Published in the Proceedings of the 1997 ACM SIGMETRICS Conference, June 1997.

File System Aging—Increasing the Relevance of File System Benchmarks

Keith A. Smith
Margo I. Seltzer
Harvard University
{keith,margo}@eecs.harvard.edu

Abstract

Benchmarks are important because they provide a means for users and researchers to characterize how their workloads will perform on different systems and different system architectures. The field of file system design is no different from other areas of research in this regard, and a variety of file system benchmarks are in use, representing a wide range of the different user workloads that may be run on a file system. A realistic benchmark, however, is only one of the tools that is required in order to understand how a file system design will perform in the real world. The benchmark must also be executed on a realistic file system. While the simplest approach may be to measure the performance of an empty file system, this represents a state that is seldom encountered by real users. In order to study file systems in more representative conditions, we present a methodology for aging a test file system by replaying a workload similar to that experienced by a real file system over a period of many months, or even years. Our aging tools allow the same aging workload to be applied to multiple versions of the same file system, allowing scientific evaluation of the relative merits of competing file system designs.

In addition to describing our aging tools, we demonstrate their use by applying them to evaluate two enhancements to the file layout policies of the UNIX fast file system.

1 Introduction

The increasing prevalence of I/O-intensive applications, such as multi-media applications and large databases, has placed increasing pressures on computer storage systems. In response to these pressures, researchers have investigated a variety of new technologies for improving file system performance and functionality. Disk arrays (RAIDS) were proposed as an alternative to large, high-performance, expensive disk systems [13]. Redundancy leads to higher availability while the multiple disk system provides opportunities for increased disk bandwidth on large I/O requests and increased parallelism for small I/O requests. The log-structured file system (LFS) [16] was proposed as a way to address the small-write performance problem [13] of RAID devices and the increasing fraction of disk traffic due to writes. The AutoRAID storage system [23] combines the benefits of RAID and LFS to construct a high performance multi-disk system that trades off performance and space utilization by moving data between RAID and mirrored store. A variety of strategies for application assisted prefetching and caching [3][9][14] have been explored as mechanisms to better utilize I/O systems by taking advantage of application-specific knowledge of I/O patterns.

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

In order to accurately assess the utility of any of these technologies, researchers need tools that allow them to understand the behavior of their file systems in realistic conditions. In laboratory settings, "realistic conditions" are usually simulated by the use of benchmark programs. A variety of benchmarks have been developed that are useful for predicting the performance of certain types of workloads. Some benchmarks, such as TPC-B [21] simulate specific application workloads. Other benchmarks, such as LADDIS [24], measure particular file system characteristics that are of interest in a wide range of applications. LADDIS is designed to measure responsiveness and scalability in NFS file system environments. The suite of benchmarks from the Transaction Processing Council (e.g., TPC-A, TPC-B, and TPC-C) was designed to quantify performance of on-line transaction and decision support applications. Webstone [22] is a more recent benchmark designed to measure the performance and scalability of Web servers.

A benchmark representative of a realistic workload is only half of the problem. To accurately characterize the performance of a file system, the benchmark itself must be executed in an environment similar to the conditions under which the file system will be used in the real world. Unfortunately, the latter requirement seems to have been widely ignored by file system researchers. Standard practice in file system research is to perform benchmarking on empty file systems, a condition that is typical of few real world environments.

In this paper, we propose a methodology for artificially *aging* a file system by simulating a long term workload on it. By aging a file system prior to running benchmarks, the resulting benchmark performance resembles that of the real file system from which the workload was generated. Just as different benchmarking programs are used to simulate different application workloads, different aging workloads can be used to simulate different execution environments.

In the next section, we motivate this work by describing some of the shortcomings associated with running benchmarks on an empty file system. Section 3 describes our file system aging technology. In Section 4, we apply our aging methodology to the evaluation of two new file layout policies for the UNIX fast file system. Section 5 compares our work with related research. Finally, in Section 6, we present our conclusions

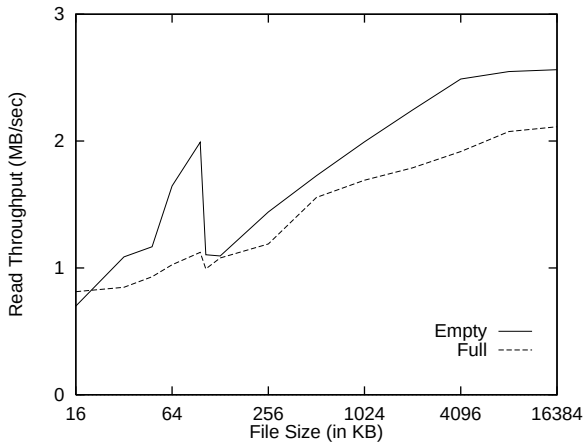

Figure 1: Effect of utilization on file system performance. This graph shows the read throughput for a range of file sizes on two UNIX file systems. The only difference between the file systems is the amount of free space available. One file system was empty when the benchmark was performed. The other file system was a duplicate of a seven month old file system that was 75% full. The contours of the lines are characteristic of the performance of the UNIX fast file system and are explained in detail elsewhere [18].

2 Motivation

Executing a benchmark on an empty file system fails to capture two important characteristics of file system behavior, both of which can have a substantial effect on file system performance. First, real file systems are almost never empty. This fact can have a profound effect on the performance of a file system. Many file systems attempt to optimize throughput by allocating physically contiguous disk blocks to logically sequential data, allowing the data to be read and written at near optimal speeds. On empty disks, this type of allocation is simple. On real file systems, which are typically highly utilized, contiguous allocation may be difficult (or impossible) to achieve due to the fragmentation of free space. As a result, new files may be more fragmented on a highly utilized file system, resulting in lower file throughput.

The second problem with benchmarking an empty file system is that it is impossible to study the evolution of the file system over time. With the passage of time, the state of a file system may change. As files are created and deleted, patterns of file fragmentation may change, as well as the relative locations of logically related objects on the disk. There are a variety of file system policies that may have no effect over the short term on an empty file system, but that can have a noticeable impact on file system performance over the long run. Decisions that a file system makes today (for example, which blocks to allocate to a new file) may affect the file system for months or years into the future.

In this section, we present an example of each of these problems, demonstrating that benchmarks conducted on an empty file system can either provide misleading results, or fail to measure the effects of significant changes to the underlying file system.

2.1 Empty file systems

The most common problem with benchmarking empty file systems stems from the fact that it is very difficult to measure the effects of file fragmentation on an empty disk. Because fragmentation is a fact of life in many file system designs, it is foolish to benchmark such file systems when they are empty, and have no file fragmentation. To demonstrate this effect, we ran a simple file system benchmark on both empty and full UNIX file systems. To measure the performance of a full file system, we copied an active file system from one of the file servers in our department onto our test machine[1]. After benchmarking this file system, we built an empty file system, with the same parameters, on the same disk, and measured its performance.

The benchmark program that we use throughout this paper measures file system throughput reading and writing files of a variety of different sizes. Figure 1 shows the read throughput for files from 16 KB to 16 MB. Throughput on the real file system is as

1. Rather than copying the entire file system, we only copied the file system's metadata. The result was that the test file system had exactly the same free blocks and allocated blocks as the original file system that we copied.

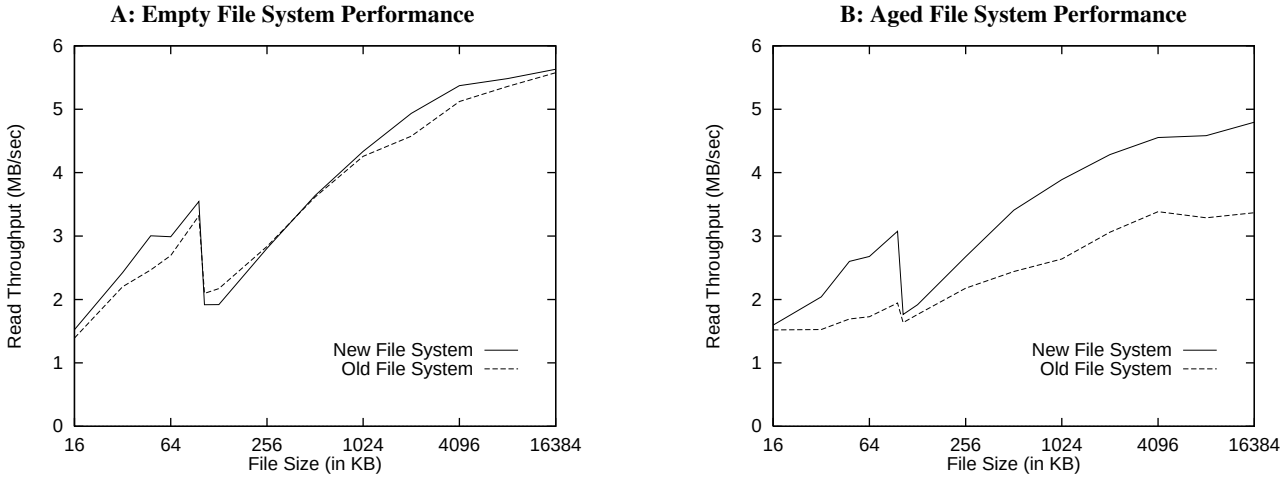

Figure 2: Effect of time on file system behavior. Each of these graphs plots the read throughput for a range of file sizes on file systems using two different block allocation strategies. In the graph on the left, performance was measured on empty file systems. In the graph on the right, performance was measured after *aging* the two file systems with a simulated ten month workload. On the empty file systems, the new algorithm performed slightly better, but the performance of the two systems was nearly identical. On the aged file systems, both file systems perform worse than in the empty case, and the new allocation algorithm provides a large improvement in read throughput. A complete discussion of this study is presented elsewhere [19].

much as 77% lower than throughput on a comparable empty file system.

2.2 Life time evolution

Most file systems attempt to optimize performance by clustering logically related data on the underlying disk(s). The effectiveness of different clustering strategies may not be apparent when observing the short term behavior of the file system. Over time, however, both free and allocated space on the disk may become fragmented, affecting the ability of the file system to perform clustering. Note that this fragmentation affects not only the sequential layout of each file's data, but also the proximity of related files on the disk, and the relative locations of a file and the metadata that describes it. In such cases, the only way to evaluate competing designs is by comparing file systems after a long period of activity.

In previous work [19], we studied the effect of one such design parameter on file system performance. The 4.4BSD fast file system [11] optimizes sequential I/O performance by allocating physically contiguous *clusters* of blocks to logically sequential file data. Over the life of a file system, as free space becomes fragmented, it becomes increasingly difficult to find contiguous free space for new clusters. In comparing two different algorithms for finding and allocating free space to new files, we discovered that they provided nearly identical performance on an empty disk (see Figure 2A). After applying a simulated ten month workload to the two file systems, however, it became apparent that there was a substantial performance difference between file systems using the two different disk allocation policies (see Figure 2B).

3 File System Aging

As the previous section demonstrates, benchmarking empty file systems cannot provide an accurate assessment of the real-world behavior of a file system architecture. In order to get a realistic picture of file system behavior, a file system must be analyzed in realistic conditions. This means that the file system should not be empty, and should have the historical state that would be developed over many months, if not years, of operation. In order to analyze file system performance in this manner, we need to apply a methodology that allows researchers to fill a file system in a realistic manner, resulting in a file system that is similar to one that had been active in real-world conditions for an extended period of time. Analyzing file system performance in this manner presents a variety of problems that do not arise when benchmarking an empty file system:

- Because different applications apply different workloads to the file system, it should be possible to simulate the effects of different file system workloads. A file system used in a traditional engineering environment for a year may behave very differently from one that has been used on a news server for a similar period of time, even if the underlying file system architectures are identical.
- The technique used to fill a file system should be reproducible, allowing scientific comparisons in a laboratory setting.
- The manner in which file systems are filled should be independent of the architecture of the underlying file system, allowing different file system implementations to be compared.

In order to study file system performance in a realistic manner, and to address the concerns listed above, we have developed a technique we call *file system aging*. We precompute an artificial workload intended to simulate the pattern of file operations that would be applied to a file system over an extended period of time. By applying the same workload to different file systems, we can see how differences in file system architecture affect the long term behavior of the file system. The aging workload is generated from snapshots and traces of a real file system. Aging workloads representative of different types of file system activity can be created using data collected from appropriate file systems.

Despite our desire for an architecture neutral file system aging technique, our existing tools have several minor dependencies on the underlying file system (FFS in our case). These dependencies are discussed in Section 3.3.

In this section we provide a brief overview of the UNIX fast file system, present the technique we use to generate aging workloads, describe the program that actually applies a workload to a test file system, and then evaluate the accuracy of our aging workload by comparing artificially aged file systems with the original file systems from which the aging workloads were generated.

3.1 The UNIX Fast File System

The Fast File System has been the *de facto* standard file system on UNIX systems for the past decade, and is only now being replaced with new journaling file systems such as SGI's XFS [20], IBM's JFS [4], and the HP-UX v10 file system. A brief overview of the relevant aspects of the UNIX Fast File System (FFS) is presented here. A more detailed explanation may be found in *The Design and Implementation of the 4.4BSD Operating System* [11].

FFS divides the disk into blocks of uniform size (typically four or eight kilobytes). These blocks are the standard unit of disk allocation. Each of these full-sized data blocks may be further sub-divided into smaller units, called *fragments*, to accommodate files that do not require an integral number of disk blocks.

The disk is also divided into *cylinder groups*, each of which is a set of consecutive cylinders. Each cylinder group is static in size (typically sixteen cylinders) and contains a fixed number of data blocks. 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.

Each file has an index node, or *inode*, that contains all of the file's metadata, including its owner, size, and time of last modification. Each cylinder group contains a fixed number of inodes. Whenever possible, FFS allocates a file's inode, as well as its data blocks, in the same cylinder group as the directory containing it. Each inode also contains pointers to the blocks that contain the file's data. Because inodes are fixed in size, they only contain fifteen of these block pointers. The first twelve block pointers refer to the first twelve data blocks assigned to the file. The final three block pointers in the inode refer to *indirect blocks*, which contain pointers to additional file blocks or to additional indirect blocks.

FFS attempts to optimize file system throughput by allocating successive blocks of a file to physically contiguous disk blocks, allowing the file to be read or written sequentially at close to the disk's maximum bandwidth. Although contiguous disk allocation usually results in optimal file throughput, FFS does not guarantee such file layout, and only uses a set of simple heuristics in attempting to achieve it.

3.2 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, resulting in a file system that is mostly full. 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 size of the traces required to do this makes this strategy impractical. Instead, we generated aging workloads from two sets of file system data that were already available to us. In doing so, we sacrifice some realism in the workload, in exchange for greater flexibility in tuning the workload to our needs.

An aging workload is a sequence of file system operations, primarily file creates and deletes, that can be applied to a test file system to simulate the effects of an extended period of application activity on the file system. Each create operation specifies the size of the file to be created.

To generate an aging workload, we used a set of file system *snapshots* collected from a file system on a local file server. These snapshots, originally gathered for a different research project [18], were collected nightly from approximately fifty file systems on five different file servers over periods of time ranging from one to three years. 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, inode generation number, file type, file size, and a list of the disk blocks allocated to the file.

By using a sequence of snapshots of one file system, we generate an aging workload modeled on the actual activity on that file system during the period of time covered by the snapshots. Because we have snapshots from a variety of different file systems, we can generate aging workloads that are representative of different file system uses. The extended period of time covered by the file system snapshots allows us to build an aging workload that simulates many months of file system activity.

Generating a workload from a sequence of traces is a three step process. First, the target file system must be populated by initializing it to a state similar to the first snapshot of the original file system. Next, we create a skeleton of the workload by comparing successive pairs of snapshots and generating a workload to account for the changes on the original file system between the two snapshots. Finally, we flesh out the workload by adding the creation and deletion of a variety of short-lived files.

The first step in creating an aging workload is to generate a sequence of file system operations that will bring the test file system into a state similar to the one represented by the first snapshot of the original file system. Because the only state that we are trying to reproduce is the set of files that exist on the file system, this is a simple matter of creating each file in the initial snapshot. The actual create operations are sorted based on the inode change times of the files in the snapshot in the expectation that this will be a reasonable approximation of the order in which the files were created on the original file system.

Next we generate the skeleton of the aging workload. By comparing the inodes listed in successive pairs of snapshots, we generate a list of the files that were created, deleted, modified, or replaced between the times of the two snapshots. The major difficulty at this stage is determining the sequence in which these actions occurred, as the snapshots do 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 the file operations generated by comparing successive snapshots. The inode change time, recorded for each file in a snapshot, indicates the last time that the file's metadata was modified. Such modifications include the original creation of the file, and the allocation of new disk blocks to the file. As previous studies have shown that files are typically written in one burst, and are seldom modified after they are first written [1][12], we used the inode change time on a newly created file to approximate the time at which the file was created. When a file was deleted between two snapshots, there was no information providing hints about the time it was deleted. We randomly assigned times to the file deletions that occurred between two snapshots. This was an ad hoc decision made to expedite the development of our file system aging workloads. A more careful analysis of file deletion times in real file system traces might provide a more accurate solution and improve the realism of our aging workloads.

When the same inode was listed in two successive snapshots, but with different file attributes, one of two things may have happened on the original file system; the file was either modified, or replaced. The inode generation number allows us to determine which of these actions actually occurred. If the generation number is the same in both snapshots then we know that the file was modified. In this case we place a "file modification" operation in the aging workload, and assign it a time corresponding to the inode change time in the later snapshot. If the generation number is different between the two snapshots, then the original file must have been deleted, and a newly created file assigned the same inode number. In this case, we place two operations in the workload, a delete, and a subsequent create. We determine the time of the create as described above, and place the delete immediately prior to it.

After processing the snapshots in this manner, the workload is missing an important component of real file system activity. Any file that was both created and deleted between the same pair of snapshots will not appear in any snapshot. Trace-based file system studies [1][12] have shown that most files live for less than the twenty-four hours between successive snapshots. These files may have a significant effect on the state of the longer lived files on the file system.

To approximate the effect of these short-lived files, we must add additional file operations to the workload generated from the snapshots. In order to add this additional workload, we must answer two questions—what operations should we add, and where (both physically and temporally) should we add them?

To determine what file operations we should add to the aging workload, we examined the patterns of activity displayed by short-lived files in a seven day trace of NFS requests to a Network Appliance file server [8]. For each day in the trace, we made a list of the active directories, and then created a profile of the short-lived file activity in those directories. The result was 449 different profiles, each containing a list of create and delete operations on short-lived files that occurred on one day in one directory. For each day in the aging workload, we selected 25 of these profiles at random and added them to the aging workload[2].

Given a day of activity from the aging workload, and a set of short-lived file profiles, we integrate the two by finding the most active directories[3] in that day of the aging workload, and randomly distributing the profiles among them. We time-shifted each profile so that it coincided with the peak of activity in the directory to which it was added.

The NFS trace that we used to generate our profiles of short-lived file activity was originally collected during a study of cleaning algorithms for log-structured file systems [2], and was generated from a server used for a typical academic workload, consisting of text editing, compilation, executing simulations, etc. We therefore only use this trace to generate aging workloads from file systems that were used in similar environments. In order to generate aging workloads for other types of file system activity, such as database or news servers, we will need to use different traces to ap-

2. We actually scaled the number of short-lived file profiles that we used based on the size of the file system from which we generated the aging workload, adding one profile for every 40 MB on the original file system.

3. Since our file system snapshots do not preserve the names of the files in them, we actually used the most active cylinder groups instead of the most active directories. This is a reasonable approximation since FFS allocates all of the files in a directory to the same cylinder group.

proximate the activity to short-lived files.

3.3 Replaying the Workload

To age a file system, we apply an aging workload generated as described above to an empty file system. In all of our measurements, we use a target file system that is the same size as the file system from which the aging workload was generated, although an aging workload could also be used on larger file systems. The aging program reads records from the workload file, performing the specified file operations. Although the aging workload includes timestamps for each file operation, we simply execute the requests as fast as possible. Replaying the workload in real time was unnecessary for our purposes, because in FFS (and many other file systems) the order in which requests are received by the file system, not the relative times of the requests, determines the behavior of the file system.

The task of replaying an aging workload was complicated by the fact that the file system snapshots did not provide pathnames for the files. Because FFS exploits expected patterns of locality by allocating files in the same directory to the same cylinder group on the disk, the algorithm used by the aging program to assign files to directories can have a major impact on the accuracy of the aging simulation.

Due to the absence of the original pathnames in the file system snapshots, we decided that it would be sufficient to create the files in the correct cylinder groups. 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 received the same set of allocation and deallocation requests that were presented to the corresponding cylinder group on the original file system from which the snapshots were generated. We used each files'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 start the aging process with an empty file system. The first step is 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 use its inode number to compute the cylinder group to which it was allocated on the original file system, and place 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 guarantees that all of the files that are in the same cylinder group on the original file system are also in the same cylinder group on the aged file system.

There are two drawbacks to this approach. First, 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 directories should be negligible, however, as the space that they occupy is much less than that of the files being manipulated during the aging simulation. The second drawback is that by exploiting these details of the FFS implementation, we are limiting the applicability of our file system aging tools to file systems that use some physical partitioning to improve the clustering of logically related data.

3.4 Workload Verification

In order to evaluate the realism of our simulation, we compared a test file system aged using our techniques with the real file system from which the aging workload was generated. Because our test file system necessarily starts in an empty state, we generated our aging workload from a file system for which we have snapshots starting the day it was created. This file system, which contains the home directories of several graduate students studying parallel computing, was not one of the file systems that we used in deriving the aging methodology. The aging workload that we generated from this file system simulates 215 days (approximately seven months) of activity on a one gigabyte file system. The workload contains approximately 1.3 million file operations that write 87.3 gigabytes of data to the disk and takes 39 hours to replay on a generic FFS implementation. At the end of the workload, the file system is 65% full.

We ran this aging workload on a test file system that was configured with the same file system parameters as the original system and compared the resulting state of the test file system with the state of the original file system at the end of the sequence of snapshots. In this discussion, we refer to the original file system from which the aging workload was generated as the *real file system*, and we refer to the test file system that was aged using our artificial workload as the *simulated file system.* Table 1 describes the hardware configuration that we used both to age the simulated file system, and for the benchmarks described in Section 4.

One of the primary changes observed in many file system architectures as a system ages is an increase of file fragmentation on the disk. Therefore we started by comparing several aspects of fragmentation on the real and simulated file systems. 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

CPU Parameters		Disk Parameters		File System Parameters	
CPU	Intel Pentium Pro	Disk Controller	NCR 53c825	Size	1024 MB
Clock Speed	200 MHz	Disk Type	Fujitsu M2694ES	Fragment Size	1 KB
Memory	32 MB EDO RAM	Total Disk Space	1080 MB	Block Size	8 KB
Bus Type	PCI	Rotational Speed	5400 RPM	Max. Cluster Size	56 KB
		Cylinders	1818	Rotational Gap	0
		Heads	15	Cylinder Groups	63
		Avg. Sectors/Track	94	*Heads*	*19*
		Track Buffer	512 KB	*Sectors/Track*	*111*
		Average Seek	9.5 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 we generated the aging workload, despite the fact that they do not match the underlying hardware.

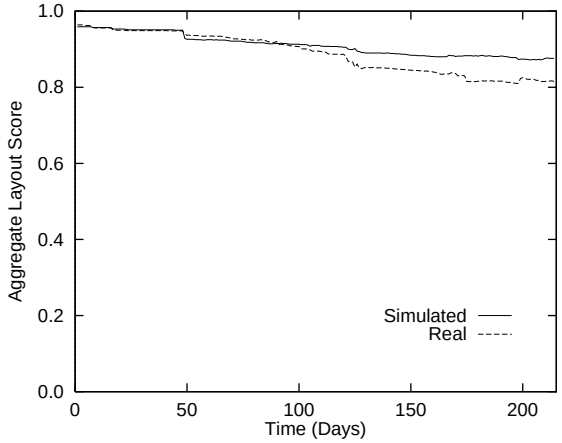

Figure 3: Real vs. simulated file system. This chart plots the aggregate layout score for each day in the seven month simulation period. The "Simulated" line shows the fragmentation on the artificially aged file system. The "Real" line shows the fragmentation on the original file system from which the aging workload was generated. Although the two file systems behave similarly for the first half of the simulation, the aging workload fails to capture several of the large changes in the original file system workload during the later half of the simulation period.

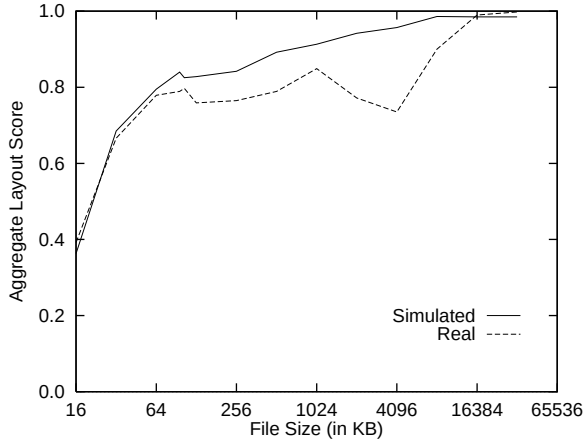

Figure 4: Fragmentation as a function of file size. File sizes were rounded up to an even number of file blocks, and the aggregate layout score was computed for files of various sizes on the real and simulated file systems. The results are graphed here. Both file systems suffer from extreme fragmentation of small files (< 32 KB). On the real file system, file layout declines noticeably for large files (2 MB – 4 MB). A similar decline is not seen on the simulated file system.

that file's blocks that are optimally allocated. An optimally allocated block is one that is contiguous with the preceding 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 not defined for one block files, since they cannot be fragmented. A file with a layout score of 1.00 is perfectly allocated; all of its blocks are contiguously allocated on the disk. A file with a layout score of 0.00 has no contiguously allocated blocks.

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

At the end of our simulation period, the aggregate layout score on the real file system was 0.815, compared to 0.876 on the simulated file system.[4] Thus, although our aging workload does cause fragmentation on the file system, it does not generate as much fragmentation as occurred on the real file system. Figure 3, which presents a time series of the aggregate layout scores for both file systems over the 215 days of the simulation, indicates that although the aggregate layout score of the simulated file system tracked the real file system very closely for the first half of the simulation, during the second half of the simulation, the aging workload failed to replicate several large changes in file fragmentation on the real file system.

To get a better understanding of the fragmentation differences between the real and simulated file systems, we sorted the files on both file systems by size and computed the aggregate layout scores for files of a variety of sizes. The results are shown in Figure 4. Although the two file systems have similar layout scores for small files (up to 64 KB), for larger files, our simulated file system has higher layout scores, indicating that it failed to capture all of the fragmentation that actually occurred on the real system. It is these large files that cause the aged file system to have a higher aggregate layout score then the real one. The difference in layout scores is most noteworthy for files of 2 MB – 4 MB. We are unsure of the cause of this discrepancy. Many files of these sizes on the original file system are unusually fragmented, and have layout scores of less than 0.5. On other file systems that we have examined, large files do not exhibit this degree of fragmentation. We have speculated that file activity concurrent with the creation of these large files, and taking place in the same cylinder groups, may have caused this fragmentation, but we do not have the data necessary to confirm this hypothesis.

To summarize, our simulated aging workload mimics the real file system from which it was derived in the steady increase in fragmentation over time. However, the total amount of fragmentation on the simulated system is less than on the real file system, largely because the simulated file system failed to replicate several large changes in fragmentation seen on the real file system. The fundamental cause of this inaccuracy in our aging workload is that when we did not have sufficient information to perfectly reconstruct the workload on the real file system, we made randomized decisions. The two most important areas where we did this were in assigning times to file delete operations and in simulating the activity of short-lived files on the file system. In real file system workloads, there are dependencies between these operations and the other activity occurring on the file system. An accurate model of these interdependencies would allow us to make more realistic decisions regarding file delete times and short-lived file activity. The absence of such a model decreases the verisimilitude of our aging. Nevertheless, these tools are still effective for evaluating the impact of design decisions on the long term behavior of a file system.

4 Applications of Aging

In an earlier study [19], we used aging to analyze the effectiveness of an improved block allocation scheme in FFS. On an empty file system, the original and improved schemes were virtually indistinguishable, but on an aged file system the improved scheme resulted in performance improvements of up to fifty percent. Aging enables us to explore the long term effects of a number of

4. Note that these seemingly high layout scores—more than 80% of the blocks on both the real and simulated file systems were optimally allocated—are typical of FFS. On all of the file systems in our snapshot library, we seldom see an aggregate layout score of less than 0.7 except on news servers, which are subject to extreme file fragmentation.

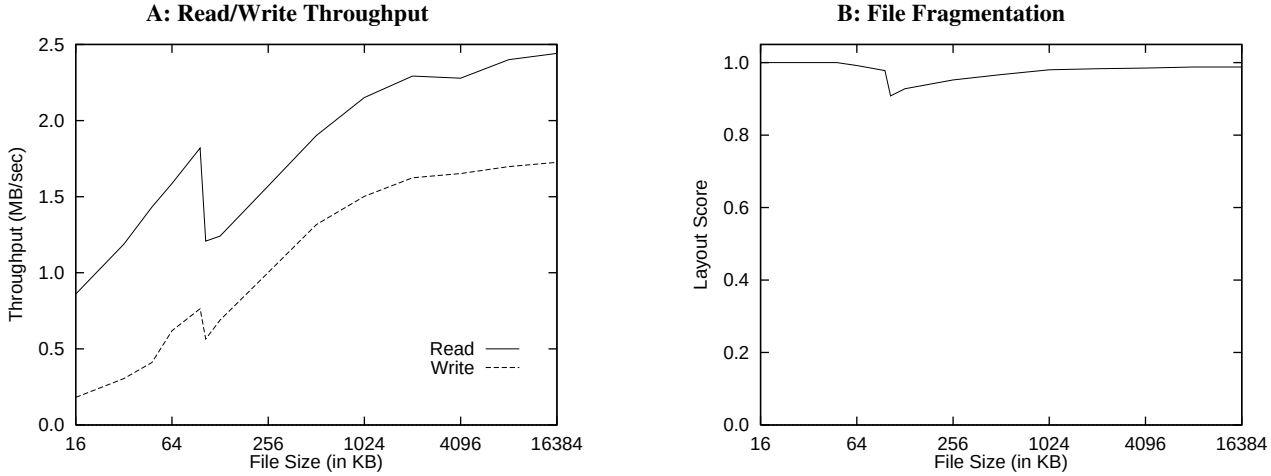

Figure 5: Performance baseline. These charts show the performance of our baseline file system in the file throughput benchmark. The benchmark was executed after the file system was aged using the workload described in Section 3.2. Graph A plots read and write throughput as a function of file size. Graph B plots the layout scores of the test files created during the benchmark. The sharp drops in all of these graphs as the file size passes 96 KB corresponds to the point where FFS allocates the first indirect block to a file (see Section 4.1).

policy decisions and file system features. In this section, we will use our aging methodology to answer the following questions about FFS layout.

- Indirect blocks (blocks that contain pointers to data blocks) are always allocated to new cylinder groups. This imposes a sharp performance penalty on midsize files (i.e., 104 KB to 256 KB). If we allocate the first indirect block of a file in the same cylinder group as the start of the file, how does this affect performance? Are there any undesirable side effects?
- Fragments (partial blocks) are rarely allocated adjacent to the preceding block of their file. Placing fragments adjacent to their preceding blocks may improve performance, but it may also lead to more internal fragmentation. Is changing fragment allocation beneficial?

The basic technique used in exploring these two issues was to propose and implement a modification to FFS. We then aged two file systems that differed only in this modification, and ran a variety of benchmarks on the aged file system to evaluate the effect of the proposed change on the long term behavior of the file system.

In order to compare the performance of two file systems, we used two simple benchmark programs. The first measures the file system throughput sequentially reading and writing files of a variety of sizes. Each run of the benchmark measures the read and write performance for one file size. The benchmark operates on 32 MB of data, which is 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 is divided into subdirectories, each containing no more than twenty-five files. This increased the number of cylinder groups exercised during the benchmark.

The benchmark executes in two phases:

1. **Create/write:** All of the files are created. For file sizes of 4 MB or less, the entire file is created with one write operation. Large files are created using as many 4 MB writes as necessary. This phase measures write throughput, including the time required to create new files and allocate disk space to them.
2. **Read:** The test file system is unmounted and remounted to flush the file cache. Then the files are read in the same order in which they were created. As with the create phase, I/O is performed in 4 MB units.

For each file size in our tests, we executed this benchmark ten times, averaging the resulting throughput measurements. In all test cases, the standard deviation was less than 1% of the average throughput.

This benchmark is unrealistic on one important sense. Real file system workloads seldom create large batches of files of the same size. Actual usage patterns typically interleave the creation and deletion of files of a variety of sizes, possibly resulting in more file fragmentation than we would see in the sequential I/O benchmark described above. Our second benchmark attempts to address the problem, by exploiting the more "realistically" created files that are left on the test file system at the end of the aging workload.

Previous research has shown that most older files are seldom accessed [17], 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 thirty days of the aging workload. These files represent 9.5% of the files on the aged file system (3,207 out of 33,797 files), and use 92.3 megabytes of storage (14.5% of the allocated disk space).

Our second benchmark measures file system throughput when reading and writing this complete set of "hot" files. To limit the amount of time spent seeking from one file to the next, we sorted the files by directory, so multiple files would be read from one cylinder group before moving to another. To preserve the file layouts, we overwrite the files during the write phase of this test.

We use FFS enhanced with the improved block-clustering algorithm [19] mentioned in Section 2.2 as our baseline system. The performance of this file system (after aging) is shown in Figure 5.

4.1 Indirect Block Allocation

Each time an indirect block is allocated to a file in FFS, the file system assigns that block, and all of the data blocks it references, to a different cylinder group than the previous part of the file. The new cylinder group is chosen by selecting the next cylinder group on the disk that has at least an average number of free blocks (relative to the rest of the file system).

This scheme seems undesirable, as it forces long seeks at periodic locations in large files. For very large files, however, these extra seeks are typically amortized over the transfer of the entire file,

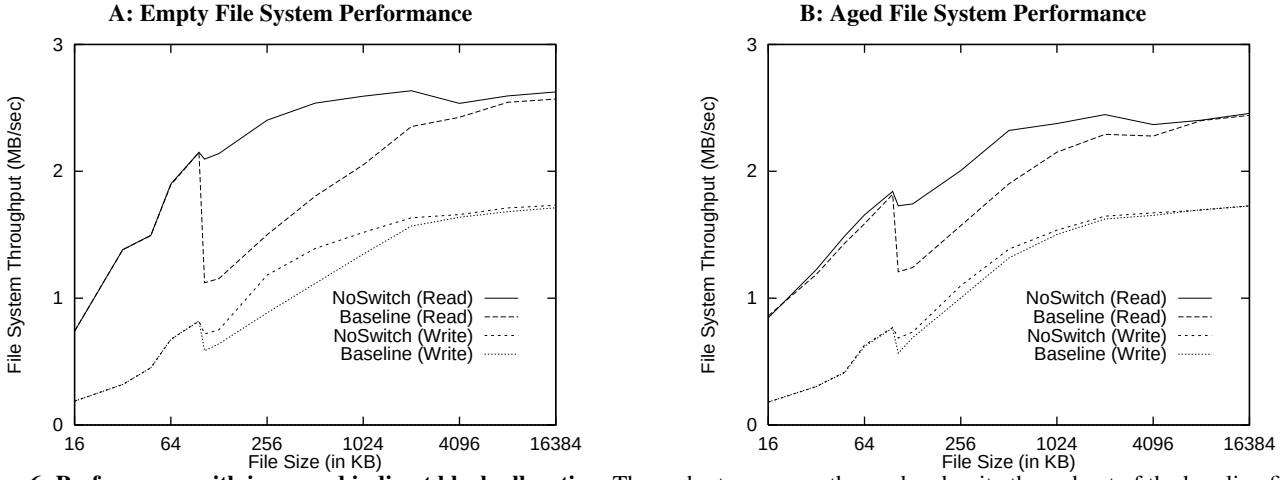

Figure 6: Performance with improved indirect block allocation. These charts compare the read and write throughput of the baseline file system to a file system that does not switch cylinder groups when allocating the first indirect block ("NoSwitch"). The graph on the left shows this comparison on an empty file system; the graph on the right shows this comparison on aged file systems. The NoSwitch file system offers higher throughput in both cases, but the magnitude of the improvement, indicated by the area between the NoSwitch and Baseline lines in the graphs, is significantly smaller on the aged file system.

and have a negligible effect on I/O throughput. Given a typical file system block size of 8 KB, this policy will force a change of cylinder groups every 16 MB of the file. We also speculate that switching cylinder groups may be useful in practice, as it prevents a single large file from consuming all of the free space in a cylinder group.

Unfortunately, there is one glaring problem with this policy of switching cylinder groups with the allocation of each indirect block of a file—in FFS the first indirect block is allocated after only the twelfth data block of a file. On an 8 KB file system, this means that FFS imposes an extra seek after the first 96 KB of a file. For medium size files of a few hundred kilobytes, this extra seek can have a noticeable impact on performance. The effect of this extra seek is apparent in the performance of our baseline file system in Figure 5A. The layout score of the test files drops from 0.98 to 0.91 when the first indirect block is allocated (between 96 KB and 104 KB) and both read and write performance decline precipitously at the same point. There is a larger drop in read performance (33%) than in write performance (25%) because the indirect block not only causes a seek during the read, but also interferes with file prefetching, as the blocks referenced from the indirect block cannot be prefetched until the indirect block itself has been read from the disk.

To address this problem, we modified FFS to not switch cylinder groups until it allocates the second indirect block in a file. (In our file systems, this occurs when the file size reaches approximately 16 MB). We call the implementation of FFS that includes this enhancement *NoSwitch*. We expected this minor enhancement to have the effect of improving file throughput for files of a few hundred kilobytes. Larger files should not see very much improvement as the savings from eliminating one seek are amortized over the time it takes to read or write the entire file. We used our throughput benchmark to compare the performance of the NoSwitch file system to our baseline file system on both empty and aged partitions. The results are shown in Figure 6.

As expected, read and write throughput to files of a few hundred kilobytes improves on the NoSwitch file system. Note that there is still a slight performance drop as file size passes 96 KB and the first indirect block is used. This occurs because in addition to transferring the file data, the file system must also transfer the indirect block. Comparing the performance on empty and aged file systems in Figure 6 we see that the NoSwitch system outperforms our baseline in both cases. The magnitude of the performance improvement, as shown by the area between the pairs of curves for the two file systems, is smaller on the aged file system. In the best case (104 KB files) the NoSwitch file system improves performance by 87% on an empty file system, but only by 43% on an aged file system.

If our only concern were whether the NoSwitch file system would improve performance, we would not have needed to run our benchmarks on an aged file system. By using an aged file system, however, we can more accurately assess the magnitude of the performance improvement. The use of an aged file system also allows us to assess an adverse side effect of this enhancement. As described earlier, FFS attempts to exploit locality of reference by co-locating all of the files in a directory in the same cylinder group as the directory itself. In a directory with many files, some of which are large, the original scheme of switching cylinder groups after only twelve blocks of a large file may have ensured that a single large file did not consume all of the free space in a cylinder group, forcing subsequently allocated files to be placed in other cylinder groups, thus destroying the desired locality.

To study this effect, we examined the state of the baseline and NoSwitch file systems after they had been aged. If the NoSwitch file system caused an increase in the number of files displaced from the cylinder group of their directory, we would expect to see a larger number of files where the first data block of the file is in a different cylinder group from the file's inode. (FFS also tries to locate a file's inode in the same cylinder group as its directory.) We counted the number of these "split files" on the two file systems, and for each such file, determined how many cylinder groups separated the file's inode and its first data block. The more intervening cylinder groups, the longer the seek required to read the file's data after reading its inode. The results are summarized in Table 2.

The NoSwitch file system has more than twice as many of these split files as the baseline file system, indicating that not switching cylinder groups when the first indirect block is allocated does, in fact, cause highly utilized cylinder group to run out of free space. On the baseline file system, most of the split files require relatively short seeks; in more than half the cases, the file's data is only one cylinder group away, and in almost all cases, the data is within ten cylinder groups of the file's inode. In contrast, a third of the split files on the NoSwitch file system involve seeks of more than 10 cylinder groups.

In an attempt to balance the performance gain for large files, which are not allocated in one cylinder group, against the potential performance loss from the longer seeks required to read the extra

	Baseline	NoSwitch
Number of split files	4312	9155
% of all files that are split	13	27
% of one cyl. group splits	58	37
% of < 10 cyl. group splits	95	67

Table 2: Number of split files on NoSwitch file systems. This table compares the number of split files (files where the inode and the first data block are in different cylinder groups) on the baseline file system and on the aged file system with the NoSwitch enhancement. The four rows of the table present, respectively, the total number of split files on each file system, the percentage of all files on each file system that are split, the percentage of split files where the data block is only one cylinder group away from the inode, and the percentage of split files where the data block is no more than ten cylinder groups away from the inode. Both file systems had sixty-three cylinder groups.

	Baseline	NoSwitch
Aggregate Layout Score	0.928	0.931
# of split files	327	594
Read bandwidth (MB/sec)	0.810	0.835
Write bandwidth (MB/sec)	0.494	0.495

Table 3: Performance of recently modified files on NoSwitch file system. This table presents the read and write throughput of the files modified during the last thirty days of the aging workload on the baseline and NoSwitch file systems. The aggregate layout scores of the files used during this test, and the number of these files where the first data block was located in a different cylinder group than the file's inode ("split files") are also presented. Throughput measurements are the averages of ten test runs. All standard deviations were less than 0.2% of the reported means.

split files that are generated on the NoSwitch file system, we turn to the results of our hot file benchmark. The results of this benchmark, which are summarized in Table 3, show that the NoSwitch file system offers a modest improvement in read throughput, with virtually no change in write throughput. This performance improvement offered by the NoSwitch file system suggests that the performance gained via better layout of larger files outweighs the performance lost by increasing the number of split files. The improvement in performance is small enough, however, that it may be an artifact of the particular workload that we are using, and this file system modification may not be universally applicable. It is important to note, however, that we would have had no means to evaluate this trade-off if we had only benchmarked NoSwitch on an empty file system.

4.2 Fragment Allocation in FFS

To limit the amount of internal fragmentation caused by small files, FFS allows a single file system block to be subdivided into *fragments*. The minimum fragment size is determined at the time that the file system is created, and blocks may only be divided into pieces that are integral multiples of the fragment size. For files with no more than twelve data blocks (i.e., files that do not use any indirect blocks), a partial block containing an integral number of fragments may be used as the last data block instead of a full-sized file system block. On our test file system, for example, the block size was 8 KB and the fragment size was 1 KB. Thus, a 30 KB file would be allocated as three file blocks, followed by a partial block containing six fragments.

While this scheme is efficient in reducing the amount of disk space wasted by internal fragmentation, the algorithm used to allocate fragments to files results in suboptimal file layout. When allocating a fragment, FFS first attempts to find a free fragment of the appropriate size in the same cylinder group as the file. If such a fragment is not available, FFS will divide a larger free fragment. Finally, if no fragment of an appropriate size is available, FFS will allocate an entire file system block, and divide it into fragments. Thus the primary goal of the fragment allocation algorithm is to limit the amount of free space that exists in fragments. The downside of this approach is that the fragment at the end of a file is seldom allocated near the preceding block of the file. In Figure 4, for example, we see that the layout scores of small files are much lower than those for other files, indicating that small files are more fragmented. This fragmentation is almost entirely due to the fragment allocation policy. On the baseline file system, for example, only 36% of two block files are allocated with their two blocks contiguous on disk. Of the two block files where the second block is a full block rather then a fragment, however, 87% are allocated contiguously.

Ideally, we would like the fragment at the end of a file to be contiguous with the preceding block of the file. To this end, we modified the fragment allocation algorithm used by FFS. Our new algorithm always attempts to allocate the block immediately adjacent to the previous file block. If that block is available, it is broken into fragments, and the unused portion is marked as free. If the desired block is not available, we return to FFS's original fragment allocation policy. For small files, where the only data block is a fragment, we always use the original FFS policy, hoping to fill in the free fragments created when full blocks are broken up to provide contiguous fragments for larger files. We refer to the version of FFS that uses this new fragment allocation policy as *SmartFrag*.

We used our sequential I/O benchmark to compare the performance of the SmartFrag file system to that of our baseline system. Since we were interested in the behavior of files that use fragments, we focused on small files in running this benchmark. Figure 7 shows the layout score of the small files created by running the benchmark on the two aged file systems. Figure 8 presents the measured performance of the two versions of FFS on both empty and aged file systems.

Figure 7 shows that the SmartFrag scheme dramatically decreases file fragmentation for files that use fragments. Both the SmartFrag and baseline file systems achieve nearly perfect layout for file sizes that are an integral multiple of the eight kilobyte disk block size. For intermediate sizes, however, SmartFrag eliminates almost all of the fragmentation seen on the baseline system.

This difference in file layout translates to the performance differences seen in Figure 8. The saw-tooth effect in the read performance on all of the tested systems is caused by changes in the performance characteristics of the file systems when fragments are used. All file sizes that are even multiples of the file system block size do not require fragments. Note that at these file sizes, the performance of the baseline file system is the same as on the SmartFrag file system, as they both use the same file layout algorithm. For file sizes that are not integral multiples of the file system block size, SmartFrag outperforms the baseline system due to the improved allocation of fragments for these files. Both the SmartFrag and baseline systems have decreased throughput for files that use fragments because FFS issues a separate I/O request to the disk driver for the fragment, regardless of whether the fragment is contiguous with the previous file block.

The differences in write throughput are much smaller because

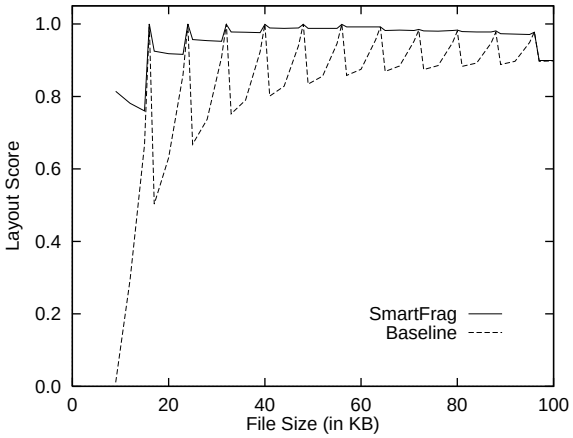

Figure 7: File layout with smart fragment allocation. This graph shows the amount of file fragmentation for small file sizes on the baseline and SmartFrag file systems. The layout score is plotted for the files created for the throughput test in Figure 8B.

in FFS write throughput is dominated by the time required to synchronously update on-disk metadata each time a file is created [18].

The write performance on all of the tested systems also shows an unexpected jump when the file size reaches 64 KB. This is the result of a performance bug in our version of FFS (and in all other versions of FFS derived from 4.4BSD). Until a full cluster (64 KB) of data has been written to a file, FFS does not use clustered writes. The result is that for smaller file sizes, FFS issues one write request for each file block, regardless of the layout on disk. At these file sizes, the overhead of performing these individual I/O operations completely masks any performance differences caused by the fragment allocation policy. For files larger than 64 KB, we see that the SmartFrag file system provides improved throughput for file sizes that require the use of a fragment.

The potential downside to the SmartFrag strategy is the amount of fragmentation of free space that it causes. Most of the data on a file system is allocated in full-sized file system blocks. If too much of the file system's free space is in fragments instead of full-sized blocks, the file system may run out of free blocks while there is still a sizeable amount of free space in fragments. Seltzer et al. have described a particularly spectacular instance of this problem [18]; one of their news servers reported that it was out of free space despite the fact the file system had more than ninety megabytes free—all in fragments.

To evaluate how much the SmartFrag file system increases the fragmentation of free space, we compared the number of free blocks and free fragments on the baseline and SmartFrag file systems. As expected, both file systems had the same amount of free space. On the SmartFrag file system, however, twice as much of this space was in fragments (5% vs. 2.5% of free space). Because the total amount of fragmented free space is relatively small on both file systems, we feel that this side effect of the SmartFrag allocation scheme is tolerable; it is unlikely to cause problems until the file system is very close to maximum capacity.

5 Related Work

The use of traces and simulated workloads is not a new idea in file system research. These tools have been used for a wide variety of purposes. In general, the two mechanisms have been used interchangeably, depending on the availability of appropriate file system traces, and the ease of parameterizing a workload for simulation.

Several studies have used file system traces to characterize file system workloads, studying such variables as file size distributions, file access times, and patterns of access within a single file. Ousterhout et al. performed such a study on the 4.2BSD UNIX file system [12]. In a follow up study, Baker et al. examined changes in file access patterns six years later, and also investigated issues of file sharing in the Sprite distributed operating system [1]. Ramakrishnan et al. performed a similar analysis of file system traces collected from large customers of Digital Equipment Corporation [15].

File system traces and simulated file system workloads have also been used to as input for file system simulations and to stress test new file system architectures. Both simulated workloads and file system traces have been used by different researchers to evaluate garbage collection strategies for log-structured file systems [16][2]. Dahlin et al. used a week-long trace of a large NFS installation to drive simulations of a new caching scheme for distributed

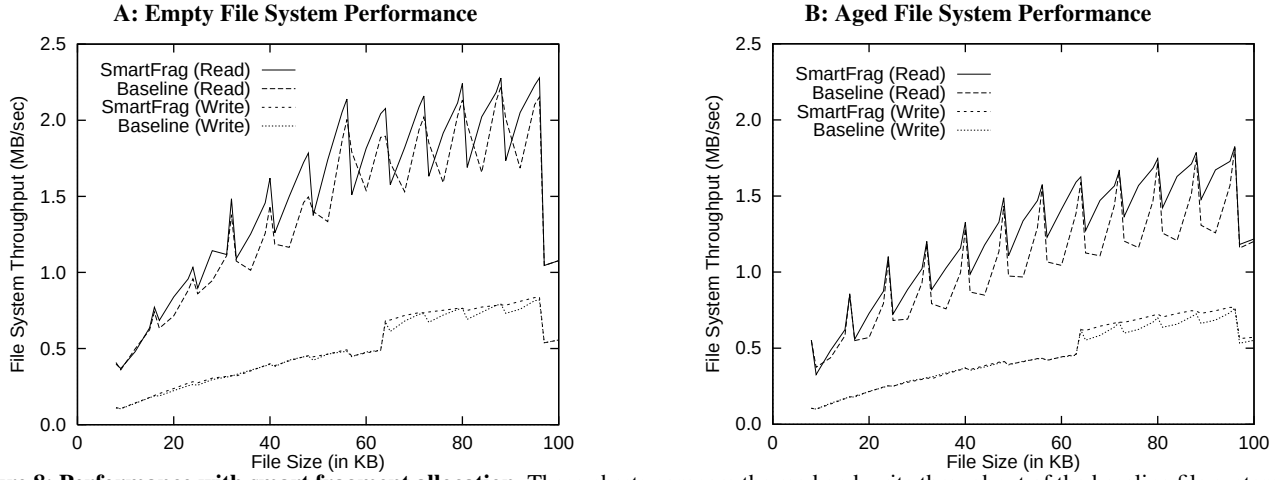

Figure 8: Performance with smart fragment allocation. These charts compare the read and write throughput of the baseline file system to a file system that uses our improved fragment allocation algorithm ("SmartFrag"). The graph on the left shows this comparison on an empty file system; the graph on the right shows this comparison on aged file systems. The saw-tooth effect shows the effect of changing fragment size on file system performance. The peaks represent file sizes that are an integral number of blocks. File sizes that require the use of a fragment do not perform as well because reading or writing the fragment requires an extra I/O operation. The step in write performance at 64 KB files is the result of a performance bug in FFS, described in the body of this paper. All of the performance curves drop precipitously after 96 KB because FFS switches cylinder groups at this point.

file systems [5].

As described above, simulated file system workloads have been generated in a variety of ways and used for a variety of purposes. The most important difference between our work and prior applications of these tools is not in the methodology used to generate the workloads, but rather in the application of the workload. Our file system workloads are not, in themselves, used as benchmarks or stress tests (although they are quite effective at the latter task). Instead we use a long-term file system workload to prepare test file systems for the application of other benchmarks and measurement tools.

Not all file system benchmarking has been conducted on empty file systems. Seltzer et al. [19] examined the effect of varying the amount of free space on a log-structured file system while running a transaction processing benchmark. Although this study did show that performance can vary depending on how full a file system is, it did not address the question of how the file system should be filled.

Herrin and Finkel [7] tested their Viva file system using an aging technique that created and deleted files at random, selecting file sizes from a hyperexponential distribution. Ganger and Kaashoek [6] used a similar technique in testing their clustering FFS. In both cases, the goal of aging was to reproduce the type of on-disk layout that might be experienced after a file system had been in use for an extended period of time. These aging techniques were not, however, based on an actual file system workload.

6 Conclusions

The behavior of a file system can change dramatically with the passage of time. As a file system is filled, or as successive generations of files are created, modified, and deleted, the performance characteristics of the system also change. By ignoring these changes in file system behavior, researchers fail to accurately assess how file system designs will respond to real-world conditions. Not only do active file systems behave differently from empty ones, but there are also a variety of file system design decisions whose full effects are only apparent after a long period of use.

In order to accurately evaluate the long-term behavior of competing file system architectures, we have developed a process for artificially aging a file system by replaying a long-term workload on a test file system. As demonstrated by our evaluation of two new file layout policies for the UNIX fast file system, this technology allows for the scientific evaluation of design decisions that may have no discernible effect on the short-term characteristics of file system behavior.

7 References

[1] Baker, M., Hartman, J., Kupfer, M., Shirriff, K., Ousterhout, J. "Measurements of a Distributed File System." *Proc. 13th SOSP.* Monterey, CA. Oct. 1991. pp. 198–212.

[2] Blackwell, T., Harris, J., Seltzer, M. "Heuristic Cleaning Algorithms in Log-Structured File Systems." *Proc. 1995 USENIX Conf.* New Orleans, LA. Jan. 1995. pp. 227–288.

[3] Cao, P., Felton, E., Li, K. "Implementation and Performance of Application-Controlled File Caching." *Proc. 1st OSDI.* Monterey, CA. Nov. 1994. pp. 165–177.

[4] Chang, A., Mergen, M., Rader, R., Roberts, J., Porter, S. "Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors." *IBM Journal of Research and Development.* Vol 34, No. 1. Jan. 1990. pp. 105–109.

[5] Dahlin, M., Mather, C., Wang, R., Anderson, T., Patterson, D. "A Quantitative Analysis of Cache Policies for Scalable Network File Systems." *Proc. 1994 SIGMETRICS.* Nashville, TN. May 1994. pp. 150–160.

[6] Ganger, G., Kaashoek, M. F. "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files." *Proc. 1997 USENIX Conf.* Anaheim, CA. Jan. 1997. pp. 1–17.

[7] Herrin, E., Finkel, R. "The Viva File System." Tech. Report No. 225-93, University of Kentucky, Lexington. 1993.

[8] Hitz, D., Lau, J., Malcolm, M. "File System Design for an NFS File Server Appliance." *Proc. Winter 1994 USENIX Conf.* San Francisco, CA. Jan. 1994. pp. 235–246

[9] Kimbrel, T., Tomkins, A., Patterson, R.H., Bershad, B., Cao, P., Felton, E., Gibson, G., Karlin, A., Li, K. "A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching." *Proc. 2nd OSDI.* Seattle, WA. Oct. 1996. pp. 19–34.

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

[11] McKusick, M., Bostic, K., Karels, M., Quarterman, J. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, Reading, MA. 1996. pp. 269–284.

[12] Ousterhout, J., Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J. "A Trace-Driven Analysis of the UNIX 4.2BSD File System." *Proc 10th SOSP.* Orcas Island, WA. Dec. 1985. pp. 15–24.

[13] Patterson, D., Gibson, G., Katz, R. "A Case for Redundant Arrays of Inexpensive Disks (RAID)." *Proceedings of the 1988 SIGMOD Conference on Management of Data.* Chicago, IL. June 1988. pp. 109–116.

[14] Patterson, R., Gibson, G., Ginting, E., Stodolsky, D., Zelenka, J. "Informed Prefetching and Caching." *Proc. 15th SOSP.* Copper Mountain, CO. Dec. 1995. pp. 79–95.

[15] Ramakrishnan, K., Biswas, P., Karedla, R. "Analysis of File I/O Traces in Commercial Computing Environments." *Proc. 1992 SIGMETRICS.* Newport, RI. June 1992. pp. 78–90.

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

[17] Satyanarayanan, M. "A Study of File Sizes and Functional Lifetimes." *Proc. 8th SOSP.* Pacific Grove, CA. Dec. 1981. pp. 96–108.

[18] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V. "File System Logging Versus Clustering: A Performance Comparison." *Proc. 1995 USENIX Conf.* New Orleans, LA. Jan. 1995. pp. 249–264.

[19] Smith, K., Seltzer, M. "A Comparison of FFS Disk Allocation Algorithms." *Proc. 1996 USENIX Conf.* San Diego, CA. Jan. 1996. pp. 15–25.

[20] Sweeney A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., Peck, G. "Scalability in the XFS File System." *Proc. 1996 USENIX Conf.* San Diego, CA. Jan. 1996. pp. 1–14.

[21] Transaction Processing Performance Council. "TPC Benchmark B Standard Specification." Waterside Associates. Fremont, CA. August 1990.

[22] Trent, G., Sake M. "WebSTONE: The First Generation in HTTP Server Benchmarking." Available as http://www.sgi.com/Products/WebFORCE/WebStone/paper.html.

[23] Wilkes, J., Golding, R., Staelin, C., Sullivan, T. "The HP AutoRAID hierarchical storage system." *Proc. 15th SOSP.* Copper Mountain, CO. Dec. 1995. pp. 96–108.

[24] Wittle, M., Keith, B. "LADDIS: The Next Generation in NFS File Server Benchmarking." *Proc. Summer 1993 USENIX Conf.* Cincinnati, OH. June 1993. pp. 111–128.