MODELS AND MEASUREMENTS OF FILE SYSTEM PERFORMANCE

Arvin Park

Richard J. Lipton

CS-TR-067-86

December 1986

Models and Measurements of File System Performance

Arvin Park

Richard J. Lipton

Department Computer Science
Princeton University
Princeton, New Jersey 08544

December 31, 1986

Abstract

File system buffering strategies can produce unexpected system behavior. For instance, writing one byte to a file can take more than twice as long as writing a 4096 bytes to the same file. To examine this curious phenomenon and other aspects of file system performance, we develop models for file system behavior which factor in contributions of processor speed and buffer cache organization. These models agree very nicely with a set of performance measurements conducted on a VAX-11/750 running the UNIX 4.3 BSD operating system.

We use these same models to predict the performance of recently proposed parallel mass storage architectures. We demonstrate that these architectures can potentially provide orders of magnitude more file system bandwidth than conventional non-parallel systems now provide.

Keywords and Phrases - *Disk Interleaving, File Systems, Mass Storage, Performance Evaluation.*

1. Introduction

File system performance can vary dramatically between different types of data transfer operations. Bandwidth is usually higher for large block transfers, and is greatly reduced for smaller transfers. This is an accurate generalization, but exactly how does this tradeoff operate? To examine this and other characteristics of file system performance we develop a set of simple models. These models predict file system performance on both read and write operations as a function of the size of a transfer request. In section three we validate our models with a set of performance measurements on a VAX-11/750 running UNIX 4.3 BSD. These models are then used to analyze the performance of recently proposed parallel disk mass storage architectures [Kim85] [Park86] [Sale86]. These systems are shown to provide orders of magnitude more mass storage bandwidth than conventional non-parallel systems now provide.

2. Models

We are interested in how data transfer bandwidths vary as a function of the block size of a transfer request. To estimate this performance we have developed a simple model of the file system. For read operations we will call the transfer time per byte T_r, we break up T_r into three components. (i) seek and latency time t_s (ii) data transfer time from the disk into buffer cache t_d (iii) data transfer time from buffer cache into data space t_b.

$$T_r = t_s + t_d + t_b$$

The seek and latency times can vary for individual requests, but the average seek and latency times for a large number of randomly distributed data transfer operations will be fairly constant. We will call the average seek and latency time S_r. The seek and latency time per byte t_s is inversely proportionate to the block transfer size b because the cost of the initial seek time is amortized over the number of bytes transferred. $t_s = S_r/b$

If the block size of the transfer exceeds the block size of the disk drive then an individual transfer operation will involve accesses to multiple blocks and therefore multiple seek and latency times. However, with the UNIX 4.3 file system, contiguous blocks of a file are usually placed on the same cylinder group in rotationally optimized locations so that disk seeks between contiguous blocks of a file will tend to be shorter than the initial seek to the first block of a file [Mcku83]. For this reason we consider the time D_r (time for a disk block transfer) to include the average seek and latency time between contiguous blocks of a file.

Our definition of S_r must be slightly modified now. This is because the time for accessing the first disk block of a file will contain both S_r and D_r. We define S_r to be the difference between the initial seek and latency time and the seek and latency time between contiguous blocks of a file. In this way, a portion of the initial seek and latency time will appear in D_r of the first block.

The time to transfer data from disk into the buffer cache depends on the number of disk blocks transferred. If transfer requests are assumed to start on block boundaries (It has been observed that most file accesses involve complete file transfers [Oust85]. Since files begin on block boundaries our model corresponds to actual system behavior), then the disk to buffer transfer time is the same for one

byte up to d bytes (where d the number of bytes in one disk block). At $d + 1$ bytes the time doubles because two blocks have to be transferred. Block transfers of $d + 1$ bytes to $2d$ bytes require the same disk to buffer transfer time and so on. The time t_d is then given by the expression $t_d = \lceil b/d \rceil D_r/b$ (Where b is the transfer request block size, d is the disk block size and D_r is the time for a disk block read operation.)

The time to transfer data from the buffer cache to the data space is proportionate to the number of bytes transferred. The rate per byte is therefore a constant C_r. The constant C_r depends on the memory to memory transfer speed of the system which in turn depends upon the bus width, memory cycle time and CPU speed. The final expression for block read time is then:

$$T_r = S_r/b + (\lceil b/d \rceil D_r)/b + C_r$$

We are interested in the file transfer bandwidth on read operations $B_r = 1/T_r$. A graph of read bandwidth B_r as a function of transfer size (in bytes) appears in Figure 1. (We have assumed that $S =$ 40 milliseconds $D_r = 15$ milliseconds $d = 4096$ bytes and $C_r = 1$ microsecond.)

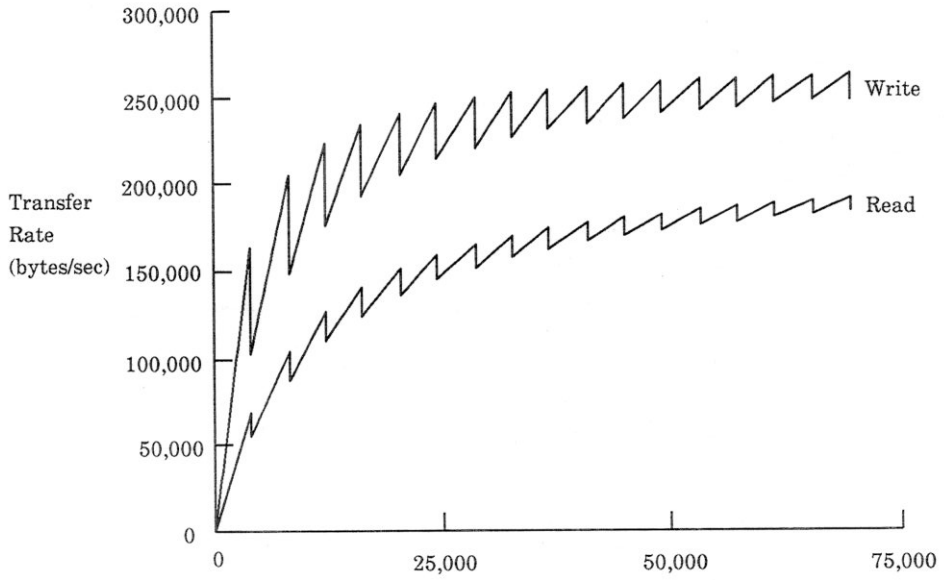

Figure 1: Read and Write Transfer Rates

The sharp discontinuities at block boundaries arise because files are transferred in block sized units from disk. Even if a transfer request exceeds a block boundary by only a single byte, an entire block transfer operation must be performed for the single byte. This boundary effect becomes less pronounced as the data transfer size increases causing the extra block transfer to be amortized over more bytes. The initial seek time causes the asymptotic convergence toward the maximum bandwidth which is similarly amortized over progressively larger data transfers.

Write operations are asynchronous in the 4.3 file system. To perform a write, the system writes the file into the buffer cache and then passes the write request to the device driver which moves the data from the buffer cache to the disk. After the request is queued, the application program is free to continue processing while the write operation is performed asynchronously. However, if the buffer

cache is full of unprocessed write requests, the application program may have to wait for the device driver to free buffers by completing an unprocessed write request.

To model simple write operations (not overwrites), we want to calculate the average write time per byte. We call this quantity T_w. Like the model for read operations T_w has three components.

$$T_w = t_s + t_d + t_b$$

Because of the asynchronous nature of the write operation, t_b can occur concurrently with t_s and t_d. Because t_b is in general much smaller than $t_s + t_d$, its contribution to the running time becomes insignificant. The asynchronous nature of the write request also reduces average seek time. This is because the write operations to disk do not have to be processed in the order in which they arrive. A group of requests can be processed in an order that minimizes seek and latency times between requests reducing the average seek and latency time for write operations S_w. Our resulting data transfer time per byte is then:

$$T_w = S_w/b + (\lceil b/d \rceil D_w)/b$$

A graph of this write transfer rate $B_w = 1/T_w$ also appears in Figure 1. (We have assumed that S_w = 10 milliseconds, D_w = 15 milliseconds, and d = 4096 bytes.) The write transfer operation behaves similarly to the read accept that there is less initial seek and latency time which allows it to converge to its maximum bandwidth more rapidly. The asynchronous operation of data space to buffer cache, and buffer cache to disk data transfers removes the performance contribution of the data space to buffer cache transfer time allowing for a higher maximum bandwidth.

The overwrite operation is more complicated to model. This is because if a disk block is only partially overwritten, the block must first be read in from disk before it can be overwritten and sent out to the disk. The time for a single block read must be included in the overwrite time if a block does not end on disk block boundaries. (Remember we assume that files begin on block boundaries.) The time for a single block read is merely $S_r + D_r + C_r b$. The rest of the write time is similar to regular write timings. Our total overwrite time then becomes:

$$T_{ow} = S_w/b + (\lceil b/d \rceil D_w)/b + (S_r + D_r + C_r b)/b \quad (\text{if } b \neq nd, \text{ for } n \text{ any positive integer } n.)$$

This write timing only applies if the transfer request is not a multiple of the disk block size. If it is a multiple, the read is not performed and its contribution is not included in the running time:

$$T_{ow} = S_w/b + (\lceil b/d \rceil D_w)/b \quad (\text{if } b = nd, \text{ for } n \text{ any positive integer } n.)$$

A graph of $B_{ow} = 1/T_{ow}$ appears in Figure 2. (We have assumed that S_w = 10 milliseconds, D_w = 15 milliseconds, S_r = 40 milliseconds, D_r = 10 milliseconds, C_r = 1 microsecond, and d = 4096 bytes.)

The sharp spikes at the block boundaries arise because write operations which are even multiples of block sizes do not require an additional block read and hence have higher bandwidths. This gives rise to some interesting performance anomalies. It takes on average over twice as long to overwrite one byte in a file as it does to overwrite 4096 bytes to the same file.

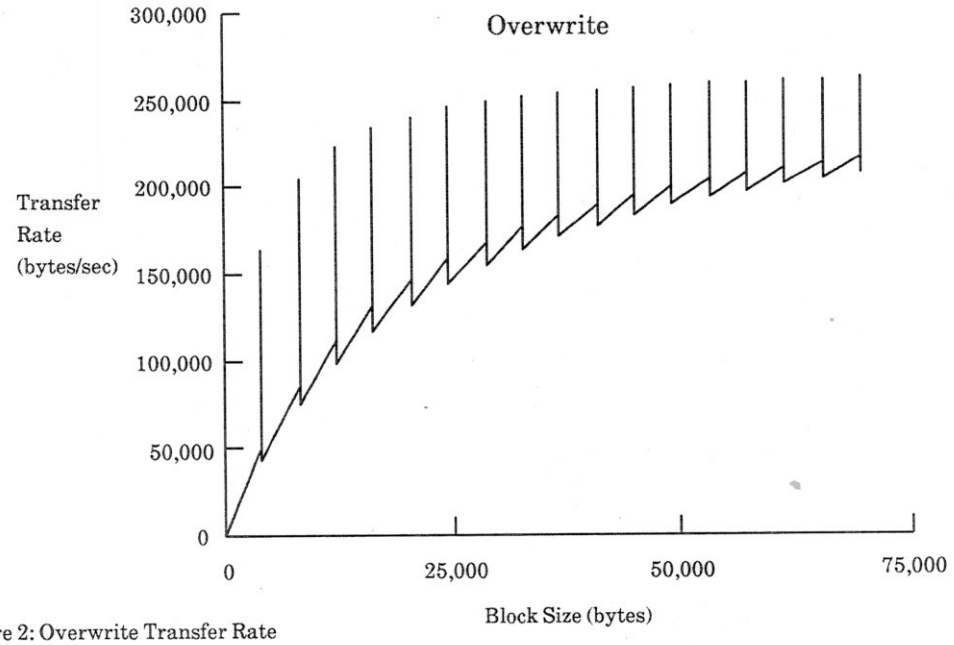

Figure 2: Overwrite Transfer Rate

We are interested in examining how file transfer performance depends on the buffer cache to data space transfer rate C. To this end we have produced the graph in Figure 3 where $S = 40$ milliseconds, $D = 15$ milliseconds, and C is varied from 0 to 1000 microseconds per byte. Increasing C has the effect of limiting the maximum attainable bandwidth. This effect is noticeable as C is increased from 0 to 1 microseconds/byte, and it clearly dominates performance as C goes to 10 , 100, and 1000 microseconds/byte.

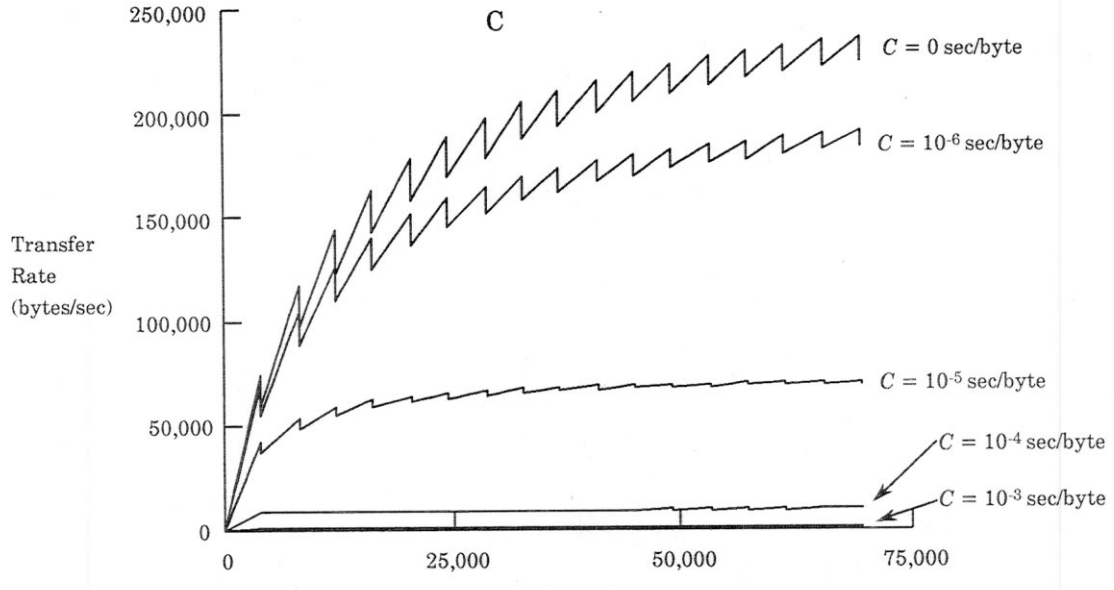

Figure 3: Dependence on C

The effect of varying the initial seek and latency time S is illustrated in Figure 4. C is held constant at 1 microsecond/byte, D = 15 milliseconds, and d = 4096 bytes. S is varied between 0 and 1000 milliseconds. Increasing S has the effect of slowing the rate at which a system approaches its maximum bandwidth. When S is 0, the maximum bandwidth is achieved immediately at the 4096 byte block size. When S reaches 1000 milliseconds, it significantly degrades performance on all reasonable transfer request sizes.

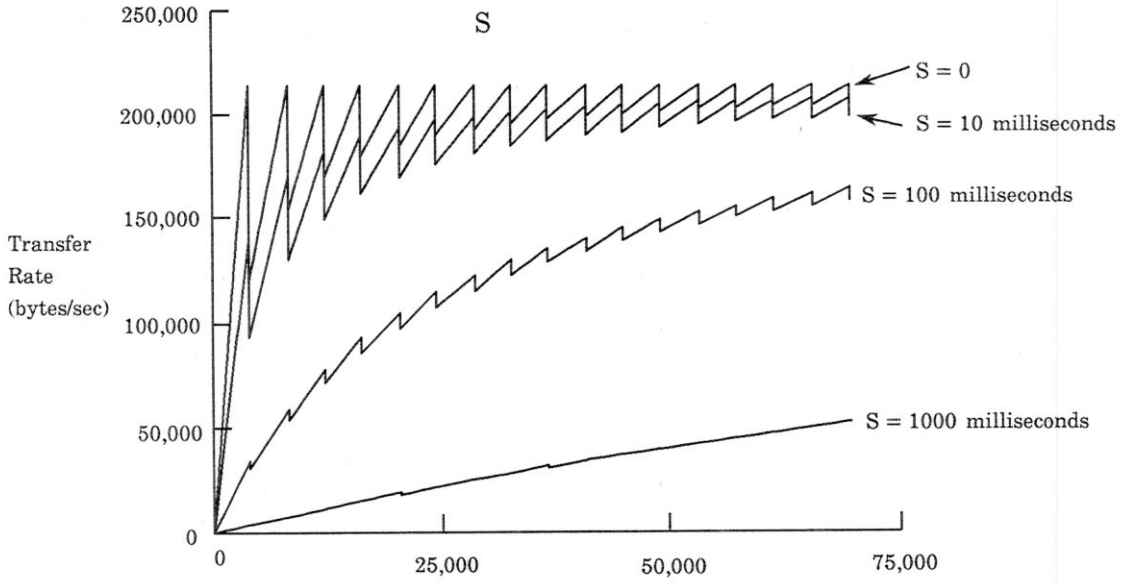

Figure 4: Dependence on S

Finally the effect of varying the contiguous block transfer speed D is illustrated in Figure 5. S is held at 40 milliseconds, C = 1 microseconds/byte, and d = 4096 bytes. D is varied between 0 and 1000 milliseconds. When D is 0, The system is mainly constrained by the initial seek time S, and the maximum bandwidth of 10^6 bytes/second imposed by the value of C. We see a smooth curve, free of the discontinuities imposed by block boundaries. As D is increased it not only accentuates breaks at block boundaries, but it ultimately limits the maximum attainable bandwidth.

To validate these models we turn to empirical results in the next section

3. File System Benchmarks

We conducted a series of performance measurements on a VAX-11/750 running the UNIX 4.3 BSD operating system. This VAX-11/750 system has a DEC UDA50 disk controller connected to DEC RA80 disk drives. The file system that was configured for a 4096 byte block size with 512 byte fragments. The buffer cache contained 77 buffers, each of size 4096 bytes (308K total). All of our benchmark codes were written in the "C" programming language.

We tested data transfer speeds for read write and overwrite operations to and from the file system using the *read* and *write* data transfer commands from the "C" programming language. We varied the sizes of the data transfer requests between 512 and 65536 bytes. We collected data points at 1024 byte increments for block sizes from 1024 to 20480 bytes and then 4096 byte increments from 20480 to

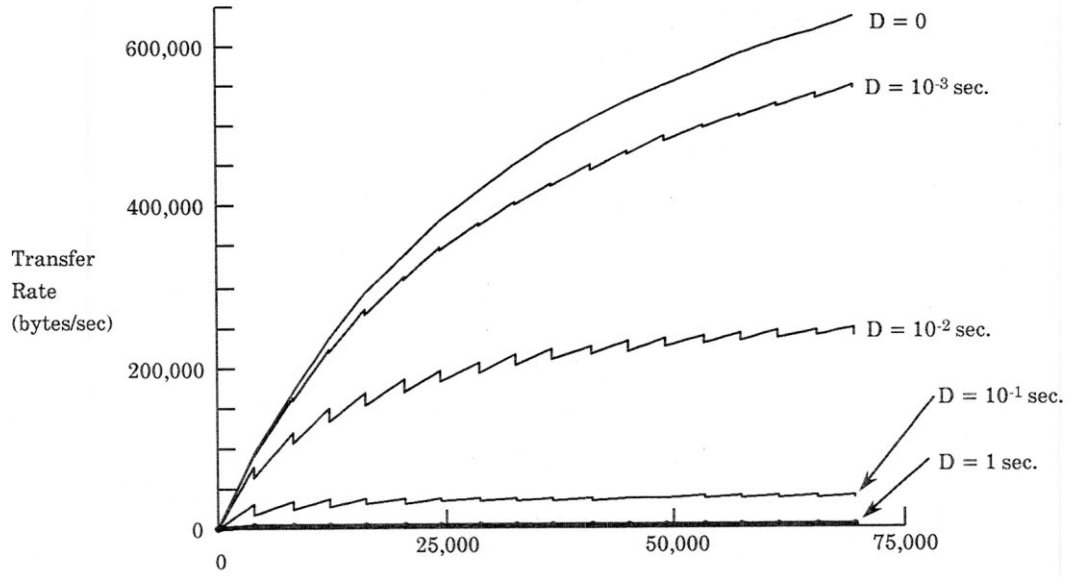

Figure 5: Dependence on D

65536 byte block sizes. Data points were collected on each side of the 4K block boundaries to catch the sharp discontinuities in data transfer speeds which exist around these boundaries.

Each data transfer was initiated at a random block boundary in an existing 4 megabyte file. (The time to open and initialize the four megabyte file was not included in the running times.) To test the speed of the read operation, the file pointer was repeatedly moved to a random block boundary, and then a read operation was performed. The random block boundary was determined in such a way that the read or write operation would never attempt to read or write past the end of the file. The read operation was performed several hundred times for a given block size and then the total elapsed wall time was measured. The data transfer speed was then calculated by dividing the total number of bytes transferred by the total elapsed time. We found that the total elapsed time could vary by as much as several percent between trials, so we repeated the file transfer benchmark forty times in succession and then averaged the resulting data transfer times before calculating the data transfer speed. Figure 6 presents the measured file system performance on read operations. This corresponds very nicely with the performance predictions of our simple model.

Overwrite timings were calculated in the same fashion as read timings. Our measurements appear in figure 7. The predicted spikes at block boundaries are readily apparent as well as the characteristic underlying performance.

Write timings were more complicated. Creating a new file before each write operation proved to be impractical because the write time would have been dominated by the large file creation time making accurate measurements impossible. We could have repeatedly appended blocks to the end of a file, but this wouldn't have exhibited the random access performance that we desired to measure. We remedied this by performing random read operations between each write. A file was initialized to contain 3 megabytes and then approximately two megabytes of block write operations were performed. In between each block write, a random read operation was performed. The time for the random read operations was then subtracted from the total time before the data transfer speed was computed. To ensure the writes always started on even block boundaries, each write operation was

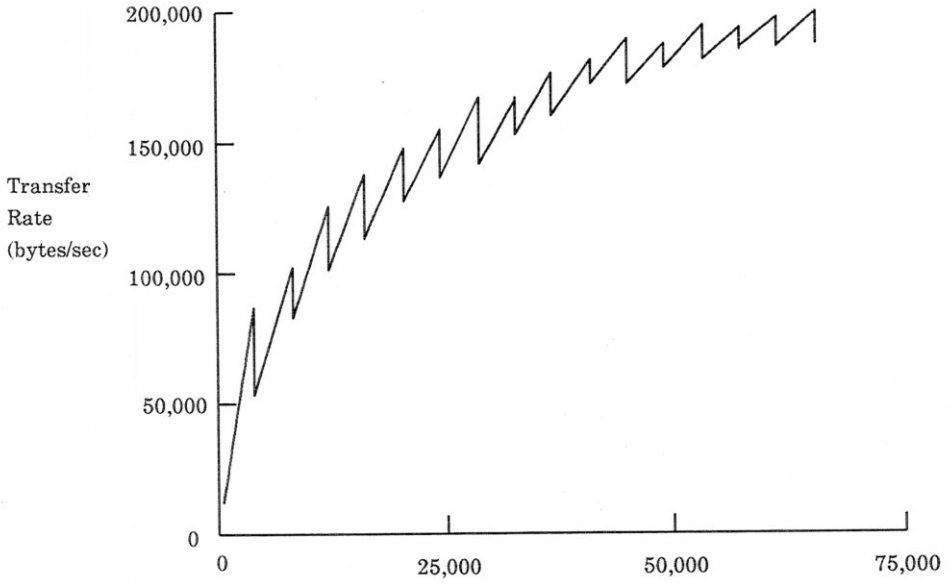

Figure 6: Test Results
Read Transfer Rate

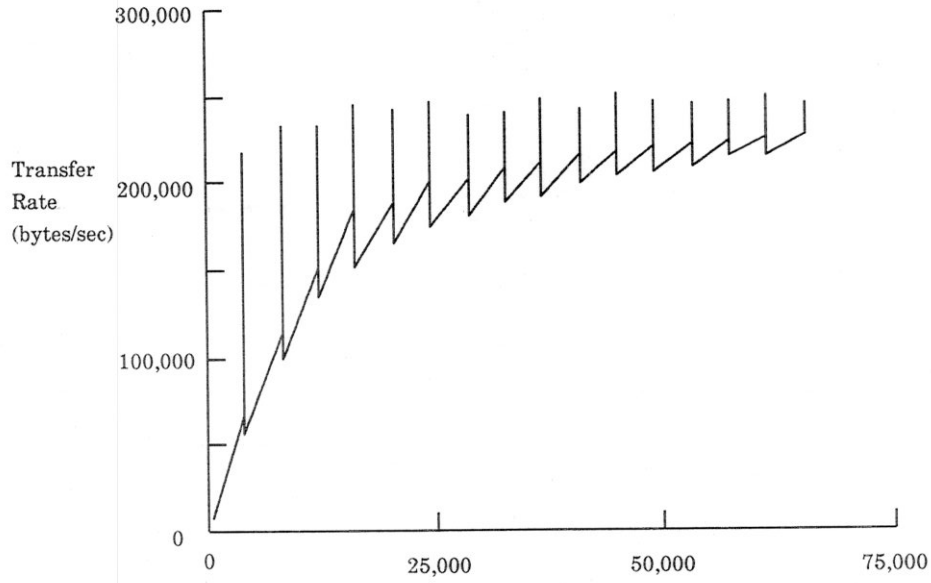

Figure 7: Test Results
Overwrite Transfer Rate

rounded up to the next largest block size multiple. This tends to require slightly more data transfer time between the user data space and the buffer cache, but previous tests have indicated that this effect accounts for at most several percent of the total data transfer time.

The write performance appears in Figure 8. After averaging forty successive trails, the write performance still exhibited a great amount of variability. Even so, the measured performance closely follows our models predictions.

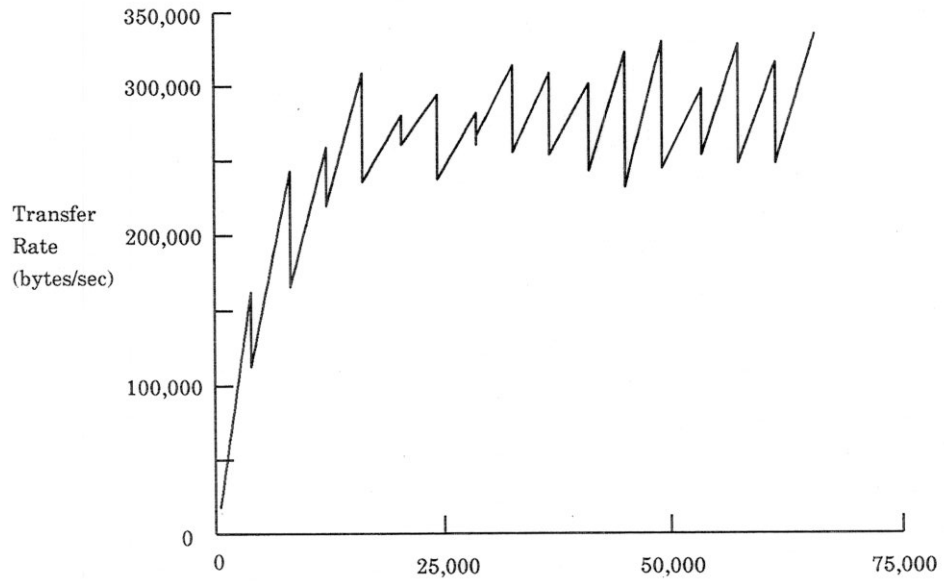

Figure 8: Test Results
Write Transfer Rate

We have demonstrated that our models very nicely predict system performance for conventional mass storage systems. In the next system we will use these same model to predict performance of recently proposed parallel mass storage systems.

4. Assessment of Parallel Mass Storage Architectures

We are interested in assessing performance of parallel mass storage architectures. These architectures provide a large amount of mass storage bandwidth by combining the bandwidth of a large number of disk drives on block transfer operations to and from mass storage [Kim85] [Park86] [Sale86]. A typical system configuration is presented in Figure 9. Blocks of data are distributed across a large number of disk drives so that a single block transfer operation can utilize the aggregate transfer rate of many parallel disk drives.

These disk drives are rotationally synchronized (using a phase-locked loop), so that seek and rotational latency times will be approximately the same for all disk drives. If the drives were not synchronized, the system would have to wait for the slowest drive before performing a data transfer operation. This worst case time can be double the average access time for a single drive. Synchronization effectively makes the parallel system behave as a single drive so that the access time of the parallel system approximates the access time to a single drive.

We have modeled the performance of a sixteen drive parallel system composed of individual drives which exhibit the same performance as the UDA50/RA80 system that we tested. Of course if higher performance disk drives are used, performance can be expected to increase proportionately. (Note that we have done nothing to modify any characteristics of the operating system or disk drive and controller. One would expect that modifications of the disk block size, changes in buffer cache replacement algorithms, and modifications to the disk scheduler will provide further performance improvements. These optimizations will be the topic of future research.) Each drive is partitioned

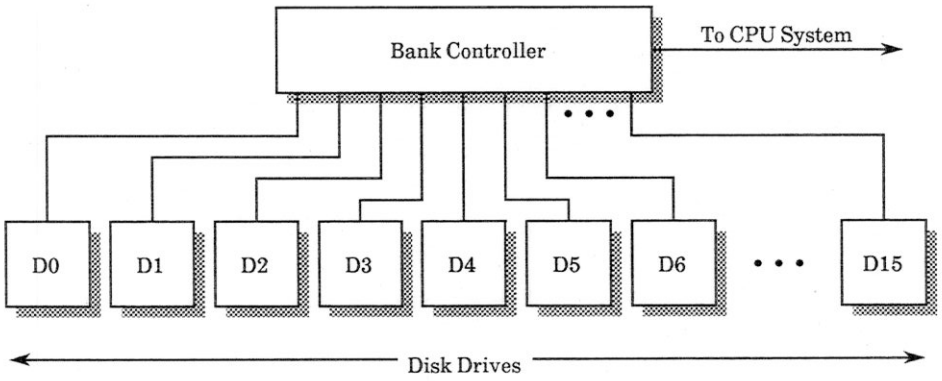

Figure 9: Parallel Mass Storage Architecture

into 4096 byte blocks. Since there are sixteen drives, each logical block is then 65536 bytes in size. We did not want to factor the contribution of processor speed into our performance numbers, so we set the buffer cache to data space transfer time "*C*" to zero. This has the effect of assuming infinite processor speed. Of course if disk to main memory bandwidth is increased sufficiently, processor speed will ultimately constrict system performance. However, we are interested in assessing the component of file system performance which is independent of processor speed.

We contrast the performance of the 16 way parallel system and a conventional non-parallel system in figures 10 and 11. Note that the peak bandwidth has improves by orders of magnitude for the larger transfer requests.

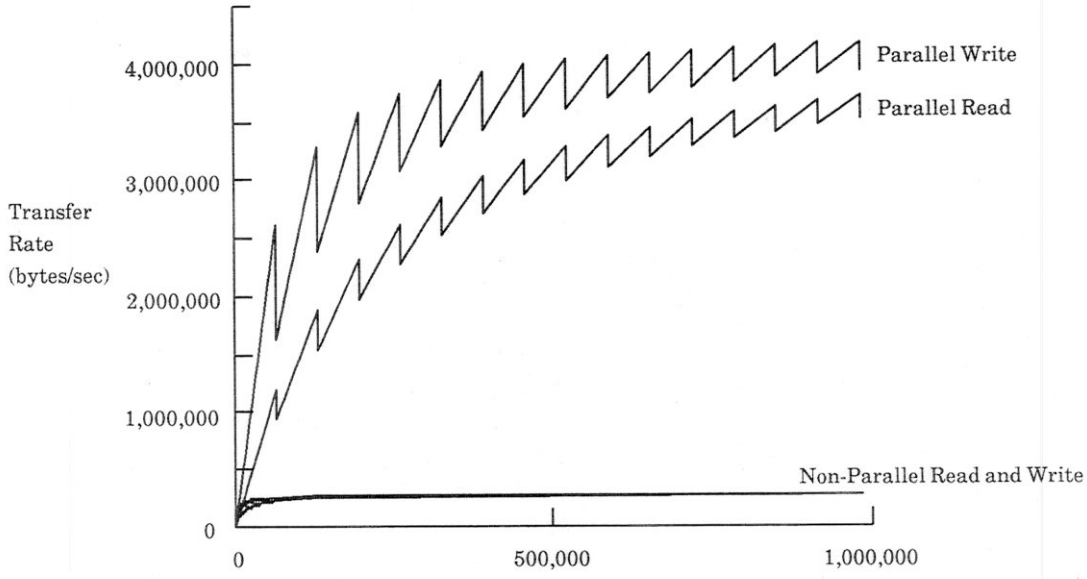

Figure 10: Parallel Read and Write Transfer Rates

However, as transfer requests decrease in size, so do the performance advantages. This is because performance on small transfer requests is constrained mainly by the seek and latency time which is not improved for these parallel architectures.

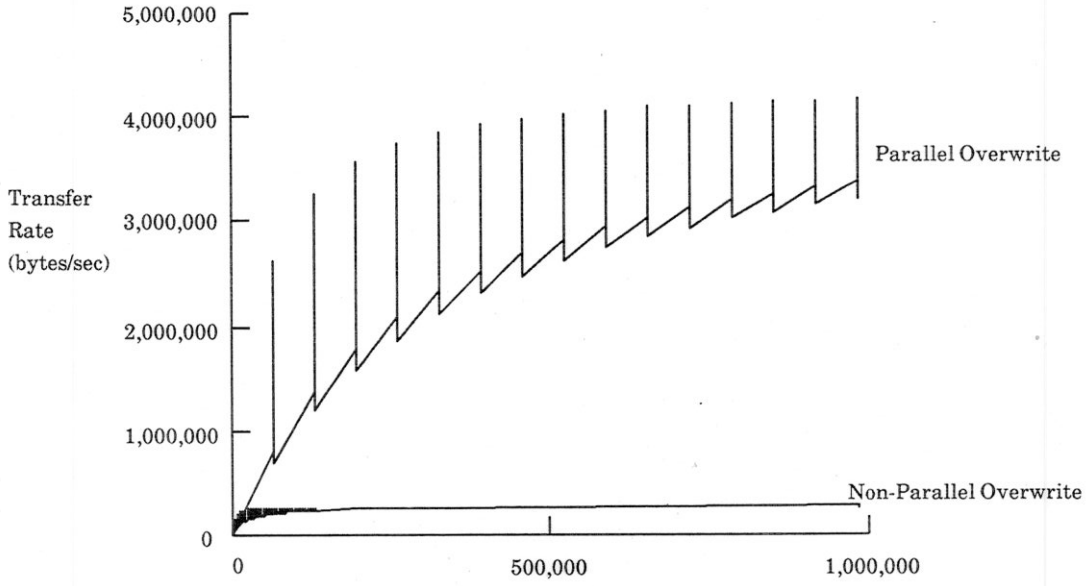

Figure 11: Parallel Overwrite Transfer Rate

To examine the performance gains on smaller transfer requests we have produced the graph of Figure 12 which is merely an enlargement of the initial portion of the graph in Figure 10. Although performance is improved many times for these smaller block sizes, we do not see the same order of magnitude gains present in larger data transfer operations.

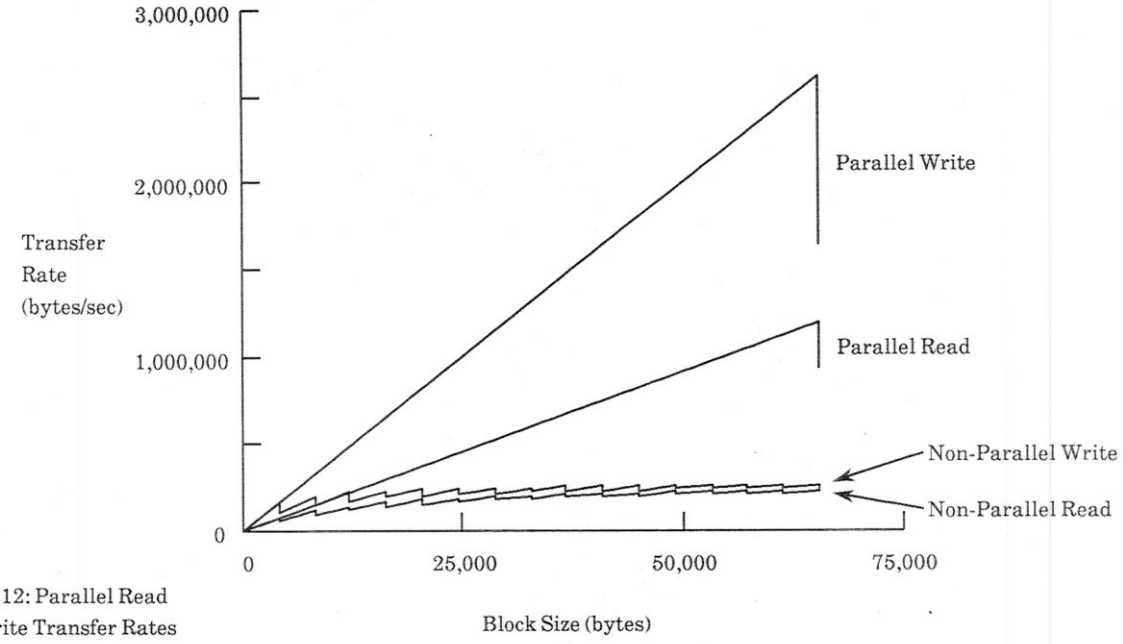

Figure 12: Parallel Read and Write Transfer Rates

We can lessen the number of smaller transfer requests by increasing the buffer cache size as long as a sequence of accesses maintains a reasonable amount of temporal locality [Oust85] [Smit85]. A system which incorporates both the high bandwidth capabilities of a parallel data transfer and the

bandwidth reducing attributes of a massive buffer cache will provide excellent performance on all tasks except ones that perform many small block accesses which exhibit neither spatial nor temporal locality.

Our research has not focused on paging performance of parallel of mass storage systems. Paging activity involves numerous transfers of small blocks of data so one would not expect paging performance to be greatly improved by parallel systems that improve performance on large block transfers. However, paging performance merits closer observation. Perhaps performance can be gained by enlarging page sizes, or compacting related pages onto larger blocks to be transferred to and from mass storage.

5. Conclusions

We have shown that file system performance can be accurately predicted with a set of simple models. These models demonstrate that mass storage bandwidths can be increased by many orders of magnitude by using parallel mass storage architectures.

Acknowledgments

We would like to thank Larry Rogers and Marc Stavely for their valuable insights into file system performance. This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Office of Naval Research under Contracts Nos. N00014-85-C-0446 and N00014-85-K-0465, and by the National Science Foundation under Cooperative Agreement No. DCR-8420948.

References

[Garc86] H. Garcia-Molina, A. Park, L. R. Rogers, "Performance Through Memory", *To appear in the Proceedings of the 1987 ACM-SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1987.

[Kim85] Kim, M. Y., "Parallel Operation of Magnetic Disk Storage Devices: Synchronized Disk Interleaving", *Proceedings of the Fourth International Workshop on Database Machines*, March, 1985, pp. 299-329.

[Mcku83] M. K. Mckusick, W. N. Joy, S. J. Leffler, R. S. Fabry, "A Fast File System for UNIX", *UNIX System Manager's Manual*, 4.2 BSD Virtual VAX-11 Version, March, 1984.

[Oust85] J. K. Ousterhaut, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, J. G. Thompson, "A Trace Driven Analysis of the UNIX 4.2 BSD File System", *Proceedings of the Tenth Symposium on Operating System Principles*, 1985, pp.15-24.

[Park86] A. Park, K. Balasubramanian, "Providing Fault-Tolerance in Parallel Secondary Storage Systems", Technical Report Number CS-TR-057-86, Department of Computer Science, Princeton University, Princeton, New Jersey, November 1986.

[Sale86] K. Salem, H. Garcia-Molina, "Disk Striping", Unpublished Report, Department of Computer Science, Princeton University, 1986.

[Smit85] A. J. Smith, "Disk Cache-Miss Ratio Analysis and Design Considerations", *ACM Transactions on Computer Systems*, Volume 3, Number 3, August 1985, pp. 161-203.