Disk Shuffling

Chris Ruemmler, John Wilkes
Software and Systems Laboratory

HPL–91–156
October, 1991

disk layout algorithms,
seek distance
optimization,
disk data placement,
disk reorganization

Adapting disk layouts to observed, rather than predicted, access patterns can result in faster I/O operations. In particular, a technique called disk shuffling, which moves frequently-accessed data into the center of a disk, can substantially reduce mean seek distances. We report here on extensions to and (partial) validation of earlier work on this approach at the University of Maryland.

Starting from disk access traces obtained during normal system use of 4.2BSD-based file systems, we established a repeatable simulation environment across a range of workloads and disk types for comparing different shuffling algorithms. We explored several of these, including how often to make layout changes, how large a unit of data to shuffle, and some mechanisms to exploit sequentiality of reference.

Our conclusions: some of the originally identified benefits are real, but sometimes performance is worse rather than better unless access interdependencies are considered. Most of the benefit can be obtained from infrequent (weekly) shuffling. Smaller quanta generally produce better results, at the expense of needing more working storage.

Overall, the benefits are small to moderate, but are likely to be much larger with file systems that do not do such a good job of initial data placement.

1 I/O performance

VLSI technology is improving computer processors much faster than mechanical disk engineering is able to improve secondary storage access times. The resulting divergence of performance threatens to limit the benefits obtainable from faster processors—the so-called "I/O gap". Therefore, speeding up disk accesses is important, but improving the underlying performance of the disk is hard: power consumption and bearing roughness limit the speed at which disks can be spun, and arm flexibility and servo power restrictions bound the rate at which heads can be moved from track to track. Instead, three software techniques have been used to improve the way that disks are used.

The first approach is to reduce the frequency of disk accesses. Caching data in high-speed primary memory can eliminate many secondary storage accesses, and can move some of the remaining ones off the critical path through read-ahead and write-behind. Caching only works well for data with moderate (or predictable) locality of references. Furthermore, the size of primary memory is constrained by its relatively high cost compared to secondary storage.

The second approach is to amalgamate several I/O operations into one to amortize the mechanical disk delays over a large transfer. This requires possibly substantial modifications to the file or virtual memory system (e.g. [Ousterhout89]), and can result in less effective utilization of the primary memory cache unless done with care [McVoy91]. Furthermore, it is usually not possible to amalgamate synchronous I/O operations without impacting I/O latency—and hence application performance. Although larger main memory caches will doubtless reduce the need for synchronous disk READs, they do not eliminate it—indeed [Braunstein89] and [Baker91] suggest that the benefits of large buffer caches are more difficult to achieve than was predicted by earlier studies such as [Ousterhout85a].

The third approach, and the subject of this paper, is to reorganize the layout of data on disk as a function of the observed workload to minimize the time spent moving the head between data blocks of interest.

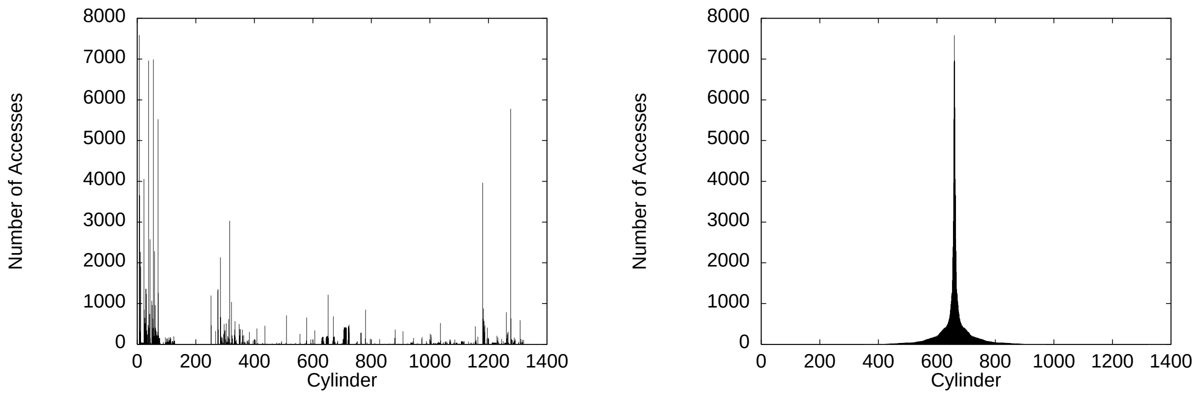
The paper is organized as follows. The next section introduces previous work in this area. It is followed by a detailed description of the hypotheses we wished to test, and the experiments we performed to do so—trace gathering, simulation, and verification. Results from these experiments are presented next with our analysis of them, and some conclusions.

2 Related work

Commercial data processing file systems have long taken advantage of large transfer sizes. For example, IBM's OS/360 and its successors allow programmers to specify file transfer units up to a full track in size, and allocation policies that grow files in units of up to many cylinders of data at a time [Clark66]; MPE XL also uses extents, but the system chooses their size as a function of file size, access type, remaining disk space, and growth rate [Kondoff88].

The early UNIX[1] file system designs were extremely clean and elegant [Ritchie74, Thompson78], but they sacrificed efficiency by allocating small (512 byte) blocks randomly across their disks, and thus were only able to extract a small fraction of the raw disk bandwidth because most of the time was spent on seek and rotational delays [McKusick84].

[1] UNIX is a registered trademark of AT&T Bell Laboratories.



a. As measured on the original layout.

b. After rearrangement into an organ pipe.

Figure 1. Cylinder access distribution over a twenty-four hour period. Measured on an HP7935 disk attached to *red*, a timesharing system running HP-UX.

Some benefits were achieved by scheduling disk operations in an order that optimized access time, such as the shortest seek time first algorithm (SSTF) and variants on it such as SCAN and V(R) [Oney75, Coffman82, Geist87]. This improved throughput in cases where there were many requests in the queue, but did nothing to improve latency for synchronous accesses—indeed, it usually got worse. Moving the disk head to the center of the disk when the disk appears to be idle can improve latencies under low load conditions [King90], but does not improve the high-load case.

The Fast File System of version 4.2BSD of the Berkeley UNIX system and its successors introduced substantial improvements [McKusick84]. The new file system increased transfer sizes by using larger blocks, but held down internal fragmentation by packing several small files (“fragments”) into a block; reduced seek times by clustering statically-related data together into “cylinder groups” (data from our UNIX systems show that 30–50% of all physical I/Os are to the same cylinder as their predecessor); and reduced rotational latency delays by using a block layout scheme optimized for sequential accesses.

The 4.2BSD scheme leaves room for improvement in two areas: it makes no attempt to optimize operations that cross cylinder group boundaries (indeed, a quadratic hash function is used to scatter file blocks widely across the disk if the current cylinder group fills up); and, although it does a very creditable job of initial data placement, it takes no notice of dynamic access patterns. Figure 1a shows a typical distribution of accesses across a disk from a 4.2BSD-based file system.

In most environments, disk accesses display significant physical and temporal locality, even in the face of increasing file buffer cache sizes [Majumdar84]. Figure 1a supports this: many cylinders are never accessed over a one-day period.

Grouping the most frequently-accessed data blocks together at the center of the disk can reduce the average seek distance substantially. A good way to do this is the *organ pipe arrangement*, formed by placing the most frequently accessed cylinder in the middle of the disk, the next most frequently accessed cylinders on either side of the middle cylinder, and so on. This arrangement is provably optimal for independent disk accesses [Wong83]. The effect of applying it to the disk shown in Figure 1a is displayed in Figure 1b.

Moving the data around to achieve such an arrangement is known as *disk shufflin*g. The algorithm is sufficiently straightforward that it can be done inside the disk controller, or inside the device driver, as in [Carson89, Vongsathorn90]. In all cases, a count is kept of the number of requests directed to each *shuffling quantum* (e.g. cylinder) over a period of time, and these counts are used to drive the rearrangement. The shuffling can be done by the disk controller itself, by the device driver, or by an outside daemon process that reads the frequency table (e.g. through an ioctl system call that queries the driver state from a program running in user mode).

A map containing the current location of each logical cylinder on disk is stored in a known, fixed place on the disk (e.g. in the first physical cylinder). This map is updated every time the disk is shuffled, and read on power-up. By recording an intentions list in the map and performing the cylinder shuffle so that there is always a valid, identifiable copy of the data stored on disk, it is possible to make the shuffling operation resilient to power failures [Vongsathorn90]. With care, the shuffling can be done while access to the file system on the disk continue, and can be made invisible to upper layers of the file system code. The rearrangement can be done during “idle” periods, to minimize the impact on normal operation.

Since real workloads do not generate independent disk requests, the straightforward organ pipe arrangement is not always optimal. For example, a large sequentially-accessed file that used to be contiguous could end up split across either side of the organ pipe, thus increasing the average seek distance. By looking for such dependency patterns in accesses, the organ pipe arrangement can be adjusted to keep sequentially-accessed data close together. Using such techniques can result in reductions of average seek distances by as much as 30% compared with straightforward organ-pipe shuffling [Carson89]. By measuring the interdependencies at a low level in the system, account is automatically taken of both sequential accesses and clusters of related accesses, such as occur sometimes when a program is started up and it reads its configuration data.

Not all disk accesses will benefit from the kind of adaptive methods discussed here. For example, data with a lifetime less than the shuffling period will probably not benefit from data shuffling. In fact, such data may degrade the shuffling algorithm’s performance by distorting access frequencies. (However, if it occupies “holes” that are frequently reused, shuffling the holes can still produce benefit.)

A variant on physical shuffling is to perform logical shuffling, for example, at the level of entire files. This is the approach adopted in the iPcress file system [Staelin91]. An obvious advantage of this technique is that it can more easily keep associated data together; disadvantages are assumptions that an entire file has a common access rate, that intra-file data associations are more important than inter-file associations, the variable-sized nature of the objects that have to be moved around, and the modifications that have to be introduced into the file system implementation(s) that are to take advantage of it.

3 Hypotheses

Cylinder shuffling was tested at the University of Maryland by Carson and Vongsathorn [Carson89, Vongsathorn90] using physical cylinders as the data unit, and shuffling the disk once a day. On their (rather slow) disk, they reported a rough halving of seek times. We decided to attempt a validation of their work on a wider range of input data and over a larger range of conditions. Basing our expectations on their work, we started out with a number of hypotheses that we decided to test:

1. Adaptive disk shuffling is beneficial even with fast disks and moderately-sized file caches.
2. The smaller the unit of shuffling, the greater the performance gain—although at the expense of requiring more internal state for access counts.
3. There is some optimum time interval between reshuffles. If rearrangement is too frequent, too much time will be spent shuffling data, and performance will decrease. If shuffling is too infrequent, performance will be lost as the layout becomes less well adapted to the current use patterns.
4. Most of the benefit can be obtained by rearranging only a small portion (e.g. 20%) of the disk.
5. Keeping track of dependencies between disk accesses *is* of value, especially for the smaller shuffling units.

We tested the validity of these hypotheses under a number of different conditions, including: several disk types; a range of workloads; different shuffling units (block, track, half-cylinder, cylinder); a range of time intervals between rearrangements (1 hour to 1 week); subsets of the total disk space being shuffled; different dependency-detection policies.

This paper reports the results of these investigations.

4 The experiments

We constructed a disk simulator and fed it with physical disk-access traces gathered from a number of our local UNIX systems, giving us the twin benefits of realistic data and experimental repeatability: the traces could be repeatedly replayed into simulations of different policies to compare their effects. This section describes the traces we gathered, the simulator, and the experiments we ran to calibrate it.

4.1 Tracing I/O accesses

Physical I/O activity was measured on a number of systems (listed in Table 1). All of these systems were running release 7.0 of the HP-UX operating system. The implementation of HP-UX is derived from 4.2BSD, while its external interface conforms to a superset of AT&T's System V validation suite [Clegg86]. All three machines had 3 MB file buffer caches.

Table 1. Systems used for tracing.

Name	*Processor*	*Disks*	*Type*
cello	HP 9000/845	HP C2204A	Timesharing; editing/mail/programming; ~20 users
red	HP 9000/840	HP 7935H	Timesharing; news/mail; ~200 users
hplajw	HP 9000/834	HP C2200A	Single user workstation; editing/mail

The traces were obtained using a software-based measurement system built into the operating system. The events recorded for these experiments captured *request enqueue time* (when the driver first sees an I/O request), *start time* (when physical I/O is initiated), and *finish time*. As well as a timestamp, each trace record included a block number, device number, transfer size and whether the I/O was a read or a write. The times were recorded to a 1 µs granularity; the clock is accurate to within 0.05%, but has a very much smaller drift rate than this.

The first trace was gathered over a 7 day period in late July and early August 1990 on system red (see Table 1). A second set of traces was gathered by monitoring the systems cello and hplajw during a much longer period: February to June 1991. Table 2 summarizes the traces used in these experiments.

4.2 The disks

Since the goal of adaptive disk layouts is to reduce seek and rotational latencies, it is important to have a good model of what these are on real disks. Our first task was to characterize the performance of the disk drives that we were going to simulate.

The graphs of seek time as a function of distance for these drives have two portions: a constant-acceleration region where the seek time is proportional to the square root of the distance, followed by a constant-velocity portion where increasing distance results in linearly increasing time. Through a series of measurements, we obtained data that allowed us to characterize the behavior of the disks over the parameters of interest to us. Table 3 shows the results. The seek time data is plotted graphically in Figure 2.

Disk track and cylinder sizes are rarely convenient multiples of the file system block size, so we also conducted a sequence of experiments to measure the effects of track and cylinder switching on data transfer rates for both reads and writes, and incorporated an approximation of these into our models.

The rate at which data comes off, or goes onto, the disk medium was calculated from the product of track capacity and rotational speed. The transfer times across the interconnect between host and disk controller were measured as a side effect of our data transfer experiments. We chose to ignore channel contention effects, as our measurements showed that they were an insignificant factor in the overall I/O times.

Table 2. Summary of the traces gathered.

ID	*System*	*Disk type*	*File system*	*Dates*	*# of I/Os*	*Reads*
C	red	HP 7935	/ (root)+swap	90.7.27 – 90.8.3	727 910	66%
E	hplajw	HP C2200A	entire file system	91.2.11 – 91.3.28	575 669	36%
F	cello	HP C2204A	/users	91.4.22 – 91.6.10	1 678 478	59%
G	cello	HP C2204A	/ (root)+swap	91.4.22 – 91.6.10	1 523 984	61%

Notes: the / file system contained only system files (e.g. /etc, /lib, /bin); the /users file system contained just users' private data, with no system files. The disk on hplajw (trace **E**) was the only disk on the machine apart from the one used for recording the traces: it included both file system and swap space.

Swap space was included in our simulations because we were interested in examining the effectiveness of shuffling algorithms that could be applied in the device driver or disk controller, where the distinctions between file system and swap areas are no longer visible.

4.3 The simulator

Our simulator was written in C++, and was designed to mimic the behavior of the operating system and disks on which the original trace data had been gathered. The software features that were modeled included the disk driver strategy / scheduling algorithm (i.e. its request reordering policy), and operating system and disk driver overheads. The disk model accounted for seek time, rotational latency, channel transfer time, and the performance effects of blocks that crossed track and cylinder boundaries.

We modeled the rotational position of the disks by calculating it from the intial start time and the nominal rotation speed.

Table 3. Summary of disk characteristics

Disk	*Type*	*Size*	*Formatted capacity*[1]	*Cylinders*	*Heads*	*Rotation speed*	*Controller overhead*	*Host interconnect*[3]
HP 7935H	removable	14”	404 MB	1321	13	2700 rpm	3.5 ms	HP-IB
HP 7937H	fixed	8”	571 MB	1396	13	3600 rpm	1.0 ms	HP-IB
HP C2200A	fixed	5.25”	335 MB	1449	8	4002 rpm	1.1 ms	HP-IB
HP C2204A	dual fixed[2]	5.25”	1.3 GB	2×1449	2×16	4002 rpm	1.6 ms	HP-FL
HP 97560	fixed	5.25”	1.3 GB	1962	19	4002 rpm	1.0 ms	SCSI–II

			Accelerating region		*Linear region*	
Disk	*Average seek*[4]	*1 cyl seek*[5]	*range*	*time*	*range*	*time*
HP 7935H	24.0 ms	3.95 ms	2–385 cyl	$2.90 + 1.05\sqrt{d}$	> 385 cyl	$16.50 + 0.018d$
HP 7937H	20.5 ms	4.87 ms	2–525 cyl	$5.28 + 0.79\sqrt{d}$	> 525 cyl	$15.15 + 0.016d$
HP C2200A	17.0 ms	2.5 ms	2–615	$3.45 + 0.60\sqrt{d}$	> 615 cyl	$10.84 + 0.012d$
HP C2204A	17.0 ms	2.5 ms	2–615[2]	$3.45 + 0.60\sqrt{d}$	> 615 cyl	$10.84 + 0.012d$
HP 97560	13.0 ms	3.64 ms	2–383	$3.24 + 0.40\sqrt{d}$	> 383 cyl	$8.00 + 0.008d$

Notes:

1. Capacity information from [HPdisks89a] and the OEM manual for the HP 97560, plus the HP Series 6000 Disk Storage System Owner’s manual.
2. A C2204A disk has two 5.25” mechanisms that a single controller makes look like one—i.e. it appears as a single disk with twice the cylinder count of the individual drives. The performance data reported here is for the individual drives. The simulations of traces from these disks used the parameters for the HP 97560 since these have similar capacity, and don’t exhibit anomalous effects such as multiple command queueing that the C2204As can use as a result of the two mechanisms.
3. Data transfer speeds to and from the host for these disks are:

HP-IB:	1 MB/s from disk to host, 1.2 MB/s from host to disk.
HP-FL:	5 MB/s
SCSI:	5 MB/s in synchronous burst mode.

4. “Average seek” is defined as the sum of all possible seek times divided by the number of seeks.
5. “1 cylinder seek” is sometimes smaller than expected because disk controllers sometimes handle this case specially.

The simulator used the same I/O request ordering algorithm as the HP-UX device drivers. This CSCAN (cyclic scan) algorithm sweeps across the cylinders from zero upwards, servicing all the requests that do not require it to change direction. When it reaches the end of the disk, it (logically) makes one long seek back to the first cylinder and starts afresh to service any remaining requests as well as new ones.[2]

Any trace-driven simulation has to consider how to model the effects of events that happen faster or slower than in the original trace. Arguably the best approach is to model "think time" in the process generating synchronous I/O requests (e.g. reads, forced-writes), and to preserve inter-arrival times for asynchronous ones (e.g. read-ahead, write-behind). Our trace data did not provide enough information to do this, so we chose instead to maintain the original request enqueue times, thus making the request rates independent of the service times.

The simulator both accepted and emitted trace-format data, as did a number of tools we constructed to acquire data, or filter traces. By this means, we were able to pipeline arbitrary combinations of filters, analysis tools, and simulations together. We used the tools to gather averages, standard deviations and frequency histograms for:

- seek distance (in cylinders)
- total I/O time (completion time minus enqueue time)
- physical I/O time (completion time minus start time: i.e. excluding time spent on the I/O request queue)

The major simplification we made was to assume that the layout of the disk could be changed instantaneously. We realize that this is unrealistic: in previous work, it took 30–60 minutes to switch between two significantly different layouts [Vongsathorn90]. However, our early results suggested that it was quite reasonable to do infrequent disk shuffling during "idle" periods, when the costs of doing the rearrangement would be immaterial.

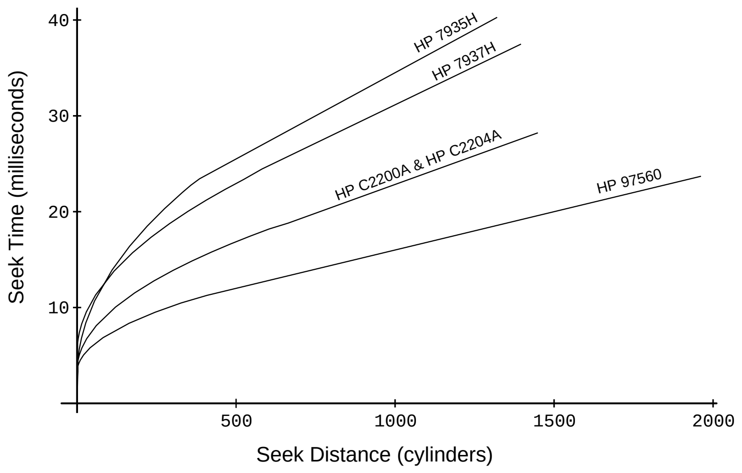


Figure 2. Seek times as a function of distance for the disks used in these experiments.

[2] The real HP-UX algorithm is rather more complex than this, and includes a great deal of code that attempts to avoid starvation effects such as undue bunching of requests at a single cylinder. The effects are only significant in pathological situations, however.

4.4 Simulator validation

In addition to constructing simulations of a variety of different experimental shuffling policies and algorithms we also constructed a base level simulation that did no rearrangements, but merely tried to mimic the observed behavior of the systems we were tracing. This was used to test the accuracy of the simulator, by comparing the output from the simulator with the original traces.

Despite our rather simplistic model of rotation positioning, the mean difference between the measured and simulated I/O durations was only 4% (6% for reads, 3% for writes). Even so, to minimize the chance that we had missed some subtle timing effects, we compared simulator runs against each other, rather than comparing simulator runs against the original trace data. This also allowed us to experiment with using different disk types than those present when the original traces were recorded.

5 Results

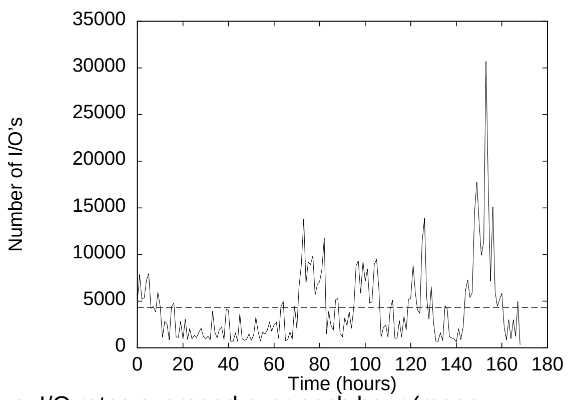
This section presents the tests we applied to our hypotheses. We began by examining an environment similar to that reported in [Vongsathorn90], with moderate loads being applied to relatively slow disks. This is the environment in which disk shuffling is likely to have the most benefit: the moderate load avoids large I/O queues (the execution of which would be optimized by the device driver scheduling policy), and the slow disks emphasize the importance of cylinder placement.

Having convinced ourselves that it was possible to get some benefits from shuffling, and having tried a number of different reorganization frequencies and dependency-detection algorithms, we gathered a great deal more trace data over a longer period of time. These traces were gathered from faster systems with more modern disks than the first set, so we rechecked our earlier results in the new context, and then proceeded to apply a model of the fastest disk available to us (the HP 97560), and evaluated the effects of adaptive layout on it.

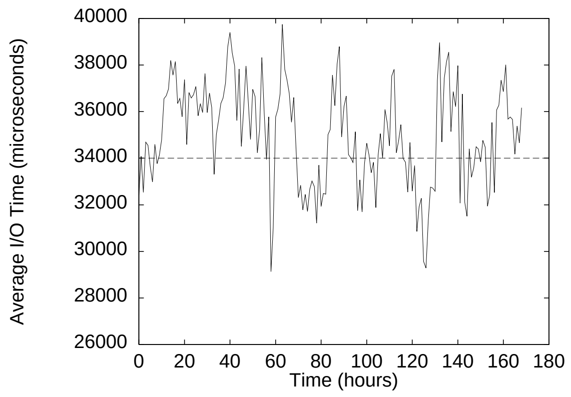
5.1 Benefits and penalties of simple shuffling

Obviously, improving the performance of I/O only brings significant benefits if the system *does* I/O. To see whether this was the case, we began by measuring the I/O rates on the root disk of red, a local timesharing system (trace **C**—see Table 2). Figure 3 shows the distribution of I/O activity over the one week period covered by this trace. Each of the large increases in activity correspond to a weekday, and the low activity period between 20 and 60 hours corresponds to the weekend.

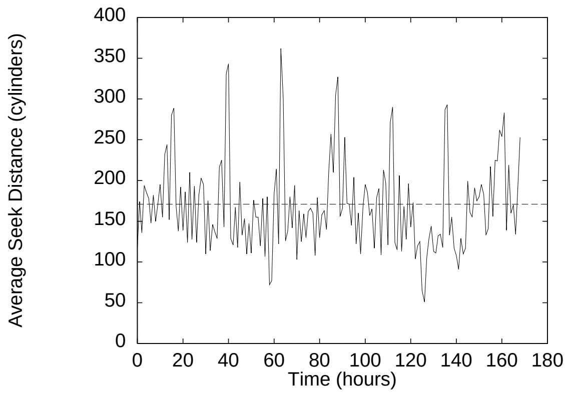
Figure 3 also shows the average physical I/O time and mean seek distances by hour throughout the week. The system is not particularly busy, with a mean arrival rate of one request per second at this disk, although the peak reaches 8.6 requests per second averaged over a whole hour. Typical physical I/Os take 32–38 ms, of which 8 ms is the data transfer time, 11 ms is average rotational latency, and 3.5 ms is controller overhead, leaving around 10–16 ms of seek time on average (these are old, slow disks!). The average seek distance is 171 cylinders, with a standard deviation of 346 cylinders. The distribution of mean seek distances (Figure 3d) shows that same-cylinder accesses are common, short seeks of less than about 50 cylinders are more common than would be expected for a random layout (this is doubtless due to the CSCAN scheduling discipline), and other seek distances occur fairly uniformly (the slope of the graph between 50 and 1321 cylinders is roughly constant).



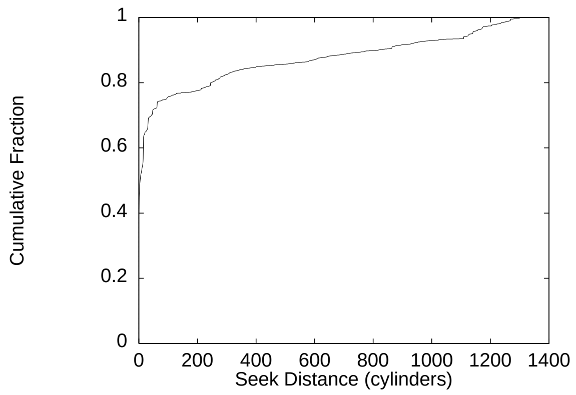
a. I/O rates averaged over each hour (mean = 4307/hour).



b. Average physical I/O time averaged over each hour (mean = 33.9 ms).



c. Average seek distance averaged over each hour (mean = 171).



d. Cumulative distribution of seek distances.

Figure 3. Trace data from root disk of machine "red" over the period 1990.7.27 – 1990.8.3. Trace **C**.

Note: the horizontal dashed lines on these (and other) graphs indicate the mean value of the datum over the interval covered by the dashed line.

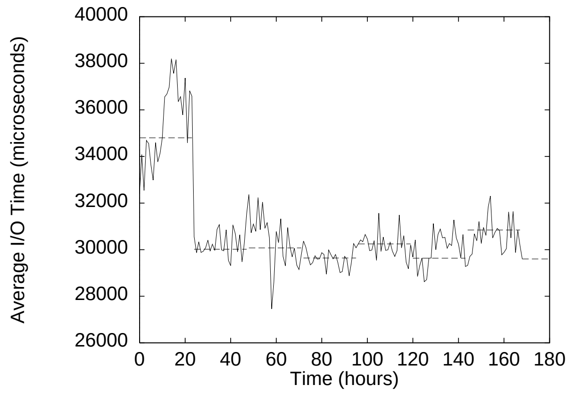
During busy periods (e.g. from hours 70–80), the average seek distance is reduced, presumably because the request scheduling algorithm in the device driver is able to do a better job with longer queues.

The effect of performing a single, straightforward shuffling of this data at the end of the first 24 hours is shown in Figure 3. The shuffling algorithm we used allocates cylinders to the organ pipe arrangement based solely on access frequency, taking no account of dependency data. The first day's results are the same as before, since the shuffling algorithm is merely accumulating access frequencies during this period. Afterward, a dramatic reduction in seek distance is observed (from a mean of 169 to a mean of 36), and a somewhat smaller reduction in average physical I/O time (from 33.9 ms to 30.3 ms) due to the reduction in seek distance.

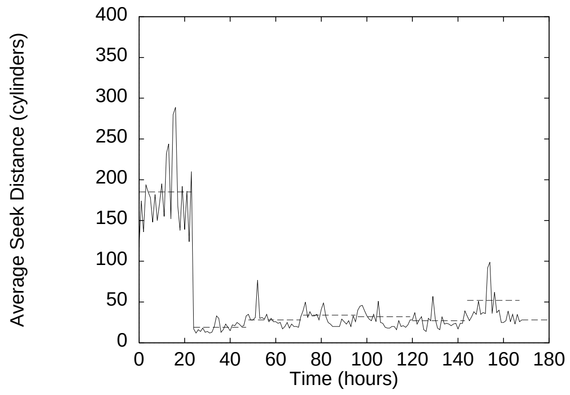
The distribution of seek distances shows clearly what has happened: the graph is skewed towards the smaller distances. Over time, this distribution becomes more uniform as the shuffling done early in the trace becomes less and less appropriate for the workload during the rest of the week.

Since the number of I/Os per hour is not constant, we take as our figure of merit the weighted percentage improvement in I/O time (Figure 3d). This value is calculated as the percentage improvement per hour multiplied by the normalized I/O rate for that period. (To avoid biasing the results with the unshuffled period, it only includes data after the first rearrangement.) For this trace, there is an average weighted improvement of 10.6% in physical I/O time.

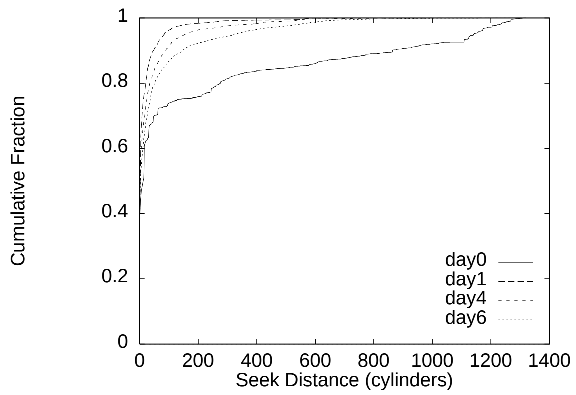
Although the results from our experiments with data from system red were encouraging, and indeed consistent with the data in [Vongsathorn90], we also wanted to validate the benefits of disk shuffling on different disks over a range of workloads. To test the stability of our results, we repeated the tests on three long traces: two from a different timesharing system (cello), one from a personal workstation (hplajw). The results are shown in Figure 5, Figure 5, and Figure 5.



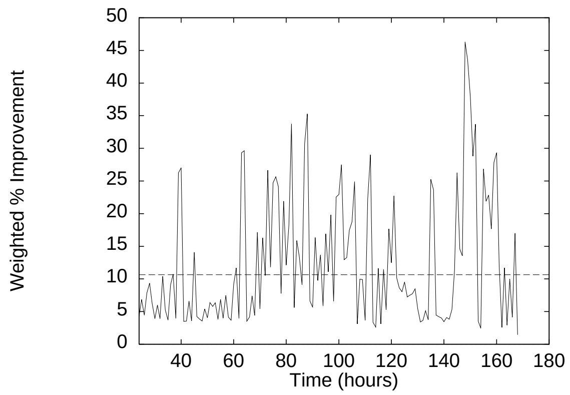
a. Average physical I/O time after shuffling (mean = 30.3 ms).



b. Average seek distance after shuffling (mean = 36 cylinders).

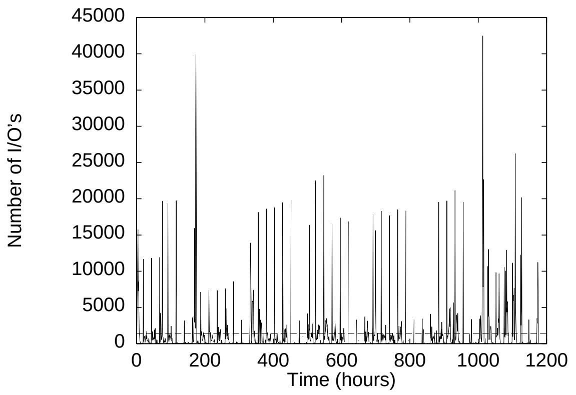


c. Distributions of seek distances after shuffling, as a function of time since rearrangement. Day 0 is before any shuffling.

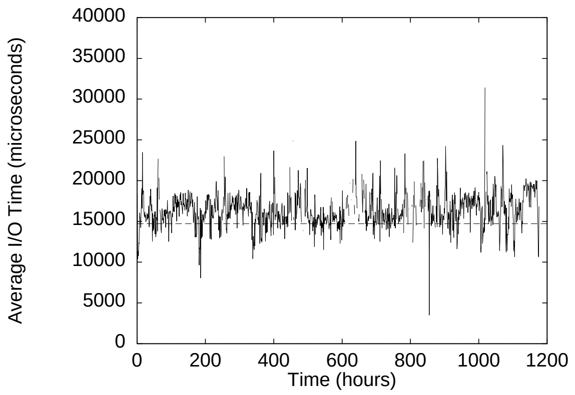


d. Weighted percentage improvement in physical I/O time (mean = 10.6%).

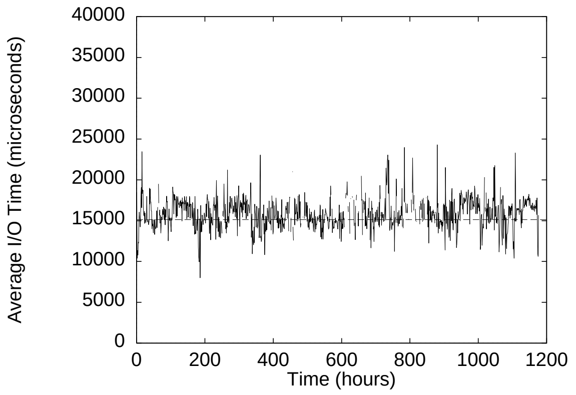
Figure 4. The same data as Figure 3, with a single cylinder shuffle occurring 24 hours into the trace.



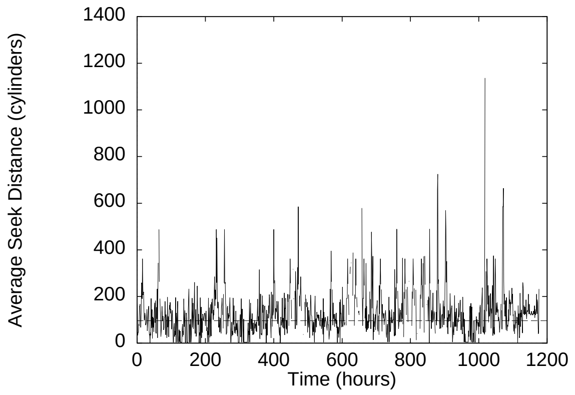
a. Hourly I/O rates (mean = 1426/hour)



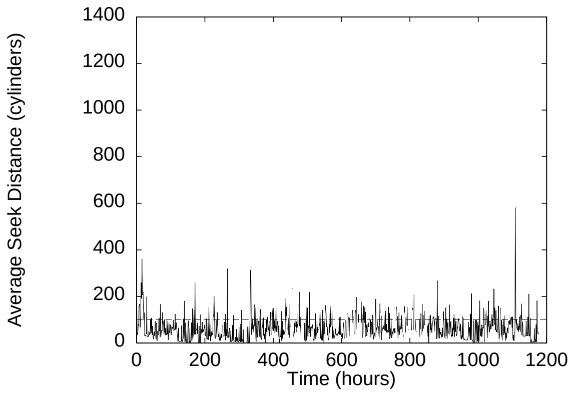
b. Average physical I/O time before shuffling (mean = 14.72 ms).



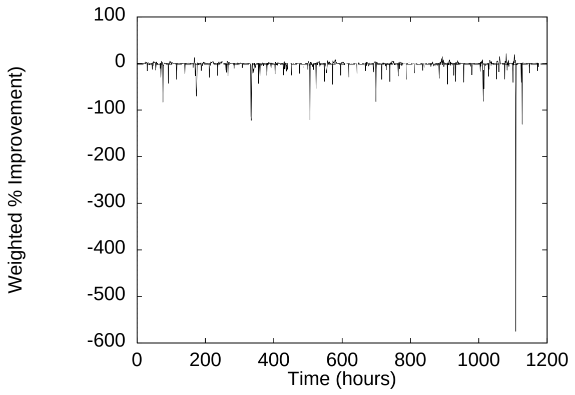
c. Average physical I/O time after shuffling (mean = 15.13 ms).



d. Average seek distance before shuffling (mean = 96 cylinders).

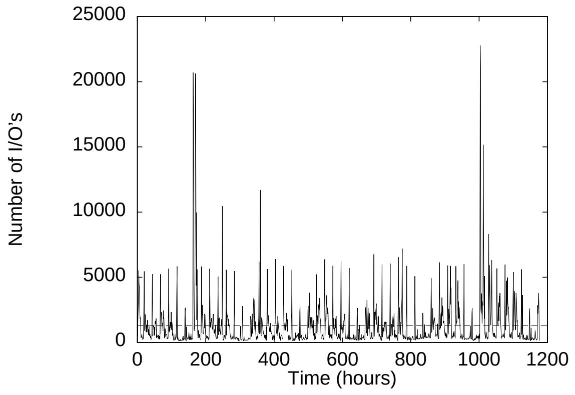


e. Average seek distance after shuffling (mean = 100 cylinders).

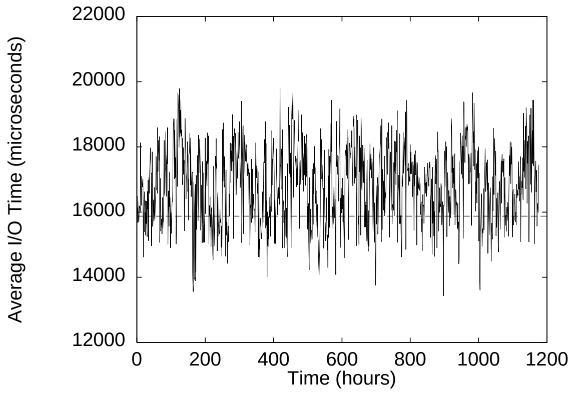


f. Weighted percentage improvement in physical I/O time (mean = –2.66%).

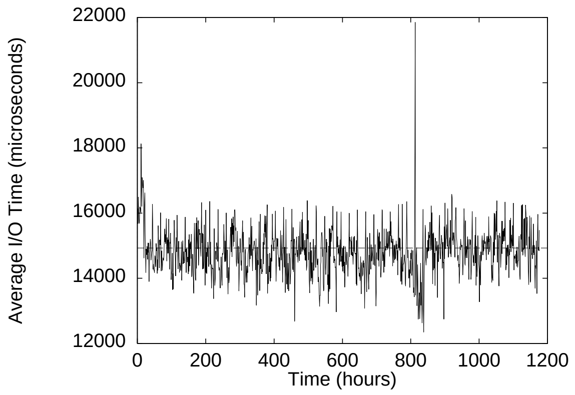
Figure 5. Trace **F** (/users) from cello before and after shuffling.



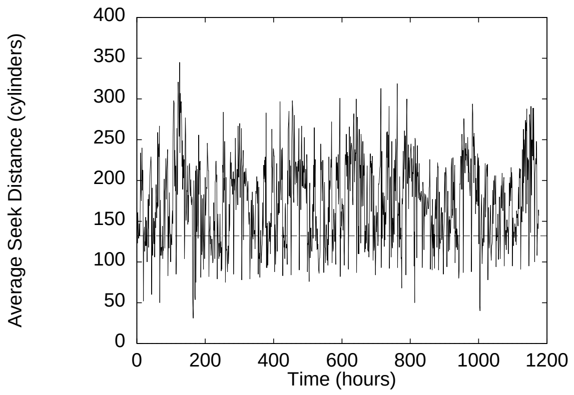
a. Hourly I/O rates (mean = 1294/hour).



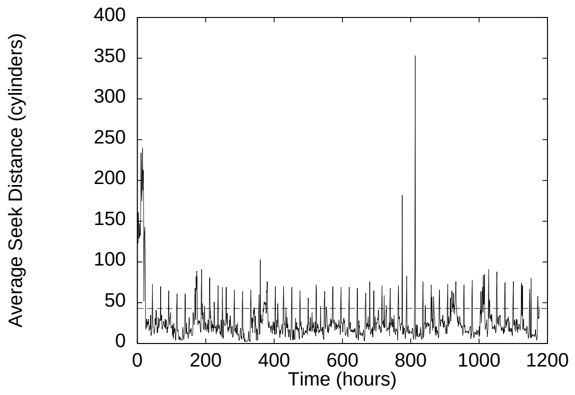
b. Average physical I/O time before shuffling (mean = 15.88 ms).



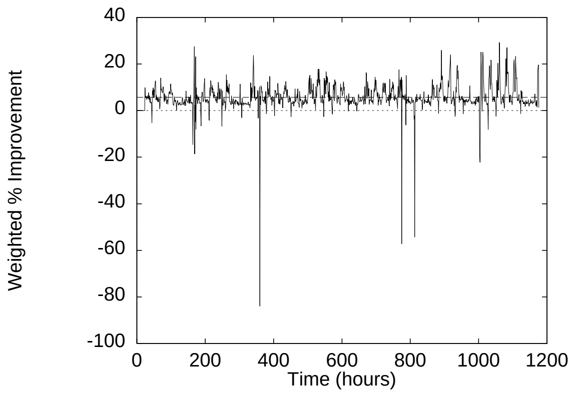
c. Average physical I/O time after shuffling (mean = 14.93 ms).



d. Average seek distance before shuffling (mean = 132 cylinders).

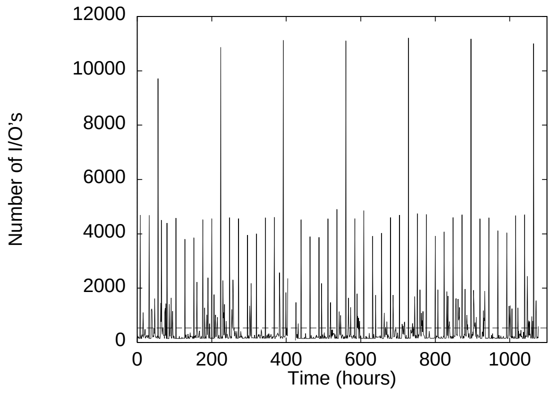


e. Average seek distance after shuffling (mean = 43 cylinders).

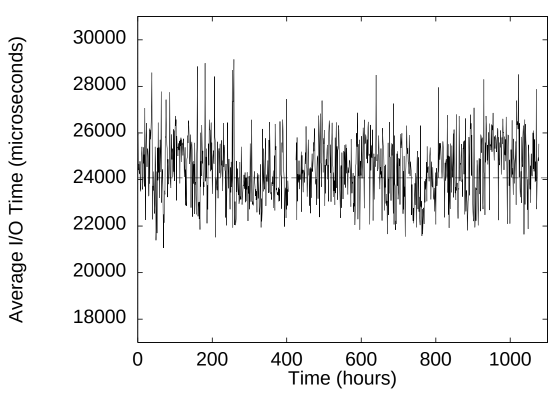


f. Weighted percentage improvement in physical I/O time (mean = 5.7%).

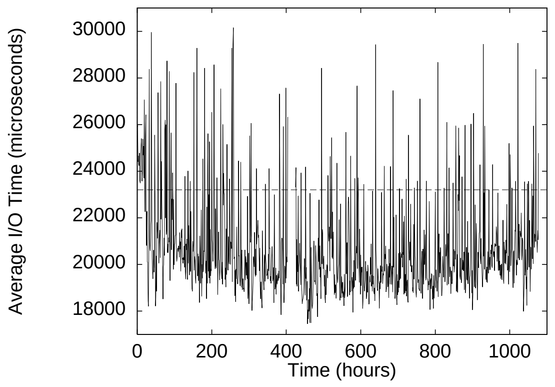
Figure 6. Trace **G** (/ plus swap) from cello before and after shuffling.
Note: the large spikes in the I/O time graphs correspond to periods with low I/O rates.



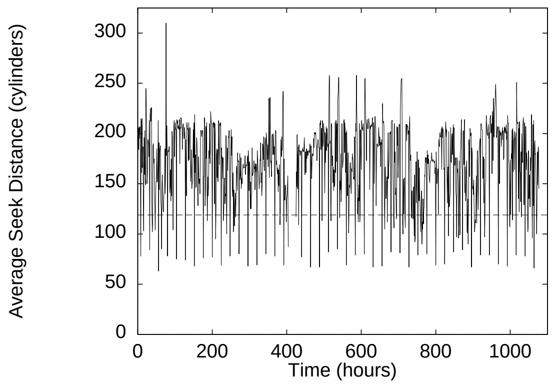
a. Hourly I/O rates (mean = 534/hour).



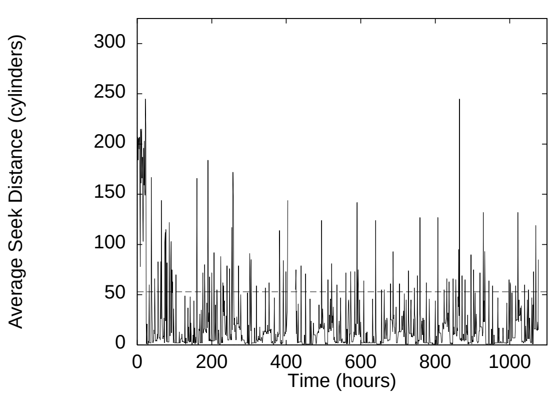
b. Average physical I/O time before shuffling (mean = 24.1 ms).



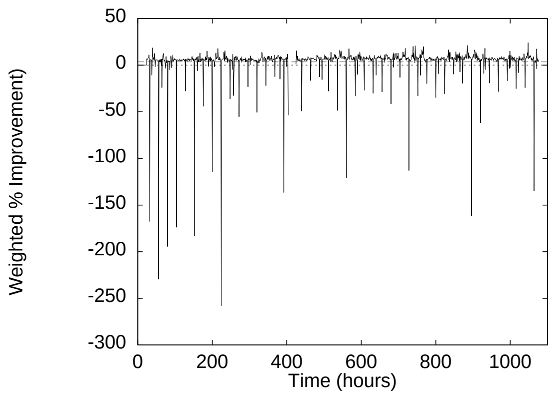
c. Average physical I/O time after shuffling (mean = 23.2 ms).



d. Average seek distance before shuffling (mean = 119 cylinders).



e. Average seek distance after shuffling (mean = 53 cylinders).



f. Weighted percentage improvement in physical I/O time (mean = 3.5%).

Figure 7. Trace **E** (/ plus swap) from hplajw before and after shuffling.
Note: one day's worth of trace data was lost during data collection (around hour 400).

These experiments show that the mean seek distance does not always improve with shuffling (trace **F**, Figure 5), with the result that shuffling can make the performance worse. When things do improve (as in the other two traces), the weighted benefit is quite small—less than 6% in these experiments. Even when the mean seek distance does decrease dramatically (e.g. from 132 to 43 cylinders for trace **G**, Figure 5), the resulting benefit is small: 6.0% of mean physical I/O time, 5.7% weighted improvement.

Why is this?

- Around 50% of the I/Os require no seeks at all, and cylinder shuffling provides no benefit to these I/O requests.
- Data that used to be contiguous before shuffling may now be split into non-contiguous areas—even, in the worst case, on opposite sides of the organ pipe. This can increase seek distances.

  In addition, what used to be a single I/O may now have to be split into two (consider a block that spanned a cylinder boundary before shuffing). The experiment shown in Figure 3 had 9490 split I/Os out of a total of 632709 I/O operations after the first day—i.e. around 1.5% of the I/Os, and a 1.5% reduction in the performance benefit from shuffling.
- As disks get faster, the benefit of reducing the seek distance decreases. For example, on the HP 7935s used on red, the system we first tested, a reduction in average seek distance of 150 to 20 cylinders results in a 52% reduction in seek time (15.8 ms to 7.6 ms), which corresponds to 21% reduction in the time for a typical 8 KB I/O.

  On the more modern HP 97560, the same seek distance reduction causes only a 38% reduction in seek time (8.1 ms to 5.0 ms), and only improves a typical I/O by 15%.

We concluded that the positive case for disk shuffling made in [Vongsathorn90] should be treated with greater caution when applied to modern disks and a wider range of workloads. To understand why, we decided to pursue a more detailed characterization of data shuffling.

We began by looking at the large negative peaks in the mean performance improvement graphs. We surmised that these coincided with the nightly backups, and confirmed this hypothesis by calculating the weighted performance improvements after excluding the backup periods (0600–0900 each day for cello; 0300–0600 for hplajw). The results are as follows:

Table 4. Effects of eliminating backup periods from the analysis.
The data shown are the mean seek distances before and after shuffling, together with the weighted percent improvements ("*benefit*") for physical I/O times.

	— including backup periods—			*——excluding backup periods——*		
Trace	*unshuffled*	*shuffled*	*benefit*	*unshuffled*	*shuffled*	*benefit*
E	119 cyl	53 cyl	3.5%	149 cyl	18 cyl	10.2%
F	96 cyl	100 cyl	–2.7%	102 cyl	49 cyl	–2.4%
G	132 cyl	43 cyl	5.7%	155 cyl	22 cyl	6.8%

These data show that the nightly incremental backup has relatively good locality of reference in the unshuffled case (the unshuffled seek distances are lower with the backup periods included), but poorer locality after shuffling (the shuffled seek distances are higher with the backup periods included). We postulate that the straightforward implementation of shuffling used here destroys the cylinder groups that a 4.2BSD-based file system uses to keep directories and the inodes

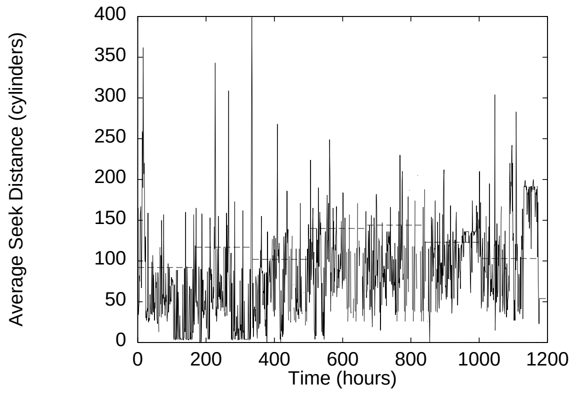


Figure 8. The long-term effects of performing a cylinder reshuffle once after 24 hours. The graph plots the mean seek distance averaged over both hours and weeks. Trace **E**.

(metadata) together. The result is increased disk arm movement when the inodes are being scanned to determine whether a file needs to be dumped. When a full backup is taken (monthly on cello, weekly on hplajw), a significant fraction of the entire file system is read, resulting in even worse locality—as the large negative peaks with this periodicity in Figure 5, Figure 5 and Figure 5 show.

For the remainder of this paper, we present data with the backup periods *included*, since this is corresponds to the case when no additional system configuration information has to be supplied to the shuffling code. We believe that performance improvements of this type (and magnitude) should not require a system administrator to provide configuration data, and, moreover, that operating without the additional information will be the common case: even if hints can be supplied, experience suggests that they will not be.

5.2 Effects of varying the shuffling interval

To determine the long-term effects of drift in the active cylinder set, we ran an experiment that performed a single shuffle once after 24 hours. Figure 8 shows the result: there is only a very gradual degradation in performance over time if no shuffling is done.

This experiment suggested that infrequent shuffling will produce most of the benefit to be obtained. To confirm this, we performed tests with shuffling at intervals of 1 hour, 4 hours, 8 hours, 1 day (24 hours), 4 days, and 7 days. The results are shown in Figure 9.

There are two kinds of behavior shown here. In the traces from cello, the shuffling interval makes little or no difference. In the trace from hplajw (trace E), the longer shuffling intervals perform better, up to around 24 hours (the period of the nightly incremental backup), after which little further benefit is obtained.

On average, the best effect was obtained for shuffling infrequently: between daily and once a week. We had hypothesized that long shuffling intervals would do best, which these data confirm. (The relative benefit from doing infrequent shuffling would be even greater if the costs of doing the rearrangement were to be taken into account.)

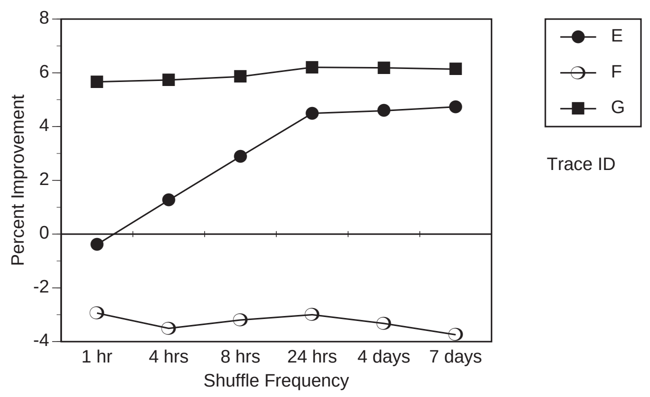


Figure 9. Effects of varying the interval between rearrangements.

5.3 Aging access count data

The simplest way to accumulate access counts is to reset them all to zero at the beginning of each shuffling interval, and only count accesses during the shuffling interval immediately preceding a reshuffle. Given that infrequent shuffling is so successful, we decided to see whether we could reduce short-term fluctuations by merging in some "history" from previous data-gathering periods. This we explored through a family of *age-weighting* algorithms, defined as follows.

aged(α): $C_{new} = \alpha\, C_{recent} + (1\text{-}\alpha)\, C_{old}$

Where: C_{recent} is the access count accumulated for a quantum during the most recent time interval, and C_{old} is the previous value of C_{new}. The case α=1.0 corresponds to considering only the most recent access count data.

We tested the range $0.2 \le \alpha \le 1.0$, but found *no* significant variation in the mean physical I/O times for these values of α.[3]

5.4 Effects of varying the shuffling quantum

We hypothesized that smaller shuffling quanta would be better. The smaller the shuffling quantum (at least down to the file system block size), the easier it would be to concentrate frequently-accessed blocks together. (Consider a cylinder with some blocks frequently accessed, and others not: the average access rate might not be enough to cause the cylinder to be moved to the center of the disk.)

We tested the effects of shuffling units of 8 KB blocks (the I/O unit of our file systems), full tracks, half-cylinders, and cylinders. The results are shown in Figure 9. The general trend is for smaller shuffling quanta to perform best, although trace **F** is a counterexample.

Early on we discovered that it is very important to keep the shuffling quantum a multiple of the file system block size. If it is not, then many I/O requests will be split across independently-migrating units, leading to a great increase in total seek distance. The quanta selected are based on *logical* cylinders or tracks rather than physical ones, since the disk drives rarely provide an

[3] For consistency, all the other simulations in this paper used the aged(0.8) weighting algorithm.

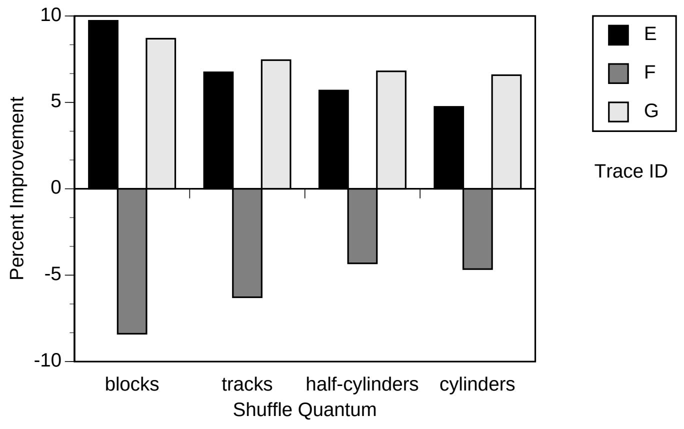


Figure 10. Effects of different sizes of shuffling quanta; shuffling once a week.

integral number of blocks on their physical tracks or cylinders. All the results reported here use shuffling quanta whose size is determined by rounding the physical unit (track or cylinder) to the nearest integral multiple of the file systems' 8 KB block size.

To confirm that the shuffling interval is independent of the shuffling quantum, we calculated the overall performance benefit from all possible combinations of our shuffling intervals and quanta. Figure 11 displays the result. The optimum point occurs at the combination of the best independently-determined values for interval and quantum: i.e. shuffling blocks every 7 days.

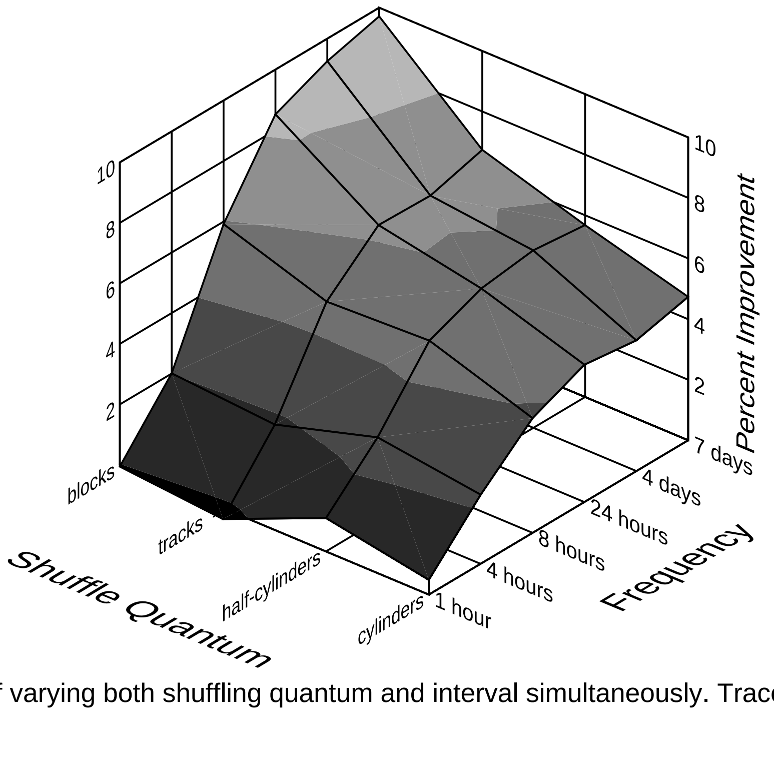
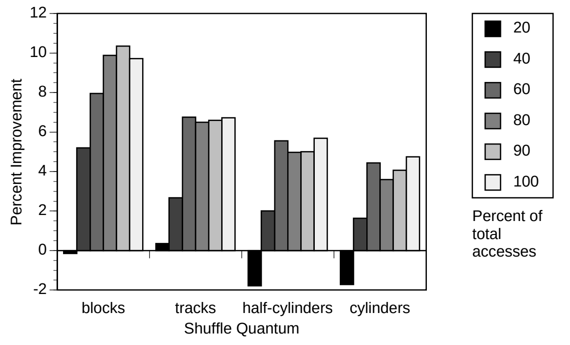


Figure 11. Effects of varying both shuffling quantum and interval simultaneously. Trace **E**.

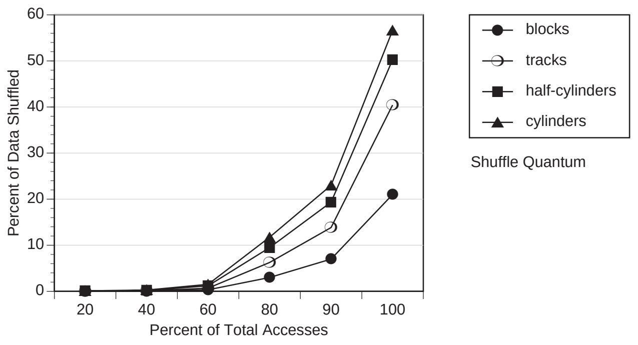
5.5 Economizing on data movement

We hypothesized that much of the benefit of disk shuffling could be obtained from reorganizing only a fraction of the data. Indeed, as the narrowness of the peak in Figure 1b shows, only a small number of cylinders are accessed with any significant frequency. In particular, we observed that with several traces, two thirds of the blocks accessed in the day were only accessed once, and were unlikely to be accessed the next day. By not considering these blocks in our shuffling placement algorithm (or rather, by trying to leave them alone), we could speed up the shuffling process with minimal effect on the performance of the resulting arrangement.

We conducted a series of experiments to determine the effects of artificially limiting the amount of data migration in each rearrangement. Figure 12 shows the effects of sorting the quanta according to access frequency, and limiting shuffling to the most active ones. It shows that the shuffling needs to cover at least 60% of the access counts to be effective, but this can be done by rearranging a small portion of the disk. Indeed, even moving all the quanta accessed by 90% of the traffic can be done by moving only 6–24% of the data—less than half that required to move all the active quanta.



a. Percent improvement achieved by limiting shuffling to the most active quanta.



b. Amount of data that has to be moved after limiting shuffling to the most active quanta

Figure 12. The effects of shuffling only the most active quanta, as determined by the percentile of total accesses that they covered. Trace **E**.

5.6 What are the best dependency algorithms?

Assuming that disk accesses are independent is not a good model of real system behavior. For example, many files are accessed sequentially [Ousterhout85a], and this access pattern is often reflected in the low level disk traffic—despite the effects of file buffer caches. Taking account of these inter-quanta dependencies in access patterns can improve the overall effectiveness of disk shuffling.

With the smallest shuffling quanta (blocks) the dependencies often represent consecutive blocks in a file. With the largest units (cylinders) the dependencies are often less pronounced, but can still occur with large files and groups of related files (e.g. those in a directory).

We knew that the BSD4.2 file system placement algorithms used in HP-UX do a good job of placing blocks of large files and the blocks of files in the same directory, so we were not expecting to see enormous benefits from doing the dependency gathering and analysis. However, we also knew that the straightforward organ-pipe arrangement performed poorly if it led to sequentially-accessed data being split on either side of the disk. When this happened, the disk arm seeked back and forth across the middle of the organ-pipe and performance decreased compared with the unadapted case.

Unfortunately, performing an optimum placement using dependency data is an NP-hard problem [Carson89], which is impractical even for cylinder-granularity shuffling (modern disks have around 1500 cylinders). Some heuristics for assigning near-optimal placements given the transition probabilities between every cylinder pair are described in [Carson89]. The simplest technique (a greedy selection of the next cylinder to place) is easy to apply, but can reach non-global minima. A rather more complex technique (simulated annealing) performs better, but is more computationally expensive—especially as the number of shuffling quanta increases.

We took a slightly different approach to the problem, and chose to exploit our expectation that much traffic is comprised of sequential accesses. To do this, we kept *forward and backward pointers* from each shuffling quantum, together with *link counts* for the number of times these pointers were "traversed". (See Figure 13.) The link count was incremented each time a quantum was preceded or followed by the quantum already recorded in the associated pointer slot (up to a maximum of 255). If a different quantum was encountered than the one already recorded, the link count was decremented; if it reached zero, the pointer's value was replaced with the new quantum number.

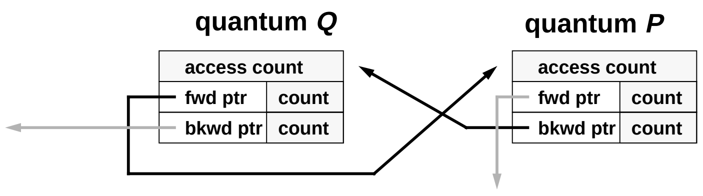


Figure 13. Links and count fields used by the dependency algorithm. Link information for two sequentially-accessed quanta is shown.

With 8 KB blocks and an 1.3 GB disk (e.g. the HP 97560), this kind of dependency data takes 1.6 MB to store. With cylinder sized quanta, it can be represented in only 20 KB.[4]

When the time came to rearrange the disk, the shuffling policy selected the next quantum *Q* to place in decreasing order of access count (i.e. activity). The *link closure L(Q)* was formed by following the forward and backward pointers from *Q*. The idea is that *L(Q)* represents a unit of sequential access—often, although not always, this will correspond to a file, or a portion of a file. The following rules determined whether *P* was included in *L(Q)*:[5]

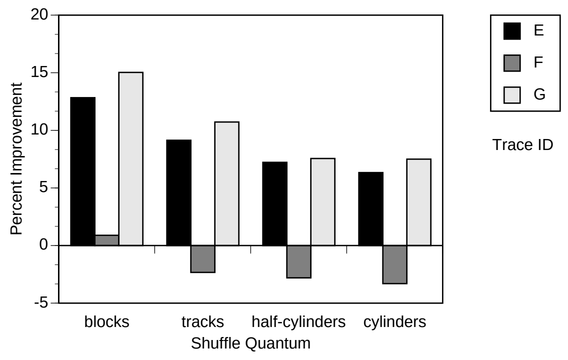
- Quantum *P* must point to quantum *Q*.
- The access count to quantum *P* must be at least half that of quantum *Q*, or *P* and *Q* must already be physically adjacent.
- There is still space for the placement (e.g. the end of the disk has not been reached).
- *P* has not already been placed.

[4] We suspect that the majority of the effect could be achieved by storing dependency data only for the most active quanta, but we did not test this. Figure 12 suggests that data on at least half of the active quanta might need to be kept for this to be effective, but this need only represent 5–10% of the total quanta.

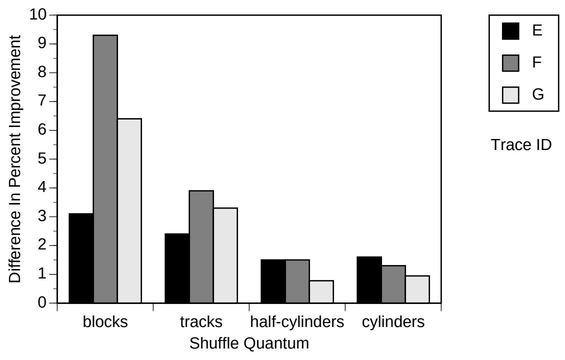
[5] The parameters to this algorithm were determined in a rather haphazard way. The rules shown here are a slight simplification of the set actually used.

All of the quanta in *L(Q)* are then placed contiguously, modulo the file placement interleave factor, if this is not 1:1.

This simple data dependency algorithm did a surprisingly good job of keeping track of dependencies. In one simulation using trace **G**, a chain of 1863 8 KB blocks (14.5 MB of data) was detected and placed as a unit. Figure 14 displays the overall improvements obtained; as expected, the benefits are largest for the smaller shuffling quanta. With the dependency algorithm in use, even trace **F** exhibits a net benefit from shuffling.



a. Weighted percent improvement in physical I/O time as a function of shuffling quantum size with dependency algorithm in use.



b. *Additional* weighted percent improvement in physical I/O time resulting from the dependency algorithm.

Figure 14. The effects of the dependency-tracking algorithm.

6 Assessment and conclusions

Dynamic disk shuffling *can* improve performance, but the benefits are not as marked as might have been imagined from the significant reduction in seek distances that are obtainable. The technique is fairly robust to variations in shuffling frequency and shuffling quantum, although it exhibits moderate sensitivity to the workload, and is less effective on modern disks because their long-distance seeks are not enormously more costly than short ones.

The best disk shuffling results come from combining relatively infrequent shuffling (once a week) with small (block or track-sized) quanta and data dependency tracking. The data aging algorithm chosen appears to be immaterial as a result of the relatively low rate of change of the active data. All these algorithms are readily performable inside a disk driver or disk drive with a modicum of temporary storage.

The net performance gains seen here are only moderate: 2–15%. This is doubtless attributable to the initially good performance of the baseline 4.2BSD file system that we used for our measurements. Despite its design emphasis on single-user performance of very large file transfers, it sets a high standard against which to measure file system improvements. We conjecture that greater benefits could be obtained with a file system that did a simpler block layout policy (e.g. MS–DOS), but we have not yet had an opportunity to test this hypothesis.

Acknowledgments

Raymie Stata first brought the work of Scott Carson and Paul Vongsathorn to our attention. In turn, they were kind enough to make available to us some of their preliminary papers and technical reports. Richard Golding, David Jacobson, Carl Staelin and Marjan Wilkes made helpful suggestions about improving the presentation of this material. This work was carried out as part of the DataMesh research project at Hewlett-Packard Laboratories.

References

[Baker91] Mary Baker, John Hartman, Mike Kupfer, Ken Shirriff, and John K. Ousterhout. *Measurements of a distributed file system.* Computer Science Division., Department of Electrical Engineering and Computer Science, University of California at Berkeley. Draft of paper submitted to SOSP. *(To be replaced by the real SOSP paper reference when that is available.)*

[Braunstein89] Andrew S. Braunstein. *File buffer cache design in computers with large physical memories.* Masters thesis. Massachusetts Institute of Technology, Cambridge, MA, February 1989. Published as Hewlett-Packard Laboratories Technical Report HPL–DSD–89–6.

[Carson89] Scott D. Carson and Paul F. Reynolds Jr. *Adaptive disk reorganization.* Technical report UMIACS–TR–89–4 and CS–TR–2178. Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, and Department of Computer Science, University of Virginia, January 1989.

[Clark66] W. A. Clark. “The functional structure of OS / 360: Part III, data management.” *IBM Systems Journal* **5**(1):30–51, 1966.

[Clegg86] Frederick W. Clegg, Gary Shiu-Fan Ho, Steven R. Kusmer, and John R. Sontag. “The HP-UX operating system on HP Precision Architecture computers.” *Hewlett-Packard Journal* **37**(12):4–22, December 1986.

[Coffman82] E. G. Coffman Jr and Micha Hofri. “On the expected performance of scanning disks.” *SIAM Journal on Computing* **11**(1):60–70, February 1982.

[Geist87] Robert Geist and Stephen Daniel. “A continuum of disk scheduling algorithms.” *ACM Transactions on Computer Systems* **5**(1):77–92, February 1987.

[HPdisks89a] Hewlett-Packard Company. *Disk product specifications and site environmental requirements*, part number 5955–3456, 15th edition, November 1989.

[Kondoff88] Alan Kondoff. “The MPE XL data management system exploiting the HP Precision Architecture for HP’s next generation commercial computer system.” *Digest of papers, Spring COMPCON’88* (San Francisco, CA), pages 152–155. IEEE, 29 February–4 March 1988.

[Majumdar84] Shikharesh Majumdar. *Locality and file referencing behaviour: principles and applications.* MSc thesis published as technical report 84–14. Department of Computer Science, University of Saskatchewan, Saskatoon, August 1984.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. “A fast file system for UNIX.” *ACM Transactions on Computer Systems* **2**(3):181–97, August 1984.

[McVoy91] L. W. McVoy and S. R. Kleiman. “Extent-like performance from a UNIX file system.” *Proceedings of Winter 1991 USENIX* (Dallas, TX), pages 33–43, 21–25 January 1991.

[Oney75] W. Oney. “Queueing analysis of the scan policy for moving-head disks.” *Journal of the ACM* **22**(3):397–412, July 1975.

[Ousterhout85a] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. “A trace-driven analysis of the UNIX 4.2 BSD file system.” *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Washington). Published as *Operating Systems Review* **19**(5):15–24, December 1985.

[Ousterhout89] John Ousterhout and Fred Douglis. “Beating the I/O bottleneck: a case for log-structured file systems.” *Operating Systems Review* **23**(1):11–27, January 1989.

[Ritchie74] D. M. Ritchie and K. Thompson. “The UNIX time-sharing system.” *Communications of the ACM* **17**(7):365–75, July 1974.

[Staelin91] Carl Staelin and Hector Garcia-Molina. “Smart filesystems.” *Proceedings of Winter 1991 USENIX* (Dallas, TX), pages 45–51, 21–25 January 1991.

[Thompson78] Ken Thompson. “UNIX implementation.” *The Bell Systems Technical Journal* **57**(6):1931–46, July-August 1978.

[Vongsathorn90] Paul Vongsathorn and Scott D. Carson. “A system for adaptive disk rearrangement.” *Software—Practice and Experience* **20**(3):225–42, March 1990.

[Wong83] C. K. Wong. *Algorithmic studies in mass storage systems*. Computer Science Press, 11 Taft Court, Rockville, MD 20850, 1983.