To appear in *Proc. Sixteenth ACM Symposium on Operating System Principles*, Saint Malo, France, October 5-8, 1997

Improving the Performance of Log-Structured File Systems with Adaptive Methods

Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello,
Randolph Y. Wang and Thomas E. Anderson

Computer Science Division, University of California, Berkeley

Abstract

File system designers today face a dilemma. A log-structured file system (LFS) can offer superior performance for many common workloads such as those with frequent small writes, read traffic that is predominantly absorbed by the cache, and sufficient idle time to clean the log. However, an LFS has poor performance for other workloads, such as random updates to a full disk with little idle time to clean. In this paper, we show how adaptive algorithms can be used to enable LFS to provide high performance across a wider range of workloads. First, we show how to improve LFS write performance in three ways: by choosing the segment size to match disk and workload characteristics, by modifying the LFS cleaning policy to adapt to changes in disk utilization, and by using cached data to lower cleaning costs. Second, we show how to improve LFS read performance by reorganizing data to match read patterns. Using trace-driven simulations on a combination of synthetic and measured workloads, we demonstrate that these extensions to LFS can significantly improve its performance.

1 Introduction

File system designs have long been driven by changes in the cost and performance of the underlying hardware. A designer must consider the relative cost per byte of memory versus disk [Rose92a], the relative performance of the CPU versus a network access versus a disk access [Dahl94], the relative magnitudes of seek time, rotational delay, and disk bandwidth [Selt90], not to mention changes in the workload placed on the file system.

As a concrete example, the management of free blocks on disk has evolved over the past two decades to reflect hardware technology changes. Early file systems, such as the original UNIX file system [Ritc74], used a simple on-disk linked list to track free blocks. Later, systems such as the BSD Fast File System (FFS) [McKu84], replaced the on-disk linked list with an in-core bitmap, allowing the file system to optimize block allocation to keep related data, such as blocks within a file, as adjacent as possible on disk. By this point, CPU cycles had become cheap enough relative to disk access costs to make the overhead of searching the bitmap small compared with the potential improvement in disk read performance from better block allocation policies. More recently, the increasing capacity of disks combined with the increasing use of RAIDs has driven some to abandon bit maps for B-trees to reduce the CPU overhead of searching for a free disk block [Swee96]; a 100 disk system today could require over 10 MB of bitmap.

These changes pose a tremendous challenge to file system designers. Although it may be possible to design a system that is efficient for today's hardware and workload patterns, file system implementations are often used for decades, long after their design assumptions are no longer valid. For example, the BSD Fast File System is still in widespread use fifteen years after it was designed. In that time, disk capacities and bandwidths have increased by over two orders of magnitude, while disk access times have improved at a much slower rate. In addition, workloads can change dramatically over periods much shorter than file system lifetimes.

In this paper, we propose and investigate a design principle for file systems, called self-tuning. A self-tuning system (1) measures the physical characteristics of the underlying hardware, (2) measures the workload placed on the file system, and (3) adapts the file system behavior to match. Building a self-tuning system requires new algorithms that monitor the environment and adjust behavior appropriately. The alternative to self-tuning is building a file system for a fixed point on the moving target of hardware and workload characteristics, and either living with the resulting system well past its applicability or rebuilding the system from scratch every few years. A plethora of knobs could be added, but they are as likely to be mistuned as well-tuned.

We explore self-tuning by means of a set of four enhancements to the design of a log-structured file system (LFS) [Rose92a]. LFS research has been a good case study of the need for adaptive methods because it has shown the difficulty of designing a file system to have good performance across a wide spectrum of workloads, even for a fixed technology point.

In LFS, disk storage is organized into a segmented, append-only log; disk writes are batched together to the end of the log. Periodically, a garbage collection process called the *cleaner* locates dead space in the log and coalesces it into large free extents that are then available for new log

This work is supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 0401156), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Exabyte, Hewlett-Packard, Microsoft, Siemens, IBM, Sun Microsystems, and Xerox Corporation. Matthews was also supported by a National Science Foundation Graduate Fellowship, Roselli by a Department of Education GAANN fellowship, Costello by a California MICRO Fellowship, and Anderson by a National Science Foundation Presidential Faculty Fellowship. The authors can be contacted at:
{neefe, drew, amc, rywang, tea}@cs.berkeley.edu.

writes.This architecture enables an LFS to offer superior performance for workloads with frequent small writes, read traffic that is predominantly absorbed by the cache, and sufficient idle time to clean the log. However, it has dramatically lower performance for some other workloads, such as those dominated by random updates to a full disk with little idle time to clean [Selt93, Selt95a].

This dichotomy in LFS performance has led to a debate among LFS researchers [Selt93, Oust95a, Selt95a, Oust95b, Selt95b, Oust95c], and has led many to conclude that LFS is an interesting, but impractical, idea. We would like to refocus the discussion away from comparing LFS and FFS, to how to design a single file system with good performance across a wide range of workloads and hardware characteristics.

We make four contributions to improving the performance of log-structured file systems. We evaluate our improvements to LFS using trace-driven simulations of both synthetic and measured workloads.

Three of our optimizations concern improving LFS write performance. First, we show how to choose the LFS segment size by trading transfer efficiency against cleaning efficiency. Second, we show how to combine traditional LFS cleaning with an alternate garbage collection mechanism called hole-plugging [Wilk96]. Our approach adapts to changes in disk utilization and workload to avoid the traditional LFS performance cliff at high disk utilizations for random updates, while still preserving the advantage LFS has at lower disk utilizations. Third, we consider how to reduce cleaning costs by taking advantage of cached data when cleaning. All told, these three optimizations can improve LFS write performance by up to a factor of four at high disk utilizations.

Our final optimization concerns improving LFS performance for reads that miss in the cache. The original LFS work was motivated by the prediction that future systems would have increasingly large memory caches relative to disk capacity, so that fewer reads would reach disk. As a result, write cost would dominate file system performance. This was a reasonable prediction when LFS was first introduced. At that time, the cost per byte of DRAM had been decreasing relative to that of disk; since then, however, this trend has reversed. Interestingly, LFS is easily adapted to improve read performance; the LFS cleaner already has a mechanism for atomically moving data around on disk and for creating large regions of contiguous space that can be used for coalescing related data together. We demonstrate that a dynamic disk reorganizer can be used to improve disk read performance by nearly a factor of two for a workload in which read patterns do not match write patterns.

Many other systems have applied self-tuning principles at some level; our work was initially inspired by these efforts. For example, TCP implementations have long measured round-trip delays to determine appropriate time-out values [Jaco88]. More recently these implementations have begun to adapt to patterns in how packets are dropped by the network under congestion [Math96]. In the file system arena, AutoRAID [Wilk96] adapts the amount of disk space devoted to mirroring vs. RAID-5 based on the percentage of free space available; AutoRAID also moves data between the mirrored region and the RAID-5 based on the pattern of writes to the data.

Our four enhancements to LFS are by no means an exhaustive list of possible applications of self-tuning to LFS. Additional opportunities include, to name just a few: altering the policy by which segments are chosen for cleaning based on whether updates occur randomly or exhibit locality; adapting the write buffer organization to segregate related data, depending on the available memory, level of multiprogramming, and sync frequency; adaptively clustering blocks during cleaning to maximize locality of future updates. Self-tuning provides a conceptual framework for developing robust solutions to these issues.

The rest of the paper describes our work in more detail. Section 2 provides background on a variety of file system paradigms. Section 3 describes our evaluation methodology, including a description of our simulator and the traces we used. Sections 4 and 5 outline how LFS can be changed to improve write and read performance, respectively. Section 6 describes related work. We summarize our conclusions in section 7.

2 Background

Much debate concerning the best paradigm for building file systems has arisen because each of the major approaches has environments for which it is well suited. Our goal in this paper is not to compare LFS directly with other file systems because such comparisons are highly dependent on workload and implementation details. Rather, our goal is to understand the strengths and weaknesses of each system and to use that information to design adaptive algorithms that allow LFS to retain its strengths as well as incorporate the strengths of other systems.

The traditional approach to building file systems has been to place most of the burden of minimizing seeks and rotational delays on the disk block allocation policy. For example, FFS [McKu84] places new data and metadata blocks on disk near other semantically related blocks (e.g., blocks within the same file or within the same directory). In update-in-place file systems such as FFS, once a block has been placed in a given disk location it does not move— all subsequent references to the block, both reads and updates, will be sent to that location. Particularly when dynamic access patterns follow semantic relationships (e.g., when large files are read or written in large chunks), this can offer good performance [McVo91, Selt95, Smit96]. However, performance can decrease over time as the disk becomes fragmented, particularly as the disk fills up, making it harder for the allocation policy to find appropriate slots for new blocks [Smit97]. Worse, this performance penalty persists; without a disk reorganizer, once the disk fills up, performance can be negatively impacted from then on.

An update-in-place approach also has significant performance costs associated with crash recovery, both during recovery itself and during normal operation. A logically atomic update to the file system may require several physical disk writes; for example, creating a new file requires disk writes to the directory containing the file, the inode describing where to find the file's data blocks, the free block list, etc. In FFS, this is accomplished by applying each update synchronously to disk in a consistent order, so that the crash recovery procedure can detect logical operations that were in progress at the time of the crash [Kowa78]. These synchronous updates can severely limit the effective disk bandwidth (although the ordering constraints can be loosened in some circumstances [Gang94]). Perhaps more importantly, crash recovery requires scanning the entire disk; for example, it can take over 10 minutes to recover a modern 9 GB FFS disk after a crash.

Write-ahead logging file systems were designed to simplify crash recovery [Hagm87, Chut92, Birr93, Cust94, Veri95, Swee96]. Write-ahead logging batches metadata updates into a log. After the log is safely on disk, the updates are copied into fixed disk locations, placed as in an update-in-place system. After most failures, only the log, rather than the entire disk, must be examined in order to recover. The log always represents a consistent set of changes to the file system. In addition to more efficient recovery, write-ahead logging can sometimes offer better write performance than simple update-in-place by batching many small writes together into one larger log write and by reordering the second in-place writes to minimize seek and rotational delay. Because the final disk location of the data is the same, in the absence of contention between reads and writes, read performance is identical whether update-in-place or write-ahead logging is used.

Log-structured file systems extend the write-ahead logging approach by treating the log itself as the only storage location [Rose92]. Both data and metadata are written to the log in large contiguous regions, called *segments*. LFS also provides periodic checkpoints which allow recovery to proceed efficiently from the most recent checkpoint to the tail of the log.

Logically, LFS treats the disk as an infinite append-only log. In practice, however, when LFS fills the disk with new log writes, it must generate new free space. Fortunately, not everything in the log is a part of the most recent version of the file system. When updated data is written to the end of the log, the previous copy of the data is still on disk in its old location and can be considered dead space or a hole in the log. (In other systems, the update would have been placed on top of the previous copy.) A garbage collecting process called the cleaner must coalesce these holes into empty segments which are then available for new log writes.

For many workloads, there is sufficient idle time in which the LFS cleaner can run without interfering with normal file system accesses [Blac95]. However, when the disk fills up, disk updates are scattered randomly across the disk, or long-term sustained disk performance is required (leaving little idle time to clean), then LFS cleaning can significantly degrade file system performance [Selt93, Selt95]. Update-in-place file systems handle these situations more gracefully because they simply pay the initial cost to place each write on top of its previous location.

Although most LFS performance evaluations have focused on write cost, read cost is also an important metric. The LFS data layout policy is simply to place blocks on disk in the order in which they are written. If reads are predominantly satisfied by the cache or if disk block reads occur in the same order in which they were written, then this simple data layout will be sufficient. However, when these conditions are not met, read cost can increase as additional seeks and rotations are required between subsequently requested blocks. Update-in-place systems group semantically related data together regardless of the write order and therefore can provide better read performance when semantic information correctly predicts dynamic read patterns. However, read patterns might also follow the temporal locality reflected in LFS, but not the semantic locality reflected in update-in-place systems.

In the rest of the paper, we discuss how to improve LFS performance, in part by modifying its algorithms to take advantage of the insights provided by the other systems.

3 Methodology

We evaluate our modifications to LFS using trace-driven simulations on both synthetic and measured workloads. In this section, we describe the simulator and the traces that we used.

3.1 The Simulator

Our LFS simulator is approximately 15,000 lines of C++ code. It allows a multitude of parameters to be varied including segment size, disk size, disk performance characteristics, and cache size. We use a segment size is 256 kB unless otherwise specified.

Our baseline disk model characterizes performance using simple seek, rotation, and bandwidth attributes. Throughout the paper, we use *access time* to refer to average seek time plus a half rotation. A disk request is modelled as taking the access time plus the request size over the disk bandwidth. Unless otherwise specified, we simulate a 15 ms access time and 5 MB/s bandwidth, typical of a mid-range disk [Sea97a]. Although simple, this model reflects the fact that most LFS implementations make no effort to opportunistically choose which segments to write or clean based on the current disk head location.

However, a more sophisticated disk model is required to study read performance. To evaluate reorganizing data for reads, we hooked our simulator to the HP97560 disk simulator from Dartmouth [Kotz94]. Our simulator can be configured to run with or without data reorganization.

Separate client and server caches can be simulated. Unless specified, the client caches are 16 MB and the server cache is 128 MB. Data is channeled into the log through one write buffer. The write buffer is flushed every 30 seconds of simulated time to capture the impact of partial segment writes; these occur in LFS when data must be committed to disk (e.g., at the end of a transaction) before an entire segment's worth of data has accumulated.

The cleaner runs when there are no more empty segments available for new data. The amount of data that the cleaner may process at one time can be varied. For the experiments presented in this paper, we allowed the cleaner to process up to 20 MB at a time. Several garbage collection methods can be chosen, including *traditional LFS cleaning*, *hole-plugging* [Wilk96] and an adaptive combination of cleaning and hole-plugging. (Each of these methods will be discussed in more detail in Section 4.2.) A variety of policies for choosing segments to garbage collect are also implemented, including *greedy*, which simply chooses the least utilized segment at each opportunity, and *cost-benefit* [Rose92a]. The cost-benefit policy chooses the segment which minimizes the formula $\frac{1+u}{a \times (1-u)}$, where u is the utilization of the segment and a is the age of the segment. Throughout the rest of the paper, we refer to this policy as *cost-age* to avoid confusion with other cost-benefit formulas presented in section 4.2.2.

Our simulator is descended from Mendel Rosenblum's LFS simulator [Rose92b]. Mike Dahlin modified this simulator to accept input from a trace file and to track cache information [Dahl95] and used it to evaluate cooperative caching in xFS [Dahl94]. We have added a write buffer and implemented the modifications being evaluated here. The benefit of this history is that the simulator has already been used in several significant LFS evaluations. As a result, there is a considerable amount of previous data with which we can compare our results.

3.2 The Traces

The simulator is driven by traces of file system activity. Each trace record represents one of the following operations: read, write, delete, truncate, sync, or attribute access. Files are specified with a unique identifier. Each record specifies which client generated the request and when the request was issued.

We use both measured traces of real systems and synthetically generated traces. We use the synthetic traces to stress the system with a specific (usually worst-case) environment. This is necessary in order to demonstrate that our self-tuning algorithms achieve robust performance across a wide range of workloads. We use the real traces to verify that we are helping, or at least not harming, average case performance.

For a measured trace, we use the Berkeley Auspex Trace [Dahl94]. This trace follows the NFS activity of 236 clients serviced by an Auspex file server over the period of 1 week during late 1993. It was gathered by snooping Ethernet packets on four subnets. The clients are the desktop workstations of the University of California at Berkeley Computer Science Division. There are approximately 4 million reads and 1 million writes, each to 8 kB blocks, in the trace. In addition, there are approximately 40,000 file deletes. There are no syncs recorded in the trace, so all partial segments are due to the flush of the write buffer every 30 seconds. Because these traces are of NFS activity, we do not see any of the accesses that hit in the local cache. Accordingly, the size of the client caches is set to zero in the simulations. The trace does not contain a checkpoint of the initial state of the file system, so to initialize the disk, we examine the trace and infer as much as we can about what existed at the beginning. The trace lacks pathname information which limited our ability to evaluate semantic reorganization.

A worst-case workload for LFS is one with random updates, no idle time, and high disk utilization. The TPC-B database benchmark is an example of such a workload and was examined in [Selt93, Selt95]. We approximate this workload with a synthetically generated random update workload that has similar worst-case characteristics for LFS. To initialize the disk, we first write enough blocks sequentially to fill the disk to the desired utilization. We then make ten times as many random updates as blocks initially written with a sync call after every fourth. All writes issue from a single client.

For many of the experiments in this paper, the results are sensitive to the amount of free space available on the disk. Traditionally, this is specified as disk utilization. However, the amount of free space relative to the amount of actively written data matters also. For example, a disk at 90% utilization will behave very differently if only 20% is being actively written than if all of the data is being actively written. For the random update workload, all of the data on disk is actively written. For the Auspex trace, we initialize the disk by inferring as much as we can about the initial state of the file system. After initialization, over 60% of the data on disk is not rewritten; it is a common characteristic of real systems that only a small portion of the disk is actively written [Ruem93]. In this paper, we specify the disk utilization, but note that a given disk utilization corresponds to a higher ratio of free space to active data for the Auspex trace than for the random update workload.

4 Improving Write Performance

LFS was designed to provide high performance for writes through large batched disk transfers. However, additional research demonstrated that cleaning overhead can result in dramatically lower write performance for some workloads [Selt93, Selt95a]. In this section, we show how self-tuning principles can be applied to LFS to provide high write performance across a broader range of workloads, even those that were previously problematic.

In our evaluation, we examine the effect of our optimizations on write cost. Write cost is the metric traditionally used in evaluating LFS write performance [Rose92b]. The original write cost model can be expressed with the following formula:

$$WriteCost \qquad (EQ\ 1)$$

$$= \frac{SegmentsTransferred_{Total}}{SegmentsTransferred_{NewData}}$$

$$= \frac{SegsWritten_{NewData} + SegsRead_{Clean} + SegsWritten_{Clean}}{SegsWritten_{NewData}}$$

The write cost is the ratio of total work to the work necessary to initially write the new data to disk. The total number of segments transferred includes both the initial writes of new data ($SegsWritten_{NewData}$) and cleaner reads and writes ($SegsRead_{Clean}$ + $SegsWritten_{Clean}$). The cleaner reads an entire segment even if only a few live blocks remain. It may be beneficial to allow the cleaner to read only the live blocks when that would take less time [Rose92b], but following the original LFS, we did not include this optimization. Ideally, the data would be written once and never moved by the cleaner; this happens if all data in the segment is overwritten before the segment is reclaimed. In the best case, then, $SegsRead_{Clean}$ + $SegsWritten_{Clean}$ is 0 and the write cost is 1.

4.1 Understanding Write Cost: The Effect of Segment Size

In this section, we discuss why segment size plays a larger role in the write performance of LFS than has been previously suggested. In [Rose92a, Rose92b], segment size is chosen to be large enough that the access time becomes insignificant when amortized over the segment transfer. In Sprite LFS, a relatively large 1 MB segment is used.

On the other hand, there is a countervailing benefit to choosing a smaller segment size. [Rose92b] observes that at smaller segment sizes the variance in segment utilizations is larger; allowing the cleaner to choose less utilized segments. In particular, smaller segments are more likely to empty completely before cleaning. Empty segments can simply be declared clean without requiring any disk transfers by the cleaner. In the limit, with one-block segments, cleaning costs would always be zero because all segments would be either full or empty and no data would need to be compacted. Of course, single block segments would eliminate any advantage from batched transfers.

In this section, we describe a way to quantify this trade-off between amortizing disk access times across larger transfer units and reducing cleaner overhead.

Figure 1 shows the results of varying the segment size for the Berkeley Auspex trace. According to the original definition of write cost in EQ 1, write cost is minimized at small segments because smaller segments reduce cleaner overhead. However, this does not reflect the inefficiency introduced by transferring smaller segments.

In EQ 2, we introduce a quantity to reflect this inefficiency. We define *transfer inefficiency* to be the ratio between the actual segment transfer time and the time it would have taken to transfer the segment at full disk bandwidth. Figure 1 plots this computed value across a range of segment sizes for a typical disk with a 15 ms access time and 5 MB/s bandwidth. As segments become large, the access time becomes insignificant relative to the time for transferring the segment, and therefore the transfer inefficiency approaches one.

$$TransferInefficiency_{Segs} \qquad \text{(EQ 2)}$$

$$= \frac{SegTransferTime_{Actual}}{SegTransferTime_{Ideal}}$$

$$= \frac{AccessTime + \frac{SegmentSize}{DiskBandwidth}}{\frac{SegmentSize}{DiskBandwith}}$$

$$= AccessTime \times \frac{DiskBandwidth}{SegmentSize} + 1$$

The write cost in EQ 1 measures the overhead of cleaning. The transfer inefficiency in EQ 2 measures the bandwidth degradation caused by seek and rotational delay. In EQ 3, we introduce a new quantity, *overall write cost,* that captures both of these effects. The overall write cost is the total time required to write new data and clean segments, divided by the time to write just the new data at full disk bandwidth. If all disk transfers are in units of full segments,

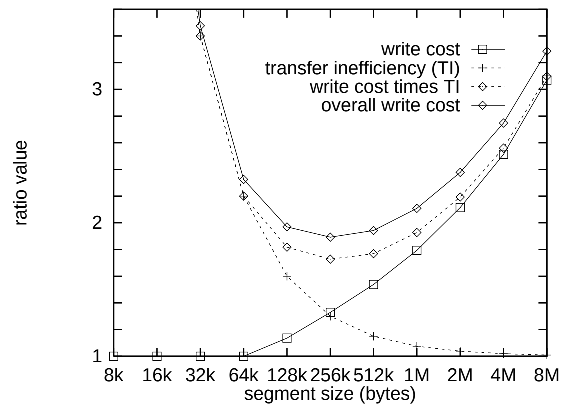

FIGURE 1. Varying segment size for the Auspex workload. Disk utilization is 85%; access time is 15 ms and bandwidth is 5 MB/s. Small segments are inefficient due to seek and rotational delay; large segments are inefficient due to fewer opportunities to find nearly empty segments. Overall write cost includes the impact of partial segments; write cost times TI does not. Write cost and overall write cost are simulated quantities. Transfer inefficiency is computed.

then this is simply the product of the original write cost in EQ 1 times the transfer inefficiency in EQ 2.

$$OverallWriteCost$$

$$= \frac{TransferTime_{Total}}{TransferTime_{Ideal}} \qquad \text{(EQ 3)}$$

when all transfers are done in units of segments

$$= \frac{SegmentsTransferred_{Total} \times SegTransferTime_{Actual}}{SegmentsWritten_{NewData} \times SegTransferTime_{Ideal}}$$

$$= WriteCost \times TransferInefficiency_{Segs} \qquad \text{(EQ 4)}$$

Figure 1 shows that this quantity does allow us to see the impact of the competing effects we have discussed. It is minimized at an intermediate segment size. Note that when the transfer inefficiency is 1, the overall write cost is equal to the original write cost. This is consistent with the assumption made in [Rose92b] that the segment size is large enough that access time becomes insignificant. In Figure 1, the difference between EQ 3 and EQ 4 is due to the impact of partial segments.

Changes in disk characteristics affect the trade-off between cleaner overhead and transfer inefficiency. Figure 2 shows that the optimal segment size for the Auspex workload is approximately four times the product of disk access time and bandwidth (i.e., four times the amount of data that could be transferred during the time necessary to position the disk head). Figure 2 shows the overall write cost curves for the disks used in Sprite LFS (17.5 ms access time and 1.3 MB/s bandwidth) and for more modern disks (15 ms, 5 MB/

s and 10 ms, 15 MB/s). This graph shows that for the Auspex workload a segment size of 64–128 kB would have been better than the 1 MB segments used in Sprite LFS. The optimal segment size has been increasing since then. This suggests that to be able to scale with disk technology improvements, an LFS file system should measure and adapt to its underlying disk performance; [Wort95] outlines a set of techniques for extracting disk parameters on-line.

Figure 3 shows overall write cost for the random update workload. Despite the inefficiency of single-block transfers, overall write cost is still lowest for single block segments (8 kB) because all cleaning overhead is avoided. (Note, however, that we do not include segment header overhead in our estimate of overall write cost.) With more than one block, there is little benefit to smaller segments. Because blocks are not overwritten in groups, segments empty slowly; even small segments stay nearly as full as the disk.

We are exploring ways to vary the segment size dynamically by enabling the cleaner to observe the average length of the runs of holes in the segments it cleans; a workload with short runs might benefit from a smaller segment size. Another possibility would be to format the disk with several fixed segment sizes. One use would be to exploit the fact that different zones of the disk have different performance characteristics; the bandwidth between inner and outer tracks can vary by as much as 50%. Another use would be to allow data to be written into the smaller segments initially and then cleaned into the larger ones. For workloads with locality, recently written data is more likely to be overwritten; this would suggest using smaller segments to maximize the likelihood of emptying segments as all of their data is overwritten. By contrast, cleaned data tends to be older and less likely to be overwritten; this suggests using larger segments to better amortize disk access times. For the random update workload, newly written segments are not any more likely to empty and so would not benefit from the smaller segments, but at least the cleaned segments could benefit from the larger ones.

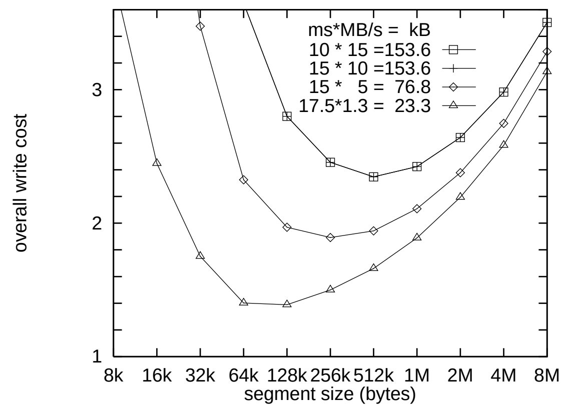

FIGURE 2. Effect of disk characteristics on overall write cost for the Auspex workload. Disk utilization is 85%. The bottom curve with access time of 17.5 ms and bandwidth of 1.3 MB/s represents the disks measured in Sprite LFS; note that Sprite chose a segment size of 1 MB. The middle curve represents the baseline disk simulated in this paper, and the top curve represents the highest performance disk available from Seagate as this paper goes to press [Sea97b]. Note that the curve is the same for different disks with the same access time bandwidth product. For all curves, overall write cost is minimized for a segment size of roughly four times bandwidth times access time. Overall write cost increases for faster disks because it is harder to match the peak disk performance.

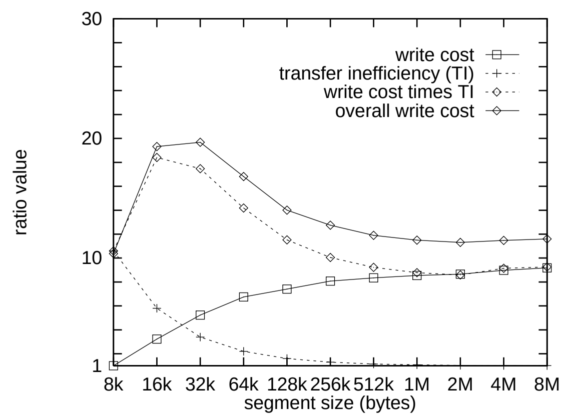

FIGURE 3. Varying segment size for the random update workload. Disk utilization is 85%; access time is 15 ms and bandwidth is 5 MB/s. One-block segments avoid all cleaning costs. Large segments benefit from larger transfers even though it is difficult to find low utilization segments to clean. Overall write cost includes the impact of partial segments; write cost times TI does not. Write cost and overall write cost are simulated quantities. Transfer inefficiency is computed. Note that the scale of the y-axis for the random workload graphs in this paper differ from that for the Auspex graphs, for example in Figures 1 and 2.

4.2 Adaptive Cleaning: Choosing the Best Garbage Collection Mechanism Based on Usage Patterns

In this section, we present an LFS cleaning algorithm that avoids the dramatic performance degradation seen at high disk utilization while retaining the good performance of traditional LFS cleaning at lower utilizations. It does this by dynamically choosing between two mechanisms: traditional LFS cleaning and hole-plugging [Wilk96]. Our adaptive method successfully chooses the lowest cost mechanism based on the observed usage patterns.

4.2.1 Comparing Traditional Cleaning With Hole-plugging

In traditional cleaning, the live blocks in several partially empty segments are combined to produce a new full segment, freeing the old partially empty segments for reuse. In many environments, traditional cleaning performs very well. Idle time can often be exploited to hide cleaning costs from users; for the workloads examined in [Blac95], 97% of cleaning could be done in the background. [McNu94] shows that cleaning costs are relatively low at disk utilizations below 80%. If segment updates show a high degree of locality, then some segments will be emptier than others and will yield more free space when cleaned.

The problem with cleaning appears at high disk utilizations, especially for workloads with many random updates and insufficient idle time [Selt93, Selt95a]. Because segments do not have a chance to empty before they must be cleaned, the cost of cleaning can skyrocket. In order to coalesce one free segment's worth of space, the cleaner must process many nearly full segments. Each segment must be read, and all but the few holes rewritten into a new segment. Recalling EQ 1, this translates into high $SegsRead_{Clean}$ and $SegsWritten_{Clean}$ and therefore high write cost. In an extreme case, the entire disk might need to be cleaned in order to coalesce a single contiguous segment.

In hole-plugging, partially empty segments are freed by writing their live blocks into the holes found in other segments. In order to produce one free segment's worth of space, we need only read one segment and rewrite each of its live blocks. These writes are more expensive per block than writing complete segments because each block write requires additional seek and rotational delay. However, despite the higher per-block cost, at high disk utilizations, hole-plugging is still better than cleaning because we avoid processing so many segments. At lower disk utilizations, the larger cost of writing individual blocks makes hole-plugging more expensive than traditional LFS cleaning.

In order to compare traditional cleaning with hole-plugging, we introduce a write cost formula for hole-plugging in EQ 5. In the traditional LFS cleaning mechanism, all transfers are done in units of whole segments. However, with hole-plugging, some transfers are done in units of whole segments (the initial writes of new data, $SegsWritten_{Data}$, and segments read to be broken up into patches for holes, $SegsRead_{Clean}$,), while other transfers are in units of individual blocks (the patches, $BlocksWritten_{Hole\text{-}plugging}$). In practice, the $TransferTime_{Block}$ would vary based on the locality of blocks written. When implementing hole-plugging, it would make sense to take advantage of this by choosing holes to plug and by ordering the block writes to minimize the total latency. We do not simulate this effect.

$$OverallWriteCost_{Hole\text{-}plugging} \qquad \text{(EQ 5)}$$

$$= \frac{TransferTime_{Total}}{TransferTime_{Ideal}}$$

$$= \frac{TransferTime_{Total}}{SegmentsWritten_{NewData} \times SegTransferTime_{Ideal}}$$

$$\text{where } TransferTime_{Total}$$

$$= TransferTime_{Seg} \times (SegsWritten_{Data} + SegsRead_{Clean}) + TransferTime_{Block} \times BlocksWritten_{Hole-plugging}$$

There are several ways that hole-plugging could be integrated into an LFS. In existing LFS implementations, each segment has a segment header that contains information about its constituent blocks. In order to maintain this structure, the header would need to be read and updated for each segment patched. Two headers per segment would be required to prevent corruption. Alternatively, the per-block information in the segment header could be distributed into individual block headers. A 512-byte block header for each 8 kB block would be an overhead of 6.25%. Interspersed block headers would also reduce read bandwidth by the same amount. In Figure 4, we evaluate the space-time trade-off between these two strategies for the random update workload. The block header approach performs better at 99% utilization than the segment header approach does at 85% utilization—more than allowing for the 6.25% space overhead. Therefore, we use the block header approach for the rest of the experiments in this paper.

Figure 4 compares the write cost of cleaning with hole-plugging. Even for this worst-case workload, cleaning performs better than hole-plugging up through 85% disk utilization. However, above 85%, the overall write cost of cleaning shoots from below 10 to above 64. Hole-plugging degrades much more gracefully, staying below 15 for the block header approach.

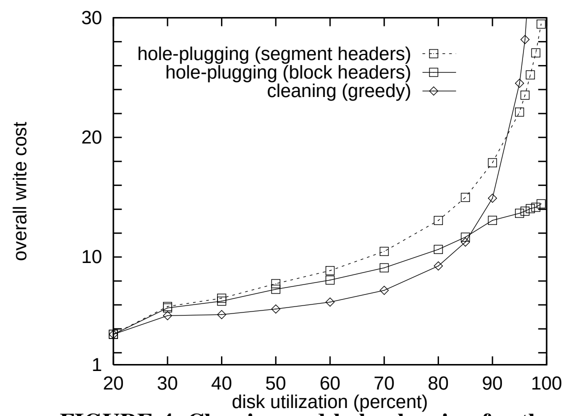

FIGURE 4. Cleaning and hole-plugging for the random update workload. Segment size is 256 kB; access time is 15 ms; bandwidth is 5 MB/s. Hole-plugging with block headers requires updating the block and its contiguously allocated header; otherwise the segment header must be read and written as well. Greedy cleaning is used because it is optimal for this workload; see Figure 6 for a comparison with cost-age. Although this point is not shown, at 99% utilization, the overall write cost for cleaning soars to 64.5.

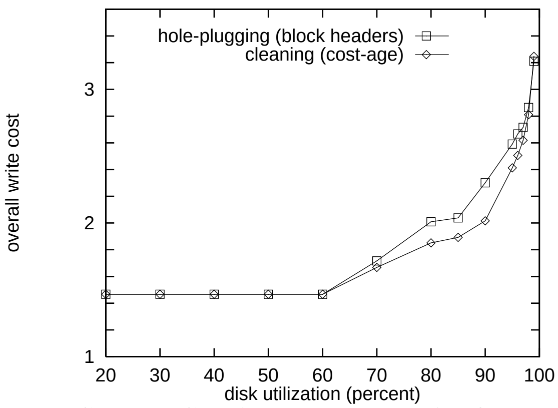

FIGURE 5. Cleaning and hole-plugging for the Berkeley Auspex workload. Segment size is 256 kB; access time is 15 ms; bandwidth is 5 MB/s**.** Cost-age cleaning is used. Cleaning performs as well or better than hole-plugging except above 99% disk utilization. Note the change in the scale of the y-axis relative to Figure 4.

Figure 5 shows the behavior of both cleaning and hole-plugging for the Berkeley Auspex workload. Cleaning performs as well or better than hole-plugging up to 99% utilization for this workload.

Hole-plugging could be considered a generalization of write-ahead logging; write-ahead logging writes the new updates to the log and then later writes them on top of the "holes" that those updates created. Write-ahead logging offers consistent performance by paying the constant cost of one batched write plus one in-place block write per block written. Similarly, we might expect hole-plugging costs to remain fairly constant. However, in Figures 4–5, the cost of hole-plugging decreases with lower disk utilization. This is because at low disk utilization many segments empty completely before they must be processed and hole-plugging, unlike write-ahead logging, can benefit from this effect.

4.2.2 Adaptive Cleaning Policy

In order to retain the good common case performance of traditional cleaning while avoiding its dramatic performance degradation at high disk utilizations, we introduce a policy that chooses adaptively between cleaning and hole-plugging at each garbage collection opportunity. (This is orthogonal to the policy used to choose which segments to clean.)

When garbage collection is needed, we first choose candidate segments for both traditional cleaning and hole-plugging. For cleaning, the candidate segments are the ones that will be compacted to form new segments. We simulated both greedy and cost-age cleaning policies. For hole-plugging, the candidate segments are those whose live blocks will be used to fill in the holes found elsewhere. As in AutoRAID, we use the least utilized segments to plug the holes in the most utilized segments.

Once we have identified the candidates, we estimate the cost-benefit of each approach with EQ 6 and EQ 7.

Cost is expressed in terms of the total time to perform the garbage collection. Benefit is expressed in terms of free space reclaimed. For hole-plugging, the cost is the time to

$$CostBenefit_{Cleaning} = \frac{TransferTime_{Cleaning}}{SpaceFreed_{Cleaning}} \qquad (EQ\ 6)$$

$$\text{where } TransferTime_{Cleaning} = (CandidatesRead + LiveBlocks/BlocksPerSeg) \times TransferTime_{Seg}$$

$$\text{and } SpaceFreed_{Cleaning} = EmptyBlocks \times BlockSize$$

$$CostBenefit_{Hole\text{-}plugging} = \frac{TransferTime_{Hole-plugging}}{SpaceFreed_{Hole-plugging}} \qquad (EQ\ 7)$$

$$\text{where } TransferTime_{Hole\text{-}plugging} = CandidatesRead \times TransferTime_{Seg} + LiveBlocks \times TransferTime_{Block}$$

$$\text{and } SpaceFreed_{Hole\text{-}plugging} = CandidatesRead \times SegmentSize$$

read the candidate segments and write their live blocks into holes found in other partially empty segments; the space freed is the size of all the candidate segments read. For cleaning, the cost is the time to read the candidate segments and rewrite their live blocks as whole segments to the end of the log; the space freed is the size of all the empty blocks found in the candidate segments.

Once we have calculated these cost-benefit estimates, we simply choose the mechanism with the lower estimate. Note that this decision applies only to the current garbage collection opportunity. At the next opportunity, we may choose the other approach.

In Figure 6, we show cleaning, hole-plugging and the adaptive policy for the random update workload. We include greedy cleaning as well as cost-age since greedy has been shown to have slightly better performance than cost-age on a random workload [Rose92a, Selt95b]. The adaptive policy correctly shifts from cleaning to hole-plugging at the appropriate point. We are indeed able to retain the good common case performance of traditional cleaning while avoiding its dramatic performance degradation at high disk utilizations.

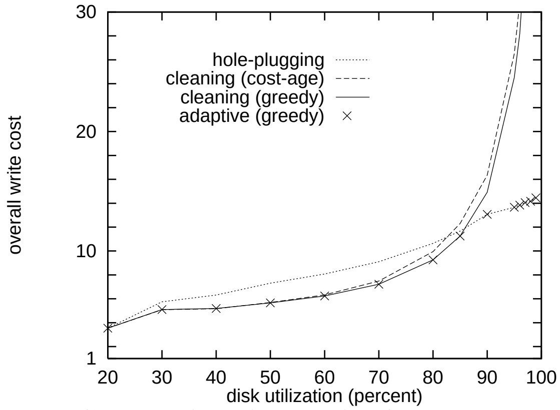

FIGURE 6. Adaptive cleaning for the random update workload. Note that the hole-plugging and greedy cleaning curves are the same as in Figure 4. The adaptive algorithm chooses between hole-plugging and greedy cleaning; it correctly follows the lower cost mechanism at each point.

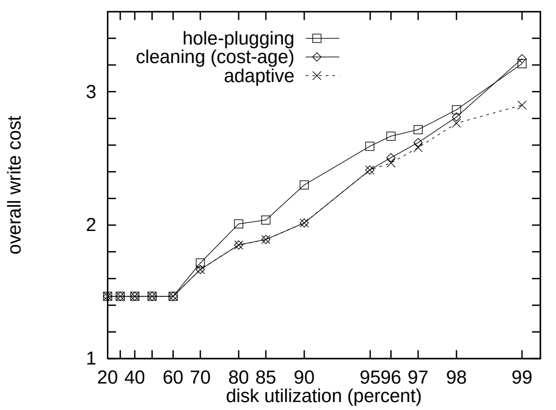

FIGURE 7. Adaptive cleaning for the Berkeley Auspex workload. Note that the hole-plugging and cost-age cleaning curves are the same as in Figure 5. The x-axis is on a reverse log scale in order to show clearly the region above 90%. Adaptive outperforms both hole-plugging and cleaning because it can choose the appropriate method at each garbage collection opportunity.

In Figure 7, we show cleaning, hole-plugging, and the adaptive policy for the Berkeley Auspex workload. Notice that at some points the adaptive policy performs better than the minimum of cleaning and hole-plugging by doing each when appropriate.

This adaptive method could also be used to adapt between any additional garbage collection mechanisms given a correct cost-benefit model of their behavior.

Changes in disk characteristics also have an impact on the trade-off between cleaning and hole-plugging, making the need for adaptive cleaning even more acute. Disk bandwidth has been improving faster than disk access times, resulting in higher relative block transfer costs. Figure 8 shows that on a faster disk the gap between hole-plugging and cleaning is larger at lower disk utilizations and that the cross-over point is later. Similarly, on RAID systems, hole-plugging would be penalized relative to cleaning because of the need to read the old data in blocks being plugged in order to update parity.

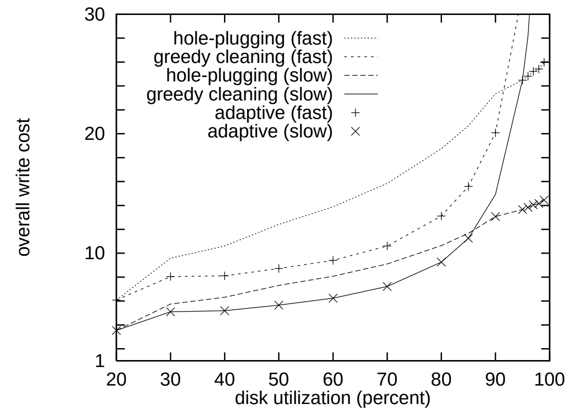

FIGURE 8. Effect of disk characteristics on the trade-off between cleaning and hole-plugging, for the random update workload. Note that the slow curves are the same as in Figure 6, using a disk with 15 ms access time and 5 MB/s bandwidth. The fast curves use a disk with 10 ms access time and 15 MB/s bandwidth. Cleaning performs relatively better than hole-plugging on the fast disk because of the larger gap between block transfer efficiency and segment transfer efficiency. Overall write cost increases for the fast disk because it is harder to match the peak disk performance.

4.3 Using Cached Data To Reduce Write Cost

In this section, we describe how to reduce cleaning costs by taking advantage of data that is already cached. When a segment is completely cached, it can be cleaned by writing its live blocks—there is no need to do a disk read. This lowers the $SegmentsRead_{Clean}$ component of write cost in EQ 1. As far as we know, no LFS implementation performs this optimization.

In exploring this possibility, we consider two different cleaning policies: normal cost-age in which cached data is not used, and a modified cost-age (*cost-age-cache*) in which fully cached segments are preferentially chosen by taking into account that a segment is cached in the cost-age formula. For this modified policy, when a segment is cached, the cost portion of the cost-age function includes only the cost to write out the live blocks and not the cost to read the complete segment.

We implemented the modified cleaning policy in our simulator by keeping an in-memory set of cached segments; its size is limited to the number of complete segments that fit in memory. As a block leaves the cache, we check this set and remove its segment if necessary. We track only segments that remain completely cached after being written; detecting when full segments re-enter the cache would complicate the implementation for only marginal benefit. We see significant improvement even though we do not take advantage of segments that are re-cached.

In Figure 9, we show the impact of increasing server

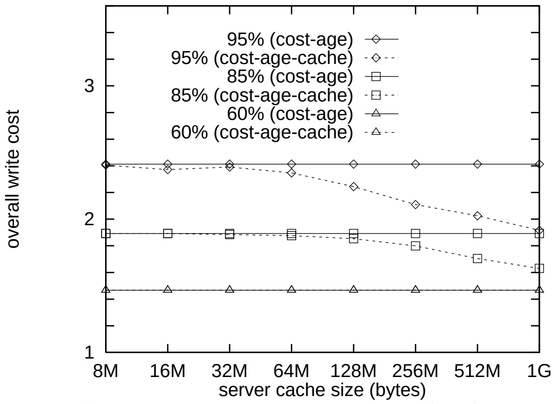

FIGURE 9. Varying server cache size for the Auspex workload. Segment size is 256 kB; access time is 15 ms; bandwidth is 5 MB/s**.** Three different disk utilizations are shown for both normal cost-age and cost-age that uses cached data. The client cache size is set to zero as described in section 3.2. This graph shows the reduction in overall write cost obtained by exploiting cached data during cleaning. The benefit is greater at higher disk utilizations.

cache size on overall write cost for the Berkeley Auspex workload at various disk utilizations. The top group of lines illustrates the behavior when the disk is 95% utilized. The next two groups of lines are with the disk at 85% and 60% utilization, respectively.

As expected, the performance of the cost-age policy is insensitive to cache size. Indeed, all of the cost-age lines are flat. For the cost-age-cache policy, there is more benefit with larger caches as one would expect.

We see incremental benefit even up through 1 GB, indicating that the working set of the Auspex trace is larger than 1 GB. This is unsurprising considering that the total data held on the server was approximately 100 GB of binaries and home directories. At a cache size of 256 MB and disk utilization of 95%, we see an 11% reduction in overall write cost corresponding to a 30% reduction in cleaning overhead.

In addition, the benefit of cleaning from cache increases as the utilization of the disk increases. To illustrate why, we consider the effect on the cost-age formula for cleaning a single segment as the utilization of that segment increases. For a highly utilized segment, we reclaim less space and therefore the read that we avoid has higher cost relative to the amount of space reclaimed. To see this quantitatively, consider the cost-age formulas. Recall from section 3.1 that when we are unable to use cached data to avoid reading the segment, the cost-age of the segment is $\frac{1+u}{a \times (1-u)}$, where u is the percentage of live blocks in the segment and a is the age of the segment. When we are able to use cached data to avoid the read, the cost-age drops to $\frac{u}{a \times (1-u)}$. Their difference, $\frac{1}{a \times (1-u)}$, is larger for segments with greater utilization.

As overall disk utilization increases, LFS will have to clean segments with higher utilization. As a result, the increased benefit for fuller segments translates directly into increased benefit for fuller disks. Interestingly, this means that using cached data is especially helpful in addressing the worst-case performance of LFS at high disk utilizations.

Also, notice that we begin to see benefit at smaller cache sizes as the utilization increases. At lower utilizations, we can wait longer to clean; therefore, we need a larger cache in order to still be holding the segments we are interested in cleaning.

4.4 Putting It All Together

Figure 10 shows the combined impact of the optimizations we have discussed in this section relative to original LFS. There is up to a 20% reduction in overall write cost for the Berkeley Auspex trace and an up to four-fold reduction for the random workload. That corresponds to a 42% and almost six-fold reduction in cleaner overhead, respectively. Log scale is used to clearly display both workloads.

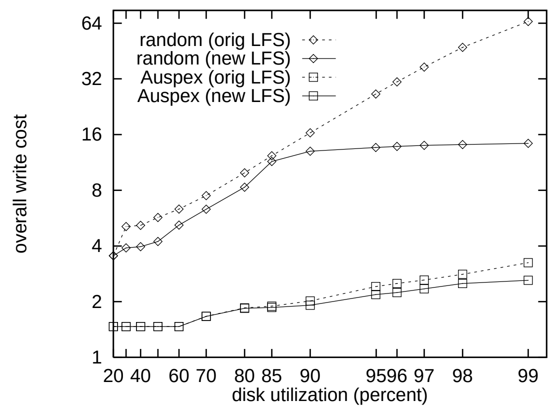

FIGURE 10. Overall write cost of original LFS versus modified LFS. Segment size is 256 kB; server cache size is 128 MB; access time is 15 ms; bandwidth is 5 MB/s; client cache size is 16 MB. Note the log scale on both axes. This graph shows the aggregate effect on overall write cost of using both adaptive cleaning and cached data. The segment size is the same for all curves. However, an additional benefit would be obtained for the Auspex trace if the Sprite LFS segment size of 1 MB was used for the original LFS curves (see Figures 1–2).

4.5 Future Work

Other opportunities exist to use self-tuning to improve LFS write performance. For example, one promising area is to adaptively exploit the differences in access characteristics of rapidly changing data versus more stable data.

For example, existing cleaning policies use the age of a segment to approximate the rate at which its blocks are being overwritten, because long term cleaning costs are minimized by aggressively cleaning segments that are partially full of relatively stable data. However, users change their working sets from time to time, resulting in old segments being rapidly updated, while newer segments are more stable. And for a random workload, using age to approximate rate of change is suboptimal. We have devised but not evaluated a self-tuning algorithm for choosing which segments to clean that takes advantage of this effect.

5 Improving Read Performance

In LFS, data that is written together is grouped together on disk. If read patterns follow write patterns, this data layout will also work well for reads. However, for workloads where this is not true, LFS can be modified to detect expensive read patterns and reorganize accordingly. In fact, the cleaner is already reorganizing data to reclaim free space for writes, and the same mechanisms can be used to reorganize data for reads.

5.1 Motivation for Reorganizing Data for Reads

LFS was originally designed to be a write-optimized file system and therefore most LFS performance evaluations have focused on write cost as their primary metric. Despite the emphasis on writes, it has been speculated that LFS would still offer good read performance.

First, data is often read as it is written. In that case, the temporal locality of LFS will be as effective as the semantic locality of update-in-place systems. Studies of traditional UNIX workloads [Bake91] show that most files are written and read sequentially. However, there are workloads for which reads patterns do not match write patterns. Even for a UNIX workload, LFS may often do a poor job of keeping the contents of a directory together. For example, if all files in the directory are actively read but only some are actively written, the actively written files will move far away from the read-only ones. As long as the data remains cached, there is no penalty; once the data is demoted, however, the penalty of fetching the files in the directory from disk will be higher than in an update-in-place system. Another problematic workload is random writes followed by sequential reads. Decision support database workloads can exhibit this pattern. Random updates are applied to the active portion of the database and then sometime later large sweeping queries read relations sequentially [Tran95].

Second, it was initially argued that almost all reads would be satisfied from large caches and therefore would be unaffected by disk layout. However, the rapid fluctuation in the relative cost per byte of disk and DRAM make such predictions uncertain at best. More importantly, the large and widening gap between CPU and disk performance has meant that file system read response times are dominated by disk accesses, even for very high cache hit rates [Dahl95].

Third, Ousterhout has argued that while fragmentation in an FFS degrades performance for both reads and writes, LFS cleaning has no ill effects on read cost [Oust95c]. However, this is not obvious since cleaning coalesces blocks from different segments together even though the contents of these segments are unrelated both semantically and temporally.

In this section, we explore one approach to reorganizing data for reads—a dynamic algorithm that operates at the granularity of blocks. Since there are many other possible approaches (for example, an algorithm based on regrouping semantic units), at this time, we do not attempt to conclude that the reorganization algorithm we are currently exploring is the best. Our goal in this paper is to explore one attractive possibility based on self-tuning principles and to reopen a discussion on the opportunity to improve LFS read performance for some workloads.

5.2 Dynamic Reorganization

The goal of dynamic data reorganization is to arrive at an optimal data layout based on observed access patterns. To accomplish this, the reorganizer must solve three problems. First, it needs to keep track of the history of previous events. Second, it must find a layout that would deliver near-optimal performance for the observed access patterns, assuming that past events are a good predictor of future access patterns. Third, it must analyze the difference between the current layout and the desired layout and if necessary issue I/O requests to correct the difference.

5.2.1 Tracking File Access History

To capture the past access pattern, we build a *block access graph*, similar to the file access graphs proposed by [Grif94], for use in prefetching file data into memory. Intuitively, reorganizing data for reads is complementary to prefetching. Prefetching must identify blocks or files that are used together to know what to pull into memory. Reorganizing data for reads requires the same information and uses it to organize file blocks so that when they are read (or prefetched) from disk it can be done efficiently.

Each node in the file access graph represents a file block. An edge connecting node A and node B denotes that block B was accessed immediately after block A. The edges are weighted by the number of such accesses.

One concern with this approach is the amount of storage required to accommodate the growth of the edge lists. We experimented with some simple approaches of pruning the graph and found that we can successfully reduce the storage costs by limiting the number of outgoing edges and pruning them in LRU order. This simple approach proves effective because the number of neighbors per node follows a bimodal distribution. Most nodes have only a few neighbors that can all be represented in limited space. A few nodes have many neighbors, but in that case it is less important to record them all because a large number of neighbors indicates that there is not a dominant access pattern for which we could optimize. The data structure we use to represent a graph node includes a block identifier, the last reference time, and a pointer to each neighbor recorded. We varied the number of neighbors recorded from 4 to 16 and saw negligible impact on graph quality. A graph node recording 4 neighbors is 48 bytes, about 0.6% overhead for an 8 kB block. When a block is not being referenced, its graph node could also be paged to disk to limit the amount of memory used.

5.2.2 Computing Optimal Layout

Once we have captured the file system events in the access graph, the next challenge is to find a disk layout strategy that will optimize for the observed usage pattern. Such a layout strategy will attempt to place blocks that are frequently accessed together close to one another in order to minimize seek and rotational delays. We observe that given an access graph, finding such a layout is an application of the more general irregular graph partitioning problem. More specifically, we must partition the file access graph into some number of roughly equal parts, such that the number of edges connecting nodes in different parts is minimized. By maximizing the number of internal edges and minimizing the number of external edges, we discover a partitioning of the file blocks such that blocks in the same partition are frequently accessed together, while blocks in different partitions are rarely accessed together.

Although the general irregular graph partitioning problem is NP-complete, there exist many heuristic solutions in the literature [Barn93,Hend93]. We have adopted a simple dynamic graph partition algorithm based on these heuristic solutions for our data reorganizer. For each read access, we create an edge between the current block and the previous block (or increment the weight of an existing edge). If the previous block is in a different partition than the current block, we shift the existing partition boundaries to bring the nodes in question closer, if doing so would result in new partitions which minimize inter-partition edges.

We have validated our dynamic algorithm by comparing the partition qualities of our algorithm with that of a well known off-line graph partitioning package [Kary95]. For file access graphs based on the Berkeley Auspex traces, the partitions produced by our data reorganizer were better than or equivalent to those generated by the off-line algorithm, even with only 4 neighbors recorded per block. This algorithm required approximately 130 μs per block read.

5.2.3 Selecting Data to Reorganize

Partitioning the file access graph allows us to identify the data blocks that should be located near each other. Based on this graph, we may need to issue I/O requests to improve the current data layout. In order to do this, the data reorganizer monitors three variables for each partition: (1) the current disk locations of the partition members and their corresponding access costs, (2) the new access costs of the partition members if they were brought together by the data reorganizer, and (3) the likelihood that the partition will be accessed again. We place the partitions into a priority queue ordered by a ranking based on these three variables. Currently, we place partitions on the queue when the expected time to read the partition exceeds the ideal and we order the queue by frequency of access. Partitions can be removed from the priority queue and reorganized at convenient times, such as during idle periods, when the partition is brought into memory, or when the partition is about to be evicted from memory.

5.2.4 Evaluation

We first evaluated the impact of reorganization for the Berkeley Auspex trace and were unsurprised to see little benefit. In order to evaluate the benefit of reorganization, we must know the default layout that was produced as the data was originally written and then subsequently cleaned. Unfortunately, we do not know the original disk layout at the start of the trace. We expect reorganization to be especially valuable for cold data and by definition, we do not see the writes for cold data in the trace.

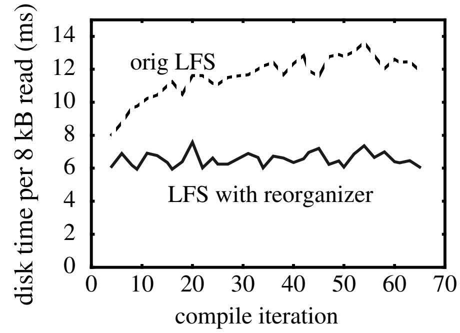

FIGURE 11. Average disk read response time per 8 kB block for the synthetic compilation workload. The y-axis reports the average time to complete an 8 kB disk read for each compile iteration; the x-axis represents the passage of time. The HP97560 disk model was used. For original LFS, the response time increases from 8 ms to around 12 ms during the measurement period. With reorganization turned on, the response time is kept stable around 6.5 ms.

We chose to use a semi-synthetic benchmark to approximate the uncached read portion of a software development workload. When a developer visits a new region of the source tree, those reads are uncached, and observed file system performance will depend on how quickly the data can be retrieved from disk. Once the data is read in, software development becomes write-dominated, until the developer changes working set, at which point the cycle repeats.

To model uncached reads in a software environment, we modified the HPUX kernel to record all file system calls generated by the following benchmark. We first recursively copy a number of C source directories. We then edit a randomly chosen source file and compile the entire subdirectory. Finally, we flush the cache to simulate the passage of time. This cycle is repeated many times. In order to get more accurate estimates of disk read performance, we hooked our simulator to the HP97560 disk simulator from Dartmouth [Kotz94]. We assume that reorganization could be accomplished in the background since we are modeling a workload for which a significant amount of time passes between the edit-compile cycles.

Figure 11 shows the average time needed to read one 8 kB block from disk as a function of the number of times we repeated the edit-compile cycle. The time is dominated by the linking stage, during which a large number of object files are read. Under LFS without data reorganization, the object files are placed close to each other initially. But as we modify the sources, the newly written object files are moved to the end of the log. As a result, the object files swept by the linking stages are gradually scattered over a large number of segments, causing a rise in I/O response time. When we turn on the data reorganizer, it correctly concludes that the object files needed in the linking phases belong to the same file access graph partitions. As a result, the I/O response time remains nearly constant with reorganization.

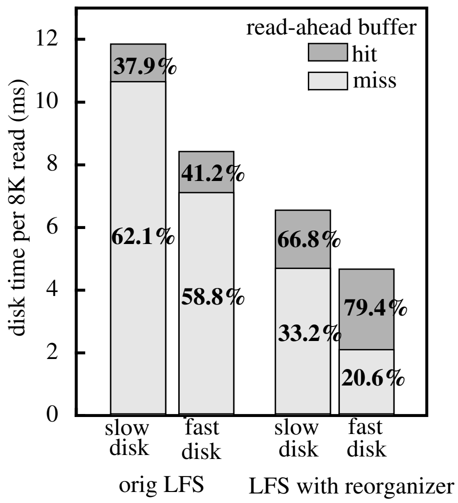

FIGURE 12. Breakdown of average disk read response time per 8K block for the synthetic compilation workload. Each bar represents the average time needed to fetch an 8K block from disk across all iterations reported in Figure 11. Slow disk refers to the original HP97560 disk model; the fast disk refers the same model with a bandwidth of 5 MB/s. The top segment in each bar shows the fraction of the read time for data accesses satisfied in the disk read-ahead buffer cache. The hit and miss rates are specified in each column. The column heights are hit rate times hit time and miss rate times miss time, respectively.

Figure 12 provides further insight into the causes of the performance difference. The HP97560 disk controller model [Kotz94] includes a 64 kB read-ahead buffer cache. The graph shows the average time a read request spends in the disk cache. As a result of periodically optimizing layout for read performance, we are able to raise the read-ahead cache hit rate from 37.9% to 66.8%. As we increase the disk speed from 2.4 MB/s to 5 MB/s, we are able to further raise the disk cache hit rate to 79.4% while an unmodified LFS can only go up to 41.2%.

Software development is not the only example of uncached read accesses that may be problematic for LFS. For example, many people store all their mail messages in a single directory and return frequently to search for a desired piece of mail. In LFS, each newly received mail message will be appended to the current tail of the log. Since these appends are separated in time, they will not be grouped together in the log. When the user tries to search through the entire directory, there will be many seeks between the scattered data blocks. Although a user may search this directory frequently, it is unlikely that the data will still be in the cache after the last search. Over time, the performance of searching through this directory will degrade. This is like the effect shown for our benchmark in Figure 11.

6 Related Work

When LFS was originally introduced [Rose92a, Rose92b], the design space of this radically new file system organization was explored with simulations using write cost as a metric. A full implementation of LFS was incorporated into the Sprite operating system and was used in a production environment. Many possible avenues of improvement to LFS were proposed, but not all of them were fully pursued, including reorganizing data for reads and the impact of different segment sizes.

[Selt92, Selt93, Selt95a] further explored the LFS paradigm. [Selt93] and [Selt95a] describe an implementation of LFS for BSD and compare it with the Berkeley Fast File System (FFS) and an enhanced extent-based FFS. This work demonstrated that cleaning costs can seriously degrade LFS performance on workloads with random updates at high disk utilization, such as the TPC-B benchmark. In addition, it is pointed out that certain modifications to an update-in-place system such as FFS can allow it to achieve some of the same benefits as LFS.

A debate ensued [Oust95a, Oust95b, Selt95b, Oust95c] concerning whether the results presented in [Selt93] and [Selt95a] were legitimate and representative. Despite the criticism of the results, they do describe some real problems with LFS that should not be ignored. In this paper, we investigate ways to enable LFS to provide reasonable performance even for these problematic workloads.

[McNu94, Blac95, Lome95] all explore enhancements to LFS. [McNu94] presents a mathematical model of garbage collection and concludes that disk utilization must be kept below 80% for LFS to provide good performance. In this paper, we explore solutions that do not require leaving 20% of the disk unused. [Blac95] considers how much of the garbage collection costs can be shifted into the background. [Lome95] argues for the use of LFS for databases and suggests detecting expensive read patterns and simply rewriting the data in the same order it is read.

[Dahl95] makes an initial evaluation of using cached data to lower cleaning costs. We continue this evaluation by considering the effects of additional parameters and workloads.

Our section on adaptive cleaning combines traditional LFS cleaning with another garbage collection mechanism, hole-plugging, that was used in HP AutoRAID [Wilk96]. We observe that hole-plugging is especially beneficial at high disk utilizations, while cleaning is better at lower disk utilizations (below 80–85%). In AutoRAID, hole-plugging, is used rather than traditional LFS cleaning or an adaptive combination of the two, because AutoRAID is structured such that hole-plugging always performs better. The AutoRAID consists of two areas: the mirrored or RAID-1 area, which houses recently updated data, and the RAID-5 area, which houses older data. The RAID-5 area is the part of AutoRAID that is log-structured. It is always at high utilization because it is constantly cleaned in order to return PEGs (segments) to the free pool where they can be used for mirrored writes or fresh demotions into the RAID-5 log. In addition, the updates to the RAID-5 area are fairly random because the mirrored storage area absorbs the updates to the hot data. The remaining update stream reaching the RAID-5 applies to cold data and shows very little locality in practice. Thus, the AutoRAID environment is the worst possible case for traditional cleaning (high utilized segments that are updated randomly) [Stae96]. AutoRAID does PEG-cleaning only in the special case that there are no holes to be plugged; that is when all PEGs but one are full or empty. In this case, the live blocks from that single PEG are appended to the end of the RAID-5 write log [Wilk96].

There is a significant amount of research into altering disk layout to improve read performance. [Wong83, Vong90, Ruem91, Akyü95] discuss the benefits of placing the most frequently accessed data in the middle of the disk where it is most likely to be close to the disk head. These systems do restructuring at either the cylinder or block level. They use the disk controller or the device driver, not the file system, to monitor access frequencies and the move data. [Akyü95] limits the number of blocks for which information must be maintained much like we do; however we are maintaining information about the relationships between blocks where they are maintaining access frequencies. [Ruem91] evaluates the benefits for an update-in-place file system and conjectures that the benefits would be even greater for a file system that did not do such a good job of initial data placement. [McDo89, Stae91] explore file system directed reorganization at the granularity of whole files. Update-in-place systems such as FFS [McKu84] reduce average disk access times by collecting statically related data in cylinder groups.

The problem of organizing data for reads is very similar to prefetching. In order to facilitate prefetching, [Grif94] proposes a file access graph similar to the one we use to reorganize data for read accesses. [Kroe96] uses a data compression technique, prediction by partial match, to predict file system activity and thus improve the effectiveness of prefetching.

We observe that partitioning the access graph used to capture file system activity is an application of the more general irregular graph partitioning problem. Although the general problem is NP-complete, there exist many heuristic solutions in the literature [Barn93, Hend93].

There is a recent trend towards incorporating LFS techniques into other file system architectures. Network Appliance's file system, WAFL, improves write performance for their RAID array by writing multiple blocks in a stripe [Hitz95]. Sweeney et al. recently incorporated location-independent inodes, an idea from LFS, into XFS [Swee96], a write-ahead logging file system. As in LFS, location-independent inodes would make it easier to incorporate a disk reorganizer. In contrast, most traditional file systems, such as FFS, fix the disk location of a file's inode (containing the table of pointers to the file's data blocks) when the file is first created; because a physical pointer to the inode can be embedded in any number of directories, moving an inode in FFS could require scanning the entire disk.

7 Conclusions

In this paper, we have argued that self-tuning—measuring both the underlying hardware and the workload of a system, and then dynamically adapting to match—is a powerful paradigm for improving the performance of file systems. We illustrated self-tuning principles with four optimizations to log-structured file systems: choosing the segment size to trade-off transfer efficiency against cleaning efficiency; dynamically choosing the cleaning method to provide good performance across the spectrum of disk utilizations and workload patterns; factoring in cache contents to reduce the cost of cleaning; and re-organizing disk contents to improve read performance. We believe that together these improvements make log structure a much more attractive alternative for file system design.

8 Acknowledgments

We would like to thank Mike Dahlin for his help with earlier versions of this work, Margo Seltzer for helping in understanding how to integrate hole-plugging into BSD/LFS and for her encouragement, John Wilkes, Carl Staelin and Terry Burkes for answering numerous questions about HP AutoRAID and for their many helpful comments, and John Ousterhout and Eric Anderson for their feedback on earlier drafts. We would also like to thank the program committee, as well as the other anonymous referees, for their comments that were a great help in revising the paper.

9 References

[Akyü95] S. Akyürek and K. Salem. Adaptive Block Rearrangement. *ACM Transactions on Computer Systems*, 13(2):89–121, May 1995.

[Ande96] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, and R. Wang. Serverless Network File Systems. *ACM Transactions on Computer Systems*, 14(1):41–79, Feb. 1996.

[Bake91] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. *Proc. Thirteenth ACM Symposium on Operating Systems Principles*, pp. 198–212, Oct. 1991.

[Barn93] S. Barnard and H. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Proc. Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pp. 711–718, 1993.

[Birr93] A. Birrell, A. Hisgen, C. Jerian, T. Mann, and G. Swart. The Echo Distributed File System. Technical Report 111, Digital Equipment Corp. Systems Research Center, Sep. 1993.

[Blac95] T. Blackwell, J. Harris, and M. Seltzer. Heuristic Cleaning Algorithms in Log-Structured File Systems. *Proc. 1995 Winter USENIX Conference*, pp. 277-288, Jan. 1995.

[Chen94] P. Chen, E. Lee, G. Gibson, R. Katz, and D. Patterson. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–188, Jun. 1994.

[Chut92] S. Chutani, O. Anderson, M. Kazar, B. Leverett, W. Mason, and R. Siedbotham. The Episode File System. *Proc. 1992 Winter USENIX Conference*, pp. 43–60, Jan. 1992.

[Cust94] H. Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[Dahl94] M. Dahlin, C. Mather, R. Wang, T. Anderson, and D. Patterson. A Quantitative Analysis of Cache Policies for Scalable Network File Systems. *Proc. SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 150–160, May 1994.

[Dahl95] M. Dahlin. Serverless Network File Systems. PhD Thesis. University of California, Berkeley, Dec. 1995.

[Gang94] G. Ganger and Y. Patt. Metadata Update Performance in File Systems. *Proc. First Symposium on Operating Systems Design and Implementation*, pp. 49-60, Nov. 1994.

[Gold95] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is Not Sloth. *Proc. 1995 Winter USENIX Conference*, pp. 201–202, Jan. 1995.

[Grif94] J. Griffioen and R. Appleton. Reducing File System Latency Using A Predictive Approach. *Proc. 1994 Summer USENIX Conference*, pp. 197-207, Jun. 1994.

[Hagm87] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. *Proc. Eleventh ACM Symposium on Operating Systems Principles*, pp. 155–162, Oct. 1987.

[Hart93] J. Hartman and J. Ousterhout. The Zebra Striped Network File System. *ACM Transactions on Computer Systems*, 13(3):274-310, Aug. 1995.

[Hend93] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[Hitz94] D. Hitz and J. Lau and M. Malcolm. File System Design for an NFS Server Appliance. *Proc. 1994 Winter USENIX Conference*, pp. 235-246, 1994.

[Jaco88] V. Jacobson and M. Karels. Congestion Avoidance and Control. *Proc. SIGCOMM Conference on Data Communication*. Nov. 1988.

[Jaco91] D. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical Report HPL-CSP-91-7rev1, Hewlett-Packard Laboratories, Palo Alto, CA, Mar. 1991.

[Kary95] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. Technical Report TR 95-035, University of Minnesota, 1995.

[Kotz94] D. Kotz, S. Toh, and S. Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Technical Report PCS-TR94-220, Dartmouth, Jul. 1994.

[Kowa78] T. Kowalski. FSCK: The UNIX System Check Program. Technical report, Bell Laboratory, Murray Hill, NJ, Mar. 1978.

[Kroe96]. T. Kroeger and D. Long. Predicting Future File-System Actions From Prior Events. *Proc. 1996 USENIX Conference*, pp. 319-328, Jan. 1996.

[Lome95] D. Lomet. The Case for Log Structuring in Database Systems. *Int'l Workshop on High Performance Transaction Systems*, Sep. 1995.

[McDo89] M. McDonald and R. Bunt. Improving File System Performance by Dynamically Restructuring Disk Space. *Proc. Phoenix Conference on Computers and Communication (Scottsdale, AZ)*, pp. 264-269, Mar. 1989.

[McNu94] B. McNutt. Background Data Movement in a Log-Structured File System. *IBM Journal of Research and Development,* 38(1):47-58, 1994.

[McKu84] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

[McVo91] L. McVoy and S. Kleiman. Extent-like Performance from a UNIX File System. *Proc. 1991 Winter USENIX Conference*, pp. 33–43, Jan. 1991.

[Math96] M. Mathis and J. Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. *Proc. SIGCOMM Conference on Data Communication*. Aug. 1996.

[Oust95a] J. Ousterhout. A Critique of Seltzer's 1993 USENIX Paper.http://www.sunlabs.com/people/john.ousterhout/seltzer93.html, 1995.

[Oust95b] J. Ousterhout. A Critique of Seltzer's LFS Measurements. http://www.sunlabs.com/people/john.ousterhout/seltzer.html, 1995.

[Oust95c] J. Ousterhout. A Response to Seltzer's Response. http://www.sunlabs.com/people/john.ousterhout/seltzer2.html, 1995.

[Ritc74] D. Ritchie and K. Thompson. The UNIX Timesharing System. *Communications of the ACM*. 17(7), pp. 365-375, Jul. 1974.

[Rose92a] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb. 1992.

[Rose92b] M. Rosenblum. The Design and Implementation of a Log-structured File System. PhD Thesis. University of California, Berkeley, Jun. 1992.

[Ruem91] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-91-156. Hewlett-Packard Laboratories, Palo Alto, CA, Oct. 1991.

[Ruem93] C. Ruemmler and J. Wilkes. A Trace-driven Analysis of Disk Working Set Sizes. Technical Report HPL-OSR-93-23. Hewlett-Packard Laboratories, Palo Alto, CA, Apr. 4, 1993.

[Sea97a] Seagate Technology, Inc. Hawk 2XL Family 3.5-inch Driives, http://www.seagate.com/disc/hawk/hawk2xlscsi3.shtml, 1997.

[Sea97b] Seagate Technology, Inc. Cheetah Family 3.5-inch Form Factor. http://www.seagate.com/disc/cheetah/cheetah.shtml, 1997.

[Selt90] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisted. *Proc. 1990 Winter USENIX Conference*, pp. 313–324, Jan. 1990.

[Selt92] M. Seltzer. File System Performance and Transaction Support. PhD Thesis. University of California, Berkeley, Dec. 1992.

[Selt93] M. Seltzer, K. Bostic, M. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. *Proc. 1993 Winter USENIX Conference*, pp. 307–326, Jan. 1993.

[Selt95a] M. Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. *Proc. 1995 Winter USENIX Conference*, pp. 249–264, Jan. 1995.

[Selt95b] M. Seltzer and K. Smith. A Response to Ousterhout's Critique of LFS Measurements. http://www.eecs.harvard.edu/~margo/usenix.195/ouster.html, 1995.

[Smit96] K. Smith and M. Seltzer. A Comparison of FFS Disk Allocation Polices. *Proc. 1996 USENIX Conference*, pp. 15-26, Jan. 1996.

[Smit97] K.Smith and M. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. *Proc. SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Jun. 1997.

[Stae91] C. Staelin and H. Garcia-Molina. Smart Filesystems. *Proc. 1991 Winter USENIX Conference*, pp. 45–51, Jan. 1991.

[Stae96] C. Staelin. Discussion at UC Berkeley. Personal Communication. Nov. 1996.

[Swee96] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. *Proc. 1996 USENIX Conference*, pp. 1-14, Jan. 1996.

[Tran90a] Transaction Processing Performance Council. *TPC Benchmark B Standard Specification*. Waterside Associates, Fremont, CA, Aug. 1990.

[Tran90b] Transaction Processing Performance Council. *TPC Benchmark C Standard Specification*. Waterside Associates, Fremont, CA, Jul. 1990.

[Tran95] Transaction Processing Performance Council. *TPC Benchmark D Standard Specification*. Waterside Associates, Fremont, CA, Apr. 1995.

[Veri95] Veritas Software. The VERITAS File System (VxFS). http://www.veritas.com/products.html, 1995.

[Vong90] P. Vongsathorn and S. Carson. A System for Adaptive Disk Rearrangement. *Software: Practice and Experience.* 20(3):225-242, Mar. 1990.

[Wilk96] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108-136, Feb. 1996.

[Wong83] C. Wong. *Algorithmic Studies in Mass Storage Systems*. Computer Science Press, 1983.

[Wort95] B. Worthington, G. Ganger, W. Patt and J. Wilkes. Online Extraction of SCSI Disk Drive Parameters. *Proc. SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 146-156, May 1995.