Teachable Smart Disks

Kent Hunter
mulhern

December 20, 2002

Abstract

In this paper we advocate the use of programs generated in the operating system and transmitted to an on-disk interpreter. A program generated by the OS stores information known to the OS. The program extracts information from the disk by making calls to an on-disk library. Using the OS information and the disk information the program makes calls to the on-disk library to instruct the disk to perform disk operations.

Such an interface between the OS and the disk allows a much greater degree of flexibility than the usual limited abstraction that the disk provides, through a number of intervening layers, to the OS. An OS can implement any of a variety of algorithms for disk scheduling, block placement, on-disk cache control, etc. by combining information known at the OS level and information known at the disk level in a program executed at the disk.

1 Introduction

In this paper we advocate the use of programs generated in the operating system and transmitted to an on-disk interpreter. A program generated by the OS stores information known to the OS. The program extracts information from the disk by making calls to an on-disk library. Using the OS information and the disk information the program makes calls to the on-disk library to instruct the disk to perform disk operations.

Such an interface between the OS and the disk allows a much greater degree of flexibility than the usual limited abstraction that the disk provides, through a number of intervening layers, to the OS. An OS can implement any of a variety of algorithms for disk scheduling, block placement, on-disk cache control, etc. by combining information known at the OS level and information known at the disk level in a program executed at the disk.

In the last decade there have been a number of developments in computer technology which makes this approach attractive. It has become economically viable to manufacture disks with an on-disk control and a buffer cache. The number of outstanding writes queued at a disk has grown much larger [SCO90, AUS98, SADAD02]. On the other hand, the type of interface between an OS and a disk has remained much the same—the disk exports an abstraction to the disk controller, and the disk controller exports an abstraction to the disk driver which in turn exports an abstraction to the filesystem. These multiple layers of abstraction work together to obscure disk information which the OS could use in its decision[EK95].

One solution to this problem would be to alter the abstraction that the disk exports and make corresponding alterations in the abstraction that the disk controller and driver support. However, this approach suffers from a number of weaknesses. It would require modifying portions of not only the OS and the disk, but also every layer between them. In any case, it would not really solve the problem as the interface would still be limited by what the designers considered to be useful. Even more important, the state of a disk changes rapidly in real time as the platters rotate and the head is positioned. This information is

useful to a program running at the disk but could not be communicated in a timely fashion to the OS.

The approach outlined above is superior in simplicity and flexibility. It is simpler because it requires limited alterations to the intervening layers between the OS and the disk proper—merely enough to allow a program generated by the OS to be sent to the disk. The alterations on the disk itself are more extensive but pleasingly simple. A disk access library, consisting of low level queries and instructions on the disk is installed. The program sent by the OS makes use of this library to examine and control the disk.

In the following section we discuss a motivating example demonstrating how combining information available at the OS and at the disk can facilitate the implementation of an optimized disk scheduling and layout algorithm. Then we discuss the design of the on-disk library. Then we discuss our experimental methodology.

2 Motivating Example: Disk Writes

In the following discussion we assume that a single new file is being written to disk. The algorithm can be extended to multiple files or to old files where blocks are being appended at the end.

When a new file is written to disk generally the procedure is somewhat as follows. The filesystem determines the logical blocks for the file and sends each block with its block number to the disk. The disk receives this data in the order it was sent and writes each block to the physical disk. Some scheduling algorithm, e.g. the elevator algorithm, may be used by the disk controller to change the order in which the writes are done to maximize throughput at the disk. At some point the locations of the file blocks are written out to the disk. For example, in the Unix filesystem, the metadata for a file is stored as an inode and that inode contains the numbers of the individual blocks in the file[Tan01].

It is clear that the interface between the disk and the file system is inflexible. The OS mandates exactly where each block should be placed on disk even though there is no information at the OS level about the physical location of the block. For example, the filesystem disk layout algorithm may be designed to allocate all the blocks of a file contiguously. However, the incomplete information it has about the physical disk layout may result in the blocks being placed in surprising locations on the physical disk. In general, it is implicit in the filesystem layout algorithm that a contiguous run of logical block numbers will correspond to a contiguous run of physical block numbers. However, a defect may arise in the disk during disk operation. In that case, the logical to physical block mapping will be very different for the physical block associated with the defect. Furthermore, the OS has no information about the physical state of the disk at the time the write is about to occur. Consequently, the blocks it has designated may be located so that the disk head must spend a long time repositioning before it is able to write.

In its simplest form our procedure is the following. The OS sends to the disk the blocks to be written, the inode associated with the blocks and a program to implement the following disk layout/scheduling algorithm. The empty block "nearest" to the current position of the disk head is located. In this case "nearest" is supposed to mean closest in terms of positioning time. The actual implementation of our algorithm does not necessarily find this "nearest" block, but will find some approximation (see Section 5). The first block is written. The inode is updated with the logical block number corresponding to the physical block. This process continues until all the blocks have been written. Then the inode is written out to disk and the logical block numbers for the blocks that were just written is communicated to the OS so that the in-memory filesystem information can be updated.

It is necessary to implement our algorithm in this way because it requires both on-disk information and filesystem information. The logical to physical block mappings are available at the disk but not generally exported to the filesystem. The current position of the diskhead is only meaningful at the time the block is about to be written. On the other hand it is

in the filesystem that the organization of files and directory structures is maintained. The OS information is encapsulated in the program that the OS sends to the disk. The disk information is extracted by the program when it is executing at the disk. The program uses all this information to make scheduling and layout decisions, which it then communicates to the disk.

The algorithm described above is intended to optimize for speed of writes. However, it may have the side effect of optimizing for locality as well as it would skip bad sectors developed during operation. One potential problem is that the communication between the filesystem and the disk is no longer one way as the disk must send back to the filesystem the locations of the blocks that were just written. Also, our algorithm does only one thing well, i.e. write out new blocks quickly. It may scatter the blocks of a file which would result in poor disk performance if the file is read sequentially in the future.

We believe that the defects of our algorithm do not detract at all from the utility of our general idea. What we are demonstrating is not a specific disk scheduling/layout algorithm but rather how programs constructed by the operating system and executed on disk can be used to implement OS/disk functionality that would be difficult or impossible to implement with the standard OS/disk interface. The algorithm can always be changed and refined to accomplish whatever the algorithm designer considers to be a desirable goal.

3 On-disk Library

The on-disk library consists of the methods which the OS generated program can invoke to acquire information about the disk or to issue instructions to the disk.

Information about the disk can be of two types. The first is basic physical information about the disk, e.g. the location of the disk head or the location of a physical block given a logical block number. This information need not be restricted merely to the physical disk. If the disk has an on-disk cache then the block numbers of the blocks resident in the cache might be useful information. The other kind of information is predictive. For example, given the current location of the disk head, the location of a physical block, and numerous other parameters of the disk, what would be the estimated time required to reposition the head at that block? We consider it better design to supply this information through the disk library rather than have it calculated by the program executing on the disk. A program that the OS generates *could* calculate this information by discovering the relevant parameters of the disk. However, implicit in this approach would be some assumptions about what the relevant parameters of the disk are and how they affect positioning time. Inevitably these assumptions would turn out to be false in some instance. It seems far better to include the disk specific calculations along with the disk state in the on-disk library.

An important consideration is whether this predictive information could actually be calculated in a sufficiently timely fashion at the disk. The DiskSim implementation uses a fairly simple threepoint curve model to calculate seek times. The code required is quite small and could certainly be executed in a fraction of the time that it takes a disk to rotate through even one block. Assuming that the necessary calculations are no more complex than this and that disk technology does not change radically the results of the predictions could be calculated in a short enough time to be useful to the program running at the disk.

Instructions to the disk consist of the basic disk operations, reads, writes, and seeks. Instructions to the cache would be cache operations, e.g. flush or retain a cache entry.

See Appendix A for a listing of the methods available.

4 Experimental Methodology

We used the DiskSim Simulation Environment to conduct all our experiments [Gan95, GWP99, Gan99]. DiskSim is a highly-configurable disk system simulator developed at the University of Michigan. It simulates an entire I/O subsystem including disks, intermediate

controllers, buses, device drivers, request schedulers, disk block caches, and disk array data organizations.

DiskSim can also be incorporated into a full system-level simulator. The system-level simulator generates an I/O request and passes it to the disksim module. The I/O request consists of a starting logical block number, the number of blocks to be read/written, and a flag indicating whether the operation is a read or a write. The disksim module processes the request, possibly generating more I/O requests, which it in turn processes. When all the outstanding I/O requests have been processed, the disksim module returns control to the system-level simulator. In this way, we are able to simulate the execution of a program actually at the disk level as we ensure that each I/O request sent to the disk is completely processed before the next one is sent.

Although our conception requires that a program be sent to the OS and executed at the disk in our experiments we simply incorporated our code into the body of the system-level simulator. The system-level simulator and the rest of DiskSim are then compiled together into a single executable.

5 Results

Our experiments focused on measuring the performance of the disk.

We used four different ways of organizing disk writes for the same sets of blocks. The first and second simulate the behavior of the disk using the normal OS/disk interface while the third and fourth use our algorithm to order disk writes.

The first category, *Ungrouped Blocks*, indicates that a random sequence of block numbers is sent (presumably by the OS) to the disk. This category is really intended as a baseline, and should simulate the behavior of an OS that makes no attempt to write files in sets of contiguous blocks. In the second category, *Grouped Blocks*, each block number has approximately a fifty-percent chance of being the immediate numerical successor of the previous block number, and the sequence of blocks is still sorted before being sent to the disk. This simulates the behavior of an OS which batches block writes before sending them to disk. Neither of these models effectively take into account complex filesystem behavior, such as trying to keep a file entirely in one cylinder group. One can imagine that such a policy would result in lower seek times than those we report, but it seems unlikely that it could achieve seek times comparable to our algorithm.

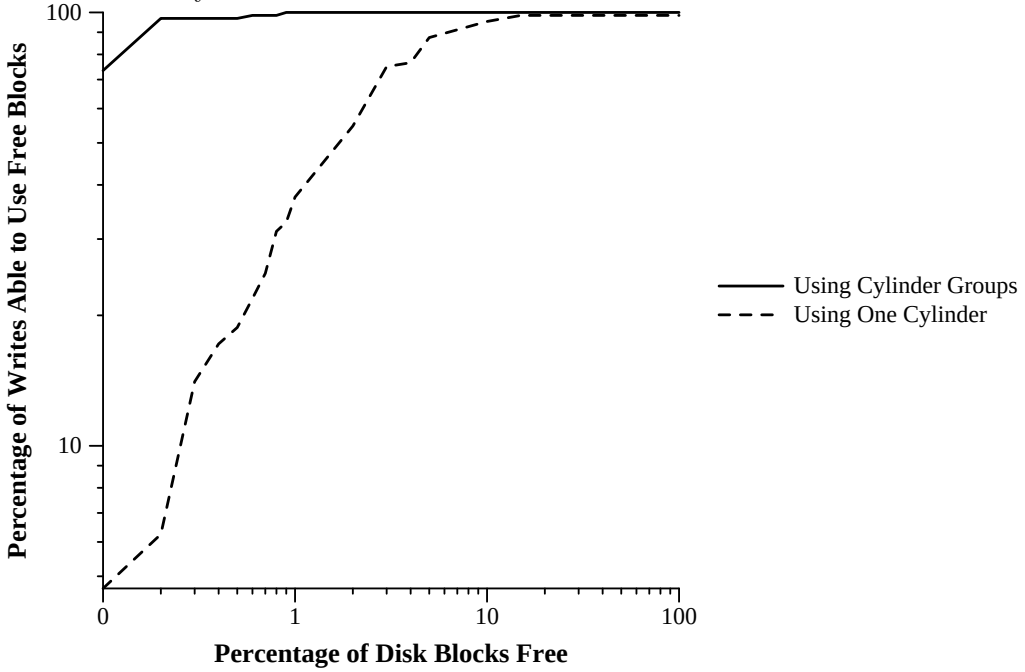
The third and fourth category use two variations of the algorithm described on page 2 to order disk writes. The third category, *Simple Algorithm*, works as follows. The OS sends a default block number with each block. If an empty block is found on the track at which the disk head is presently located, the block is written on that empty space, otherwise the default block number is used. This use of a default block number can have the effect of relocating the disk head to a different track where there may be more free physical blocks available.

For the fourth category, *Cylinder Groups*, all the tracks in the current cylinder group[1] are searched (rather than just the current track). The idea behind this variation is that the OS may prefer to try to localize related blocks in the same cylinder group. For the following experiments, a cylinder group was taken to be a set of fifty cylinders, so that the ranges of cylinders from 0 through 49 and 50 through 99 are the first two cylinder groups on a disk.

It turns out that there is little difference, at least at the level we considered, in effectiveness between the Simple Algorithm and Cylinder Groups. Figure 1 indicates that as long as more than 10% of disk blocks are available for the algorithm to use, even the Simple Algorithm can usually find blocks on the current track. While this graph only shows our algorithms' success in finding room on a track for 64 blocks, this implies that we will endure a seek only once for every 64 blocks written, at the very worst. Further, consider that a scripted interface could be used to move the read/write head to a track with more free

[1] A cylinder group is a group of cylinders that are intended to hold related data[MJLF84].

Figure 1: **Finding Free Blocks:**Even with only a small percentage of blocks free, both of our algorithms can usually find room on a track for 64 blocks.



blocks available during idle periods. Throughout the rest of this section, we will consider only the case in which our algorithm is always able to find a free block on the current track, or within the current cylinder group, because, as Figure 1 shows, this is indeed the common case.

Now we compare our algorithm to the traditional method of block placement. Figure 2 illustrates the effectiveness of combining information from the operating system and the disk. By doing so, we have achieved a significant reduction in the average time taken to position the disk's head.

Further investigation reveals that this savings in positioning time comes entirely from eliminating seeks. The breakdown of positioning times shown in Figures 3 and 4 demonstrate that while our algorithm nearly eliminates seek times entirely, it also results in increased rotational latency. This was not what was intended: our algorithm was supposed to find not just any free block on the current track, but in fact the "closest" block in terms of rotational distance.

What went wrong? In this algorithm's original implementation, we eliminated one layer of abstraction too few. The program that produced the results discussed above searched for free blocks in order of physical block number, starting from the block currently under the read/write head and "moving around the track", or so we'd assumed. In fact, considering consecutive physical block numbers does not appear to provide this behavior. When the algorithm is modified so that it actually considers the physical *positions* of each block (rather than the physical block numbers) and finds the one that is smallest angular distance away from the current head position, we find that the rotational latencies for our algorithm are at least as good those of traditional block-placement, and sometimes better (see Figure 5). This provides a much more satisfying overall performance, shown in Figure 6. It seems as if there ought to be room for further improvement here.

Figure 2: **Positioning Time:** This includes both seek times and rotational delays. Our algorithms are the two left-hand columns in each group.

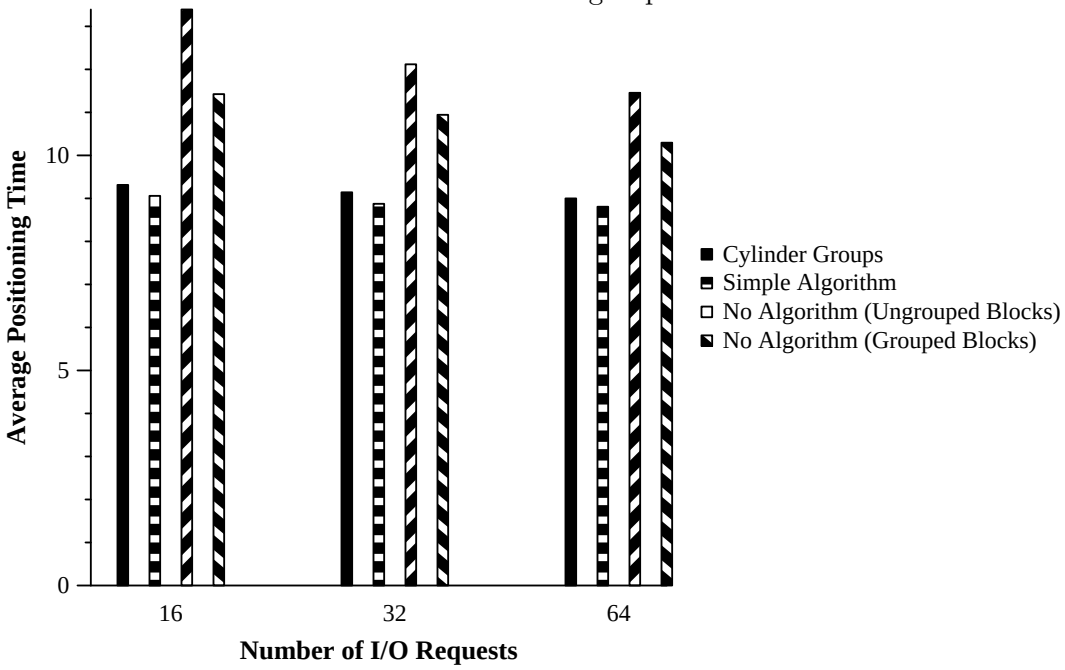


Figure 3: **Seek Times:** Our algorithm drastically reduces average seek time by eliminating nearly all seeks during writes.

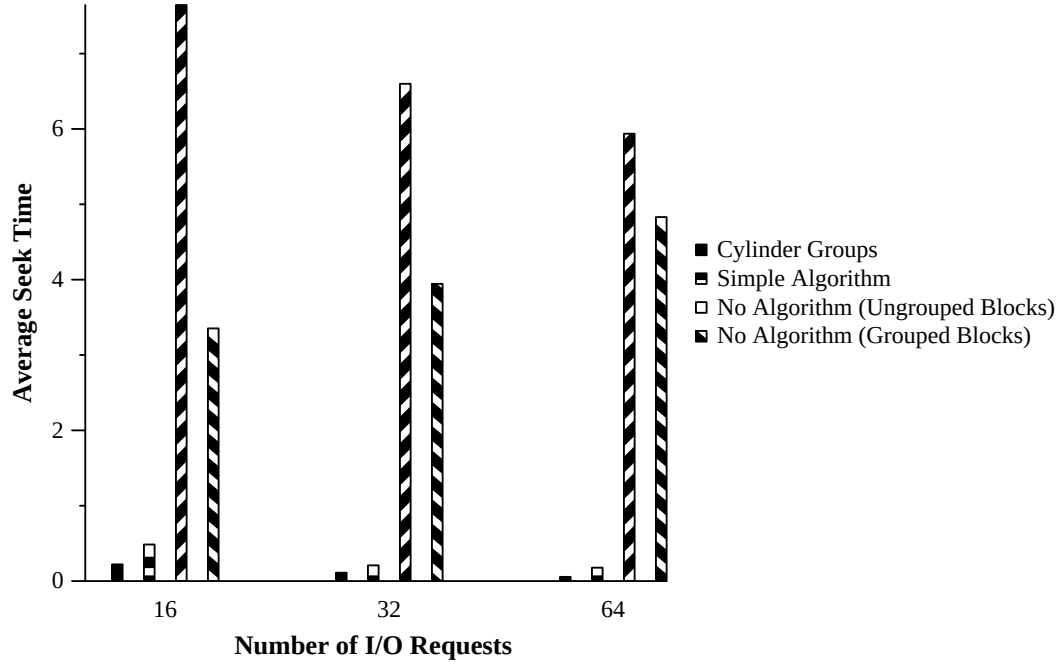


Figure 4: **Rotational Latency:** Our algorithm was supposed to find a rotationally optimal free block, but here it seems to increase the rotational latency significantly.

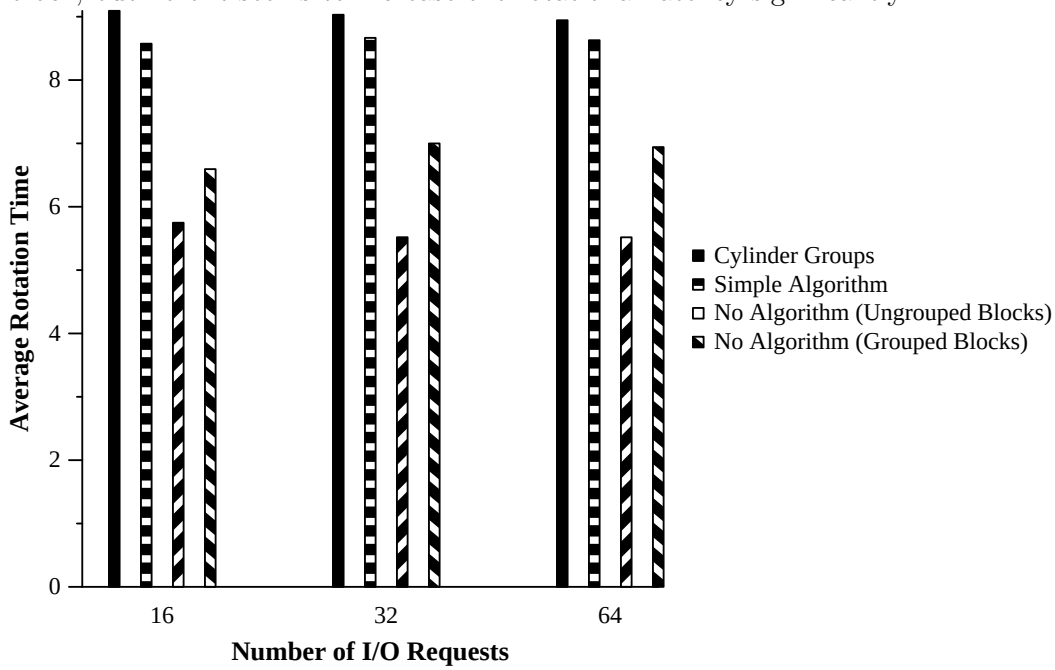


Figure 5: **Good Rotation Times:** By digging past another abstraction (physical block numbers), we can also improve rotational performance.

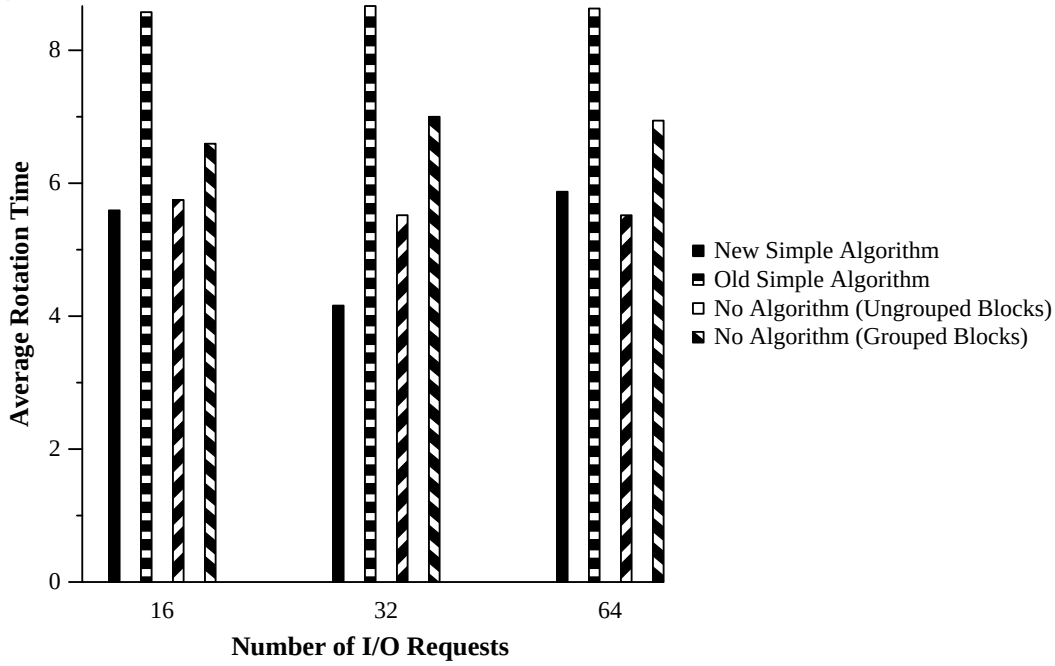
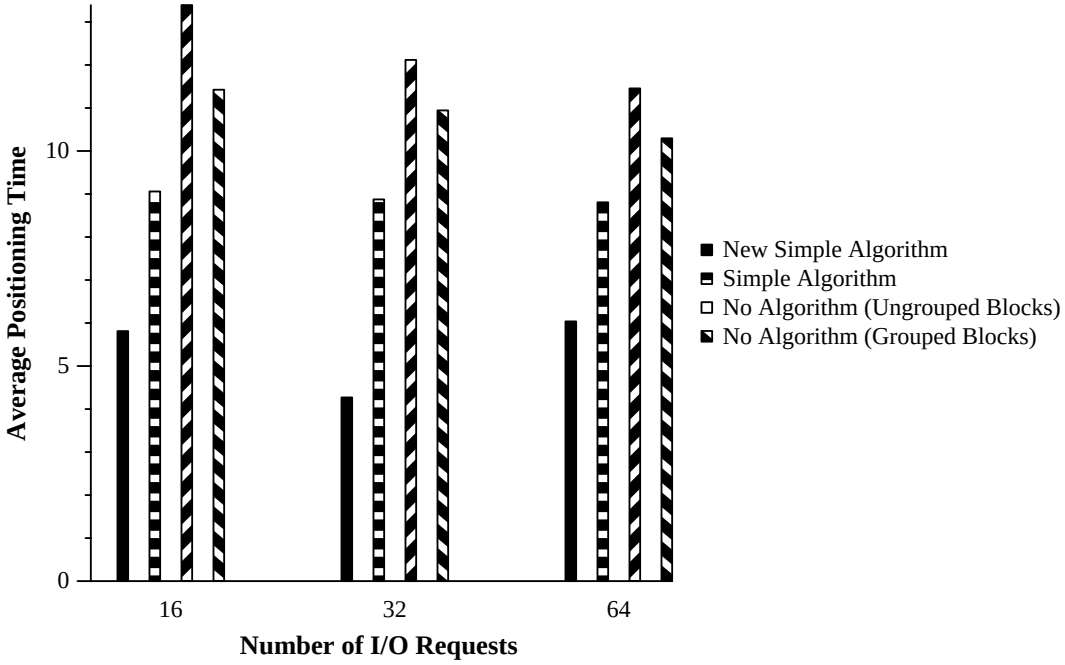


Figure 6: **Total Positioning Times with Improved Rotation** Improving the rotational awareness provides a more clear win for our algorithm.



6 Technical Note

DiskSim's system-level simulation interface turned out to be readily adaptable to our needs. Consequently, that some of the application code was confusingly written turned out not to be the obstacle that it originally seemed it would be. However, one way in which DiskSim seemed deficient was that its software data structures did not always correspond to the structures that we feel would actually occur in a disk. For example, a disk must keep some record of a logical to physical block mapping. In DiskSim however, there is a method which takes a logical block number and returns a physical block location and there is a corresponding method which takes a physical block location and returns a logical block number. These methods are parametrized on different disk organization values but strangely the corresponding logical to physical and physical to logical methods seem to be parametrized on different values.

7 Related Work

The notion of offloading some computation to an on-disk component is not new. The authors of [AUS98] were interested in optimizing the behavior of database algorithms over large databases that might be distributed among multiple disks. Thus, a portion of the algorithm that can be localized to a single disk is executed on the disk. For example SQL SELECT is run at a single disk on the local disk processor. The result of the query is buffered on the local disk cache, then returned to the host processor.

Our work differs from this work in a number of ways. The most important is that the *disklets* that run at the host processor do not discover any information about the state of the disk itself. The only information that is used is the application algorithm, encoded in the control structure of the disklet, and the OS or application knowledge of the filesystem. A second difference, is that the disklet is not generated by the OS but is part of some user level application. Therefore the safety constraints on the disklet must be much stronger

than the safety constrains on a program generated by the OS itself.

In many ways, what we have done is more similar to (indeed, inspired by) work involving scriptable RPC for active storage [SADAD02], which suggested expanding the interface presented by a file server in much the same way that we suggest expanding the interface to a local disk. This was shown to be useful in improving efficiency. For example, "partial–page writes," where a small portion of a page is written at the client, could be optimized using a script that performed the write on the server, avoiding unnecessary network traffic. In addition, scripts could be used to change policies (e.g., to alter cache consistency guarantees). Many of the ideas presented could apply to local disks as well as client/server systems.

Our work is different from the work discussed above in that our programs operate at a very low level. That is, they make queries about the actual physical state of the disk itself and they issue low level disk commands—reads, writes, and positioning instructions.

An interesting point to note is that the *Simple Algorithm* has somewhat the same effect at the disk as a log-structured file system has as the OS level[RO92] while the *Cylinder Group* algorithm has somewhat the same effect at disk level as the Berkeley Fast File System has at the OS level[MJLF84].

8 Conclusions

We set out to demonstrate that the OS/disk interface that we propose can successfully combine OS level and disk level information to extend or enhance system functionality. We chose our disk write example because its algorithm required both OS level information and disk level information. Our results demonstrate that the algorithm does what it was intended to do, i.e. decrease the positioning time required to write sequences of blocks to disk. More importantly we have shown the utility of our OS/disk interface.

Such an interface provides enough functionality to implement scheduling algorithms that are entirely different from the one presented here. As on-disk computing power increases, it may be possible to move almost the entirety of the filesystem logic out of the OS and on to the disk, where it will be fully able to exploit low-level disk information. Furthermore, we believe that enhancements that exploit this interface do not need to be restricted to disk scheduling/layout algorithms. Another possibility would be to synchronize the on-disk cache with the OS's file cache to avoid duplication of cache entries in both places; or to move the read head to an optimal position during idle time; or to tell the disk to do a read without caching it. Other possibilities are likely to suggest themselves to the reader.

References

[AUS98] A. Acharya, M. Uysal, and J. Saltz. Active disks: Programming model, algorithms and evaluation. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[EK95] Dawson R. Engler and M. Frans Kaashoek. Exterminate all operating system abstractions. In *Fifth Workshop on Hot Topics in Operating Systems*, May 1995.

[Gan95] Gregory R. Ganger. *System-Oriented Evaluation of I/O Subsystem Performance*. PhD thesis, University of Michigan, Ann Arbor, June 1995.

[Gan99] Gregory R. Ganger. The disksim simulation environment, 1999. http://www.ece.cmu.edu/ ganger/disksim.

[GWP99] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Pratt. *The DiskSim Simulation Environment Version 2.0 Reference Manual*, December 1999.

[MJLF84] 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):181–197, 1984.

[RO92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[SADAD02] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Evolving rpc for active storage. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 264–276, October 2002.

[SCO90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proceedings of the 1990 Winter Usenix*, January 1990.

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

A On-disk Library Methods

The following should be regarded as a sample of what we would consider a reasonable API for an on-disk library to provide. Not all of this functionality was implemented on top of DiskSim (indeed, much of it was already there). Neither is all of this necessary for the algorithm discussed in this paper, but it might be useful for ideas to be implemented in the future (see page 9).

number-of-disks Returns the number of disks in a disk array.

current-surface(diskno) Returns the surface (which side of which platter) where the read/write head of `diskno` is currently located.

current-sector(diskno) Returns the sector where the read/write head of `diskno` is currently located.

current-cylinder(diskno) Returns the cylinder where the read/write head of `diskno` is currently located.

blocks-per-cylinder(diskno,cylno) Returns the number of blocks on cylinder `cylno` of disk `diskno` (maybe with some indication of bad sectors).

is-cached(diskno,blockno) Returns true if and only if the data corresponding to (logical) block number `blockno` is cached on disk `diskno` (the cache contents should probably be available to the script in the form a data structure, as well).

flush-cache-block(diskno,blockno) If the data from (logical) block number `blockno` is cached at disk `diskno`, it is flushed to the disk and removed from the cache.

write-block(diskno,blockno,data) Writes a block containing `data` to disk `diskno` at (logical) block number `blockno`.

read-block(diskno,blockno) Reads the block corresponding to (logical) block number `blockno` form disk `diskno` and returns that data.

logical-to-physical-blockno(logical-blkno) Returns the physical block number of the block corresponding to the logical block `logical-blkno`

physical-to-logical-blockno(physical-blkno) Returns the logical block number of the block corresponding to physical block `physical-blkno` (the easy availability of this transformation is what allows the methods above to deal strictly with logical block numbers—it should be fast).

position-to-physical-blockno(diskno,surfaceno,cylno,sector) Returns the physical block number of the block located at the position given by the arguments.

physical-blockno-to-position(diskno,physical-blkno) Returns the surface, cylinder, and sector at which `physical-blkno` is located on disk `diskno`.

seek-position(diskno,surfaceno,cylno,sector) Instructs the read/write head to seek to the designated position without performing any I/O operations.