Intelligent Methods for File System Optimization

Leo Kuvayev
Department of Computer Science
University of Massachusetts
Amherst, MA 01002
kuvayev@cs.umass.edu

C. L. Giles and **J. Philbin** and **H. Cejtin**
NEC Research Institute
4 Independence Way
Princeton, NJ 08540
{giles,philbin,henry}@research.nj.nec.com

Abstract

The speed of I/O components is a major limitation of the speed of all other major components in today's computer systems. Motivated by this, we investigated several algorithms for efficient and intelligent organization of files on a hard disk. Total access time may be decreased if files with temporal locality also have spatial locality. Three intelligent methods based on file type, frequency, and transition probabilities information showed up to 60% savings of total I/O time over the naive placement of files. More computationally intensive hill climbing and genetic algorithms approaches did not outperform statistical methods. The experiments were run on a real and simulated hard drive in single and multiple user environments.

Introduction

The performance of computer systems has rapidly improved during the last 40 years. However, the speed of I/O components still lags behind the speed of all other major components in most computer systems. Since most of the I/O deals with transferring information between computer memory and disks, it requires the movement of physical parts that have inertia. A major bottleneck is created where computer cycles and user time are wasted waiting for the data transferring to complete.
[1]

Typically, disk caching has been used to maintain frequently used files in main memory in order to minimize disk accesses. However, many files are not accessed frequently enough to stay resident in cache. It still may be possible to decrease the total access time if files with temporal locality also have spatial locality. This would decrease the amount of mechanical movement within the disk drive leading to increased throughput and potentially decreased mechanical wear.

We have investigated several placement algorithms using two independent data sets. The experiments were conducted on simulated hard disks of various sizes as well as on a raw hard disk. It is worth noting that the proposed methods will also apply to similar physical devices such as CD–ROMS, DVDs, tape drives and MO drives. These media all have similar mechanical limitations that prevent the further improvements in the I/O speed.

File System Optimization

A typical hard disk consists of several *platters* that rotate together on a spindle. Each platter has tracks on its surface where data is stored. A *sector* is the smallest unit of a track that can be read or written independently. Tracks vary in the number of sectors they can hold due to the different circumferences. To simplify the simulated experiments the tracks are assumed to be of the same size. A *cylinder* consists of all tracks with the same circumferences located on different platters.

The goal is to minimize the average and total access time. *Access* time consists of seek time and latency time. *Seek* time is the time it takes for a drive head to position itself over the correct track. *Latency* time is the time it takes to rotate the disk and reach the correct sector. This paper focuses on minimizing seek time component of total access time. We present the various placement strategies for reorganizing the files on disk and thus reducing the average head movement.

Mathematically we can write the problem as follows. Suppose we have a finite collection of file accesses $\{\xi_1, ..., \xi_k\}$ where $\xi_i \in \{f_1, ..., f_n\}$ is an access of one of the n files. Let d be some distance measure. We would like to find such placement of files $\mathcal{F}$: $\{f_1, ..., f_n\} \longrightarrow \Re$ that the average seek time $\sum_{i=1}^{k} \frac{d(\mathcal{F}(\xi_{i+1}) - \mathcal{F}(\xi_i))}{k}$ is minimal.

If we know the transition probabilities among the file accesses the optimal solution could be found in $O(n!)$ time by looking at all possible file placements $\mathcal{F}$. This solution is infeasible for typical file systems. (Horne, Giles, & Philbin 1996) propose to place files one by one at locally optimal locations. Such locations are

[1]Published in the *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artifical Intelligence Conference*, p, 528-533, MIT Press, Cambridge Mass. 1997.

determined by evaluating a cost function, which measures the expected amount of head movement. This approach uses previous history to predict the transition probabilities of file accesses. It could be implemented in $O(n^3)$ time, where $O(n^2)$ is spent to calculate the cost and $O(n)$ to find a location with the minimal cost. If the probability matrix is sparse then the complexity can be reduced to $O(n^2 \log n)$ by incrementally computing the expected cost. However, for a large file system with many thousands of files even the quadratic solution is prohibitively expensive. We propose several strategies that can be implemented in linear time.

Related Work

Problems with file system performance have been dealt with extensively in the literature, see (Smith 1981) for a survey. The work by (McKusick *et al.* 1984) proposes to partition a disk into cylinder groups consisting of several consecutive cylinders. The files that are frequently accessed together, e.g. inodes of the files in the same directory, are stored in the same cylinder group.

(Madhyastha 1996) devised a trained learning mechanism to recognize access patterns and automatically select appropriate caching strategies. The pattern classification was performed by neural networks.

(Shih, Lee, & Ong 1990) presented algorithms for an adaptive prefetch design. They are based on the observed sequentialities of files during program execution. The history of cache hits and misses for each file was used to calculate the optimal prefetch amount.

(Rosenblum & Ousterhout 1991) argued that files tend to be read in the same patterns that they are written and proposed to write data sequentially on disk. Other work by (English & Stepanov 1992) investigated several heuristics for dynamical placement of file blocks. Our work is focused instead on global reshuffling of file system that reduces access time for reading.

Another approach for optimizing file systems includes (Tuel 1978) where the optimal reorganization intervals for linearly growing files is discussed. Disk striping, proposed in (Ganger *et al.* 1993), uniformly spreads data sets across the disks in the subsystem and essentially randomizes the disk accessed by each request. This randomization effectively handles both fixed and floating load imbalance.

More details on ways to optimize a file system can be found in (Kuvayev *et al.* 1996).

Various Placement Strategies

There are many possible file placement strategies. In this study, we experimented with five of them. We call them *naive*, *contiguous*, *type based*, *frequency based*, and *markovian* placements. Their pictorial representation is shown in Figure 1.

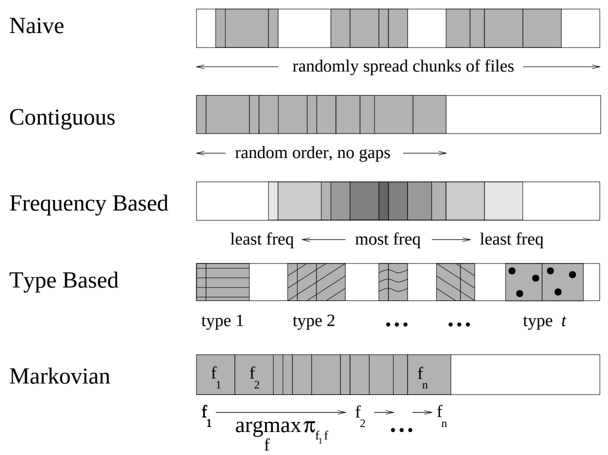

Figure 1: Five placement strategies used in experiments.

Naive Placement. We use *naive* placement to simulate the current state-of-the-art placement of files on a typical hard drive. We assume here that blocks belonging to one file are written sequentially and contiguously. This can be a desired property since most of the files are accessed sequentially. This is not always the case on actual disks but more sophisticated systems will attempt to keep files together. We also assume that the files are scattered along the whole drive in sequencies of a random length. Having gaps on a simulated hard drive is necessary to simulate deletion of files. Even if the files are initially placed contiguously on a real hard drive, some files are deleted and free slots of different sizes appear. All other placement strategies are compared against this strategy.

Contiguous Placement. *Contiguous* placement is the simplest strategy that could be used during weekly backups. First all data is copied to a tape and the disk is formatted. Then the data is placed back on a hard disk leaving no gaps between the files. The idea is to pack the files as densely as possible so that they occupy the fewest number of tracks. As a result, we expect the disk head to travel less than with the *naive* placement. The files are restored from the tape in the order they were originally backed up. This placement is also a good benchmark for other non-random placement strategies.

Frequency Based Placement. This placement strategy places the files by sorting them according to their stationary probabilities. The stationary probabilities are based on the number of times a particular file has been accessed. They are computed from the accumulated logs of file accesses. The most frequent files are placed first on the middle tracks while the less frequent files occupy the positions toward the edges of the disk. One expects that the head will move most of the time along the busy middle part while making occasional trips to the edges. This strategy requires the number of hits to be recorded for each file.

Type Based Placement. The *type based* approach utilizes the locality of accesses among the files of the same type or belonging to the same group. The files that are believed to be related are grouped into categories. Files of the same category are placed close to each other. In order to implement such placement a hard drive is partioned into several segments with each segment corresponding to a particular category. Files are placed on a corresponding segment of a hard drive. For example, one category could be a LaTeX package, various fonts, and viewing utilities. The segments are allocated with enough free space so that they are not filled up too often. The head movement between segments could be potentially bigger than in the *contiguous* placement. However, we expect it to be infrequent compared to the movement within a segment. To implement this strategy the file type must be known when the file is first created. The classification can be accomplished by looking at the file extension or by consulting a type dictionary.

Markovian Placement. This strategy employs transition probabilities to find a locally optimal placement of the files. We assume that file accesses $\{\xi_i\}$ form the first-order Markov chain. To predict an access ξ_{i+1} we need to know $\Pr(\xi_{i+1}|\xi_i)$. The transition probabilities π_{ij} of file i followed by file j are calculated based on the historic data. We start with any file f_1 and place the file f_2 that has the highest probability to be accessed next, i.e. $f_2 = argmax_f\{\pi_{f_1 f}\}$. The same is repeated for f_3 and so on. If during placement it happens that the next best file is already placed on a disk, then the next file is randomly selected from the remaining pool until all files are placed. Thus, we place all files contiguously with the ordering that greedily minimizes the traveling distance of the head. Of course, the algorithm can be extended to higher order Markov processes at significantly more computational cost.

Experiments on a real hard drive

This section demonstrates the potential improvement gains on an actual hard drive. The traces of file accesses were obtained from a single user workstation running Linux operating system. All file accesses during the three day collection period were traced and collected in a log file. There are 1.1 million entries in the log that represent user and system requests and access 400 Mbytes of files. The traces contain time, the access type (read, write, or execute), the inode number, the offset within the file and the number of bytes read or written.

The training and test traces are the same for the experiments on a real hard drive because of the difficulty of creating new files with the OS simulator. However, the simulated experiments in the next section were run on previously unseen test traces. Seek time could not be clocked separately on a real hard drive, therefore, we measure the time spent on reading the complete log. As shown in Tables 1 and 2 the savings are significant even though we only minimize the seek component of total I/O time.

Non-cached accesses

A hard drive with 2 Gbyte capacity has been used as a raw device. There is no file system on the drive. Files are simply chunks of blocks of the size obtained from the trace information. The raw device allows us to place files exactly to our specifications without changing the code for the file system in the OS. The experiments were run on SGI-ip22, 100Mhz machine. Its raw device references are never cached.

Table 1 shows the results for five placement strategies discussed above. The 'ave seek' column contains the length of the average seek in Mbytes which equals the total seek distance divided by the number of accesses. *Markovian* placement allows significantly shorter seeks than *naive* placement.

The percentage of seeks with the distance less than 1 Mbyte is displayed in 'seek $<$ 1Mb' column. The short seeks are especially important because they are likely to be on the same cylinder and do not cause head movement. The short seeks are mostly responsible for the savings in I/O time.

'I/O time' shows the total I/O time spent to run through all accesses in the log. This time constitutes time spent on seeks, rotational latency, and transfer of data. The time for the disk with a file system can be less or more due to slower I/O and cache savings.

The same run was repeated several times in order to compute standard error of I/O time. The small variance in timing from run to run is attributed to CPU handling other light weight processes. The hard disk itself was entirely dedicated to the experiment. The percentage savings over the *naive* are shown in the last column. All improvements yield more than 99% significance when a t-test is performed on the samples.

Cached Accesses

It is important to know how much of the savings are attributed to the cached accesses. We repeated the experiments on a Linux machine with Pentium-133Mhz processor and the same 2 Gbyte hard drive being used again as a raw device. In a Linux environment all accesses go through cache. We varied the cache size by booting the machine with 16 and 64 Mbytes of available cache. The amounts of cache actually used were a little less since some cache was taken by the data structures and the log file.

It can be concluded from Table 2 that the savings are bigger when there is cache available since some of the accesses do not hit the hard disk. Overall the potential

placement strategy	ave seek	seek < 1Mb	i/o time	st error	improvement
Naive	68.5 Mb	79.6%	4341 sec	19.29	—
Contiguous	29.4 Mb	79.8%	4174 sec	28.07	3.8%
Frequency Based	12.7 Mb	82.3%	3979 sec	59.70	8.3%
Type Based	25.5 Mb	85.5%	3701 sec	23.82	14.7%
Markovian	14.6 Mb	89.7%	3629 sec	46.20	16.4%

Table 1: Performance of a raw device without cache.

Placement	i/o time w/ small cache	i/o time w/ large cache	improvement
Naive	1655 sec	1377 sec	—
Contiguous	1567 sec	1206 sec	5.3–12.4%
Frequency Based	1288 sec	950 sec	22.2–31.0%
Type Based	831 sec	593 sec	49.8–56.9%
Markovian	706 sec	494 sec	57.3–64.1%

Table 2: Performance of a raw device with cache.

savings could be in the 16–64% range with the best placement depending on the amount of available cache.

In both experiments, we trained and tested all algorithms on the same log. Thus, the percentage of savings serves only as an upper bound. However, at least in a single user case the new accesses will come from the same distribution, and the savings should approach the upper bound as more historic data is accumulated. The following section discusses the more revealing experiments with unseen test data.

Simulated hard drive experiments

In this section, we discuss experiments on a simulated hard drive. The traces were obtained from single and multiple user environments. In both cases the training and testing data sets were different. The entire file system was kept in memory. The access time was based on the travel distance between the current disk head position and the next destination. To convert the distance to the actual time we linearly interpolated between minimum and maximum seek times.

Single user environment

We simulated all five placement strategies on hard drives of three sizes: 0.5 Gbyte, 1 Gbyte and 4 Gbyte. The minimum and maximum seek time were taken from Seagate "Barracuda" drive specs. The drives contained the same number of tracks, 3711, and different number of platters, 5, 10, and 40 respectively.

The traces are the same as in the previous experiment. We split the traces in two halves. The files for the first half were placed according to the strategies defined above. *Frequency based* and *markovian* placements used these data to estimate stationary and transition probabilities. The files from the other half are to simulate the next day usage, therefore the newly created ones were written in the first available space forward from the disk head.

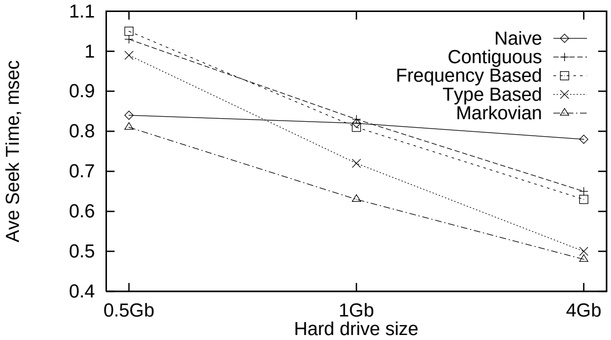

Figure 2: Performance under five placement strategies. Data was obtained from a single-user Linux workstation.

The types for the *type based* placement were collected in a file dictionary of 57 types. The examples of types are GIF binary, ascii text, Linux executable and others.

All placements but *naive* performed better on more spacious drives, as shown in Figure 2. With more space available on each track the savings become more apparent. On the small drive the data occupies 80% of the drive so the contiguous placements are not much different from the *naive* one. The *naive* placement has the advantage of putting files in the order in which they arrive and utilizes some of the correlation among the files, thus explaining its advantage on the small drive.

The *markovian* approach showed the best performance for all drive sizes. It improves by 38.5% over the *naive* placement and significantly over all others on the medium and large drives.

Applying search methods

The methods that we described and implemented above are not the only ones applicable to the problem of file system optimization. In this section we show that search methods are less successful on this problem. We applied hill climbing and genetic algorithms for searching in the space of available orderings and demonstrate that the *markovian* and *type based* placements outperform the ones found by both search methods.

An element of the search space consists of n file ids $f_1, \ldots, f_n$ that constitute the order in which files would be placed on a disk. To evaluate the ordering we use the same training traces that we used for previously discussed placement strategies. The files are placed on a 4Gb simulated disk described in the previous section and the training traces are run on that disk. The average seek time per access is the value returned by an evaluation function.

Hill climbing algorithm is a common search method. The orderings are altered slightly and evaluated on the training traces. The best ordering improves gradually over time.

Genetic algorithms is another popular search method (Holland 1975). The idea is to mimic the nature's laws of evolution and natural selection. Initially there is a population of 300 random file orderings. All of them are tested on the historical traces and sorted in the ascending seek time order. The first third of the population survives to the next round of evolution while the rest of the orderings are discarded. The standard practice is to select individuals with a probability proportional to their fitness. However, the deterministic selection was significantly faster and produced similar results in our case. Then we apply crossover and mutation operations to produce new orderings. The new orderings are similar to their ancestors so that the overall fitness of the population tend to increase. The testing, selection, and reproduction processes are repeated until the fitness of the population converges to a local or possibly the global minimum.

During crossover the randomly chosen segment is exchanged between two parents. Two newly produced orderings are normalized to ensure that each file occurs only once in the ordering. On each iteration we repeat a crossover operation 100 times to create 200 new children. Then each file block is mutated, i.e. swapped, with a probability of 10% to maintain the diversity among the population.

In both simulations we used the 4Gb hard drive configuration described in the previous section. The time reported in Figure 3 can be compared with one in Figure 2. We repeated the simulations five times starting with a different random population each time. The top curves are the results of the best individual on the unseen half of the traces. The bottom two curves are the performances on the training half. Unfortunately, there is no computationally feasible way to reach the global minimum even on the training data. The complete search needs to test $O(n!)$ of different file orderings which is inadequate even for a minuscule file system.

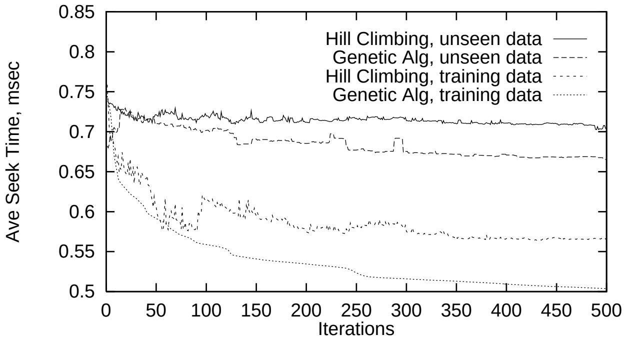

Figure 3: Performance of hill climbing and genetic algorithms placement strategies. The same 4Gb hard drive was simulated as in Figure 3. The performance improves significantly on training data and slightly on test data.

Even if we could reach the global minimum on the training data, it may not be the best choice for the test data. As shown on the top two curves in Figure 3, the performance on the test data improves slightly in the beginning and continues to oscillate as the ordering gets significantly better on the training data. We have shown that two ingeniously implemented though computationally feasible search techniques do not match the performance of *markovian* and *type based* placements on the unseen data.

The file system can be significantly optimized for a single user. In the next section we discuss a distributed environment where the correlation among file accesses is expected to be weaker.

Multiple user environment

The traces for this experiment follow the NFS activity of 236 clients serviced by an Auspex file server over the period of one week during late 1993. They were gathered by Cliff Mather by snooping Ethernet packets on four subnets. The clients are the desktop workstations of the University of California at Berkeley Computer Science Division. The traces can be downloaded from now.cs.berkeley.edu/Xfs/AuspexTraces/auspex.html.

The only difference between these data and the data for the single user experiment is the additional information about which computer and which file system were processing the access. We did not use this information; instead, we assumed that the accesses come from one workstation and go to one file system. The distributed origin of file accesses make the locality principle less advantageous.

We did not use the *type based* strategy in this experiment since there was no type information available in the Auspex data set. The simulated 10 Gbyte hard

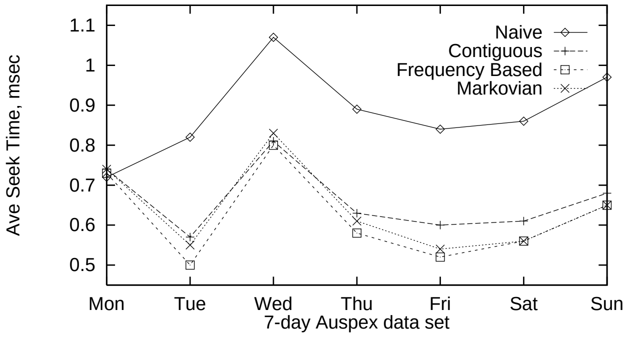

Figure 4: Performance under four placement strategies. Daily data, Auspex trace, was obtained from a network of workstations. All intelligent methods significantly outperform *naive* placement.

drive contained 4000 tracks, 40 platters, and 64 sectors. All other parameters were the same as in the single user experiments.

The results are shown in Figure 4. It could be seen that all contiguous placements performed significantly better than the *naive* one. With each new day there was additional statistical information available but it did not help the *frequency based* and *markovian* placements to increase the edge over the *contiguous* one. We believe that a lack of pattern in the accesses reduced the advantages of the intelligent optimization methods in this environment.

Conclusions

Three intelligent placements: *frequency based*, *type based*, and *markovian* demonstrated an impressive advantage over the *naive* placement which corresponds to the current state-of-the-art on most machines. They also consistently outperform a randomly contiguous allocation.

On single user data the *markovian* placement showed undisputably the best performance improving the average seek time by 38.5% over the *naive* approach. A user may have certain patterns in file accesses that will be reflected in the transition probabilities. The stronger the patterns are, the more predictable file accesses will become. In a multi-user environment, it performed only slightly better than the *contiguous* case and still beat the *naive* case by 32.9%. We also demonstrated savings on a real hard drive reducing the total I/O time by up to 64.1% using the *markovian* strategy.

Markovian placement could be performed once a week during backup time. It could make a pleasant Monday morning surprise for users to see their I/O time reduced by 10–20%. Other methods such as hill climbing and genetic algorithms did not outperform the statistical methods in finding the better file placement. Search methods are also more computationally intensive.

The next step is to implement the *markovian* strategy at the operating system level. The file accesses will be accumulated in a log that will be used during a weekly reshuffle. The tradeoff between extra logging time and reduced I/O time will ultimately determine the usefulness of the proposed idea.

The current research applies not only to hard drive technology but also to other types of media where mechanical properties create a bottleneck and locality of accesses could reduce a seek time. Such media are read/write compact disks, optical disks, and tape drives. In the future, storage devices will tend to become more capacious. The seek time or the time to find the information on a device will play a more significant role, as the time for transferring information continually shortens. This research has shown the savings that can be achieved by using the intelligent placement of data.

References

English, R., and Stepanov, A. 1992. Loge: a self-organizing disk controller. In *USENIX Winter*, 238–252.

Ganger, G.; Worthington, B.; Hou, R.; and Patt, Y. 1993. Disk subsystem load balancing: disk stripping vs. conventional data placement. In *IEEE*.

Holland, J. 1975. *Adaptation in Natural and Artificial Systems*. University of Michigan Press.

Horne, B.; Giles, C.; and Philbin, J. 1996. A method for optimizing file system organization. Technical report, NECI, Princeton, NJ.

Kuvayev, L.; Giles, C.; Philbin, J.; and Cejtin, H. 1996. The impact of intelligent methods on file system optimization. Technical report, NECI, Princeton, NJ.

Madhyastha, T. 1996. Intelligent, adaptive file system policies. *Frontiers of Massively Parallel Computation*.

McKusick, M.; Joy, W.; Leffler, S.; and Fabry, R. 1984. A fast file system for unix. *ACM Transactions on Computer Systems* 2(3):181–197.

Rosenblum, M., and Ousterhout, J. 1991. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, 1–15.

Shih, F.; Lee, T.-C.; and Ong, S. 1990. A file-based adaptive prefetch caching design. In *IEEE*.

Smith, A. 1981. Input/output optimization and disk architectures: A survey. *Perform. Eval.* 1:104–117.

Tuel, W. 1978. Optimum reorganization points for linearly growing files. *ACM Transactions on Database Systems* 3(1):32–40.