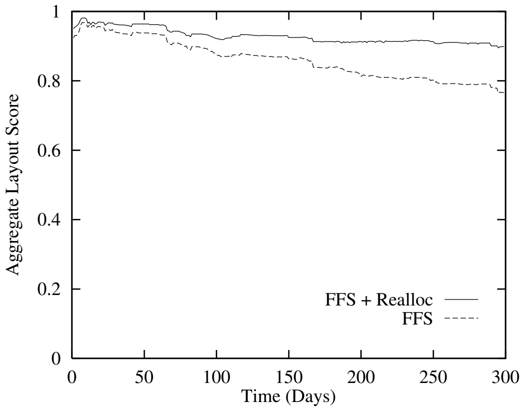

Figure 2: Aggregate Layout Score Over Time: FFS vs. realloc algorithm. This chart plots the aggregate layout score on each day of the ten month simulation period. The "FFS" line shows the aggregate layout scores on the file system that used the original FFS disk allocation algorithm. The "FFS + Realloc" line shows the aggregate layout scores on the file system that used the new realloc allocation algorithm.

The difference in fragmentation between the two file systems is the result inaccuracies in the aging workload. Section 3.1 discussed the approximations that were necessary due to the incomplete information contained in the file system snapshots used to generate the workload. Despite these differences, the aging workload is realistic in many ways. As on the real file system, the layout score of the artificially aged file system decreases steadily over time, occasionally declining quickly for a day or two, sometimes remaining almost constant for many weeks.

4 Comparison of Allocation Algorithms

To compare the FFS disk allocation algorithm to the realloc algorithm, we aged two file systems that differed only in the disk allocation algorithm that they used. These tests were also performed on the hardware configuration described in Table 1. After each simulated day during the aging, we computed the aggregate layout score for the two file systems. The results are plotted in Figure 2.

The file system that used the realloc allocation algorithm exhibited less fragmentation (higher layout scores) for the entire duration of the 300 day

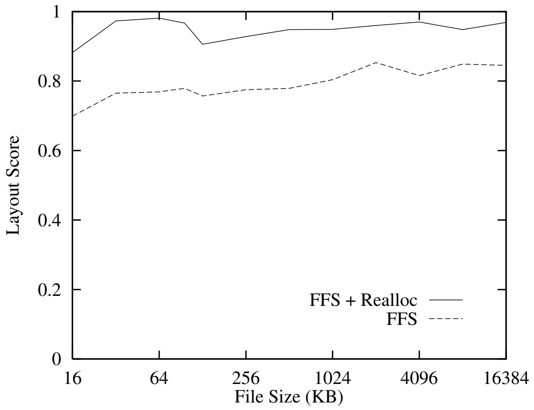

Figure 3: Layout Score as a Function of File Size. The "FFS" line was generated from the aged file system that used the original FFS allocation algorithm. The "FFS + Realloc" was generated from the aged file system that used the realloc enhancements.

simulation. The difference in aggregate layout score between the two file systems increased over time, from a difference of 0.026 (0.950 vs. 0.924) after the first day of the simulation, to a difference of 0.133 (0.899 vs. 0.766) at the end of the simulation. In other words, by the end of the simulation only 10.1% of the file blocks were non-optimally allocated when using the realloc algorithm, in contrast to 23.4% when the realloc code was not used—an improvement of 56.8%.

To understand the types of files that derive the most benefit from the realloc algorithm, we sorted the files on both file systems by size, and computed the aggregate layout score for files of a variety of sizes. The results are shown in Figure 3. This graph shows that the realloc disk allocation algorithm produces better file layout (i.e., less fragmentation) for all file sizes, and near optimal layout for files smaller than the file system cluster size. Surprisingly, two block files have a lower layout score than slightly larger files when the realloc algorithm is used. This is due to a quirk in the disk allocation code, which does not invoke the realloc functionality until a file fills the second block. The lower layout score for two block files in Figure 3 is the result of files that are big enough to use two blocks (instead of one block and some number of fragments) but do not completely fill the second block.

Both file systems in Figure 3 exhibit a drop in layout score when file size passes twelve blocks (96 KB). Files larger than twelve blocks require an indirect block, which is always allocated in a different cylinder group than the first part of the file. The result is that all files of more than twelve blocks contain at least one non-optimal block (the thirteenth), lowering the average layout score for these files. As the file size grows past thirteen blocks, the effect of this mandatory seek becomes smaller, and the layout score rises again.

5 Performance

We expected the decreased fragmentation seen when using the realloc algorithm to lead to better file system performance. In this section we present the results of several benchmarks that quantify the performance difference between file systems using the two allocation algorithms.

The most important difference in the two disk allocation algorithms is the long-term effect that they have on file layout. To account for this in benchmarking the two FFS implementations, we ran all of our benchmarks on file systems that were aged using the ten month aging workload described in Section 3.

One set of benchmarks measures the performance of sequential reads and writes to files of varying sizes. A second benchmark uses the files left on the file systems at the end of the aging process to compare the performance of files that were created in a more realistic manner.

5.1 Sequential I/O Performance

Our first measurement compared the file systems using a benchmark of sequential read and write performance. The benchmark operated on thirty-two megabytes of data, which was decomposed into the appropriate number of files for the file size being measured. Because FFS allocates all of the files in a single directory to the same cylinder group, the data was divided into subdirectories, each containing no more than twenty-five files. This distributed the benchmark data across more cylinder groups than if all of the test files had been placed in one directory.

The benchmark executed in two phases:

1. **Create/write:** All of the files were created. For file sizes of four megabytes or smaller, the entire file was created with one write operation. Large files were created using as many four megabyte writes as necessary.
2. **Read:** The files were read in the same order in which they were created. As with the create phase, I/O was performed in four megabyte units.

We ran this benchmark for a variety of file sizes from sixteen kilobytes (two file blocks) to thirty-two

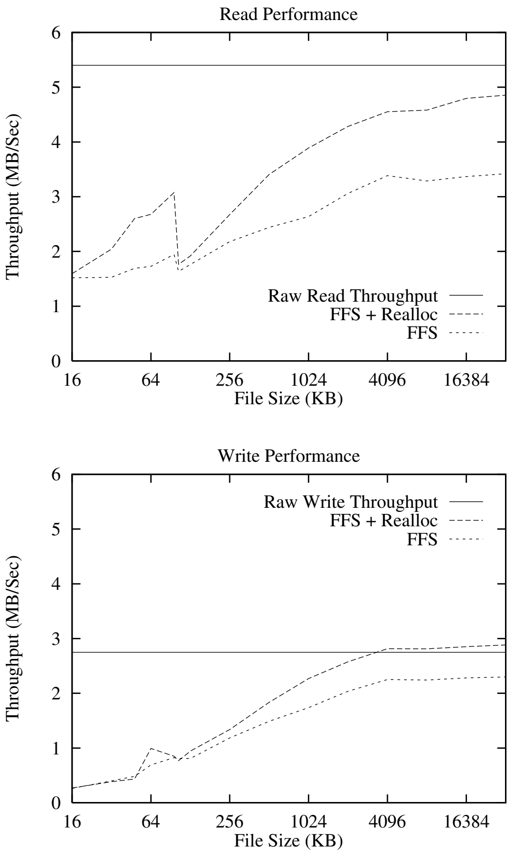

Figure 4: Sequential I/O Performance. These graphs show the read and write performance of the sequential I/O benchmark on the two FFS implementations. The throughput reading and writing the raw disk are also shown. In the top graph, showing read performance, we see that when the realloc algorithm is used, performance improves by as much as 58%. Similarly, as shown in the bottom graph, the realloc algorithm improves write performance by as much as 44%. For large file sizes, performance using the realloc algorithm exceeds the throughput to the raw disk. This surprising result is due to lost rotations when writing the raw disk. All benchmarks were executed ten times and had standard deviations smaller than 1.5% of the mean data value.

megabytes. The results, presented in Figure 4, show that the FFS implementation that included the realloc algorithm performed better for nearly all file sizes. The sharp dip in all of the performance curves at 104 KB corresponds to the file size at which FFS begins to use indirect blocks. Because FFS allocates indirect blocks, and the data blocks to which they point, in a different cylinder group than the previous part of the file, a large performance penalty is incurred at this file size. The overhead of this seek between cylinder groups is amortized as the file size grows, improving throughput for larger file sizes.

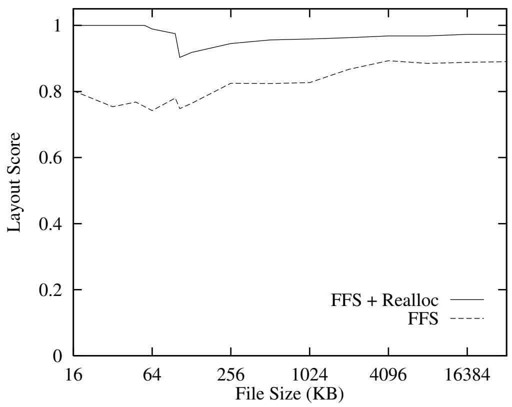

Figure 5: File Fragmentation During Sequential I/O Benchmark. This graph shows the average layout score of files created by the sequential I/O benchmark as a function of file size. The "FFS" line was generated from the aged file system using the original FFS disk allocation algorithm. The "FFS + Realloc" line was generated from the aged file system using the realloc allocation algorithm. For all file sizes, the realloc algorithm produced better file layout. For files up to 56 KB (7 blocks) the realloc algorithm achieved perfect layout.

Small file reads exhibit a large performance difference between the two FFS implementations. 96 kilobyte files, the largest size possible without an indirect block, have 58% greater read throughput on the file system with the realloc disk allocation algorithm. This performance improvement is directly attributable to the better layout attained when using the realloc algorithm. Figure 5, which graphs the average layout score of the files created for each run of the benchmark, shows that for files of up to fifty-six kilobytes (the file system cluster size) the realloc algorithm attained perfectly contiguous file layout.

The improvement in create performance when using the realloc algorithm is less noticeable than the corresponding change in read performance, especially for smaller file sizes. This smaller performance difference is due to the synchronous metadata updates that FFS performs when creating a file. These metadata updates dominate the total run time of the create benchmark, and differences in file layout have little effect on the performance of small file creates.

For larger files, there was a more noticeable improvement in write performance when using the realloc algorithm. Large files (four megabytes and larger) perform 25% better, and 64 kilobyte files perform 44% better using the realloc algorithm than they do on the original FFS.

It is interesting to note that write performance when using the realloc algorithm drops after 64 kilobytes, unlike read performance which does not drop off until the first indirect block is allocated at 104 kilobytes. This is an artifact of the maximum disk transfer size imposed by the hardware (64 kilobytes). As Figure 5 indicates, most files between 64 and 96 kilobytes are allocated completely contiguously, despite the fact that they require more than one cluster on the disk. When writing to such a file, the first 64 kilobytes of data is transferred in one request, and the remaining data in a second request. By the time the second request has been issued, however, the disk has rotated past the location where the data is to be written, adding the latency of an extra disk rotation to the I/O time. This phenomenon does not occur when reading the same file because of the read-ahead performed by the track buffer [Seltzer95].

These lost disk rotations between sequential write requests also explain why the write throughput to large files when using the realloc algorithm exceeds the write throughput to the raw disk. When writing to the raw disk, all writes are sequential, and a rotation is lost between each transfer. When the realloc algorithm is used, large files achieve good, but not perfect layout (as shown in Figure 5). These imperfections actually improve write performance, as a small seek between transfers is preferable to a lost rotation. A similar benefit is not seen for large files using the original FFS allocation algorithm because the resulting layout is more fragmented. The overhead of these additional seeks exceeds the savings from avoiding extra rotations.

The performance improvement seen when using the realloc algorithm was larger than we had anticipated. Before running the sequential I/O benchmark, we had expected to see performance differences of no more than 15%, in line with previous research comparing the performance of contiguous and fragmented FFS files [Seltzer95]. The larger than expected performance improvements seen in our tests of the realloc algorithm are explained by comparing our hardware configuration to the one used in the earlier research. Although the two systems had comparable disks, the SparcStation 1 used in the earlier study provided substantially less I/O bandwidth than the PCI bus in our current test configuration. As a result, the ratio of seek time to transfer time was higher on the PCI-based system, and reducing the seek time resulted in larger performance improvements (expressed as a percentage of the total I/O time) than were possible on the SparcStation.

	FFS	**FFS + Realloc**
Layout Score	0.80	0.96
Read Throughput	1.65 MB/sec	2.18 MB/sec
Write Throughput	1.04 MB/sec	1.25 MB/sec

Table 2: Performance of Recently Modified Files. This table presents the read and write throughput of the files modified during the last month of the aging simulation. The aggregate layout score of these files is also presented. The “FFS” column provides the measurements on a standard FFS file system. The “FFS + Realloc” column presents the same measurements on an FFS that includes the realloc disk allocation algorithm. Each of the throughput tests was run ten times. All standard deviations were less than 2% of the corresponding mean value.

5.2 Existing File Performance

One important aspect of the previous benchmark is not representative of many types of real world file system usage. On a real file system, files are deleted as well as created; this may create small holes of free space that cause fragmentation in subsequently created file. Unlike the sequential I/O benchmark, the aging workload interleaves many create and delete operations, possibly resulting in more file fragmentation than is produced by the sequential I/O benchmark. To gauge the effect of using more “realistically” created files, we ran some benchmarks using the files that were present on the test file systems at the end of the aging process.

Previous research has shown that most older files are seldom accessed [Satyanarayanan81], and therefore that the most active files on a file system tend to be relatively young. We approximated the set of “hot” files on our simulated file system by using all of the files that were modified during the last month of the aging workload. These files represent 10.5% of the files on the aged file system (929 out of 8774 files), and use 59.5 megabytes of storage (19% of the allocated disk space). Since these files cannot all fit in the buffer cache, their layout and performance should have a large effect on the overall performance of the file system.

We measured the file system throughput when reading and writing this complete set of files. To limit the amount of time spent seeking between files, we sorted the files by directory, so multiple files would be read from one cylinder group before moving to another. In order to preserve the layout of the original

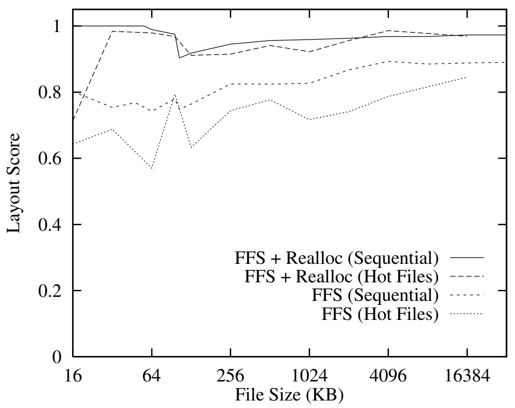

Figure 6: Layout Score of Hot Files. This graph plots the layout score of the hot file set on the two file systems as a function of file size. For comparison, the layout scores of the files produced by the sequential I/O benchmarks (from Figure 5) are also graphed.

files, the write phase of this benchmark overwrote the existing files. Thus the write performance does not include the overhead of creating the files or of allocating disk space to them (these overheads were included in the sequential write performance measurements of Section 5.1). We also computed the aggregate layout score of the files used in this test. The results, presented in Table 2, show that the FFS with the realloc algorithm had 32% higher read throughput and 20% higher write throughput than the original FFS. The performance difference between the two files systems on this benchmark is consistent with the sequential I/O performance measurements shown in Figure 4.

Figure 6 shows the layout score of the hot files plotted as a function of file size for the two file systems. For comparison, we also present the same data for the files from the sequential I/O test (copied from Figure 5). These data show that although the sequential I/O tests produced better layout than the "hot" files under the original FFS implementation, with the realloc algorithm, the layout of the hot files is almost identical to that of the files produced by the sequential I/O test. This indicates the ability of the realloc algorithm to produce near optimal file layout in a variety of circumstances. In the hot file benchmark on the file system that used the realloc algorithm, the layout score for two block files is lower than any other layout score measured in this test. The poor layout of two block files is a result of the same behavior described in Section 4.

6 Future Work

We believe that file system aging can be used to address two issues frequently overlooked in file system performance analysis. This first is the simple fact that real-world file systems usually operate at close to full utilization, unlike the empty file systems that are often used when analyzing file system behavior in a laboratory setting. The second issue that file system aging allows us to address is the impact of specific design decisions on the long term behavior of a file system. The interaction of the disk allocation algorithm and file layout examined in this paper is one such issue. There are a variety of other issues in FFS and other file systems that readily suggest themselves for similar analysis.

In order to apply the file system aging technique to other file systems, we need to generalize the way the aging workload is replayed. The current program makes an important assumption about the behavior of the underlying file system in the way it assigns file operations to directories. More work also needs to be done to make the aging program work on file systems where the idle time between file operations can effect the behavior of the file system itself. An example of this is the timing of cleaner execution on a log-structured file system [Rosenblum92].

We also plan to generate a variety of different aging workloads representative of different file system usage patterns, such as news, database, and personal computing workloads. By analyzing the demands of different workloads, we hope to determine the file system design parameters that are best suited for each type of workload.

7 Conclusions

Our simulations and benchmarks provide conclusive evidence of the improved file layout and file system performance achieved using the realloc disk allocation algorithm. A simulation of ten months of file system activity shows that the reallocation algorithm decreases the number of intra-file disk seeks by more than 50%. With the exception of the mandatory seek imposed by FFS when a file becomes large enough to require an indirect block, the realloc algorithm produces nearly optimal file layout.

In all of the benchmarks that we conducted, an FFS using the realloc code outperformed a file system that did not include this enhancement. The improved file layout achieved by the realloc algorithm improved read and write performance for large files by up to 16%. Read performance for files up to 96 kilobytes improved by as much as 20%. The synchronous metadata updates performed when creating a file limited the performance improvements for writing small files.

8 Availability

The source code for the aging tool and the benchmarks along with the aging workload are available on the World Wide Web at:
`http://www.eecs.harvard.edu/~keith`

9 Acknowledgments

We would like to express our gratitude to the many reviewers for their useful guidance and suggestions, and to offer special thanks to Christopher Small and Jackie Horne for their last minute proof-reading.

10 References

[Bach86] Bach, M., *The Design and Implementation of the UNIX Operating System,* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Baker91] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J., "Measurements of a Distributed File System," *Proceedings of the Thirteenth Symposium on Operating Systems Principles,* Monterey, CA, October 1991, pp. 198–212.

[Blackwell95] Blackwell, T., Harris, J., Seltzer, M., "Heuristic Cleaning Algorithms in Log-Structured File Systems," *Proceedings of the 1995 USENIX Technical Conference,* New Orleans, LA, January 1995, pp. 277–288.

[CSRG94] Computer Systems Research Group, University of California at Berkeley, *4.4BSD-Lite Source CD-ROM,* USENIX Association and O'Reilly & Associates, Sebastopol, CA, 1994.

[Hitz94] Hitz, D., Lau, J., Malcolm, M., "File System Design for an NFS File Server Appliance," *Proceedings of the Winter 1994 USENIX Conference,* San Francisco, CA, January 1994, pp. 235–246.

[Leffler89] Leffler, S., McKusick, M., Karels, M., Quarterman, J., *The Design and Implementation of the 4.3BSD UNIX Operating System,* Addison-Wesley Publishing Company, Reading, MA, 1989.

[McKusick84] McKusick, M., Joy, W., Leffler, S., Fabry, R., "A Fast File System for UNIX," *ACM Transactions on Computer Systems,* Vol. 2, No. 3, August 1984, pp. 181–197.

[McVoy90] McVoy, L., Kleiman, S., "Extent-like Performance from a UNIX File System," *Proceedings of the Summer 1990 USENIX Conference,* Anaheim, CA, June 1990, pp. 137–144.

[Ousterhout85] Ousterhout, J., Costa, H., Harrison, D., Kunze, J., Kupfer M., Thompson, J., "A Trace-Driven Analysis of the UNIX 4.2BSD File System," *Proceedings of the Tenth Symposium on Operating Systems Principles,* Orcas Island, WA, December 1985, pp. 15–24.

[Peacock88] Peacock, J., "The Counterpoint Fast File System," *Proceedings of the Winter 1988 USENIX Conference,* Dallas, TX, February 1988, pp. 243–249.

[Rosenblum92] Rosenblum, M., Ousterhout, J., "The Design and Implementation of a Log-Structured File System," *ACM Transactions on Computer Systems,* Vol. 10, No. 1, February 1992, pp. 26–52.

[Satyanarayanan81] Satyanarayanan, M., "A Study of File Sizes and Functional Lifetimes," *Proceedings of the Eighth Symposium on Operating Systems Principles,* Pacific Grove, CA, December 1981. pp 96–108.

[Seltzer93] Seltzer, M., Bostic, K., McKusick, M., Staelin, C., "The Design and Implementation of the 4.4BSD Log-Structured File System," *Proceedings of the Winter 1993 USENIX Conference,* San Diego, CA, January 1993.

[Seltzer95] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V., "File System Logging Versus Clustering: A Performance Comparison," *Proceedings of the 1995 USENIX Technical Conference,* New Orleans, LA, January 1995, pp. 249–264.

[Smith94] Smith, K., Seltzer, M., "File Layout and File System Performance," Harvard University Computer Science Department Technical Report TR-35-94.

Keith A. Smith (keith@eecs.harvard.edu) is a graduate student in computer science at Harvard University. His research interests include file system design and performance, extensible operating systems, and the search for the perfect pizza recipe. Keith received his B.S. in computer science from Yale University in

1987. Prior to entering the Ph.D. program at Harvard, Keith worked for VenturCom, Inc., developing a real-time UNIX for the x86 architecture.

Margo I. Seltzer (margo@eecs.harvard.edu) is an Assistant Professor of Computer Science in the Division of Applied Sciences at Harvard University. Her research interests include file systems, databases, and transaction processing systems. She is the co-author of several widely-used software packages including database and transaction libraries and the 4.4BSD log-structured file system. Dr. Seltzer spent several years working at start-up companies designing and implementing file systems and transaction processing software and designing microprocessors. She is a Sloan Foundation Fellow in Computer Science and was the recipient of the University of California Microelectronics Scholarship, the Radcliffe College Elizabeth Cary Agassiz Scholarship, and the John Harvard Scholarship. Dr. Seltzer received an A.B. degree in Applied Mathematics from Harvard/Radcliffe College in 1983 and a Ph. D. in Computer Science from the University of California, Berkeley in 1992.