Matthew J. Rattigan
Department of Computer Science
University of Massachusetts Amherst
June 30, 2005
Synthesis Project Writeup

# Learning Filesystems

**Abstract:** In this paper, we explore the tradeoffs involved in designing an adaptive, high-performing filesystem allocation policy. Through simulation, we examine the performance of typical log-based and logical allocation strategies. We also present several alternative algorithms, and attempt to gain insight into why they fail to outperform the (much simpler) traditional algorithms.

## Introduction

As technology has advanced, computer components have grown rapidly in terms of speed and capacity. While processor and memory speeds have increased dramatically, hard disk throughput has remained relatively constant. As a result, today's I/O intensive applications are often disk bound. To address this problem, a great deal of research has been dedicated to improving the filesystem allocation policies that govern the use permanent storage, as even the most modern operating systems employ simplistic algorithms for performing disk layout. Researchers believe that a well-designed allocation strategy can improve data locality, thus reducing hard drive seek times and increasing I/O throughput.[14]

While the statement of the filesystem allocation problem is quite straightforward, finding an effective solution is a complex undertaking. The complex interactions between data layout and usage patterns are difficult to characterize. In this paper, we explore the tradeoffs involved in designing an adaptive, high-performing filesystem allocation policy. Through simulation, we examine the performance of typical log-based and logical allocation strategies, as well as several alternative, adaptive algorithms, none of which are able to outperform existing methods. Furthermore, we attempt to explain why existing policies hold up so well.

## Background

While hard disk capacity has increased, the basic design of drives has remained the same. Disks contain (sometimes multiple) platters and heads, and the chief source of latency is "seek time" --- the time that it takes the disk head to move from one part of the disk to another in search of a specific data block. File locality can greatly alleviate the seek problem, and as a result the physical layout of the files on disk can have a tremendous effect on system performance.[14]

Traditionally, filesytem allocation algorithms have fallen into two main categories. The first and most commonly deployed family of filesystems is based on the Berkeley Fast Filesystem (FFS).[6] Past studies have demonstrated that in many workloads, files that reside within the same logical directory tend to be accessed together. These FFS filesystem (and its descendents) attempt to exploit this property to increase file locality on disk.[2] Disk blocks are divided up into several continuous "block groups", typically 100MB or more in size. The FFS allocator attempts to place "sibling files" (files that reside in the same logical directory) within the same block group, minimizing seek times when the siblings are accessed together.

Hard disk drives have two basic operations: reading and writing. The FFS paradigm does not optimize itself for either operation, as it makes the implicit assumption that both reads and writes are equally correlated with the logical structure of the filesystem hierarchy as defined by the user. Thus in a sense, the FFS scheme is equally optimized for disk reads as well as writes, to whatever degree of access locality achieved is dependent on degree to which files in the same directory are concurrently read from or written to.

The second major category of filesystems is the "log-structured" filesystems (LFS), based on the work of Rosenblum and Ousterhout.[10,12] The LFS allocation policy is temporally rather than logically based. In this scheme, all file write operations are continuously appended to the end of the disk's "log". Thus files that are created at the same point in time are located in close physical proximity, and read locality is eschewed in favor of write locality (though in practice some amount of read locality is achieved to the extent that files created at the same time are accessed together in the future). In addition to creating new files at the frontier of the log, the LFS policy carries out all file modifications in this manner. Thus, if a 1k chunk of a 100k file is altered, the altered fragment is written to the end of the log, which may be in a completely different part of the disk than the rest of the file. Given that most files are read in their entirety, this willingness to constantly fragment files might seem risky. However, this weakness is balanced out by the fact that an LFS system virtually eliminates write-based seeks.

There is an additional bit of overhead associated with LFS systems, however. As the disk fills up, eventually the frontier of the log reaches the end of the disk. Under most circumstances, the disk may still have much unused space when it reaches this point, since files that have been removed or modified along the way have orphaned deallocated (but not reused) blocks in the regions where they were originally stored. In order to reclaim these "dirty" blocks, the LFS invokes a "cleaner". The cleaner's job is to identify segments of the disk (typically 512k in size) that have small proportions of live data in them. The (hopefully few) blocks of live data are copied to the end of the log in the same manner as a normal file write, and segment is returned to the system as a continuous block of free space suitable for log-based writes.

Both paradigms have strengths and weaknesses. Rather than focus solely on which performs better overall, we attempt to examine the specific situations where each scheme was able to achieve locality and minimize seeks. The goal of this work is to formulate a "hybrid" filesystem, similar in spirit to the work by Muller or Salmon et al.[7,11]

Previous attempts to create a "best of both worlds" allocation policy have relied on static heuristics for determining how to handle different types of files (typically segregating files into two or more size categories). We believe that a more principled approach, based on simple probabalistic models of file access patterns can inform allocation and produce a system that outperforms both FFS and LFS.

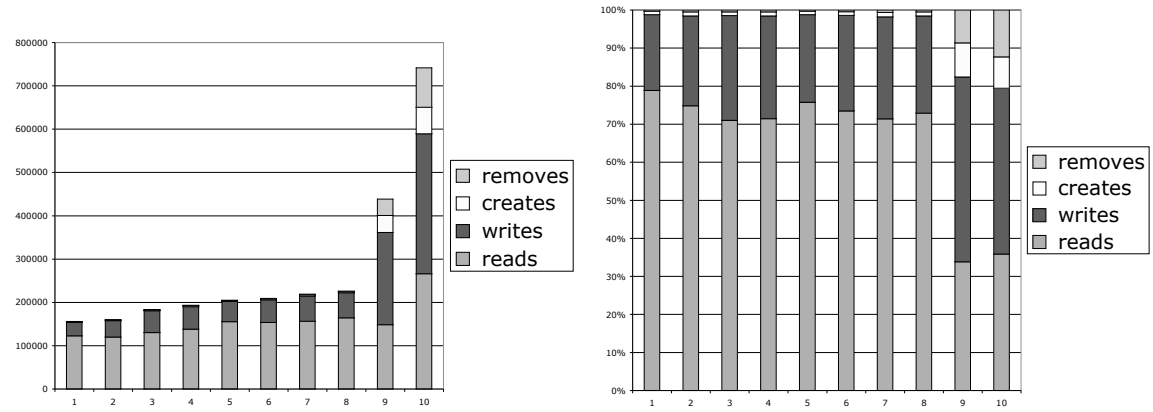
**Data**

The first data set we examined was collected from a single Linux workstation using a file trace utility written by Jim Cipar at the University of Massachusetts. While the data collected was high quality, there was simply not enough of it to provide sufficient sample size for any statistical analysis. Furthermore, the program responsible for collecting traces relied on functionality implemented in an unstable branch of Linux kernel source and was deemed unfit for use on a high traffic fileserver or webserver.

As an alternative, we examined a series of NFS traces collected by Margo Seltzer et al. at Harvard in the fall of 2001.[3] We focused our efforts around the "EECS" data set, consisting of general usage traffic from computer science and electrical engineering workstations. Since the traces capture the activity of multiple clients simultaneously accessing the same fileserver, there is an added layer of complexity added to any allocation strategy, as the usage patterns of a single client can be effectively randomized by the server traffic coming from other clients. In order to control for this effect, we chose to focus on the traffic associated with a single client at a time, and filtered the raw trace data accordingly.

For the analysis below, we worked with week-long time slices of NFS calls from ten different workstations (while the aggregated traces contain calls from 117 unique clients, the ten we analyzed represent over 42% of the total traffic). The distribution of call types can be seen below in Figure 1. Though NFS lookup and getattr calls were used to infer the logical filesystem hierarchy from the trace data, for the purposes of analysis only file read, write, create, and remove calls were considered, as they represent the "core" functionality inherent to any filesystem.

Since the data were anonymized, we do not know anything more about the clients from which they were collected. However, as we can from the breakdown of call types found Figure 1b, the ten clients seem to represent two distinct classes of user. The first type, represented by bars 1-8, fits the traditional definition of a "typical" user: most operations are reads (70-80%), with very few create or remove calls. The second class, represented by bars 9 and 10, depicts a different type of user, characterized by higher volume and greater proportions of write, create, and remove calls. For the purposes of this discussion, we will refer to these two classes of users as "read-based" and "write-based", respectively.




(a) (b)

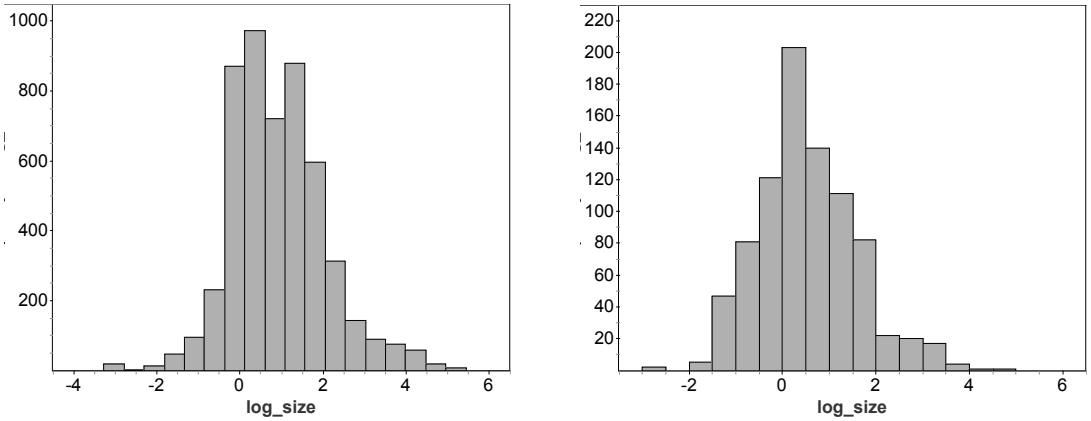
**Figure 1. Distribution of NFS call types over different data sets.**

The table in Figure 2 offers a quick characterization of the data for a single typical specimen of a read-based and write-based client (our analysis has shown that for the most part, clients are statistically interchangeable within these two classes). While the trace for the write-based client contains almost double the number of calls for its read-based counterpart, its activity is centered around far fewer files. From Figure 3, we see that the distributions of file size are similar for each, with the vast majority being of small size.

| | **read-based** | **write-based** |
|---|---|---|
| number of files | 5302 | 863 |
| total accesses | 217726 | 438638 |
| Reads | 117426 (54%) | 148477 (34%) |
| Writes | 57727 (27%) | 213049 (49%) |
| Creates | 2461 (1%) | 39262 (9%) |
| Removes | 504 (0%) | 28478 (6%) |
| mean file size (kb) | 682.1 | 174.9 |
| median file size (kb) | 5.7 | 2.5 |
| total read volume (Mb) | 731.0 | 750.6 |
| total write volume (Mb) | 625.5 | 466.3 |

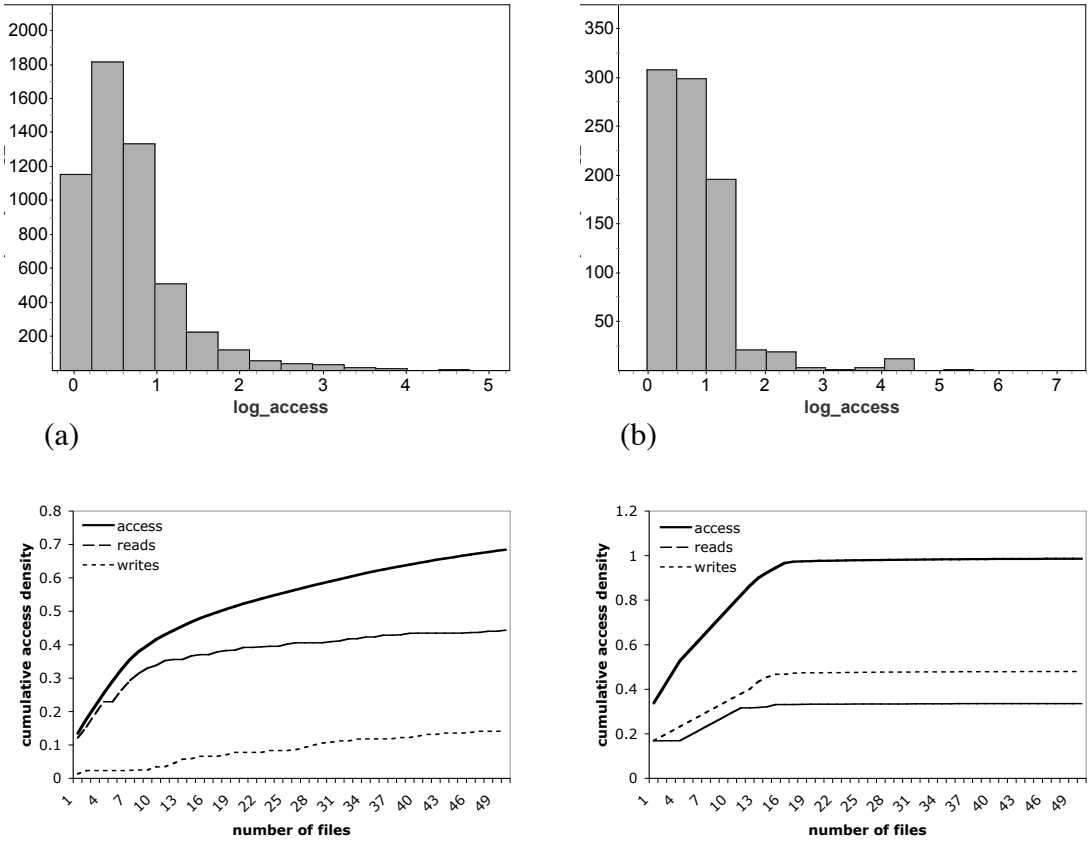
**Figure 2. Descriptive attributes from examples of read-based and write-based file traces.**

A notable aspect of both traces is the uneven distribution of activity among different files. In each trace we examined, the majority of traffic was associated with tiny fraction of the files in the filesystem. In the cases below, the top 1% of files is responsible for 69% and 74% of all NFS calls, respectively. The charts in Figure 4 detail the overall distribution of activity among individual files, as well as the cumulative density of activity for the top fifty files in each trace. While the distribution of activity toward a small number of files (<< 1%!) is more pronounced in the write-based data set, the trend is the same. Of course, this aspect of the data has important implications on the performance of any filesystem layout scheme, and will be discussed in more detail later.




(a) (b)

**Figure 3. Logarithmic distributions of file size for read-based (a) and write-based (b) file traces.**

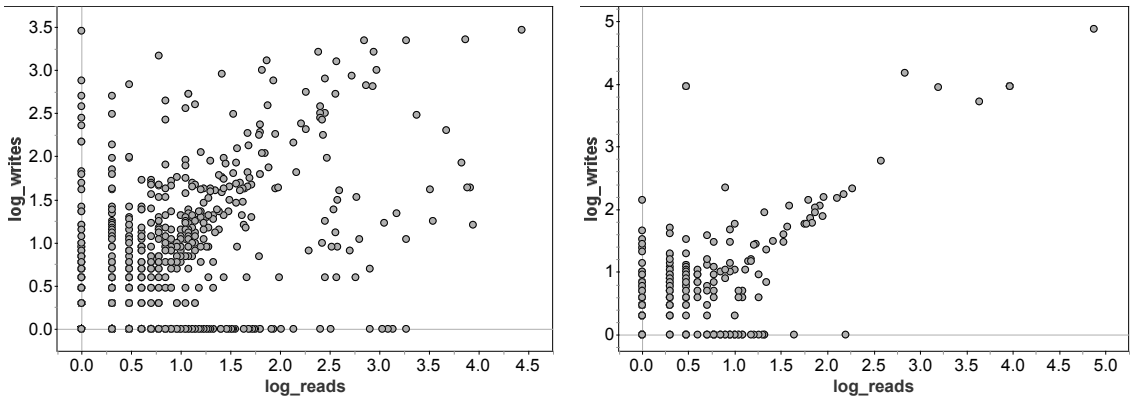



(a) (b)

(c) (d)

**Figure 4. Distributions of file access frequency across individual files for read-based (a) and write-based (b) traces, along with cumulative density distributions for the fifty most active files in each (c, d).**

Finally, Figure 5 depicts a breakdown of read calls and write calls for individual files. While there seems to be a tendency for files to be "active" (lots or read calls and write calls) or "inactive", there seems to be no clearer clusters between the call types.




(a) (b)

**Figure 5. Corresponding read and write calls counts for read- and write-based traces.**

## Standalone Models

Our first goal was to try to find evidence of latent structure within the data than could potentially guide an adaptable file allocation policy. To this end, we constructed static models of file access patterns, similar to some of the original work performed on the EECS traces.[3] These studies assert the usefulness of being able to predict individual file access attributes for use in hints systems. The goal of our experiments was to categorize the data and its structure in the hopes of lending insight into allocation policies. We created several standalone models to predict the probability of future file access types for individual files based on their "intrinsic" attributes (file size, mode, uid, gid, and extension).

Figure 6 depicts an example of a decision trees constructed for predicting file activity (in this case, whether or not a file will be written to). These models were created using the Weka data mining toolkit's J48 tree algorithm, which is based on Ross Quinlan's C4.5 algorithm. Decision trees are "selective" models, meaning that the algorithm determines which attributes are statistically relevant to determining class value, and arranges them hierarchical ly according to their predictive ability. By examining which attributes are used to create splits in the tree (and at what level), we can get a feel for which pieces of information are the most useful when making predictions. In the example below, the model first looks at the size of the file being classified ("discsize"). If the file size is zero, we determine that the file will not be written to. If the size is nonzero, we then examine the uid of file.

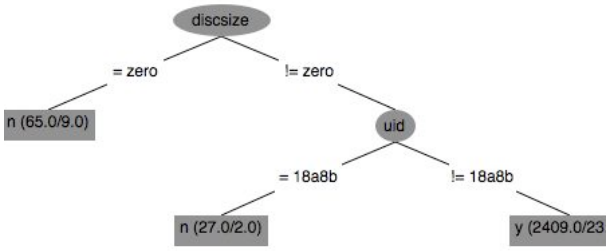
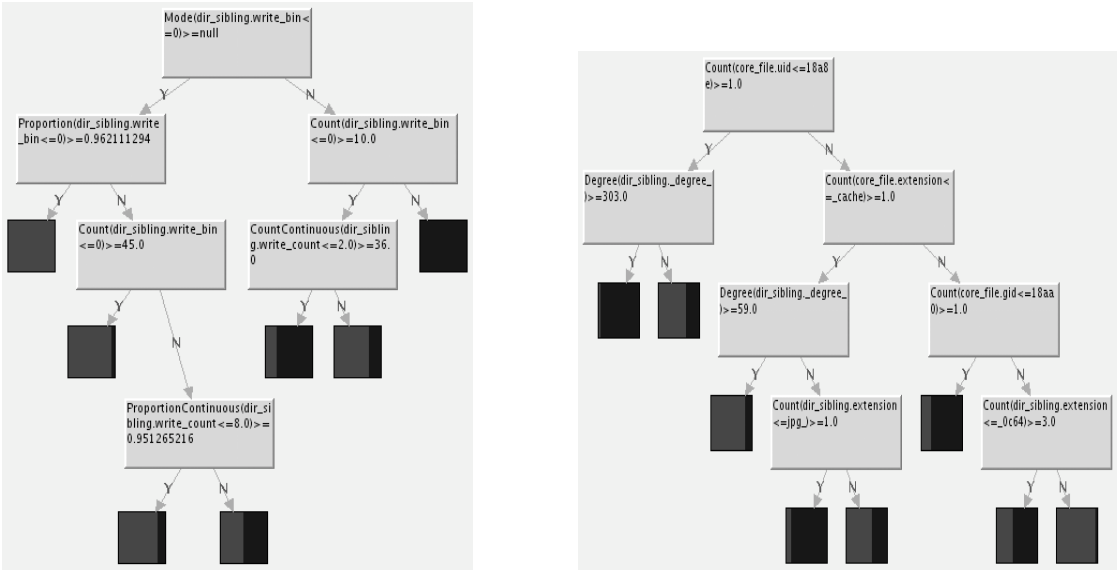



**Figure 6. Example of a decision tree for predicting file writes from file attributes, created with the Weka data mining software package.**

Our simple trees performed moderately well on our data. The file read model produced a contingency table with a corrected contingency coefficient value of 0.67 (perfect would be 1.0). Its accuracy was 94.1%, which is actually smaller than the default accuracy (determined by always predicting the majority class) of 96.0%. We fared better when predicting file writes; here, our model produced contingency coefficient of 0.91, along with an accuracy measure of 98.7% (96.3% default). On a general level, we can conclude that with our simple models, file writes are easier to predict than reads.

In addition, we created a "relational" data set from the data and loaded into Proximity in an attempt to learn Relational Probability Trees (RPTs).[8] A relational data representation allows for encoding the statistical dependencies between data instances in the form of "links". In our case, links were created to reflect the logical structure within the file hierarchy. In addition to considering the attributes of each individual file when determining its class, the RPT learning algorithm is allowed to consider the attributes of neighboring (linked) files as well. Thus instead of just splitting on attribute values as in the Weka trees, RPTs search a space of possible aggregations over the groups of attributes from all connected files. In addition, the RPT has access to purely structural attributes, such as the size of the neighborhood being considered.

The trees learned for predicting file writes can be seen in Figure 7. In the first tree (a), the best feature (utilized in several splits) is the proportion of "sibling" files that have been written to. This confirms the utility of the guiding principle behind FFS --- that is, the best indicator of file write operations is the accesses patterns of the files in its directory. The second tree (b) was constructed without supplying the class label (written) of neighboring files to the algorithm. This model splits first on the uid of the file being classified (the "core file"), then makes sub-decisions be examining either the number of siblings the file has ("degree dir_sibling") or the core file's extension. The tree below performed comparably to the standard decision trees, with a predictive accuracy of 97%.




(a) (b)

**Figure 7. Relational probability trees learned from the trace data for predicting file writes.**

While these preliminary results are encouraging, we must keep in mind the fact that these models are working offline and with perfect information. Further, it is unclear how to translate these findings into an online allocation policy. Intuitively, the predictive power of co-reads/writes within directories would associate a policy that takes the logical structure into account with better locality based performance; however, as we shall see this is not necessarily the case.

## Simulation

We constructed a filesystem simulator to evaluate the relative performance of different allocation strategies. The simulator models the behavior of an actual hard disk drive, given some basic assumptions about physical drive characteristics and operating system behavior with regard to file cacheing and prefetching. Our disk model itself is heavily abstracted: the simulator represents disk storage as a single continuous array of blocks of a fixed size (4096 bytes for our experiments). Filesystem metadata is simplified; while each file is assigned to an inode that must be updated each time one of its blocks is modified, we do not simulate indirect blocks or system-wide metedata structures found in both FFS and LFS systems. To measure seek behavior of the disk under different layout strategies, we utilize a linear model of seek time based on the physical distance between disk locations as measured in blocks, added to a "minimum movement time" (MMT) penalty that is associated with any movement of the disk head. For our experiments below, we based our simulation on a Seagate Barracuda hard disk drive, as characterized by Talagala et al.[16] In this case, the MMT is equivalent to traversing approximately 1.8 GB of physical disk space in a seek. Through the course of our experiments, we found that the MMT was a difficult factor to overcome when choosing a layout strategy,

as the benefits of locality are often trumped by the penalty incurred from moving the disk head at all.

In addition, our simulator models two separate buffer caches of varying sizes. The first of the two simulates the physical cache of the drive itself (usually modeled at 64 tracks, or 2MB). Since the cacheing algorithms for actual hard disks in production are not publicly available, we were forced to make further assumptions regarding cache behavior. In our model, when a data block is read from disk, its entire track is loaded into the buffer cache. It remains there until it is evicted by another incoming file as determined by a "least recently used" (LRU) policy. In addition, we also modeled an operating system level file cache (128MB). The file cache operates on a block level (typically 4k each), and also utilizes an LRU policy. Additionally, the file cache performs a simple form of prefetching when consecutive blocks in a file are read, in accordance with the behavior of modern Linux kernals.

In both cases we assume a zero access penalty for reading files resident in the cache. Thus blocks that are repeatedly accessed incur no seek penalty. While the simplifications mentioned above may be a bit heavy handed, they do not unfairly favor or penalize one scheme over another, so it is unlikely that our findings would not be applicable to an actual deployed system.

Since the traces being used are small slices of time (one week), some additional assumptions were made concerning the initial state of the disk at the start of a trace. Using the modification times (mtime) for each file are contained in the trace data, we were able to "pre-build" the filesystem: for each file with an *mtime* value that was previous to the start time of the trace, we created an artificial write call to initialize the file to the correct size. While this obviously will not accurately represent the true state of the filesystem at the start of the trace, it will be a much closer approximation of behavior than working with a disk that is initially empty. We would also like to note that these “pre-write” calls were not counted in any of the measurements discussed below.

## Experimental Results

Our first set of experiments was an attempt to examine the effects of different simulation parameters on the performance of LFS and FFS systems. By exploring different scenarios, we hoped to gain insight into how each filesystem functions.

### Disk Size

It was previously reported that the FFS algorithm was especially sensitive with regard to disk size. Given that files are evenly spread across the entirety of the disk, having a larger disk meant spanning longer distances to seek to files found in different portions of the disk. Thus, we concluded that having lots of free disk space severely degraded performance! The effect can be seen in Figure 8a below. However, these conclusions were drawn from data supplied by a simpler implementation of the simulator; specifically, the seek model did not include an MMT penalty for each operation, and thus

seek times were based entirely on physical disk address proximity. A revised graph is shown in Figure 8b. Here, the introduction of a sizeable MMT largely washes out any differences in aggregate seek times produced by disks of varying sizes.

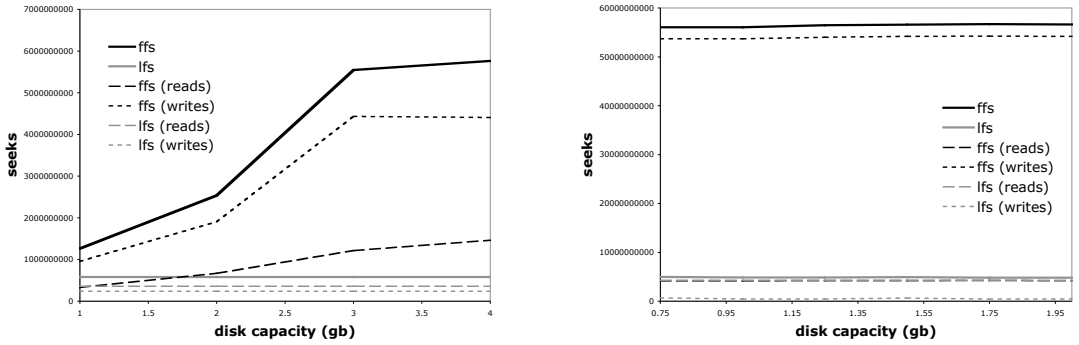
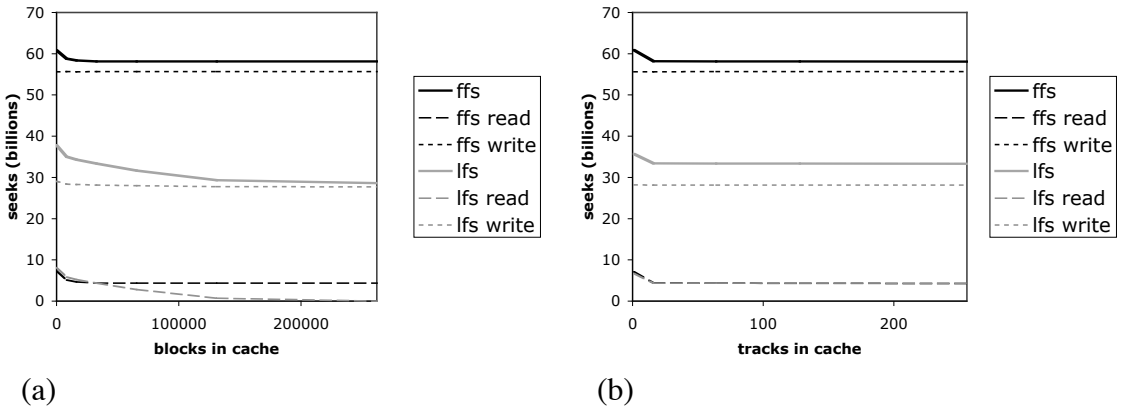



**Figure 8. (Non-) effects of disk size on FFS performance. Previously reported (and erroneous) results (as derived from a simplified seek model that did not incorporate MMT are shown in (a), revised results in (b).**

We've included the contrast here because it highlights a very important point: compared to the penalty associated with performing any seek at all, there's not much difference between "short seeks" and "long seeks". Indeed, this fact does not bode well for file layout optimization. As these results demonstrate, simply improving data locality is not enough to really improve performance. As a result, new policies must reduce the total number of seeks, which requires (potentially impossibly) accurate anticipation of file usage patterns.

## Cache Size

The next set of simulations compares the effect of cache size on a file trace. Since read-based seeks are a performance bottleneck for both FFS and LFS (especially in the case of the latter), we needed to be sure that increasing the cache size did not trump the utility of any allocation strategy decisions. As we can see from Figure 9, the larger the cache, the more overall seek performance is improved; however, the performance curve levels off pretty quickly for both LFS and FFS. In addition, increasing the size of the hard disk cache does not seem to have an effect after a minimum size of around 64 tracks is reached. This would lead us to believe that in terms of seek performance, the hard disc buffer cache is actually extraneous to the much larger operating system file cache.




(a) (b)

**Figure 9. Effect of file cache size (a) and hard disk cache size (b) on seek performance under FFS and LFS policies.**

In addition, all of the above figures illustrate some of the key performance tradeoffs associated with the FFS and LFS paradigms. Clearly, the LFS strategy results in far fewer seeks than FFS. This is largely due to the amount of seeks incurred while writing files under FFS. Since files are organized on disk according to their logical directory structure, the drive must continually seek to write files that are stored in separate directories, and this dominates performance. In contrast, LFS is able to minimize write seeks, and while it performs better overall, its main performance bottleneck is related to reading files (though, as expected, the problem is partially alleviated as the cache gets larger).

Another notable aspect of these results is the fact that as far as read-based seeks are concerned, FFS slightly outperforms LFS regardless of cache size. This would suggest that the FFS notion of locality determined by the logical file structure is sound.

It should be noted that these effects may be exaggerated by the fact that we are largely ignoring the penalties associated with the cleaner for the LFS, as the cleaner does not get invoked on the disk filesystem until its free space is limited. Proper cleaning policy is worthy of an entire study in itself, and beyond the scope of the current work. We use a the selection criterion described by Seltzer et al in [12], and invoke the cleaner when the filesystem is over 50% full and more than 10% of the blocks used can be reclaimed. It is worth mentioning that if we assume that cleaning is performed during idle time, we can ignore its associated performance penalty entirely.[5] In fact, since by relocating existing files the cleaner may increase locality our model is missing out on possible optimizations in the cases where the cleaner does not get invoked.

For the sake of completion, we also tested the effects of altering the “block group” size for FFS, as well as the “segment size” for LFS. Neither produced any significant differences in performance.

**Alternative Allocation Policies**

The results presented us above led us to believe that for the types of workloads found in our trace data, an LFS approach is superior to an FFS strategy. The next step was to construct a new, better policy based on the data produced by the simulator. Unfortunately, our efforts to create a high performance, adaptable filesystem allocation policy were largely met with failure. Overall seek performance is depicted in Figure 10 below, and details about each approach are in the sections following.

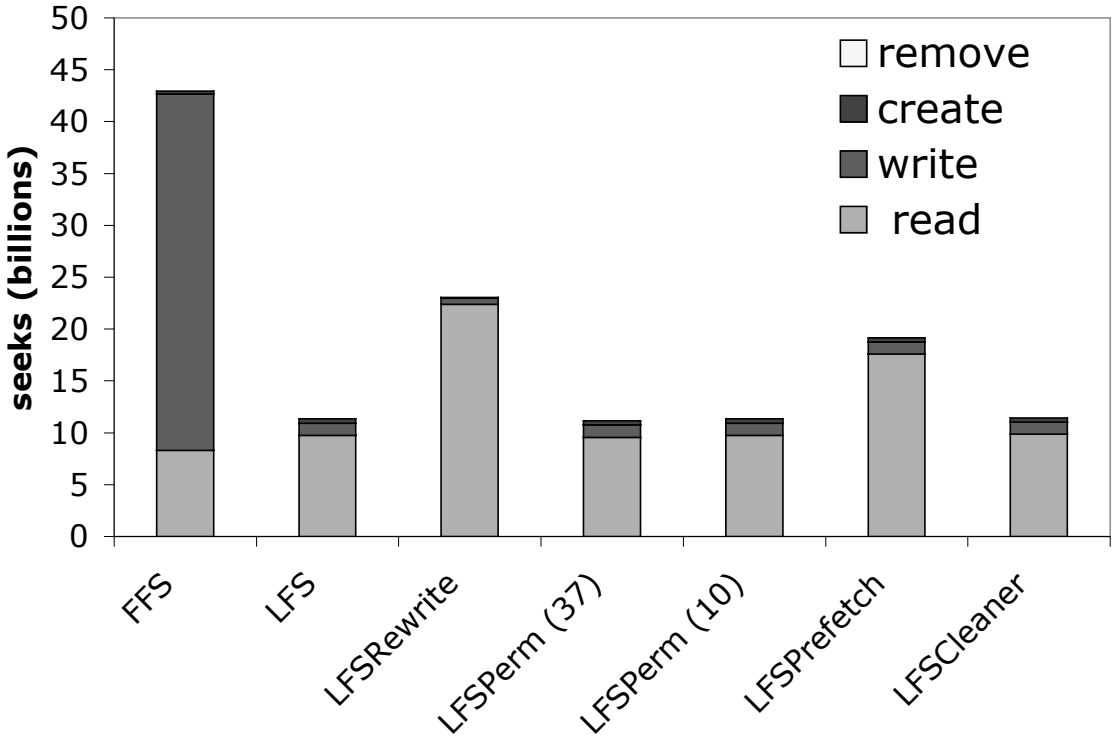



**Figure 10. Simulated performance of several filesystem allocation policies, broken out by call type.**

**LFSRewrite**

Our first attempt at constructing a better performing filesystem centered around trying to marry the strengths of the FFS and LFS approaches. While in general, LFS outperforms FFS by a wide margin. Since the former is optimized to eliminate write-based seeks, we wanted to see if we could introduce some amount of temporal locality into the placement of files being read as well. The algorithm hinges on the fact that write operations come "cheaply" in LFS, and that the main seek penalties are incurred when seeking long distances to read a block that is scattered on a different part of the disk. The LFSRewrite algorithm reorganizes file locations as files are sent in and out of the file cache. When a block is cached, it remains in the cache until it is discarded in favor of a more recently used block. The LFSRewrite algorithm relocates the discarded block to the current writing frontier of the disk. Since write operations are largely seekless, we hoped that be relocating recently to the "hot" part of the disc we could improve performance.

This approach performed the worst of those that we implemented (or, more precisely, the worst of those that were implemented and being reported). Upon examination of the file traces, we found that we were being "outdone" by the LRU. It is only useful to relocate a file that is going to be read again in the near future. Such "hot files" tend to stay in the cache anyway, and therefore are never candidates for relocation. Therefore, the majority of the files that were being relocated were never read again --- resulting in lots of extra writes. Note that in the chart above, seeks incurred from rewriting are categorized as "readseeks", as they are triggered by a file read that ejects data blocks from the file cache.

**LFSPerm**

The next strategy comes out of the analysis of which files are incurring the most seeks. As we see in Figure 11, the seek penalties are unevenly distributed among files --- in a typical trace, the top 1% of files account for 70% of all seeks. In theory, if we could identify these "hot" files, an adaptive system could learn to give them cache priority over other files whenever possible, cutting down on read seeks. We implemented an offline version of this algorithm in two variations: LFSPerm37 permanently caches the top 1% of files (37), while LFSPerm10 only caches the top ten. Our approach is as follows: first we analyzed the input trace to identify the hot files. The LFSPerm allocator is then supplied with a list of hot files to permanently cache whenever possible.

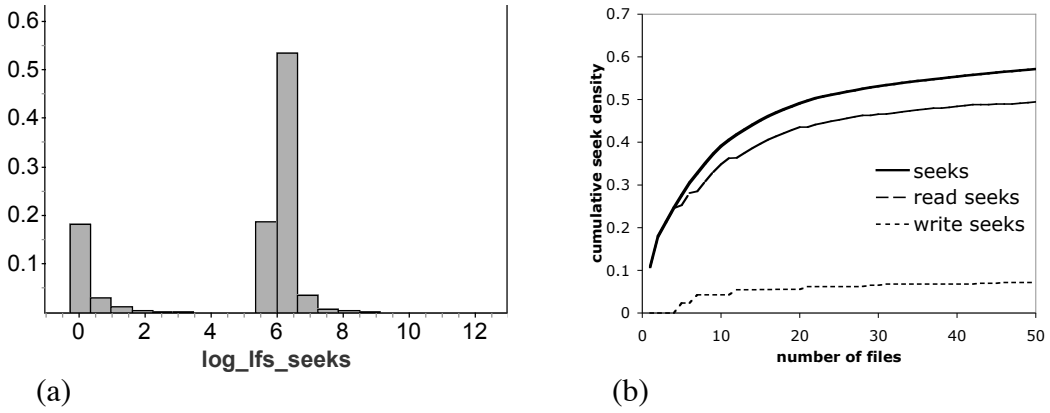
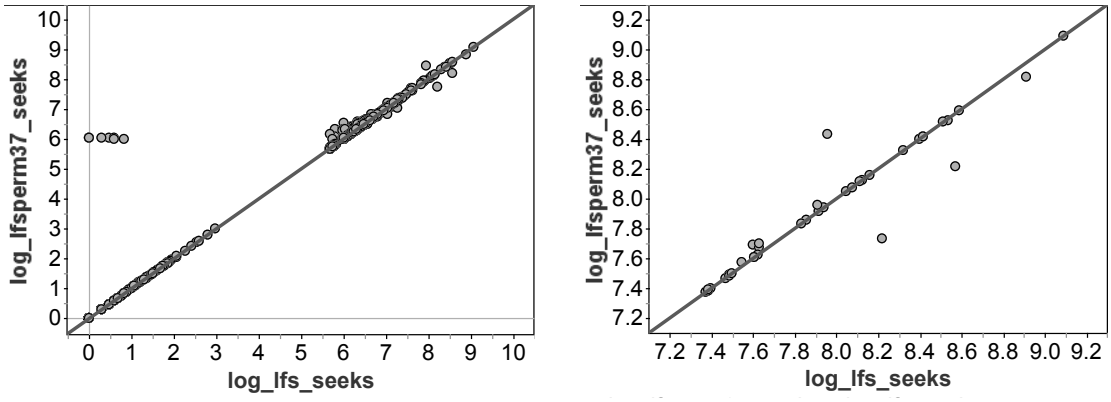



**Figure 11. Distribution of seek penalties across individual files for LFS allocation policy (a), along with cumulative density for the top fifty files (b).**

Unfortunately, LFSPerm was the worst performing policy we implemented. A bit of insight into the effects of this strategy can be seen below in Figure 12. This first chart plots the seeks incurred by each individual file under both schemes. The only remarkable deviation in behavior comes from a cluster of files that register higher seek counts under the new policy. This is due to the fact that even though only a small number of files are being put in the cache permanently, they are sizeable enough to essentially "take over" the cache, so as a result the other 99% of the files are never being cached, resulting in much higher penalties.




(a) (b)

**Figure 12. Comparison of seeks incurred under LFS and LFSPerm37, plotted for all individual files (a) and "hot" files (b).**

## LFSPrefetch

Our next approach stems from some analysis performed on the relationships between different hot files in terms of access patterns. We hoped to exploit concurrent accesses of hot file pairs by expanding the prefetching functionality of the file cache. Normally, when consecutive blocks of a file are read, the operating system assumes that the file will be read further, and actively caches upcoming blocks. Our idea was to extend the prefetching to "partner" files that are commonly accessed along with the file being read.

First, we calculate the co-access patterns of each pair of files by counting the number of times they are both read within a sliding window of time (in our case, 30 seconds for a week-long trace). We then perform a randomization test to calculate which pairs are linked beyond what we'd expect from random chance. This is a subtle distinction; without the randomization test, two heavily read files may appear to be linked, when in reality they are simply both accessed frequently (the "regulars" at a coffee shop may often be found there at the same time, yet never show up together). Once the list of statistically valid "hot pairs" is determined, we pass it along to the LFSPrefetch allocator and rerun the trace.

The effects of the prefetching are shown in Figure 13. The plots depict the amount of seeks incurred for individual files under the basic LFS strategy and LFSPrefetch. Figure 13a shows the results for all of the files in the trace. Again we can see the tradeoffs inherent to working with a finite cache. For a subset of files, the prefetch is a win and we can eliminate seeks related to reading them. However, this comes at the cost of prematurely evicting other files from the cache, which in turn cost more seeks. As seen in 13b, the hot files themselves unfortunately perform pretty badly. In most of the cases, the aggressive prefetching is overkill --- hot files are prefetched when the do not need to be, and this incurs read seeks. The overall effects can be seen in the histograms in figure 13c. Clearly, we have shifted some of the mass of the distribution to the right, resulting in an the overall performance hit seen in the bar graph above.

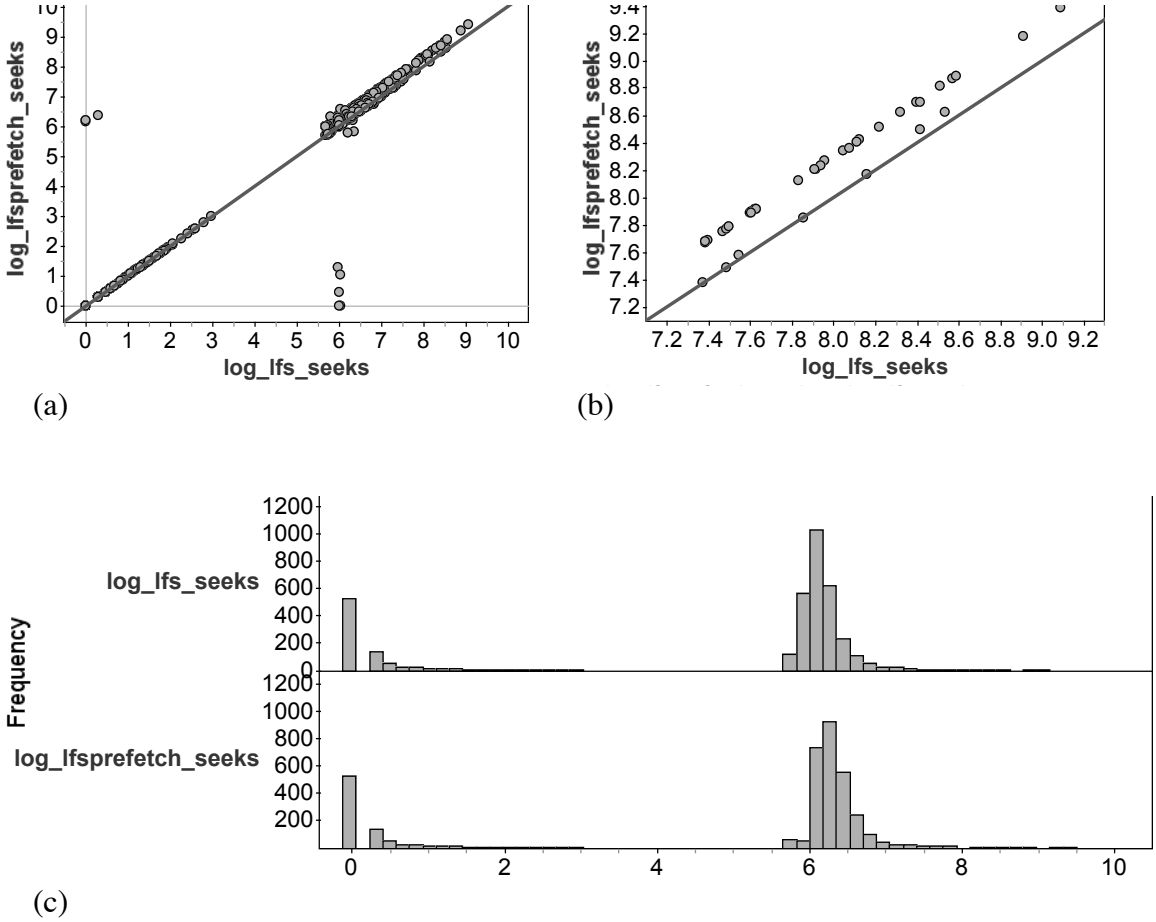



**Figure 13. Comparison of seeks incurred under LFS and LFSPrefetch, plotted for all individual files (a) and "hot" files found in prefetched pairs (b). Histogram depicting frequency distribution of seeks across files (c).**

## LFSClean

The LFSClean allocator is yet another variant on LFS. Like in the case of the LFSRewrite, this policy attempts to alleviate the penalties incurred when making long seeks across the disk to read file blocks. The "normal" LFS allocator maintains a frontier on the disk, where all data is written, allowing the disk to achieve optimal bandwidth performance when writing. However, a single read call sandwiched between a series of writes can be quite a disruption --- the disk head must travel all the way to the block being read, then all the way back to before it can resume writing. The LFSCleaner attempts to avoid this situation by opportunistically relocating the writing frontier on the disk as the head is moved about by different reads. Under this policy, whenever a write request is made, the allocator checks to see where the disk head is in relation to the current frontier. If it is far away (as a result of a previous read call), it then examines the contents of the segment where it is currently located. If that segment is fragmented beyond given threshold, the allocator cleans that segment and relocates the writing frontier to the current position. This preserves continuous writing while also introducing an element of locality with recently read blocks.

Once again, though, our efforts to be clever are met with non-results. The culprit here is the nature of seek penalties themselves. While this scheme does save write-based seeks incurred after a distant file read, the act of (possibly prematurely) cleaning a segment balances out any possible gains. It should be noted that if the MMT of the disk head is set to 0, this algorithm slightly outperforms LFS, as it alleviates the problem. This illustrates the main hurdle that improved allocation strategies face: locality, and therefore decreased seek distance are not enough to show real improvement.

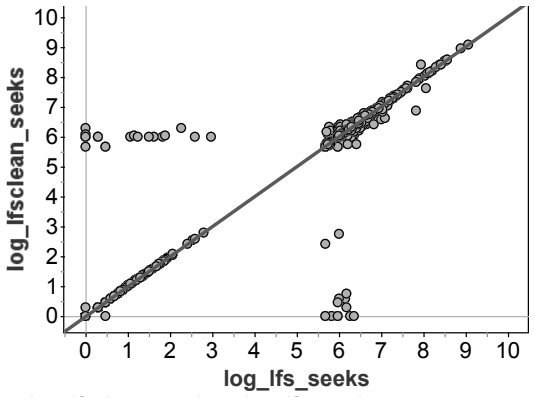



**Figure 14. Comparison of seeks incurred under LFS and LFSClean, plotted for all individual files.**

## Discussion

So how to explain our lack of success? Certainly, pinpointing the reasons why each approach failed is simple enough, as exhibited above. Conversely, though, it is very difficult to pinpoint which facets of an algorithm will make it successful (of course, if we could do this, we would, and the performance graphs above would look a lot different). While we do not have definitive answers about what makes this problem so difficult, we do have some intuitions.

First of all, the problem and solution space are both incredibly complex, and not well understood, a fact that has left some researchers in the field wondering if the disk allocation problem is even solvable.[11] Aside from being unable to devise effective solutions, it is unclear how to gauge how effective the current solutions are. We currently do not have a method for an optimal layout of data given a trace, and therefore are unable to see how close a given layout is to optimal (or explore the properties of an optimal layout in hopes of emulating them). We suspect that we can reduce the much studied "offline linear access problem" to our optimal layout problem. As the former has been shown to be NP-hard[1], doing so for a trace made up of thousands of data blocks is computationally infeasible. Approximation may exist, though, and it is probably worthy of future investigation.

Another inherent difficulty stems from the lack of file attributes on which to base layout decisions. Figure 15 shows graphical representations of file size, file owner permissions,

and file extension and how they relate to seek penalties. In each case, there is little or no correlation between attribute values and resulting seeks. Graphs depiction individual operation counts or cache behavior are similarly unhelpful, and learned models incorporating combinations of attributes are unable to perform above default levels of accuracy.

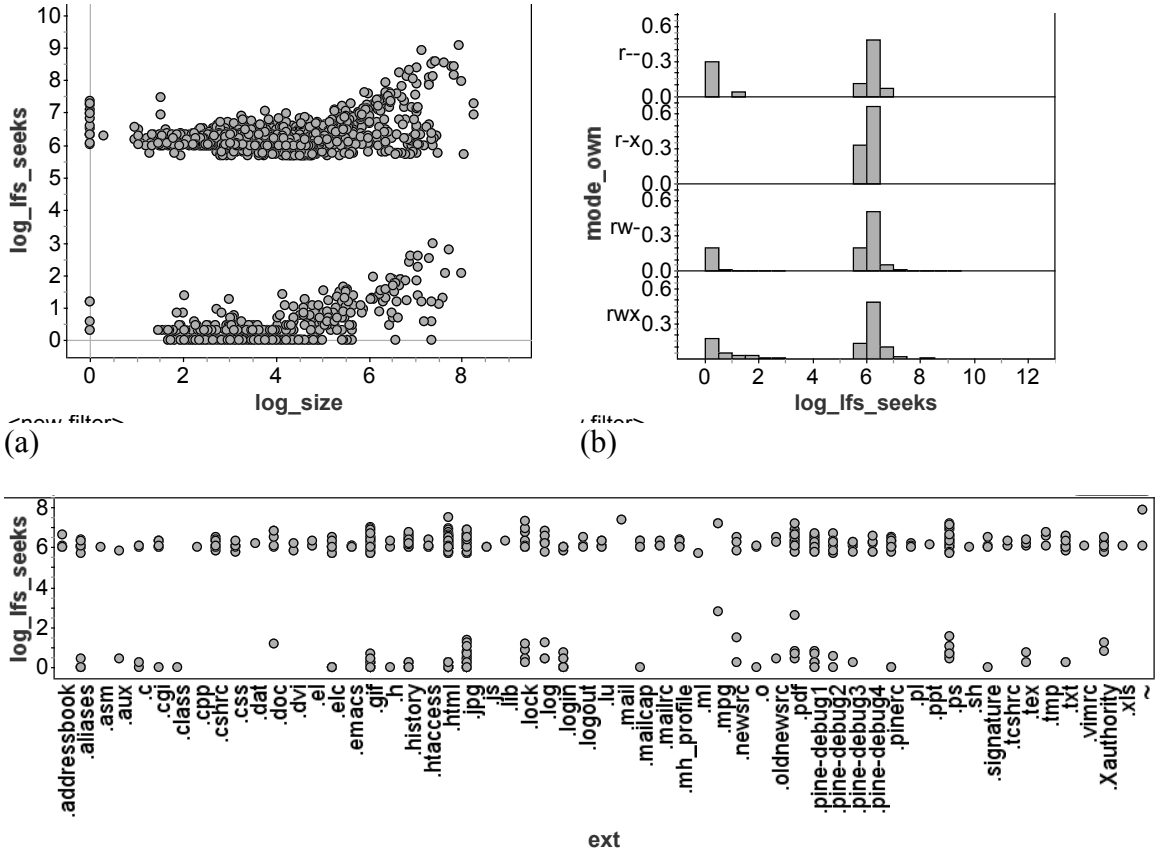



**Figure 15. Graphical depictions of unhelpful attributes: file size (a), owner permissions (b), and file extension (c).**

The skewed distribution of file activity also makes this a difficult environment to perform learning. The curves in Figures 4 and 11 illustrate the crux of the problem: no matter what your layout policy, a few dozen files (out of thousands) are going to be associated with the majority of the disk movement. With the attributes we have, differentiating these hot files from the masses is a near impossible task.

Figure 16 below depicts the relative rankings of files by seek total under LFS and FFS. While some files incur a large (or small) number of seeks under both policies, most do not. This is strong evidence that there is very little about the files themselves that can indicate access patterns.

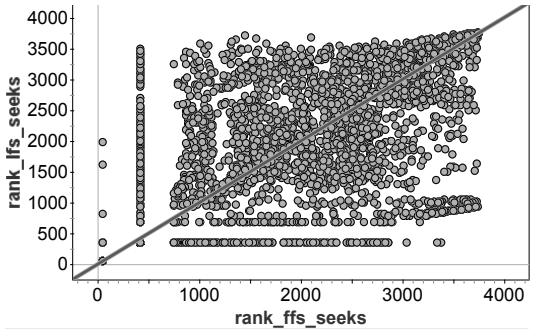



**Figure 16. Relative rankings of individual files by seek penalties incurred for FFS and LFS... a bit of a mess.**

Finally, our ability to improve hard drive performance through filesystem layout may be limited by the physical characteristics of hard disks themselves. Given the minimum movement time (MMT) of a drive head, in many ways a seek is a seek is a seek, regardless of the geographic distance covered on disk. Thus an allocation policy that shortens seek distances will have little actual effect on performance. Rather, to make an actual impact the layout algorithm must reduce the number of seeks performed. Unfortunately, doing so requires a high level of predictive precision with regards to user behavior, and very well may be impossible to achieve in a noisy data environment, as exemplified by the examples presented above.

## References

1. Ambühl, C. Offline List Update is NP-hard. Proceedings of 8th Annual European Symposium on Algorithms (ESA), pages 42-51.

2. Card, R. T. Ts'o, and S. Tweedie. Design and Implementation of the Second Extended Filesystem. In Proceedings of the First Dutch International Symposium on Linux, number ISBN 90 367 0385 9. Laboratoire MASI - Institut Blaise Pascal and Massachussets Institute of Technology and University of Edinburgh.

3. Ellard, D., M. Mesnier, and E. Thereska, Gregory R. Ganger, and Margo Seltzer. Technical Report TR-14-03, Division of Engineering and Applied Sciences, Harvard University.

4. D. Liben-Nowell, J. Kleinberg. The Link Prediction Problem for Social Networks. Proc. 12th International Conference on Information and Knowledge Management (CIKM), 2003.

5. Lumb, C., J. Schindler, G. Ganger, and D. Nagle. Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives. Proc. Symp. Operating Systems Design and Implementation, 2000.

6. McKusick, M., W. Joy, S. Leffler, and R. Fabry, "A Fast File System for UNIX", ACM Transactions on Computer Systems 2, 3 (August 1984), 181-197.

7. Muller, K. and J. Pasquale. "A High-performance Multi-structured File System Design," Proceedings of the 1991 ACM Symposium on Operating System Principles, Asilomar, CA, October 1991.

8. Neville, J., D. Jensen, L. Friedland and M. Hay (2003). Learning Relational Probability Trees. Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.

9. Reingold, N., and J. Westbrook: Off-Line Algorithms for the List Update Problem. Inf. Process. Lett. 60(2): 75-80 (1996)

10. Rosenblum, M., and J. Ousterhout. The design and implementation of a log-structured file system. Proceedings of the 13th Symposium on Operating System Principles, pages 1--15, October 1991.

11. Salmon, B., E. Thereska, C. Soules, J. Strunk, and G. Ganger. Challenges in Building a Two-Tiered Learning Architecture for Disk Layout. Carnegie Mellon University Parallel Data Laboratory Technical Report CMU-PDL-04-109. August, 2004.

12. Seltzer, M., K. Bostic, M. McKusick, and C. Staelin. "An Implementation of a Log-Structured File System for UNIX", Proceedings of the Winter 1993 USENIX Conference, San Diego, CA, January 1993, 307-326.

13. Seltzer, M., and M. Stonebraker. Read Optimized File Systems: A Performance Evaluation Proceedings 7th Annual International Conference on Data Engineering, Kobe, Japan, April 1991, 602-611.

14. Smith, K., and M. Seltzer. File Layout and File System Performance. Harvard University Technical Report TR-35-94, December 1994.

15. Smith, K., and, M. Seltzer. A Comparison of FFS Disk Allocation Polices. Proceedings of the 1996 Usenix Technical Conference, San Diego, CA, January 1996.

16. Talagala, N., R. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. University of California, Berkeley 1999 number CSD-99-1063.

17. Zhang, Z., and K. Ghose. yFS: A Journaling File System Design for Handling Large Data Sets with Reduced Seeking., Proceedings of the USENIX Symposium on File Systems and Storage Technlogies (FAST '03), 2003.