# Loge: A Self-Organizing Disk Controller

Robert M. English, Alexander A. Stepanov
Software and Systems Laboratory
HPL-91-179
December, 1991

disk, controller,
intelligent controller

While the task of organizing data on the disk has traditionally been performed by the file system, the disk controller is in many respects better suited to the task. In this paper, we describe Loge, a disk controller that uses internal indirection, accurate physical information, and reliable metadata storage to improve I/O performance. Our simulations show that Loge improves overall disk performance, doubles write performance, and can, in some cases, improve read performance. The Loge disk controller operates through standard device interfaces, enabling it to be used on standard systems without software modification.

Internal Accession Date Only

# Loge: a self-organizing disk controller

*Robert M. English & Alexander A. Stepanov*
*Hewlett-Packard Laboratories*

## Abstract

While the task of organizing data on the disk has traditionally been performed by the file system, the disk controller is in many respects better suited to the task. In this paper, we describe Loge, a disk controller that uses internal indirection, accurate physical information, and reliable metadata storage to improve I/O performance. Our simulations show that Loge improves overall disk performance, doubles write performance, and can, in some cases, improve read performance. The Loge disk controller operates through standard device interfaces, enabling it to be used on standard systems without software modification.

*...only craft and cunning will serve,*
*such as Loge artfully provides.*
*— Richard Wagner, Das Rheingold, Scene II*

## 1 Introduction

Loge[1] is a disk controller that organizes data based on the I/O stream, rather than on interpretations of file system structure. Unlike a conventional design, where data location is a static decision made when blocks are allocated, Loge places blocks dynamically, choosing an optimal location for each write at the time the data is written to the disk. Write performance on Loge is largely independent of the I/O stream, and in many cases approaches the sequential throughput of the device. Even on a highly fragmented, nearly full disk, Loge can achieve half of the sequential throughput of the device.

Loge is a write-optimized design—several studies, most recently [Baker91a] have shown that the proportion of writes in the I/O stream is increasing over time—but read performance does not necessarily suffer. Files tend to be read in the same patterns that they are written, so that a device which performs well on writes will often perform well on reads [Ousterhout89]. Even when this is not true (for example, when a file is generated slowly but read quickly), the data structures in Loge allow the device to reorganize data and improve performance, simply by copying files to a better location. Our simulations show that, at least against some traces, Loge can outperform a standard disk on reads by about ten per cent.

Loge combines a number of techniques. It introduces a layer of indirection within the controller to allow data relocation on every write. It uses reverse indexes stored in the data blocks to update mappings reliably and efficiently. It uses time stamps and shadow pages to allow atomic updates. None of these techniques is particularly new, but only recently has their combination in a disk controller become economical. For example, Loge uses single-level, main memory data structures to achieve good read and write performance

1. Loge (pronounced loh-ghee), is the Germanic god of fire.

by avoiding additional disk accesses. This approach is only feasible because of the decrease in RAM prices with respect to disk prices.

The rest of this paper describes these results in greater detail, as well as giving a full description of the Loge design. It also describes several advantage of intelligent storage devices over traditional file systems as storage managers, and some effective ways to combine the two approaches.

## 2 Related work

The concept of an intelligent controller is not new, but we cannot find an example that plays as aggressive a role as we envision. Closest, perhaps, are dedicated file servers such as those sold by Auspex [Nelson91], but such systems have high-level interfaces and are typically much more complicated and expensive than a simple peripheral. [Menon89] proposes using similar techniques for parity updates in disk arrays, but stops short of advocating them for single disks.

Much has been written on the issue of disk data organization, from file system formats that optimize sequential performance—either by controlling individual block placement [McKusick84, McVoy91] or by allocating files in extents [Peacock88]—to theoretical studies on minimizing head movement [Wong83]. These differ from Loge primarily in that they view data location on disk as static, rather than dynamic. Even with adaptive techniques such as the cylinder and block shuffling [Vongsathorn90, Musser91, Ruemmler91, and Staelin91], the time scale for reorganization is much longer than a single transaction. In all of these strategies, the high overhead for relocating data limits their effectiveness and prevents them from responding to transient disk behavior.

In some respects, this work is similar to the Log-based file system (*LFS*) work [Rosenblum91]. Like LFS, Loge emphasizes write performance and uses the order in which data is written to determine placement. Unlike LFS, Loge does so in a single-level structure without modifying file system software, an approach we find at once simpler and more powerful. LFS-style write policies, for example, can be implemented in Loge, and the result should perform at least as well as LFS.

## 3 Terminology

A few terms will be useful for the following discussion. A *block address* is the logical tag associated by a host with a fixed size array of data (e.g., an 8KB file system block). A *segment address* is the physical address of a location on disk capable of storing a block. *Block* and *segment* refer to the combination of address and either data or storage, respectively. Disk head position is described by a segment address.

When we speak of the *distance* from segment A to segment B, we refer to the amount of time it takes to move the disk head from A to B. Segment B is close to segment A if the time to go from A to B is small. It should be noted, however, that because disks only spin in one direction, the distance function is not symmetric. Since there may be several segments equidistant from A, there will, in general, be no single closest segment, but it will sometimes be convenient to refer to a *closest segment*, in which case any segment with the smallest distance will suffice.

Lastly, we borrow a measure of write performance from [Rosenblum91], the *write cost*, defined as the ratio between the total time spent in an I/O transaction and the time spent actually transferring data from the medium. Unlike [Rosenblum91], however, we base our write cost on the raw transfer rate of the medium, rather than the sequential throughput of the disk for large transfers, since the latter includes a number of seeks and head switches, during which the disk is not transferring data.

## 4 Loge data structures and operation

Loge contains only two primary data structures: an indirection table and a bitmap of available blocks. Though some of the placement and reorganization heuristics described later require additional structures, none of them are essential for normal operation. The indirection table stores the addresses of the physical

*segments* indexed by logical block address. The *free map* is used to find potential storage locations. Both of these structures are kept in dedicated RAM on the controller.

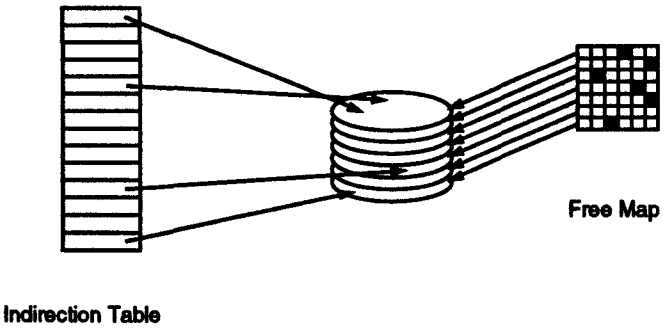



Figure 4.0: Loge data structures

In response to a read request, the controller looks up the segment address in the indirection table and starts the necessary transfer. In response to a write request, the device selects a segment from the free map that can be reached quickly from the current head position, writes the block to it, frees the old segment, and installs the new segment address in the indirection table.

## 4.1 Recovery

To ensure reliable operation, Loge includes the block address and a time stamp in the blocks being written. This information is written into the sector headers of the disk along with normal ECC information, and is invisible to the host computer. The combination of time stamps and inverted addresses allows Loge to recover from any non-media failure in the time it takes to scan the disk, about 11 minutes on a 1.3GB disk capable of reading 2MB/s. As transfer rates increase and form factors decrease, this number will shrink.

More aggressive Loge implementations could use small amounts of non-volatile storage or even full battery backup for the indirection table to reduce the recovery time, but the inverted indices provide excellent protection against catastrophic failures and should probably be maintained even so.

## 4.2 Costs

In order for Loge to be practical, it must not significantly increase the cost of a disk. Battery-backed RAM has exceptional performance, and any disk-based scheme must maintain the current cost advantage of disks over RAM in order to be feasible. Fortunately, the dramatic decreases in RAM costs help Loge almost as much as they help solid state systems.

For each block accessible from the host, Loge requires about 3 bytes of storage. A 1GB disk contains 2 million 512-byte sectors, so each entry in the indirection table requires at most 22 bits of address, with one additional bit for the free map. Even allowing for data structure overhead, this should require only 4 bytes per block, which for a 4KB block size, amounts to 0.1% of the secondary storage space, or 1MB for a 1GB disk. Assuming that RAM is 10–20 times more expensive per bit than disk storage [Anderson91], this amounts to only 1–2% of the cost of the disk.

## 4.3 Generic segment allocation

Segment allocation in Loge is a local policy, rather than a global one. Block assignment is based primarily on the relationship between the current head position and the available space, not on semantic interpretations of the block address itself. Block address need not be totally ignored, but local properties dominate. All of these strategies can be described as "greedy," in that they write to the closest available block. and all have a common structure. The free map, in particular, is organized to facilitate their searches.

A typical heuristic begins by calculating the time at which the disk will be positioned to write, and then determines the cylinder range which can be reached in that time. The heuristics then search the segments in this range, and return the first free segment that they find. The heuristics differ only in their search orders and in how they define an available segment.

To make this search more efficient, the free map is organized as a collection of columns, with each column corresponding to the set of segments on the disk at the same rotational position on a cylinder. Each column is represented as a bit vector, with each bit position corresponding to a disk surface. Heuristics can thus scan the disk column by column, rather than segment by segment.

# 5 Performance analysis

## 5.1 Read performance

In general, Loge read performance is equivalent to that of a regular device with the same layout. Since the layout of a Loge disk is determined by the I/O trace, however, it depends strongly on the application. We will not, therefore, attempt to characterize read behavior with great precision, but limit ourselves to qualitative statements and general observations about the device structure.

First, files that are written sequentially tend to be read sequentially, while files written randomly tend to be read randomly. One would therefore expect that a device which performs well on writes should perform reasonably on reads. In particular, we can say that a system capable of writing a file in a particular time is also capable of reading it in the same order in the same time, and that that places a lower bound on read performance. If the file is generated slowly, this limit may not be interesting, but in many applications, files are written as fast as they are read.

Second, reads and writes do not interfere on Loge to the same extent as they do on standard drives. On a standard drive with a standard file system, the write location is independent of any knowledge of the current head position. If a drive is used for reads and writes simultaneously, the read commands and the write commands interfere with each other, causing excessive head motion and poor performance. On Loge, writes are scheduled close to the current head position, minimizing read/write interference.

Last, while the Loge design does not specifically target read performance, the Loge data structures support aggressive data movement at the device level, allowing the types of block and cylinder shuffling described in [Staelin91, Vongsathorn90, Musser91, and Ruemmler91]. These report performance improvements of between ten and thirty per cent over standard file system organizations and are well-suited to a Loge-style device. Since disks are idle most of the time, there will normally be time to fit data movement into the gaps, and to schedule it so that it does not interfere with normal operation, even when the device is relatively busy. Whereas host-initiated shuffling techniques are normally limited to off-hours operation, Loge can shuffle data constantly.

## 5.2 Write performance

To develop a feel for Loge write performance, we will discuss write performance under three sets of conditions: highly-ordered, random, and badly fragmented. For the highly-organized case, we consider the period either immediately after initialization or after an idle period that has allowed Loge to rearrange itself into an optimal write configuration. In the random case, we will assume that the disk has been entirely randomized, and that available segments are scattered uniformly across the disk. In the pathological case, we will construct a situation where the disk organization prevents efficient reads and writes. The analysis presented here is not intended to be exhaustive. A real disk system contains a large number of variables that we do not address. Rather, the intent is to provide some intuition into the behavior of these types of devices and motivate the heuristic and simulation studies that follow.

To analyze these situations, we will use an idealized disk that spins at 3600RPM, has ten platters (with 19 usable surfaces), 2000 cylinders, and tracks 32KB long. The seek profile for this disk will be based on the HP97560 disk. This idealized device has a maximum sequential throughput of 1.6MB (200 8KB blocks) per second, and a data capacity of 1.25GB. The write cost for long sequential transfers is 1.25 (20% of the time

is spent seeking or shifting between tracks). For simplicity, we will consider only Loge devices with a block size of 8KB. There is an important relationship between block size, seek time, and performance which will become clear in the following discussions, but performing the calculations for other block sizes is straightforward.

### *5.2.1 Well-ordered performance*

With an 8KB block size and 32KB tracks, Loge has four segments per track, and reaches a segment every 4.16ms. We call this time one *rotational period* and say that the disk has four *rotational positions*. In one rotational period, the head can seek five cylinders in either direction. In this highly-ordered case, we assume that every fifth cylinder has exactly one empty track.

If a write request comes in at the end of a random read, the controller is always able to find a free segment after skipping one rotational position, for a total write time (seek + rotational latency + data transfer) of 8.3ms, and a write cost of 2. If a write request arrives at a random time, we must add an additional one-half rotational period for a total write time of 10.4ms and a write cost of 2.5. If a write occurs immediately after another write, then the head is already in position for the next transfer, no seek is required, the write time is 4.2ms, and the write cost is 1. For 32KB transfers arriving randomly, the total write time is 23ms, for a write cost of 1.38 or 10% less than sequential disk bandwidth. For long transfers, the controller must initiate a seek of 4.2ms after every 32KB, leading to a transfer rate of 1.6MB, the full, large-transfer sequential bandwidth of the disk.

What must be emphasized here is that the total amount of free space reserved is only one track every five cylinders, or a little more than 1% of the disk. While this level of performance could not be sustained against a random I/O stream (which causes the free segments to fragment and quickly degrades into the next case), it compares favorably with that of extent-based and conventional file systems running against nearly empty disks, and shows that the amount of order that must be maintained to achieve good performance is quite small. Restoring a random disk to this state, for example, takes less then 15 seconds for the disk under discussion

### *5.2.2 Performance against a randomized disk*

Suppose that, instead of a highly-ordered disk, we assign blocks and free segments randomly across the disk. At any moment, the probability that we will be able to write to the next segment that the head will reach is equal to the percentage of free segments on the disk. If the next segment is allocated, then we have at least one rotational period in which to search for a free segment. In that time, the head can seek five cylinders in either direction for a total of 11 cylinders which, with 19 segments per cylinder, gives us 209 segments to choose from. A simple calculation shows that with 1% free, there is an 88% chance of finding a free segment within that time. With 2% free, the probability rises to 98.5%. As the time available to search rises, the number of reachable segments rises dramatically, making it virtually certain that a free segment will be found in the next position. Similar calculations show that, on a long write, the expected value for the time spent seeking or waiting for the spindle is equal to 1.11 times the data transfer time for a 99% full disk and is equal to the transfer time for a 98% full disk, yielding write costs of 2.1 and 2, respectively. For single writes with random arrival, we will, on average, have to wait an additional one-half rotational period. Since the extra time can be spent seeking, the probability of finding a free segment in the second location is approximately one in both cases, and the total write cost is 2.5, twice that of a sequential transfer.

For large writes and more sparsely populated disks, we can take this approach a bit further and look for sequential free segments. With 10% free space, have an 88% chance of finding a pair of sequential blocks within five cylinders. If we write to these segments rather than to isolated segments, our write cost for large transfers drops from 1.9 to 1.5. For a disk with 35% free space, the probability of finding a free track (four consecutive free segments) within five cylinders is 95%, and even on a randomized disk, throughput nears full sequential bandwidth. It should be noted, however, that this performance cannot be sustained indefinitely for a random write stream since sequential free areas will be consumed faster than they will be produced.

### 5.2.3 Performance after fragmentation

All of the calculations for fragmented disks assume that writing does not destroy the random distribution of free segments. Clearly, if segments are allocated within a narrow range of cylinders and freed across the entire disk, the number of available segments in that cylinder range will drop and performance in that range will degrade. The number of free segments in other areas of the disk, on the other hand, will increase, and performance there will improve.

In extreme cases, the head can become trapped in nearly full areas, and performance can remain poor for extended periods of time. It is possible, for example, for a disk that has 35% free blocks globally to have localities with 2% free space. This is enough to increase the write cost from about 1.25 to 2, a degradation of 60%. The main goal of search policy design is to avoid such situations, either by preventing exhausted regions from occurring or by moving the head away from them if they do.

There are two types of behavior worth particular note.

Consider first the case of a disk used only for writing. Once an area of the disk has been depleted, the head will move to one side of the depleted region and begin to write there. Since the depleted region acts as a barrier to head movement, once the head begins to travel in a certain direction, it will tend to continue in that direction until it reaches another depleted area, then it will find another relatively empty region and repeat the process. If the head remains close to a depleted region, the number of available segments is less than it would be on a randomized disk by up to significant factor. Some of the search heuristics described below exhibit this type of behavior and can perform poorly as a result.

A second cause of poor performance is bias. If the head spends most of its time over a subset of the disk, the number of free segments in that subset will decrease. Since those regions are, by definition, the regions where the head spends most of its time, the overall performance of the device will suffer. Bias is one of the most difficult problems to avoid, since it can result from many sources. All of the search heuristics described below, for example, suffer from it to some extent; most read traces contain some amount of bias; and most of the cylinder and block shuffling techniques described in the literature increase bias in order to improve read performance.

### 5.2.4 Trade-offs between reads and writes

The write costs described above for single transactions assume that the disk is idle when a request arrives, but this need not be the case. If the controller makes the assumption that it should always be prepared to write as quickly as possible, it can reduce the write cost for a single transaction to about 1.5 by making sure that the head is always in transit to the next free segment (a technique similar to [King90]. Following the same reasoning that allowed writes to every other rotational period in the analysis above, we find that the disk head can be ready to write within one rotational time period. This means that, for a random arrival, the average seek and rotational latency amounts to half of a rotational period. The total transaction thus requires only 1.5 times the media transfer time, a write cost of only 20% more than that of sequential transfers.

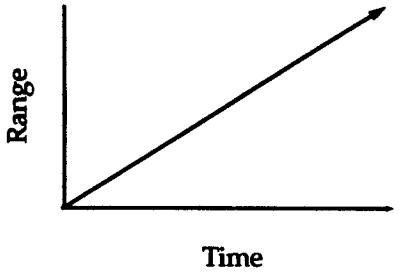
This technique hurts read performance slightly in some cases and helps it in others. If a read request occurs in the direction that the head is moving, read performance benefits slightly because the head is already in flight. If not, read performance suffers due to the necessity of stopping the head and accelerating in it the opposite direction.

The costs and benefits of these events will depend on the characteristics of the disk arm mechanism, and are difficult to estimate for an arbitrary disk, except to say that neither is larger than the cost of a minimum seek. What is clear, however, is that the expected cost for reads is minimized if the head is always travelling in the direction where reads are likely to occur, towards the center of the disk. Unfortunately, such a policy will also deplete the center of the disk, leading to poor write performance.

## 6 Search heuristics

These search heuristics all follow the basic framework outlined above. They generate a list of columns to be searched, and then search the list in order to determine which of several available segments should be selected. The differences between the heuristics lie entirely in their search orders and in their definitions of available segments.

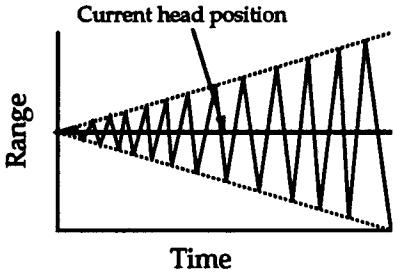
### 6.1 Linear scan



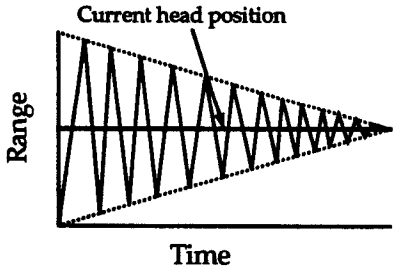

In *linear scan* (*LS*) the cylinder range is searched in ascending cylinder order until the first free segment is found. This heuristic is biased toward the beginning of the disk, which results in poor performance under certain circumstances. Consider, for example, the behavior of *LS* when confronted by a long series of writes. Any time that the heuristic can find an efficient way of moving the head downward, it will do so. The beginning of the disk becomes depleted, but the heuristic attempts to move the head back into this depleted region at every opportunity. The end of the disk, on the other hand, is nearly empty, but the head never makes it into that region, so the space is simply wasted. However, if the number of reads in the trace is high enough, this heuristic can perform well, because the reads will keep the head from staying near the beginning of the disk.

### 6.2 Closest reachable

In *closest reachable* (*CR*) the cylinder range is searched starting from the current head position, alternating above and below in order of increasing distance. This heuristic moves the head very slowly and can spend a great deal of time in depleted areas of the disk. On long sequences of writes, the head quickly becomes trapped behind small depleted areas and then stays close to them, reducing the pool of available segments as described above. This heuristic performs poorly on all traces, and is included primarily for illustrative purposes.



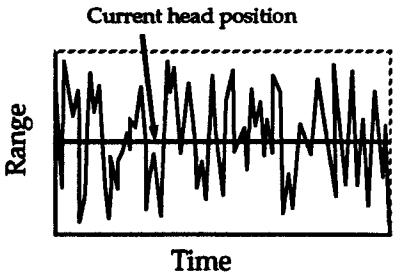

### 6.3 Furthest reachable



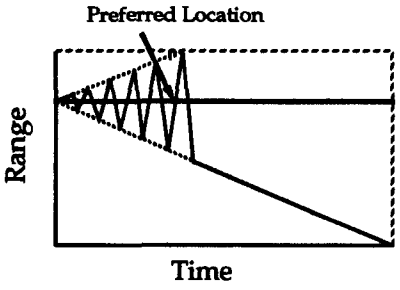

*Furthest reachable* (*FR*) is the opposite of *CR*. Rather than searching from the head position in order of increasing seek distance, it searches in order of decreasing seek distance. As a result, the head responds to crowded areas of the disk by moving large distances, enabling it to find relatively uncrowded areas where it can write with greater efficiency. *FR* exhibits a periodic bias. After long sequences of writes, the free segment density of the disk oscillates with a period equal to the length of the first long seek of the device. This shows up in generated traces as anomalous behavior under write-dominated loads, where *FR* performs better with a light scattering of reads than with only writes.

### 6.4 Random

*Random* (*R*) is a compromise intended to move the head quickly while avoiding the periodic behavior exhibited by *FR* during long sequences of writes: Since the search order is random, the free segment distribution remains mostly flat. Its performance on traces is slightly worse than *FR*, however, since it does not escape from crowded areas as rapidly. Under extended write loads, *random* exhibits a slight bias toward the middle of the disk, simply because it needs to allocate from the middle in order to reach either end.



### 6.5 Preferred location




*Preferred location* (*PL*) is an attempt to use the global information known to the file system to inform the local segment selection process. Each block is assigned a preferred segment address (derived from a sequential block:segment mapping with regularly placed free tracks) and the search heuristic attempts to place the block as close to that location as possible. If the disk is in a well-ordered state, and the trace is largely sequential, then this heuristic performs very well. The head travels toward the preferred cylinder writing sequentially. Once there, it finds abundant free segments in which to write. If the disk is in a highly disordered state, the head travels toward the preferred location, but once there, cannot find enough room to write, and from then on behaves similarly to *CR*.

### 6.6 Sequential groupings

The *sequential grouping* (*SG*) heuristic searches for sequences of free segments to write to (see the discussion of fragmented performance above). In addition to improving read and write performance by improving sequentiality, it also increases the speed with which the head finds empty areas. Since relatively empty areas are far more likely to have sequential free space than relatively full ones, searching for sequential groupings is an effective method for finding these areas.

*SG* is different from the previous heuristics in that it does not affect the search order, but the definition of an available segment. *SG* is thus independent of the other heuristics in this list, and can be combined with any of them.

## 7 Simulations

Our simulations had three goals. First, we wanted to test the behavior of Loge on real traces, to determine whether Loge would benefit real systems. Second, we wanted to probe the behavior of the different heuristics to determine their properties. While some properties are easy to infer from the description of the heuristic, others are much easier to discover by running the heuristic under controlled conditions and observing the result. In addition, we wanted to observe the effects of varying initial conditions. Some heuristics that perform quite well under certain conditions break down under others, and we needed to explore these phenomena. Third, we wanted to gain insight into the steady state behavior of the heuristics. We wanted to know what the disk would look like after running the algorithm over a period of time, so that we could compare steady state conditions with the conditions needed to run the algorithms well.

The simulator itself models the data flow through the various steps in the I/O path, with the level of detail at each step under the programmer's control. Using this model, it is straightforward to set up a highly detailed and accurate simulation of the entire I/O path, though reducing the level of detail in order to improve performance requires care to avoid introducing inaccuracies, and designing I/O paths without bottlenecks can be a tricky business in itself. The current simulator supports only a single outstanding I/O, and does not perform disk arm scheduling or command queueing, both of which would be valuable in a Loge controller. Since host-based scheduling is ineffective on Loge and the simulator cannot perform read scheduling within the device, all traces were scheduled first-come, first-served.

Interpreting the real traces is made difficult by the fact that there is no one representative workload, and complicated further by the inherent problems of running simulations against recorded I/O traces. We collected a number of traces from local time sharing systems and workstations. The workloads on these machines are dominated by text processing, electronic mail, and cpu-intensive simulations, so the applicability of these traces is limited to similar environments. In addition, since all we have are the recorded arrival times of individual requests, we cannot determine which reads or writes were synchronous (and thus affected user performance); neither can we determine how changing the response

time of a single request will affect the arrival rates of later requests. We can determine how fast a device runs a specific trace, but we cannot say with certainty how that will affect application performance.

While we do not present the results here, our preliminary simulations showed that sequential grouping heuristic improved performance whenever there was a difference. In some of the simulation runs, little fragmentation occurred, and there was no measurable difference between the two. When tested against a randomized initial disk, however, sequential grouping led to markedly better performance, suggesting that it significantly reduces the effects of fragmentation. The *Red* performance numbers, in particular, were the result of applying sequential grouping to a slightly modified device

## 7.1 Results

### *7.1.1 Cello: /usr/spool/news*

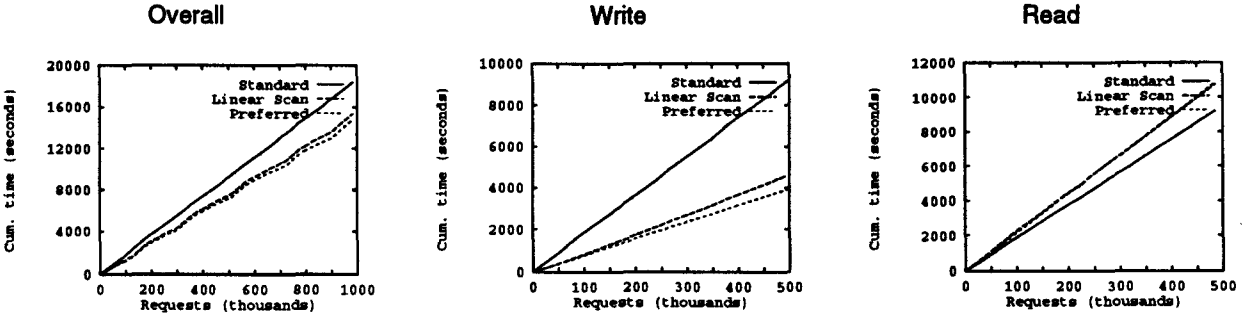



Figure 7.1.1: Performance of */usr/spool/news*.

In figure 7.2.1, we show plots of the cumulative response time for a trace of one million I/O transactions (half reads and half writes) against a standard disk and a Loge disk with two different heuristics. Write performance on the Loge disk is, on the average 130% better than that on the standard disk. Read performance drops by 18%, and overall performance improves by 25%. The *LS* heuristic performs about the same as the *PL* heuristic for reads, but *PL* outperforms *LS* by 14% on writes.

The device being simulated in these tests is derived from an actual HP97560 drive, which is similar to the canonical drive described above except that each track is 36KB and the disk spins at 4002 RPM. In order to have the block size divide evenly into the track size, we used 4KB blocks instead of 8KB blocks. These changes can have a dramatic impact on performance. With a block size of 8KB, the minimum seek time is slightly less than the time to transfer a block, which allowed the device to transfer at half the peak bandwidth of the disk even when the disk was nearly full and sequences of free blocks were rare. With a block size of 4KB, each seek lasts two rotational periods, and throughput drops to a third of peak performance.

If that were the only performance problem and everything else were equal, we would expect to see the standard disk outperform Loge on reads by about 3.2ms on blocks written after the disk became fragmented, but there is another factor. Since the tracks on this drive contain 10% more data and the disk spins 10% faster, the time in each rotational period is significantly shorter. Two rotational periods are only 0.09ms longer than the settle time of the disk, not long enough for the disk arm to seek to a different track. The head can switch from one surface to another, but cannot change cylinders. This raises the average rotational miss from 3.2ms to 4.8ms, matching the observed read degradation.

It is also interesting to compare the write performance of different heuristics. *Linear scan* always seeks toward the beginning of the disk, while *Preferred location* attempts to store blocks near their "natural" positions. Initially, the two algorithms have similar performance, but over time, *linear scan* begins to run out of free segments near the beginning of the disk and its performance suffers. Since *preferred location* sends

different blocks to different areas of the disk, this effect is not pronounced and performance stabilizes. In figure 7.2.2, average response time in milliseconds is plotted over time. The upper curve is *LS*, the lower *PL*.

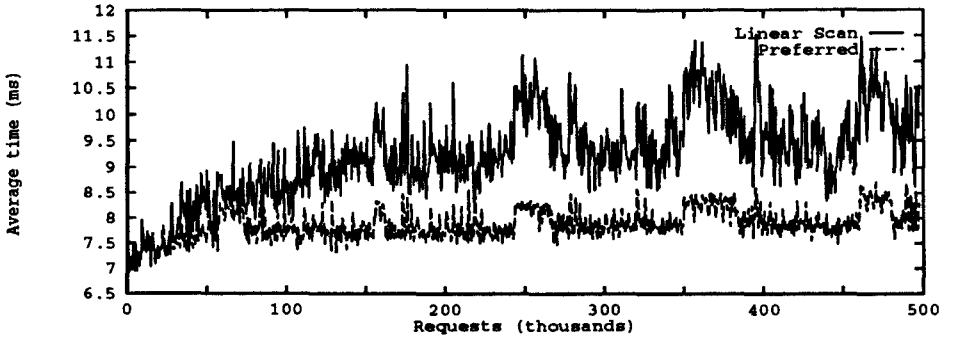



Figure 7.1.1: Linear scan vs. preferred location

### *7.1.2 Cello — root + swap*

We chose to examine */usr/spool/news* for the simple reason that it had the highest access and modification rates on our system. The same results, however, were demonstrated on other file systems:

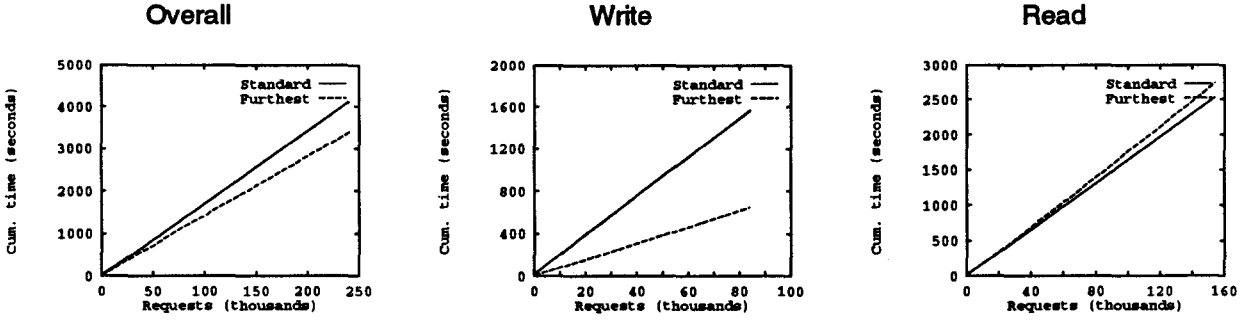



Figure 7.1.2: Performance on root and swap.

In this case, running against the root and swap file systems on cello, Loge improves write performance advantage by 140%, degrades read performance by 8%, and improves overall performance by 20%, against a trace with a read:write ratio of 2:1. Since the number of writes is lower, the read performance degradation is smaller. The effect of the write performance increase is less, but the overall performance of the disk remains about the same.

### *7.1.3 Red—root + swap + /usr*

Finally, to prove that Loge does not always hurt read performance, we would like to present some results of a slightly different simulation. Red, like Cello, is a time-sharing machine, but it is more lightly loaded and uses somewhat smaller disks. We modified the simulator to use a 1KB block size, and ran the *PL* heuristic with sequential grouping, obtaining results shown in figure 7.1.3.

On this trace, the 1KB version of Loge outperformed the standard disk by 164% on writes, 8% on reads, and 36% overall. Whether similar performance can be achieved on other workloads is not clear. It may be, for example, that the trace and the simulated disk were poorly matched, and that a trace of a tuned system would have outperformed Loge on reads. Even then, this result suggests that Loge tunes itself to workloads and can compensate for poor tuning at the file system level.

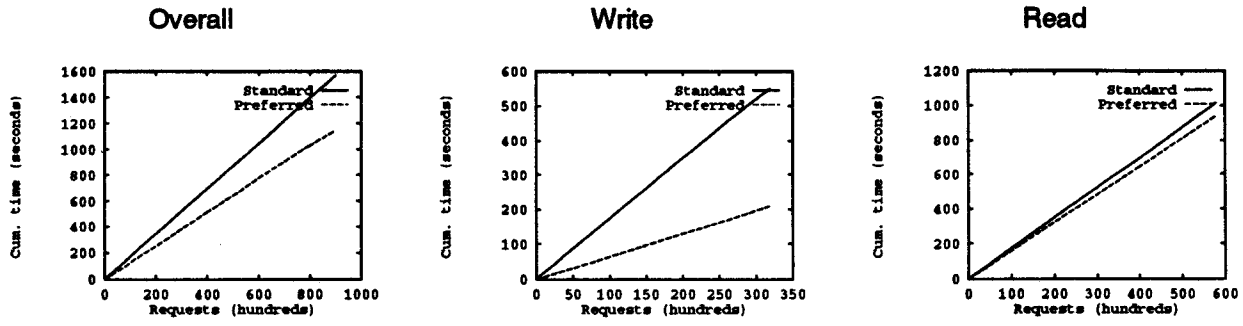



Figure 7.1.3: Performance of 1KB Loge device.

### 7.2 Idle time

Our future plans for Loge include shuffling data in the background to improve read performance. While these simulations do not address this issue directly, the traces suggest that we will have ample idle time to perform these tasks. The busiest trace, that of */usr/spool/news* on cello, keeps the disk busy only one-sixth of the time, the others far less. Even if read performance based on Loge write policies never matches that of a file system based approach, we will have time to reorder the device, and either match or exceed the read performance of a standard disk.

## 8 Future directions

Simulations of Loge will continue with the investigation of the effects of command queueing and controller-based I/O scheduling. We intend to expand our inquiries into the use of similar techniques in disk arrays, and to incorporate disk shuffling techniques to improve read performance. We would like to expand our collection of I/O traces to include a wider variety of workloads, to get a better understanding of the range of applicability of our techniques. And we would also like to go beyond refinement of the techniques described in this paper to address some of the larger issues of system design illustrated by Loge.

### 8.1 On-line data reorganization

Since actual data locations are invisible, Loge can rearrange the disk without interacting with the host. This gives us the chance to use the data shuffling techniques to improve read performance, as well as some the log-structured file system techniques to improve write performance.

In order to shuffle efficiently, we need to distinguish periods of idleness. Loge differs from conventional systems, however, in that the amount of disruption is smaller, making the periods of idleness that can be taken advantage of smaller as well. In particular, since the amount of interference is limited to about the length of an average seek, Loge can take advantage of idle periods of one-half second and degrade performance by only about 2% in the worst case. While we are interested in methods to determine optimal idle thresholds and adjust them to current load characteristics, these preliminary calculations suggest that they may not have a dramatic effect on overall performance.

#### *8.1.1 Block sorting*

An interesting alternative to adaptive block shuffling approaches is to return blocks to sequential order during idle periods. This allows us to take advantage of any read optimizations the file system may have already performed, and, based on our current simulations, would improve read performance over the Loge heuristics. The main advantage of this technique over the adaptive techniques is its simplicity. Loge would not need to keep statistics or additional maps of preferred locations, but could simply calculate the preferred location from the block address.

#### *8.1.2 Cluster recognition*

The block-level adaptive algorithms in the literature do a poor job of recognizing and maintaining sequentiality. A file that is always read sequentially may get split into many pieces in an organ pipe algorithm, and even if kept on the same cylinder, would rarely get stored in sequential order. File-oriented algorithms do not suffer from this phenomenon, but since Loge sees only blocks, these algorithms cannot easily be used. One clear area for future work is detecting access clusters from the block-level traces. Once such clusters are recognized, Loge can cache place them sequentially on the disk, prefetch subsequent blocks, and cache the initial blocks to reduce latency.

### 8.2 Optimizations

The current design of Loge is simple and straightforward, but may not be entirely optimal. Since it ignores most address information from the host, it must keep large tables to determine data location. Conversely, since the host does not know actual data locations, Loge must do its own disk sorting, implying a device which supports command queueing.

#### *8.2.1 Sorted Loge*

One way to address both is to limit blocks to a small range of cylinders. Seek-based disk sorting algorithms running on the host will then do an effective job of scheduling for seek latency (because the blocks will be approximately where the host believes them to be). Loge will still be able to limit rotational latency for writes, since it have access to a range of cylinders at allocation time. Because blocks are limited to small ranges of cylinders, this approach will prevent a disk from becoming biased, but at the expense of longer seeks for every write. Finally, since the blocks can only reside in a small number of cylinders, the address needed to identify that location gets smaller, leading to a substantial reduction in the size of the indirection table.

#### *8.2.2 Compressed Loge*

Another way of reducing the size of the indirection table is to assume that all but a small number of blocks are at their natural locations, and store in the indirection table only the addresses of "misplaced" blocks. If we assume, for example, that at least 95% of the blocks reside in their home locations at any given time, then we need only provide an indirection table for the 5% of the address space; we can calculate addresses for the rest. Since will be a hash table rather than a direct map, it will be more expensive per entry than the standard Loge design, but if the reduction in entries is large enough, the total usage will decrease. In the example above, assuming that the resulting hash table requires four times as many bytes per entry as the simple map, the resulting table is one-fifth that of the standard device. Clearly, since the vast majority of blocks are in their standard locations, host-based disk sorting will be effective.

### 8.3 Loge techniques in disk arrays

The applications of Loge to disk arrays extend much further than use on parity drives as discussed in [Menon89]. Unlike traditional disk arrays, for example, a properly designed Loge array reduces latency as well as improving throughput. A write request to a Loge array can use, not just the first available segment on a single device, but the first available segment across the entire array. If the spindles are properly synchronized, each additional spindle reduces the write latency. In a RAID-style device, rather than updating the parity disk synchronously, with the implicit read-write cycle, a Loge array can mirror initial writes, and then update the parity disk at its leisure. A Loge array can also use multiple devices to reorganize data, dedicating one device to reading and one to writing to double throughput, or even feeding a disk reorganization algorithm directly from the data stream, rewriting the data onto the second device as it is read from the first, or automatically shift data from one device to another to balance load.

### 8.4 Semantic interpretation vs. storage management

A traditional file system design contains two components—a semantic interpreter and a storage manager—that we see as, if not orthogonal, then at least separable. The storage manager simply allocates and manages storage, and can be combined seamlessly with a wide variety of semantic interpreters. Similarly, the

semantic interpreter can be designed to know little or nothing about the underlying storage, so that it can be used against any type of storage, disk, RAM, or optical jukebox. We believe that this interface between these two layers can be the current read/write interface, with some minor extensions to control storage capacity and deallocation.

#### *8.4.1 Extended address space — a controller based storage manager*

In a traditional file system, storage management and semantic interpretation are combined into a single structure. A simple approach to separating these two functions is to build a storage manager that provides a large, sparse address space to the semantic layer, which organizes its structures within that space so that they do not interfere. As an example, consider a storage manager that provides a flat, 64-bit address space to the semantic layer. A Posix-style file system could then be built by simply allocating one 32-bit address range to each file and using the high-order 32-bits for file identifiers. The only additional interfaces necessary between the storage manager and the file system are an interface for freeing storage and another to track storage usage.

It is worth emphasizing that this framework exists within the current SCSI-II protocol. SCSI already allows a 32-bit block address space which, with 1KB blocks, corresponds to a 42-bit byte address space, and can be carved quite nicely into identifiers and file extents (though in a slightly more complicated way than the 64-bit space described above). A reserved area of the disk could be used to read status information from the device, and blocks could be freed by overwriting them with zeros. SCSI even provides a repeating write command that allows large contiguous address spaces to be overwritten with a data pattern without requiring that the host transmit the pattern more than once.

### 8.5 Transaction support

Since all updates to Loge are shadow paged, all single-request updates to Loge are naturally atomic. Extensions to provide multiple-request transactions or concurrent transactions are straightforward, but are only useful if an application can take advantage of them. One potential application is the file system. Directory and file metadata updates can be made atomic simply by associating a transaction ID with the process performing the updates, and then committing the changes with the final update. The question of whether this type of support can be used by databases and transaction systems is, to us, an interesting one. Since Loge avoids most of the overhead traditionally associated with shadow paging (for example, it does not require a separate update of a disk index), it changes the equations which have favored logging databases over shadow paging designs. Whether the result in fact favors shadow paging remains to be seen, but the issue certainly needs to be revisited.

## 9 Conclusions

For many years, file system designers (ourselves included) have argued that high-level software does a better job of data placement than low-level software or a peripheral device is capable of doing, but the track record of such efforts is not particularly good. File systems tend to be overly complex and inefficient, and to have wide, complicated interfaces that make them difficult to replace as underlying technologies change. Many of the internal structures in modern file systems were designed at a time when memory was expensive and processor cycles were at a premium, and they reflect these design points even though the parameters of current systems are fundamentally different. In particular, memory is now inexpensive enough that keeping full main memory indices of secondary storage is cheap compared to the cost of the secondary storage itself, and microprocessors are cheap enough that putting a capable CPU on an I/O controller does not appreciably increase the cost of the resulting device.

In Loge, we have a device which demonstrates that, at least for the writing of data, file systems are not the best place to do data placement. The work of others on disk shuffling and abstract storage devices shows that, even for reads, heuristics that look only at access patterns can outperform conventional file system designs. From our perspective, data placement and shuffling are natural functions of I/O devices, not file systems, and while these operations can be implemented in the host operating system, doing so would be more complicated, difficult and expensive than a straightforward controller implementation such as Loge.

## 10 Acknowledgments

We would like to thank Carl Staelin and John Wilkes for their help in editing our paper, David Jacobson for setting up the simulation environment, and Chris Ruemmler for a summer spent running the early simulations. This work was carried out as part of the DataMesh research project at Hewlett-Packard Laboratories.

## 11 Availability

Loge is still in the simulation and design stage, and is not available at this time. The authors can be contacted via e-mail at *renglish@hpl.hp.com* and *stepanov@hpl.hp.com*, or via US Mail at Hewlett-Packard Laboratories, Building 1U, P.O. Box 10490, Palo Alto, CA 94303-0969.

## 12 References

[Anderson91] Dave Anderson. Data storage technology: trends and developments. Presentation at UCB RAID retreat, 1991.

[Baker91a] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 198–212. Association for Computing Machinery SIGOPS, 13 October 1991.

[King90] Richard P. King. Disk arm movement in anticipation of future requests. *ACM Transactions on Computer Systems*, 8(3):214–29, 1990.

[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.

[Menon89] Jai Menon and Jim Kasson. Methods for improved update performance of disk arrays. Technical report, RJ 6928 (66034). IBM Almaden Rescarch Center, San Jose, CA, 13 July 1989. Declassified 21 Nov. 1990.

[Musser91] David R. Musser. Block shuffling in Loge. Technical Report HPL–CSP–91–18. Concurrent Systems Project, Hewlett-Packard Laboratories, 31 July 1991.

[Nelson91] Bruce Nelson and Auspex Engineering. The myth of MIPS: an overview of functional multiprocessing for NFS network servers. Technical report 1. Auspex Systems Incorporated, February 1991.

[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.

[Peacock88] J. Kent Peacock. The counterpoint fast file system. *1988 Winter USENIX Technical Conference* (Dallas, Texas, February 1988), pages 243–9. USENIX, February 1988.

[Rosenblum91] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA), pages 1–15. Association for Computing Machinery SIGOPS, 13 October 1991.

[Ruemmler91] Chris Ruemmler and John Wilkes. Disk shuffling. Technical Report HPL–CSP–91–30. Concurrent Systems Project, Hewlett-Packard Laboratories, 3 October 1991.

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

[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.