Why Does File System Prefetching Work?

Elizabeth Shriver, Christopher Small
Bell Labs, Lucent Technologies
Keith A. Smith
Harvard University

Why does file system prefetching work?

Elizabeth Shriver
Information Sciences Research Center
Bell Labs, Lucent Technologies
shriver@research.bell-labs.com

Christopher Small
Information Sciences Research Center
Bell Labs, Lucent Technologies
chris@research.bell-labs.com

Keith A. Smith
Harvard University
keith@eecs.harvard.edu

Abstract

Most file systems attempt to predict which disk blocks will be needed in the near future and prefetch them into memory; this technique can improve application throughput as much as 50%. But why? The reasons include that the disk cache comes into play, the device driver amortizes the fixed cost of an I/O operation over a larger amount of data, total disk seek time can be decreased, and that programs can overlap computation and I/O. However, intuition does not tell us the relative benefit of each of these causes, or techniques for increasing the effectiveness of prefetching.

To answer these questions, we constructed an analytic performance model for file system reads. The model is based on a 4.4BSD-derived file system, and parameterized by the access patterns of the files, layout of files on disk, and the design characteristics of the file system and of the underlying disk. We then validated the model against several simple workloads; the predictions of our model were typically within 4% of measured values, and differed at most by 9% from measured values. Using the model and experiments, we explain why and when prefetching works, and make proposals for how to tune file system and disk parameters to improve overall system throughput.

1 Introduction

Previous work has shown that most file reads are sequential [Baker91]. Optimizing for the common case, modern file systems take advantage of this fact and prefetch blocks from disk that have not yet been requested, but are likely to be needed in the near future. This technique is effective for several reasons:

- There is a fixed cost associated with performing a disk I/O operation. By increasing the amount of data transfered on each I/O, the overhead is amortized over a larger amount of data, improving overall performance.
- Modern disks contain a disk cache, which contains some number of disk blocks from the cylinders of recent requests. If multiple blocks are read from the same track, all but the first may, under certain circumstances, be satisfied by the disk cache without accessing the disk surface.
- The device driver or disk controller can sort disk requests to minimize the total amount of disk head positioning done. With a larger list of disk requests, the driver or controller can do a better job of ordering them to minimize disk head motion. Additionally, the blocks of a file are often clustered together on the disk; when this is so, multiple blocks of the file can be read at once without an intervening seek.
- If the application performs substantial computation as well as I/O, prefetching may allow the application to overlap the two, which would increase the application's throughput. For example, an MPEG player may spend as much time computing as it does waiting for I/O; if the file system can run ahead of the MPEG player, loading data into memory before they are needed, the player will not block on I/O.

If an application spends as much time performing I/O as it does computing—successfully prefetching data will allow overlapping the two and the application's throughput will double.

This paper presents and validates an analytic file system performance model that allows us to explain why and when prefetching works, and makes recommendations for how to improve the benefit of file prefetching. The model described here is restricted in two ways.

First, it addresses only file read traffic, and does not capture file writes or file name lookup. Although this restricts the applicability of the model, we believe that there are many interesting workloads covered by the model. For example, many workloads consist almost entirely of reads, such as web servers, read-mostly database systems, and file systems that store programs, libraries, and configuration files (e.g., /, /etc, and /usr). Additionally, studies have shown that, for some engineering workloads, 80% of file requests are reads [Baker91, Hartman93].

It has long been the case that in order to improve performance, file systems use a *write-back* cache, delaying writes for some period of time [Feiertag72, Ritchie78, McKusick84]. The results of the study by Baker and associates showed that with a 30-second write-back delay, 36% to 63% of the bytes written to the cache do not survive, and by increasing the write-back delay size to 1000 seconds, 60% to 95% do not survive.

Second, although our model includes multiple concurrent programs accessing the file system, we limit it to workloads where each file is read from start to finish with little or no *think time* between read requests. Although this does not cover workloads such as the MPEG player discussed above, we believe that, given the relative speed of modern processors and I/O devices, in the common case think time is immeasurable. The Baker study showed that 75% of files were open less than a quarter of a second [Baker91]. This study was done on a collection of machines running at around 25 MHz (SPARCStation 1, Sun 3, DECStation 3100, DECStation 5000); on modern machines, with clock speeds an order of magnitude faster, think time should be substantially lower.

Outline of paper. Section 2 discusses in detail the modeled file system (the 4.4BSD Fast File System) and disk. Section 3 presents previous work in file systems and prefetching policies. The analytic model of file system response time and its validation are presented in Section 4 and Appendix A. Section 5 explains why and when prefetching works. We summarize our work and discuss future work in Section 6.

2 Background

In this section we describe in detail the behavior of the file system we model, and discuss the characteristics of modern disks.

2.1 The file system

The file system that we used as the basis for our model is the 4.4BSD implementation of the Berkeley Fast File System that ships with BSD/OS 3.1 [McKusick96].

Reading files. An application can make a request for an arbitrarily large amount of data from a file. To process an application-level read request, the file system divides the request into one or more block-sized (and block-aligned) requests, each of which the file system services separately. A popular file system block size is 8 KB, although other block sizes can be specified when the file system is initialized.

For each block in the request, the file system first determines whether the block is already in the operating system's in-memory cache. If it is, then the block is copied from the cache to the application. If the block is not already in memory, the file system issues a read request to the disk device driver.

Regardless of whether the desired block is already in memory, the file system may prefetch one or more subsequent blocks from the file. The amount of data the file system prefetches is determined by the file system's prefetch policy, and is a function of the current file offset and whether or not the application has been accessing the file sequentially. A read of block x from a file is *sequential* if the last block read from that file was either x or $x - 1$. By treating successive reads of the same block as "sequential," applications are not penalized for using a read size that is less than the file system's block size.

As read requests are synchronous, the operating system blocks the application until all of the data it has requested are available. Note that a single disk request may span multiple blocks and include both the requested data and prefetch data, in which case the application can not continue until the entire request completes.

Data placement on disk. A *cluster* is a group of logically sequential file blocks that are stored sequentially on disk; the *cluster size* is the number of bytes in the cluster. Depending on the file system parameters, the file system may place successive allocations of clusters contiguously on the disk. This can result in contiguous allocations of hundreds of kilobytes in size.

The blocks of a file are indexed by a tree structure on disk; the root of the tree is an *inode.* The inode contains the disk addresses to the first few blocks of a file (i.e., the first DirectBlocks blocks of the file); in the case of the modeled system, the inode contains pointers to the first twelve blocks. The remaining blocks are referenced by *indirect blocks.*

The first block referenced from an indirect block is always the start of a new cluster. This may cause the preceding cluster to be smaller than the file system's cluster size. For example, if DirectBlocks is not a multiple of the cluster size, the last cluster of direct blocks may be smaller than the cluster size.

The file system divides the disk into cylinder groups, which are used as allocation pools. Each cylinder group contains a fixed sized number of blocks (2048 blocks, or 16 MB on the modeled system). The file system exploits expected patterns of locality of reference by co-locating related data in the same cylinder group.

The file system usually attempts to allocate clusters for the same file in the same cylinder group. Each cluster is allocated in the same cylinder group as the previous cluster. The file system attempts to space clusters according to the value of the rotational delay parameter which is set using the `newfs` or `tunefs` command. The file system can always achieve this spacing on an empty file system. If the free space on the file system is fragmented, however, this spacing may vary. The file system allocates the first cluster of a file from the same cylinder group as the file's inode. Whenever an indirect block is allocated to a file, allocation for the file switches to a different cylinder group. Thus an indirect block and the clusters it references are allocated in a different cylinder group than the previous part of the file.

Prefetching in the file system cache. If the requested data are not in cache, the file system issues a disk request for the desired block. If the application is accessing the file sequentially, the file system may prefetch one or more additional data blocks. The amount of data prefetched is doubled on each disk read, up to a maximum of the cluster size. The last block of a file may be allocated to a *fragment* rather than a full size block. When this happens, the final fragment of the file is not prefetched.

It is possible for a block to be prefetched and then evicted before it is requested. If the user subsequently requests such a block, the file system assumes that it is prefetching too aggressively and cuts the prefetch size in half.

2.2 The disk

When a disk request is issued from the file system, it enters the device driver. If the disk is busy, the request is put on a queue in the device driver; the queue is sorted by a scheduling algorithm that attempts to improve response times. One commonly-used class of scheduling algorithms are the *elevator* algorithms, where the requests are serviced in the order that they appear on the disk tracks. CLOOK and CSCAN are examples of elevator algorithms. Once the request reaches the head of the queue, the request is sent to the bus controller which gains control of the bus. The request is then sent to the disk, and might be queued there if the disk mechanism is busy. This queue is also sorted to improve response time; one commonly-used scheduling algorithm is Shortest Positioning Time First, which services requests in an order intended to minimize the sum of the *seek time* (i.e., the time to move the head from the current track to the desired track) and the *rotational latency* (i.e., the time needed for the disk to rotate to the correct sector once the desired track is reached).

Seek time. Seek is the time for the actuator to move the disk arm to the desired cylinder. A seek operation can be decomposed into:

- *speedup*, where the arm is accelerated until it reaches half of the seek distance or a fixed maximum velocity,
- *coast* for long seeks, where the arm is moving at maximum velocity,
- *slowdown*, where the arm is brought to rest close to the desired track, and
- *settle*, where the disk controller adjusts the head to access the desired location.

Very short seeks (2–4 cylinders) are dominated by the settle time. Short seeks (less than 200–400 cylinders) are dominated by the speedup, which is proportional to the square root of seek distance. Long seeks are dominated by the coast time, which is proportional to the seek distance. Thus, the seek time can be approximated by a function such as

$$\mathsf{Seek\_Time[dis]} = \begin{cases} 0 & \mathsf{dis} = 0 \\ a + b\sqrt{\mathsf{dis}} & 0 < \mathsf{dis} \leq e \\ c + d\,\mathsf{dis} & \mathsf{dis} > e \end{cases} \qquad (1)$$

where a, b, c, d, and e are device-specific parameters and dis is the number of cylinders to be traveled. Single cylinder seeks are often treated specially.

The disk cache. When the request reaches the head of the queue, the disk checks its cache to see if the data are in cache. If not, the disk mechanism moves the disk head to the desired track (seeking) and waits until the desired sector is under the head (rotational latency). The disk then reads the desired data into the disk cache. The disk controller then contends for access to the bus, and transfers the data to the host from the disk cache at a rate determined by the speed of the bus controller and the bus itself. Once the host receives the data and copies them into the memory space of the file system, the system awakens any processes that are waiting for the read to complete.

The disk cache is used for multiple purposes. One is as a pass-through speed-matching buffer between the disk mechanism and the bus. Most disks do not retain data in the cache after the data have been sent to the host. A second purpose is as a readahead buffer. Data can be readahead into the disk cache to service future requests. Most frequently, this is done by the disk saving in a cache segment the data that comes after the requested data. Modern disks such as the Seagate Cheetah only readahead data when the requested addresses suggest that a sequential access pattern is present.

The disk cache is divided into *cache segments*. Each segment contains data prefetched from the disk for one sequential stream. The number of cache segments usually can be set on a per-disk basis; the typical range of allowable values is between one and sixteen.

Disk performance is covered in more detail by Shriver [Shriver97] and by Ruemmler and Wilkes [Ruemmler94].

3 Related work

Prefetching. Prefetching is not a new idea; in the 1970's, Multics supported prefetching [Feiertag72], as did Unix [Ritchie78]. Earlier work has focused on the benefit of prefetching, either by allowing applications to give prefetching hints to the operating system [Cao94, Patterson95, Mowry96], or by automatically discovering file access patterns in order to better predict which blocks to prefetch [Griffioen94, Lei97, Kroeger96]. Techniques studied have included neural networks [Madhyastha97a] and hidden Markov models [Madhyastha97b]. Our work differs from this work in three ways. First, we address only common case workloads that have sequential access patterns. Second, our model is parameterized by the file system's behavior such as caching strategy and file layout, and takes into account the behavioral characteristics of the disks used to store files. Third, our model predicts the performance of the file system.

Substantial work has been done studying the interaction between prefetching and caching [Cao95, Patterson95, Kimbrel96]. Others have examined methods to work around the file system cache to achieve the desired performance (e.g., [Kotz95]).

The benefit of prefetching is not limited to workloads where files are read sequentially; Small studied the effect of prefetching on random-access, zero think time workloads on the VINO operating system, and showed that even with these workloads the performance gain from prefetching was more than 20% [Small98].

Disk modeling. Much work has been done in disk modeling. The usual approach to analyzing detailed disk drive performance is to use simulation (e.g., [Hofri80, Seltzer90, Worthington94]). Most early modeling studies (e.g., [Bastian82, Wilhelm77]) concentrated on rotational position sensing for mainframe disk drives, which had no cache at the disk and did no readahead. Most prior work has not been workload specific, and has, for example, assumed that the workload has uniform random spatial distributions (e.g., [Seeger96, Ng91, Merchant96]). Chen and Towsley, and Kuratti and Sanders, modeled the probability that no seek was needed [Chen93, Kuratti95]; Hospodor reported that an exponential distribution of seek times matched measurements well for three test workloads [Hospodor95]. Shriver and colleagues, and Barve and associates present analytic models for modern disk drives, representing readahead and queueing effects across a range of workloads [Shriver97, Shriver98, Barve99].

4 The analytic model

In this section, we present the file system, disk, and workload parameters that we need for our model. As we present the needed file system and disk parameters, we also give the values for the platforms which we used to validate the model. We close this section with presenting the details of the model.

We used two platforms; one with a slow bus (i.e., 10 MB/s), and one with a fast bus (i.e., 20 MB/s). Details of our test/experiment platforms are in Section 5.

4.1 File system specification

Based on our understanding of the file system cache policies, we determined a set of parameters that allow us to capture the performance of the file system cache; these can be found in Table 1. For our fast machine, the SystemCallOverhead value was 5 μs and the MemoryCopyRate was 5 μs/KB.

4.2 Disk specification

To predict the disk response time, we need to know several parameters of the disk being used.

- DiskOverhead includes the time to send the request down the bus and the processing time at the controller, which is made up of the time required for the controller to parse the request, check the disk cache for the data, and so on. DiskOverhead can either be approximated using a complex disk model [Shriver97] or can be measured experimentally. In this paper we measured the disk overhead experimentally at 1.8 ms for a single file and 1.2 ms for multiple files for our slow platform and 0.34 ms for our fast platform.
- *seek curve information* is used to approximate the seek time. The seek curve information we use is $a = 0.002$, $b = 0.173$, $c = 3.597$, $d = 0.002$, and $e = 801$ as defined in equation (1).
- *disk rotation speed* is used to approximate the time spent in rotational latency. The DiskTR is the rate that data can be transfered from the disk surface to the disk cache. The disk used to validate our model spins at 10,000 RPM, giving us a DiskTR of close to 18 MB/s.
- BusTR gives us the rate at which data can be transfered from the disk cache to the host; we are bounded by the slower of the BusTR and DiskTR. On the slow platform, the transfer rate was limited to 9.3 MB/s; on the fast platform, the transfer rate was 18.2 MB/s.
- CacheSegments is the number of different data streams the disk can concurrently cache, and hence the number of streams for which it can perform read-ahead. The disk used to validate our model was configured for three cache segments; this model of disk can be configured for between one and sixteen cache segments.
- CacheSize is the size of the disk cache. From this value and the CacheSegments, the size of each cache segment can be computed. The disk used to validate our model has a 512 KB cache.
- Max_Cylinder is the number of cylinders in the disk. The disk used to validate our model has 6526 cylinders.

Table 1: File system parameters and values for validated platform.

parameter	*definition*	*validated platform*
$\mathsf{BlockSize}$	the amount of data which the file system processes at once	8 KB
$\mathsf{DirectBlocks}$	the number of blocks that can be accessed before the indirect block needs to be accessed	12
$\mathsf{ClusterSize}$	the amount of a file that is stored contiguously on disk	64 KB
$\mathsf{CylinderGroupSize}$	number of bytes on a disk that file system treats as "close"	16 MB
$\mathsf{SystemCallOverhead}$	time needed to check the file system cache for the requested data	10 μs
$\mathsf{MemoryCopyRate}$	rate at which data are copied from the file system cache to the application memory	10 μs/KB

4.3 Workload specification

The workload parameters that affect file system cache performance are the ones needed to predict the disk performance and the file layout on disk. Table 2 presents this set of parameters; most of these parameters were taken from earlier work on disk modeling [Shriver98].[1]

4.4 The model

Our approach has been to use the technique presented in our earlier work on disk modeling, which models the individual components of the I/O path, and then composes the models together [Shriver97]. We use some of the ideas presented in the disk cache model to model the file system cache.

Disk response time. The mean disk response time is the sum of disk overhead, disk head positioning time, and time to transfer the data from disk to the file system cache:

$$\begin{aligned}\mathsf{DRT} = {} & \mathsf{DiskOverhead} + \mathsf{PositionTime} + \\ & \mathbf{E}[\mathsf{disk\_request\_size}] / \min\{\mathsf{BusTR}, \mathsf{DiskTR}\}.\end{aligned}$$

(Note: $\mathbf{E}[x]$ denotes the expected, or average value for x.) The amount of time spent positioning the disk head, $\mathsf{PositionTime}$, depends on the current location of the disk head, which is determined by the previous request. For example, if this is the first request for a block in a given cluster, $\mathsf{PositionTime}$ will include both seek time and time for the rotational latency. Let $\mathbf{E}[\mathsf{SeekTime}]$ be the mean seek time and $\mathbf{E}[\mathsf{RotLat}]$ be the mean rotational latency (1/2 the time for a full disk rotation). Thus, the disk response time for the first request in a cluster is

$$\begin{aligned}\mathsf{DRT}[\text{random request}] = {} & \mathsf{DiskOverhead} + \\ & \mathbf{E}[\mathsf{SeekTime}] + \mathbf{E}[\mathsf{RotLat}] + \\ & \frac{\mathbf{E}[\mathsf{disk\_request\_size}]}{\min\{\mathsf{BusTR}, \mathsf{DiskTR}\}}. \qquad (2)\end{aligned}$$

If the previous request was for a block in the same cylinder group, the seek distance will be small. This will be the case if the previous read was to a portion of the file stored in the same cylinder group, or to some other file found in the same cylinder group. If there are n files being accessed concurrently, the expected seek distance will either be (a) $\mathsf{Max\_Cylinder}/3$, if the device driver and disk controller request queues are empty, or (b) (assuming the disk scheduler is using an elevator algorithm) $\mathsf{Max\_Cylinder}/(n + 2)$ [Shriver97].

The mean disk request size, $\mathbf{E}[\mathsf{disk\_request\_size}]$, can be computed by averaging the request sizes; these can be computed by simulating the algorithm to determine the amount of data prefetched, where the simulation stops when the amount of accessed data is equal to $\mathsf{ClusterSize}$. If the file system is servicing more than one file, the actual amount prefetched can be smaller than expected due to blocks being evicted before use. If the file system is not prefetching data, the $\mathbf{E}[\mathsf{disk\_request\_size}]$ is the file system block size, $\mathsf{BlockSize}$.

Sometimes the requested data are in the disk cache due to readahead; in these cases, the disk response time is

$$\begin{aligned}\mathsf{DRT}[\text{cached request}] = {} & \mathsf{DiskOverhead} + \\ & \mathbf{E}[\mathsf{disk\_request\_size}]/\mathsf{BusTR}. \qquad (3)\end{aligned}$$

[1]The previous disk model includes additional workload parameters that support specification of spatial locality; these are not needed for our current model since we assume that the files are accessed sequentially. The earlier disk model also supports a read fraction parameter; in this paper, we only model file reads.

Table 2: Workload specification.

parameter	*definition*	*unit*
temporal locality measures		
request_rate	rate at which requests arrive at the storage device	requests/second
cylinder_group_id	cylinder group (location) of the file	integer
arrival_process	inter-request timing (constant [open, closed], Poisson, or bursty)	—
spatial locality measures		
data_span	the span (range) of data accessed	bytes
request_size	length of a host read or write request	bytes
run_length	length of a *run*, a contiguous set of requests	bytes

File system response time. We first compute the amount of time needed for all of the file system accesses TotalFSRT, and then compute the mean response time for each access, FSRT, by averaging:

$$\mathsf{FSRT} = \frac{\mathsf{data\_span}}{\mathsf{request\_size}}\mathsf{TotalFSRT}. \qquad (4)$$

The rest of this section discusses approximating TotalFSRT.

Let us first look at the simplest case: reading one file that resides entirely in one cluster, the mean response time to read the cluster contains file system overhead plus the time needed to access the data from disk:

$$\begin{aligned}\mathsf{ClusterRT} = {} & \mathsf{FSOverhead} + \\ & \mathsf{DRT}[\text{first request}] + \\ & \sum_i \mathsf{DRT}[\text{remaining request}_i]\end{aligned}$$

where the first request and remaining requests are the disk requests for the blocks in the cluster and $\mathsf{DRT}[\text{first request}]$ is from equation (2). If n files are being serviced at once, the $\mathsf{DRT}[\text{remaining request}_i]$'s each contain $\mathbf{E}[\mathsf{SeekTime}] + \mathbf{E}[\mathsf{RotLat}]$ if n is more than CacheSegments, the number of disk cache segments. If not, some of the data will be in disk cache and equation (3) is used. The FSOverhead can be measured experimently or computed as $\mathsf{SystemCallOverhead} + \mathbf{E}[\mathsf{request\_size}]/\mathsf{MemoryCopyRate}$. The number of requests per cluster can be computed as data_span/disk_request_size.

If the files span multiple clusters, we have

$$\mathsf{TotalFSRT} = \mathsf{NumClusters} \cdot \mathsf{ClusterRT}$$

where we approximate the number of clusters as $\mathsf{NumClusters} = \mathsf{data\_span}/\mathsf{ClusterSize}$. To capture the "extra" cluster due to only the first DirectBlocks blocks being stored on the same cluster, this value is incremented by 1 if (ClusterSize/BlockSize)/DirectBlocks is not 1 and $\mathsf{data\_span}/\mathsf{BlockSize} > \mathsf{DirectBlocks}$.

If the device driver or disk controller scheduling algorithm is CLOOK or CSCAN and the queue is not zero, then there is a large seek time (for CLOOK) or a full stroke seek time (for CSCAN) for each group of n accesses, when n is the number of files being serviced by the file system; we call this time extra_seek_time.

If the n files being read are larger than DirectBlocks, we must include the time required to read the indirect block:

$$\begin{aligned}\mathsf{TotalFSRT} = {} & n \cdot \mathsf{NumClusters} \cdot \mathsf{ClusterRT} + \\ & \mathsf{num\_requests} \cdot \mathsf{extra\_seek\_time} + \\ & \mathsf{DRT}[\text{indirect block}] \qquad (5)\end{aligned}$$

where num_requests is the number of disk requests in a file. Since the location of the indirect block is on a random cylinder group, equation (2) is used to compute $\mathsf{DRT}[\text{indirect block}]$. Of course, if the file contains more blocks than can be referenced by both the inode and the indirect block, multiple indirect block terms are needed.

5 Discussion

In the introduction to this paper, we listed the reasons that prefetching improves performance: the disk cache comes into play, the device driver amortizes the fixed cost of an I/O over a larger amount of data, and total disk seek time can be decreased. In this section we discuss the terms introduced by our model and attempt to explain where the time goes, and when and why prefetching works. To do this, we collected detailed traces of a variety of workloads. These traces allowed us to compute the file system and disk response times experienced by the test machines. These response times were also used in our validations as discussed in Appendix A.

Hardware setup and trace gathering. We performed experiments on two hardware configurations: a 200 MHz Pentium Pro processor and a 450 MHz Pentium II processor. Each machine had 128 MB of main memory. We conducted all of our tracing and measurements on a 4 GB Seagate ST34501W (Cheetah) disk, connected via a 10 MB/second PCI SCSI controller (for the 200 MHz processor) or via a 20 MB/second PCI SCSI controller (for the 450 MHz processor). Our test machines were running version 3.1 of the BSD/OS operating system. The file system parameters for our test file systems are found in Table 1 and the disk parameters are in Section 4.2.

We collected our traces by adding trace points to the BSD/OS kernel. At each trace point the kernel wrote a record to an in-memory buffer describing the type of event that had occurred and any related data. The kernel added a time stamp to each trace record, using the CPU's on-chip cycle counter. A user-level process periodically read the contents of the trace buffer.

To document the response time for application-level read requests, we used a pair of trace points at the entry and exit of the FFS read routine. Similarly, we measured disk-level response times using a pair of trace points in the SCSI driver, one when a request is issued to the disk controller, and a second when the disk controller indicates that the I/O has completed. Additional trace points documented the exact sequence (and size) of the prefetch requests issued by the file system, and the amount of time each request spent on the operating system's disk queue.

The numbers discussed in this section are for the machine with the faster bus unless stated otherwise.

Disk cache. When an application makes a read request of the file system, the file system checks to see if the requested data are in its cache, and if not, issues an I/O request to the disk. The data will be found in the file system cache if they were prefetched and have not been evicted. If the data are not in the file system's cache, the file system must read it from the disk. There are two possible scenarios:

1. The data are in the disk cache as a result of readahead for a previous command, so the disk does not need to read the data again. The disk sends the data directly from the disk cache.
2. The data are not in the disk cache and must be read from the disk surface.

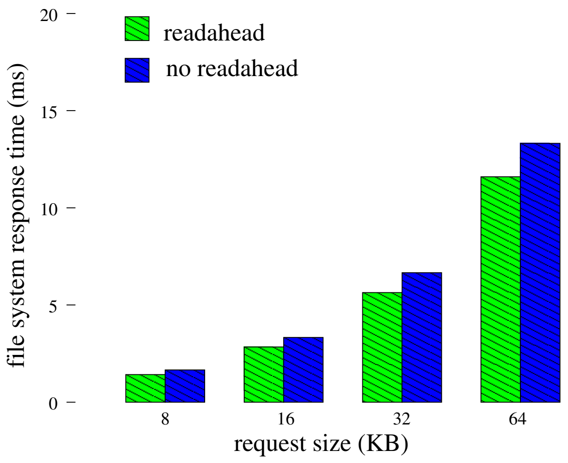

Figure 1: File system response times for a 64 KB file with and without the disk cache performing readahead. The readahead values are measured; the no-readahead values are predicted by the model.

In an attempt to quantify the effect of the disk cache, Figures 1 and 2 contain the file system response time measurements with the disk cache performing readahead and file system response time predictions without the disk cache performing readahead. The percent improvement in the response time when the disk cache is performing readahead is 17–23%.

Modern disks are capable of caching data for concurrent workloads, where each workload is reading a region of the disk sequentially. If there are enough cache segments for the current number of sequential workloads, the disk will readahead for each workload, and each workload will benefit. However, if there are more sequential workloads than cache segments, depending on the cache replacement algorithm used by the disk, the disk's ability to prefetch may have little or no positive effect on performance. In addition, disk readahead is only valuable when the file system prefetch size is less than the cluster

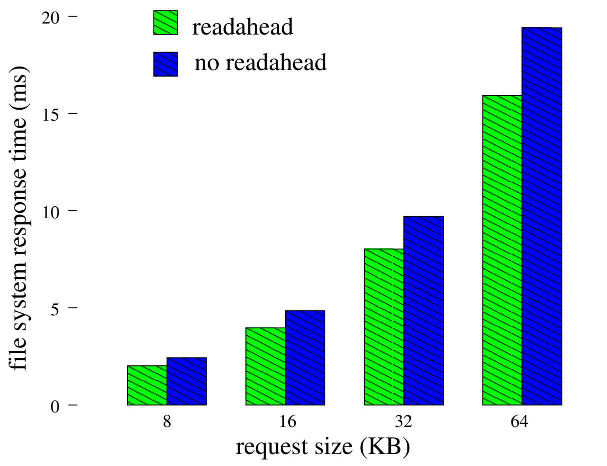

Figure 2: File system response times for a 128 KB file with and without the disk cache performing readahead. The readahead values are measured; the no-readahead values are predicted by the model.

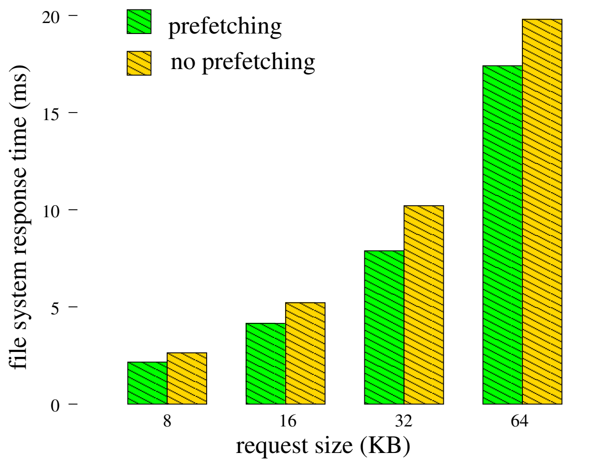

Figure 3: Measured file system response times for a 64 KB file with and without the file system performing prefetching.

size, since, after that, the entire cluster is fetched with one disk access. In the case of our file system parameters, this occurred when the file was 128 KB or smaller.

I/O cost amortization. On our slow bus configuration, we measured the disk overhead of performing an I/O operation at 1.2 to 1.8 ms, which is on the same order as the time to perform a half-rotation (3 ms). The measured transfer rate of the bus is 9.3 MB/s; by saving an I/O operation, we can transfer an additional 11 to 16 KB. As an example, assume that 64 KB of data will be used by the application. If the requested data are in the disk cache, using a file block of 8 KB will take at least 14.1 ms (1.8 ms overhead four times + 6.9 ms for data transfer);[2] a file block of 64 KB will take 8.7 ms (1.8 ms overhead + 6.9 ms for data transfer), just a little over half the I/O time.

The impact of I/O cost amortization can be seen when comparing the measured file system response time when servicing one file, with and without prefetching. Figure 3 show these times for the slower hardware configuration. With prefetching disabled, the file system requests data in BlockSize units, increasing the number of requests, and the amount of disk and file system overhead. The additional overheads increase the resulting performance by 13–29%.

[2]With the file system performing prefetching, there will be 4 disk requests having a mean disk request size of 16 KB.

When we ran our tests on the machine with the faster bus, we noted anomalous disk behavior that we do not yet understand. According to our measurements, it should (and does, in most cases) take roughly 0.78 ms to read an 8 KB block from the disk cache over the bus on this machine. However, under certain circumstances, 8 KB reads from the disk cache complete more quickly, taking roughly 0.56 ms.[3] This happened only when file system prefetching is disabled, i.e., when the file system requests multiple consecutive 8 KB blocks, and when there is only one application reading from the disk. The net result is that, in these rare situations, performance is slightly *better* with file system prefetching disabled. This behavior also was displayed with the slower bus, but as you can see in Figure 3, the bus is slow enough so that the response time with prefetching is smaller than the response time without prefetching.

Seek time reduction. As the number of active workloads increases, the latency for each workload will increase, but the disk throughput can, paradoxically, increase as well. Due to the type of scheduling algorithms used for the device driver queue, more elements in the read queue can mean smaller seeks between each read, and hence greater disk throughput. On the other hand, a longer queue means that each request will, on average, spend more time in the queue, and thus the read latency will be greater.

[3]We are in communication with Seagate in an attempt to determine why we are seeing this behavior.

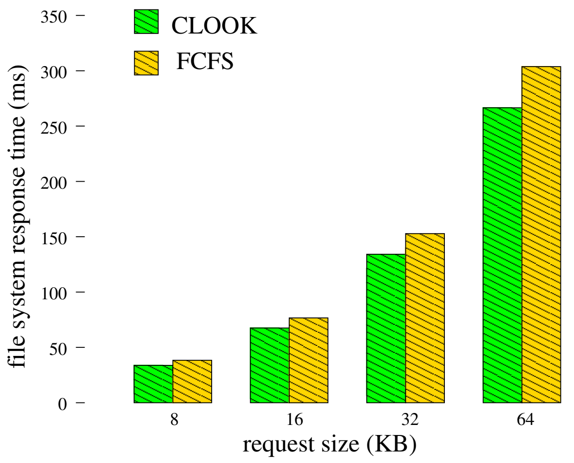

Figure 4: Measured file system response times for 8 64 KB files with the device driver using a CLOOK scheduling algorithm with a FCFS scheduling algorithm.

Figure 4 displays the file system response time with the device driver implementing the CLOOK scheduling algorithm (the standard algorithm), and implementing FCFS, which will not reduce the seek time. The performance gain from using CLOOK over FCFS is 14%.

I/O / computation overlap. As was discussed in Section 1, if an application performs substantial computation as well as I/O, prefetching may allow the application to overlap the two, increasing application throughput and decreasing file system response time. For example, on our test hardware, computing the MD5 checksum of a 10 KB block of data takes approximately one millisecond. A program reading data from the disk and computing the MD5 checksum will exhibit delays between successive read requests, giving the file system time to prefetch data in anticipation of the next read request. Figure 5 shows the file system response times with a request size of 10 KB for files of varying lengths. The figure shows the response time given no delay (representing the application having no I/O / computation overlap), with an application delay of 0.5 ms, and with an application delay of 1.0 ms (as with MD5). As the file size increases, so do the savings due to prefetching. With a 64 KB file there is a 36% improvement, compared to a 114% improvement when reading a 512 KB file.

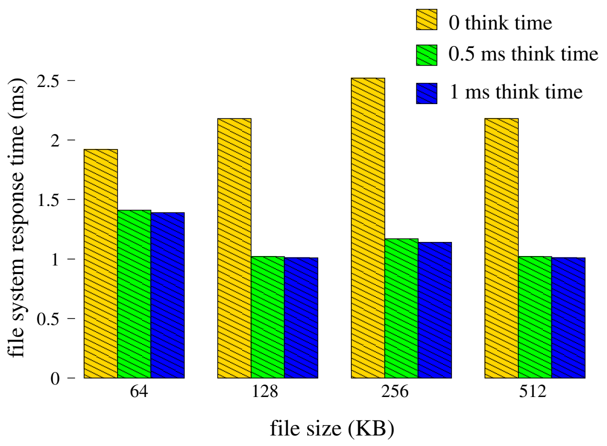

Figure 5: Measured file system response times with the application think time of 0 ms, 0.5 ms, and 1 ms.

Summary. In this section we showed how the components of prefetching combine to improve performance. In our tests, disk caching provided a 17–23% boost, I/O cost amortization yielded, 13–29%, seek time reduction (CLOOK *vs.* FCFS) 14%, and overlapping computation and I/O can save as much as 50%. Depending on the behavior of the specific application, these components can have a greater or lesser benefit; added together, the performance gain is significant.

6 Conclusions

We developed an analytic model that predicts the response time of the file system with a maximum relative error of 9% (see Appendix A). Given the wide range of conceivable file system layouts and prefetching policies, an accurate analytic model simplifies the task of setting system parameters that may improve performance enough to be worth implementing and studying in more detail.

Our model has allowed us to develop two suggestions for decreasing the file system response time. If it is reasonable to assume that the prefetched data will be used, and we have room in the file system cache, once the disk head has been positioned over a cluster, the entire cluster should be read. This will decrease the file system and disk overheads.

The number of disk cache segments restricts the number of sequential workloads for which the disk

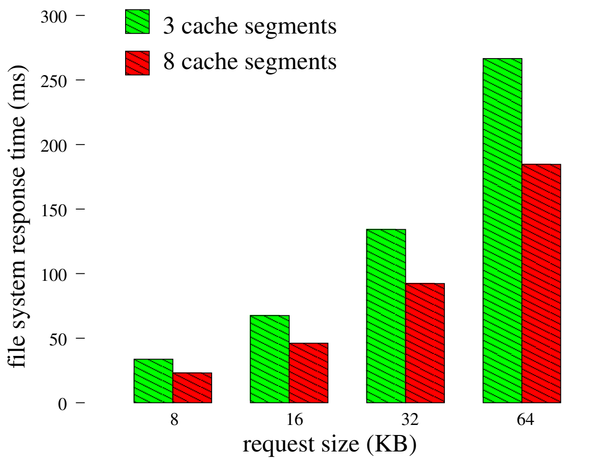

Figure 6: File system response times for 8 64 KB files with 3 and 8 segments. 3-segment values are measured and 8-segment values are predicted by the model.

cache can perform readahead, which means that if the number of disk cache segments is smaller than the number of concurrent workloads, it can be as if the disk had no cache at all. One enhancement that we suggest is for the file system to dynamically modify the number of disk cache segments to be the number of files being concurrently accessed from that disk. This is a simple and inexpensive SCSI operation, and can mean the difference between having the disk cache work effectively and having it not work at all. Figure 6 compares the measured file system response time when servicing 8 workloads with 3 cache segments and the predicted response time with 8 cache segments. This shows us a 44–46% decrease in the response time when the number of cache segments is set to the number of concurrent workloads.

Our current model does not handle file system writes; we would like to extend it to support writes. We would like to use our analytic model to compare different file system prefetching polices and parameter settings to determine the "best" setting for a particular workload. Workloads which seem promising are web server, scientific workloads [Miller91], and database benchmarking workloads.

Acknowledgments. Many thanks to Bruce Hillyer for many interesting conversations.

References

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Asilomar (Pacific Grove), CA), pages 198–212. ACM Press, October 1991.

[Barve99] Rakesh Barve, Elizabeth Shriver, Phillip B. Gibbons, Bruce K. Hillyer, Yossi Matias, and Jeffrey Scott Vitter. Modeling and optimizing I/O throughput of multiple disks on a bus. *Proceedings of the 1999 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems* (Atlanta, GA), May 1999. Available at `http://www.bell-labs.com/~shriver/`.

[Bastian82] A. L. Bastian. Cached DASD performance prediction and validation. *Proceedings of 13th International Conference on Management and Performance Evaluation of Computer Systems* (San Diego, CA), pages 174–177, Mel Boksenbaum, George W. Dodson, Tom Moran, Connie Smith, and H. Pat Artis, editors, December 1982.

[Cao94] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA), pages 165–178, November 1994.

[Cao95] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems* (Ottawa, Canada), pages 188–197, May 1995.

[Chen93] Shenze Chen and Don Towsley. The design and evaluation of RAID 5 and parity striping disk array architectures. *Journal of Parallel and Distributed Computing*, **17**(1–2):58–74, January–February 1993.

[Feiertag72] R. J. Feiertag and E. I. Organick. The Multics input/output system. *Proceedings of the Third Symposium on Operating Systems Principles* (Palo Alto, CA), pages 35–41, October 1972.

[Griffioen94] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. *Proceedings of the 1994 Summer USENIX Technical Conference* (Boston, MA), pages 197–207, June 1994.

[Hartman93] John Hartman and John Ousterhout. Letter to the editor. *Operating Systems Review*, **27**(1):7–9, January 1993.

[Hofri80] M. Hofri. Disk scheduling: FCFS vs. SSTF revisited. *Communications of the ACM*, **23**(11):645–653, November 1980.

[Hospodor95] Andy Hospodor. Mechanical access time calculation. *Advances in Information Storage Systems*, **6**:313–336, 1995.

[Kimbrel96] Tracy Kimbrel and Anna R. Karlin. Near-optimal parallel prefetching and caching. *Proceedings of the 37th Annual Symposium on Foundations of Computer Science* (Burlington, VT), pages 540–549, October 1996.

[Kotz95] David Kotz. Disk-directed I/O for an out-of-core computation. *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing* (Pentagon City, VA), pages 159–166, August 1995.

[Kroeger96] Thomas Kroeger. Predicting file system actions from reference patterns. Department of Computer Engineering, University of California, Santa Cruz, Santa Cruz, CA, December 1996. Master's thesis.

[Kuratti95] Anand Kuratti and William H. Sanders. Performance analysis of the RAID 5 disk array. *Proceedings of International Computer Performance and Dependability Symposium* (Erlangen, Germany), pages 236–245, April 1995.

[Lei97] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. *Proceedings of the USENIX 1997 Annual Technical Conference* (Anaheim, CA), January 1997.

[Madhyastha97a] Tara M. Madhyastha and Daniel A. Reed. Exploiting global input/output access pattern classification. *Proceedings of Supercomputing '97* (San Jose, CA), November 1997.

[Madhyastha97b] Tara M. Madhyastha and Daniel A. Reed. Input/output access pattern classification using hidden Markov models. *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS)* (San Jose, CA), pages 57–67. ACM Press, November 1997.

[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):181–197, August 1984.

[McKusick96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing, 1996.

[Merchant96] Arif Merchant and Philip S. Yu. Analytic modeling of clustered RAID with mapping based on nearly random permutation. *IEEE Transactions on Computers*, **45**(3):367–373, March 1996.

[Miller91] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputing applications. *Proceedings of Supercomputing '91* (Albuquerque, NM), pages 567–576, November 1991.

[Mowry96] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation* (Seattle, WA), pages 3–17. USENIX Association, October 1996.

[Ng91] Spencer W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, **40**(1):22–30, January 1991.

[Patterson95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, CO), pages 79–95. ACM Press, December 1995.

[Ritchie78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, **57**(6):1905–1930, July/August 1978. Part 2.

[Ruemmler94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994.

[Seeger96] B. Seeger. An analysis of schedules for performing multi-page requests. *Information Systems*, **21**(5):387–407, July 1996.

[Seltzer90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Proceedings of Winter 1990 USENIX Conference* (Washington, DC), pages 313–323, January 1990.

[Shriver97] Elizabeth Shriver. *Performance modeling for realistic storage devices*. PhD thesis. Department of Computer Science, New York University, New York, NY, May 1997. Available at `http://www.bell-labs.com/~shriver/`.

[Shriver98] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. *Joint International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '98/Performance '98)* (Madison, WI), pages 182–191, June 1998. Available at `http://www.bell-labs.com/~shriver/`.

[Small98] Christopher Small. *Building an extensible operating system*. PhD thesis. Division of Engineering and Applied Sciences, Harvard University, Boston, MA, October 1998.

[Wilhelm77] Neil C. Wilhelm. A general model for the performance of disk systems. *Journal of the ACM*, **24**(1):14–31, January 1977.

[Worthington94] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA), pages 241–251, May 1994.

A Validation of the model

To validate the model, we ran a set of simple microbenchmark workloads, collecting traces of both file system and disk events. From these traces, we determined the mean disk and application-level response times for a variety of synthetic and real workloads. The results in this section are presented for the machine with the slower bus.

The workloads. We created simple workloads which opened files, read them sequentially, and closed them. We recorded only the reads, i.e., not the open and close operations. We varied the request size and size of the file (which, in turn, varied $\mathsf{data\_span}$ and $\mathsf{run\_length}$). Our workloads have a closed arrival process; files greater than 96 KB spanned at least two cylinder groups.

Single files accessed. We ran our workloads using one process to access a single file; we repeated the experiments 100 times and averaged to compute the measured FSRT. Table 3 includes the measured and model-computed FSRT using equations (4) and (5), as well as the relative errors of the model. In all but one case the model's estimate is within 4% of the measured result; when reading 32 KB of a 64 KB file the model error is 9%, which is higher, but still quite good for an analytic model of this type.

Multiple files accessed. We then reran our experiments with multiple concurrent processes, each accessing a different file with the same workload specification (i.e., same request size and same file size). All of the files were opened at the beginning of the experiment; the files which were used during one experiment run were randomly chosen among a set of files that spanned the 4 GB disk. We flushed the disk cache between each of the 50 experiment runs. Table 4 presents our results using equations (4) and (5). We see that the model's error is 6% or less in all cases.

Prefetching disabled. We modified the file system to disable prefetching and reran our single file experiments. Table 5 contains our results using equations (4) and (5) with $\mathbf{E}[\mathsf{disk\_request\_size}] = \mathsf{BlockSize}$. The maximum relative error is 7%.

Table 3: Relative errors with one file accessed.

request size (KB)	*file size* (KB)	*measured* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*computed* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*error*
8	64	2.16	2.16	0%
16		4.15	4.31	4%
32		7.88	8.62	9%
64		17.41	17.24	1%
8	128	2.70	2.69	1%
16		5.41	5.37	1%
32		10.74	10.74	0%
64		21.08	21.48	2%

Table 4: Relative errors with multiple files accessed.

number of files	*request size* (KB)	*file size* (KB)	*measured* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*computed* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*error*
2	8	64	8.80	8.80	0%
	16		17.27	17.61	2%
	32		34.89	35.22	1%
	64		68.96	70.43	2%
3	8	64	12.82	12.90	1%
	16		25.60	25.81	1%
	32		50.85	51.61	1%
	64		99.98	103.23	3%
4	8	64	24.61	23.09	6%
	16		49.16	46.17	6%
	32		98.10	92.35	6%
	64		193.86	184.70	5%
8	8	64	43.83	43.52	1%
	16		87.54	87.04	1%
	32		174.06	174.08	0%
	64		344.04	348.16	1%
16	8	64	77.96	79.50	2%
	16		155.48	159.00	2%
	32		308.20	318.01	3%
	64		604.90	636.01	5%

Table 5: Relative errors with one file accessed and prefetching disabled.

request size (KB)	*file size* (KB)	*measured* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*computed* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*error*
8	64	2.64	2.46	7%
16	64	5.22	4.93	6%
32	64	10.21	9.86	3%
64	64	19.80	19.72	0%
8	128	3.26	3.07	6%
16	128	6.59	6.14	7%
32	128	13.21	12.28	7%
64	128	26.19	24.56	6%

Why does file system prefetching work?

Elizabeth Shriver
Information Sciences Research Center
Bell Labs, Lucent Technologies
shriver@research.bell-labs.com

Christopher Small
Information Sciences Research Center
Bell Labs, Lucent Technologies
chris@research.bell-labs.com

Keith A. Smith
Harvard University
keith@eecs.harvard.edu

Abstract

Most file systems attempt to predict which disk blocks will be needed in the near future and prefetch them into memory; this technique can improve application throughput as much as 50%. But why? The reasons include that the disk cache comes into play, the device driver amortizes the fixed cost of an I/O operation over a larger amount of data, total disk seek time can be decreased, and that programs can overlap computation and I/O. However, intuition does not tell us the relative benefit of each of these causes, or techniques for increasing the effectiveness of prefetching.

To answer these questions, we constructed an analytic performance model for file system reads. The model is based on a 4.4BSD-derived file system, and parameterized by the access patterns of the files, layout of files on disk, and the design characteristics of the file system and of the underlying disk. We then validated the model against several simple workloads; the predictions of our model were typically within 4% of measured values, and differed at most by 9% from measured values. Using the model and experiments, we explain why and when prefetching works, and make proposals for how to tune file system and disk parameters to improve overall system throughput.

1 Introduction

Previous work has shown that most file reads are sequential [Baker91]. Optimizing for the common case, modern file systems take advantage of this fact and prefetch blocks from disk that have not yet been requested, but are likely to be needed in the near future. This technique is effective for several reasons:

- There is a fixed cost associated with performing a disk I/O operation. By increasing the amount of data transfered on each I/O, the overhead is amortized over a larger amount of data, improving overall performance.
- Modern disks contain a disk cache, which contains some number of disk blocks from the cylinders of recent requests. If multiple blocks are read from the same track, all but the first may, under certain circumstances, be satisfied by the disk cache without accessing the disk surface.
- The device driver or disk controller can sort disk requests to minimize the total amount of disk head positioning done. With a larger list of disk requests, the driver or controller can do a better job of ordering them to minimize disk head motion. Additionally, the blocks of a file are often clustered together on the disk; when this is so, multiple blocks of the file can be read at once without an intervening seek.
- If the application performs substantial computation as well as I/O, prefetching may allow the application to overlap the two, which would increase the application's throughput. For example, an MPEG player may spend as much time computing as it does waiting for I/O; if the file system can run ahead of the MPEG player, loading data into memory before they are needed, the player will not block on I/O.

If an application spends as much time performing I/O as it does computing—successfully prefetching data will allow overlapping the two and the application's throughput will double.

This paper presents and validates an analytic file system performance model that allows us to explain why and when prefetching works, and makes recommendations for how to improve the benefit of file prefetching. The model described here is restricted in two ways.

First, it addresses only file read traffic, and does not capture file writes or file name lookup. Although this restricts the applicability of the model, we believe that there are many interesting workloads covered by the model. For example, many workloads consist almost entirely of reads, such as web servers, read-mostly database systems, and file systems that store programs, libraries, and configuration files (e.g., /, /etc, and /usr). Additionally, studies have shown that, for some engineering workloads, 80% of file requests are reads [Baker91, Hartman93].

It has long been the case that in order to improve performance, file systems use a *write-back* cache, delaying writes for some period of time [Feiertag72, Ritchie78, McKusick84]. The results of the study by Baker and associates showed that with a 30-second write-back delay, 36% to 63% of the bytes written to the cache do not survive, and by increasing the write-back delay size to 1000 seconds, 60% to 95% do not survive.

Second, although our model includes multiple concurrent programs accessing the file system, we limit it to workloads where each file is read from start to finish with little or no *think time* between read requests. Although this does not cover workloads such as the MPEG player discussed above, we believe that, given the relative speed of modern processors and I/O devices, in the common case think time is immeasurable. The Baker study showed that 75% of files were open less than a quarter of a second [Baker91]. This study was done on a collection of machines running at around 25 MHz (SPARCStation 1, Sun 3, DECStation 3100, DECStation 5000); on modern machines, with clock speeds an order of magnitude faster, think time should be substantially lower.

Outline of paper. Section 2 discusses in detail the modeled file system (the 4.4BSD Fast File System) and disk. Section 3 presents previous work in file systems and prefetching policies. The analytic model of file system response time and its validation are presented in Section 4 and Appendix A. Section 5 explains why and when prefetching works. We summarize our work and discuss future work in Section 6.

2 Background

In this section we describe in detail the behavior of the file system we model, and discuss the characteristics of modern disks.

2.1 The file system

The file system that we used as the basis for our model is the 4.4BSD implementation of the Berkeley Fast File System that ships with BSD/OS 3.1 [McKusick96].

Reading files. An application can make a request for an arbitrarily large amount of data from a file. To process an application-level read request, the file system divides the request into one or more block-sized (and block-aligned) requests, each of which the file system services separately. A popular file system block size is 8 KB, although other block sizes can be specified when the file system is initialized.

For each block in the request, the file system first determines whether the block is already in the operating system's in-memory cache. If it is, then the block is copied from the cache to the application. If the block is not already in memory, the file system issues a read request to the disk device driver.

Regardless of whether the desired block is already in memory, the file system may prefetch one or more subsequent blocks from the file. The amount of data the file system prefetches is determined by the file system's prefetch policy, and is a function of the current file offset and whether or not the application has been accessing the file sequentially. A read of block x from a file is *sequential* if the last block read from that file was either x or $x - 1$. By treating successive reads of the same block as "sequential," applications are not penalized for using a read size that is less than the file system's block size.

As read requests are synchronous, the operating system blocks the application until all of the data it has requested are available. Note that a single disk request may span multiple blocks and include both the requested data and prefetch data, in which case the application can not continue until the entire request completes.

Data placement on disk. A *cluster* is a group of logically sequential file blocks that are stored sequentially on disk; the *cluster size* is the number of bytes in the cluster. Depending on the file system parameters, the file system may place successive allocations of clusters contiguously on the disk. This can result in contiguous allocations of hundreds of kilobytes in size.

The blocks of a file are indexed by a tree structure on disk; the root of the tree is an *inode.* The inode contains the disk addresses to the first few blocks of a file (i.e., the first DirectBlocks blocks of the file); in the case of the modeled system, the inode contains pointers to the first twelve blocks. The remaining blocks are referenced by *indirect blocks.*

The first block referenced from an indirect block is always the start of a new cluster. This may cause the preceding cluster to be smaller than the file system's cluster size. For example, if DirectBlocks is not a multiple of the cluster size, the last cluster of direct blocks may be smaller than the cluster size.

The file system divides the disk into cylinder groups, which are used as allocation pools. Each cylinder group contains a fixed sized number of blocks (2048 blocks, or 16 MB on the modeled system). The file system exploits expected patterns of locality of reference by co-locating related data in the same cylinder group.

The file system usually attempts to allocate clusters for the same file in the same cylinder group. Each cluster is allocated in the same cylinder group as the previous cluster. The file system attempts to space clusters according to the value of the rotational delay parameter which is set using the `newfs` or `tunefs` command. The file system can always achieve this spacing on an empty file system. If the free space on the file system is fragmented, however, this spacing may vary. The file system allocates the first cluster of a file from the same cylinder group as the file's inode. Whenever an indirect block is allocated to a file, allocation for the file switches to a different cylinder group. Thus an indirect block and the clusters it references are allocated in a different cylinder group than the previous part of the file.

Prefetching in the file system cache. If the requested data are not in cache, the file system issues a disk request for the desired block. If the application is accessing the file sequentially, the file system may prefetch one or more additional data blocks. The amount of data prefetched is doubled on each disk read, up to a maximum of the cluster size. The last block of a file may be allocated to a *fragment* rather than a full size block. When this happens, the final fragment of the file is not prefetched.

It is possible for a block to be prefetched and then evicted before it is requested. If the user subsequently requests such a block, the file system assumes that it is prefetching too aggressively and cuts the prefetch size in half.

2.2 The disk

When a disk request is issued from the file system, it enters the device driver. If the disk is busy, the request is put on a queue in the device driver; the queue is sorted by a scheduling algorithm that attempts to improve response times. One commonly-used class of scheduling algorithms are the *elevator* algorithms, where the requests are serviced in the order that they appear on the disk tracks. CLOOK and CSCAN are examples of elevator algorithms. Once the request reaches the head of the queue, the request is sent to the bus controller which gains control of the bus. The request is then sent to the disk, and might be queued there if the disk mechanism is busy. This queue is also sorted to improve response time; one commonly-used scheduling algorithm is Shortest Positioning Time First, which services requests in an order intended to minimize the sum of the *seek time* (i.e., the time to move the head from the current track to the desired track) and the *rotational latency* (i.e., the time needed for the disk to rotate to the correct sector once the desired track is reached).

Seek time. Seek is the time for the actuator to move the disk arm to the desired cylinder. A seek operation can be decomposed into:

- *speedup*, where the arm is accelerated until it reaches half of the seek distance or a fixed maximum velocity,
- *coast* for long seeks, where the arm is moving at maximum velocity,
- *slowdown*, where the arm is brought to rest close to the desired track, and
- *settle*, where the disk controller adjusts the head to access the desired location.

Very short seeks (2–4 cylinders) are dominated by the settle time. Short seeks (less than 200–400 cylinders) are dominated by the speedup, which is proportional to the square root of seek distance. Long seeks are dominated by the coast time, which is proportional to the seek distance. Thus, the seek time can be approximated by a function such as

$$\mathsf{Seek\_Time}[\mathsf{dis}] = \begin{cases} 0 & \mathsf{dis} = 0 \\ a + b\sqrt{\mathsf{dis}} & 0 < \mathsf{dis} \le e \\ c + d\,\mathsf{dis} & \mathsf{dis} > e \end{cases} \qquad (1)$$

where a, b, c, d, and e are device-specific parameters and dis is the number of cylinders to be traveled. Single cylinder seeks are often treated specially.

The disk cache. When the request reaches the head of the queue, the disk checks its cache to see if the data are in cache. If not, the disk mechanism moves the disk head to the desired track (seeking) and waits until the desired sector is under the head (rotational latency). The disk then reads the desired data into the disk cache. The disk controller then contends for access to the bus, and transfers the data to the host from the disk cache at a rate determined by the speed of the bus controller and the bus itself. Once the host receives the data and copies them into the memory space of the file system, the system awakens any processes that are waiting for the read to complete.

The disk cache is used for multiple purposes. One is as a pass-through speed-matching buffer between the disk mechanism and the bus. Most disks do not retain data in the cache after the data have been sent to the host. A second purpose is as a readahead buffer. Data can be readahead into the disk cache to service future requests. Most frequently, this is done by the disk saving in a cache segment the data that comes after the requested data. Modern disks such as the Seagate Cheetah only readahead data when the requested addresses suggest that a sequential access pattern is present.

The disk cache is divided into *cache segments*. Each segment contains data prefetched from the disk for one sequential stream. The number of cache segments usually can be set on a per-disk basis; the typical range of allowable values is between one and sixteen.

Disk performance is covered in more detail by Shriver [Shriver97] and by Ruemmler and Wilkes [Ruemmler94].

3 Related work

Prefetching. Prefetching is not a new idea; in the 1970's, Multics supported prefetching [Feiertag72], as did Unix [Ritchie78]. Earlier work has focused on the benefit of prefetching, either by allowing applications to give prefetching hints to the operating system [Cao94, Patterson95, Mowry96], or by automatically discovering file access patterns in order to better predict which blocks to prefetch [Griffioen94, Lei97, Kroeger96]. Techniques studied have included neural networks [Madhyastha97a] and hidden Markov models [Madhyastha97b]. Our work differs from this work in three ways. First, we address only common case workloads that have sequential access patterns. Second, our model is parameterized by the file system's behavior such as caching strategy and file layout, and takes into account the behavioral characteristics of the disks used to store files. Third, our model predicts the performance of the file system.

Substantial work has been done studying the interaction between prefetching and caching [Cao95, Patterson95, Kimbrel96]. Others have examined methods to work around the file system cache to achieve the desired performance (e.g., [Kotz95]).

The benefit of prefetching is not limited to workloads where files are read sequentially; Small studied the effect of prefetching on random-access, zero think time workloads on the VINO operating system, and showed that even with these workloads the performance gain from prefetching was more than 20% [Small98].

Disk modeling. Much work has been done in disk modeling. The usual approach to analyzing detailed disk drive performance is to use simulation (e.g., [Hofri80, Seltzer90, Worthington94]). Most early modeling studies (e.g., [Bastian82, Wilhelm77]) concentrated on rotational position sensing for mainframe disk drives, which had no cache at the disk and did no readahead. Most prior work has not been workload specific, and has, for example, assumed that the workload has uniform random spatial distributions (e.g., [Seeger96, Ng91, Merchant96]). Chen and Towsley, and Kuratti and Sanders, modeled the probability that no seek was needed [Chen93, Kuratti95]; Hospodor reported that an exponential distribution of seek times matched measurements well for three test workloads [Hospodor95]. Shriver and colleagues, and Barve and associates present analytic models for modern disk drives, representing readahead and queueing effects across a range of workloads [Shriver97, Shriver98, Barve99].

4 The analytic model

In this section, we present the file system, disk, and workload parameters that we need for our model. As we present the needed file system and disk parameters, we also give the values for the platforms which we used to validate the model. We close this section with presenting the details of the model.

We used two platforms; one with a slow bus (i.e., 10 MB/s), and one with a fast bus (i.e., 20 MB/s). Details of our test/experiment platforms are in Section 5.

4.1 File system specification

Based on our understanding of the file system cache policies, we determined a set of parameters that allow us to capture the performance of the file system cache; these can be found in Table 1. For our fast machine, the SystemCallOverhead value was 5 μs and the MemoryCopyRate was 5 μs/KB.

4.2 Disk specification

To predict the disk response time, we need to know several parameters of the disk being used.

- DiskOverhead includes the time to send the request down the bus and the processing time at the controller, which is made up of the time required for the controller to parse the request, check the disk cache for the data, and so on. DiskOverhead can either be approximated using a complex disk model [Shriver97] or can be measured experimentally. In this paper we measured the disk overhead experimentally at 1.8 ms for a single file and 1.2 ms for multiple files for our slow platform and 0.34 ms for our fast platform.

- *seek curve information* is used to approximate the seek time. The seek curve information we use is $a = 0.002$, $b = 0.173$, $c = 3.597$, $d = 0.002$, and $e = 801$ as defined in equation (1).

- *disk rotation speed* is used to approximate the time spent in rotational latency. The DiskTR is the rate that data can be transfered from the disk surface to the disk cache. The disk used to validate our model spins at 10,000 RPM, giving us a DiskTR of close to 18 MB/s.

- BusTR gives us the rate at which data can be transfered from the disk cache to the host; we are bounded by the slower of the BusTR and DiskTR. On the slow platform, the transfer rate was limited to 9.3 MB/s; on the fast platform, the transfer rate was 18.2 MB/s.

- CacheSegments is the number of different data streams the disk can concurrently cache, and hence the number of streams for which it can perform read-ahead. The disk used to validate our model was configured for three cache segments; this model of disk can be configured for between one and sixteen cache segments.

- CacheSize is the size of the disk cache. From this value and the CacheSegments, the size of each cache segment can be computed. The disk used to validate our model has a 512 KB cache.

- Max_Cylinder is the number of cylinders in the disk. The disk used to validate our model has 6526 cylinders.

Table 1: File system parameters and values for validated platform.

parameter	*definition*	*validated platform*
BlockSize	the amount of data which the file system processes at once	8 KB
DirectBlocks	the number of blocks that can be accessed before the indirect block needs to be accessed	12
ClusterSize	the amount of a file that is stored contiguously on disk	64 KB
CylinderGroupSize	number of bytes on a disk that file system treats as "close"	16 MB
SystemCallOverhead	time needed to check the file system cache for the requested data	10 μs
MemoryCopyRate	rate at which data are copied from the file system cache to the application memory	10 μs/KB

4.3 Workload specification

The workload parameters that affect file system cache performance are the ones needed to predict the disk performance and the file layout on disk. Table 2 presents this set of parameters; most of these parameters were taken from earlier work on disk modeling [Shriver98].[1]

4.4 The model

Our approach has been to use the technique presented in our earlier work on disk modeling, which models the individual components of the I/O path, and then composes the models together [Shriver97]. We use some of the ideas presented in the disk cache model to model the file system cache.

Disk response time. The mean disk response time is the sum of disk overhead, disk head positioning time, and time to transfer the data from disk to the file system cache:

$$\mathsf{DRT} = \mathsf{DiskOverhead} + \mathsf{PositionTime} + \mathbf{E}[\mathsf{disk\_request\_size}] / \min\{\mathsf{BusTR}, \mathsf{DiskTR}\}.$$

(Note: $\mathbf{E}[x]$ denotes the expected, or average value for x.) The amount of time spent positioning the disk head, PositionTime, depends on the current location of the disk head, which is determined by the previous request. For example, if this is the first request for a block in a given cluster, PositionTime will include both seek time and time for the rotational latency. Let $\mathbf{E}[\mathsf{SeekTime}]$ be the mean seek time and $\mathbf{E}[\mathsf{RotLat}]$ be the mean rotational latency (1/2 the time for a full disk rotation). Thus, the disk response time for the first request in a cluster is

$$\mathsf{DRT}[\text{random request}] = \mathsf{DiskOverhead} + \mathbf{E}[\mathsf{SeekTime}] + \mathbf{E}[\mathsf{RotLat}] + \frac{\mathbf{E}[\mathsf{disk\_request\_size}]}{\min\{\mathsf{BusTR}, \mathsf{DiskTR}\}}. \quad (2)$$

If the previous request was for a block in the same cylinder group, the seek distance will be small. This will be the case if the previous read was to a portion of the file stored in the same cylinder group, or to some other file found in the same cylinder group. If there are n files being accessed concurrently, the expected seek distance will either be (a) $\mathsf{Max\_Cylinder}/3$, if the device driver and disk controller request queues are empty, or (b) (assuming the disk scheduler is using an elevator algorithm) $\mathsf{Max\_Cylinder}/(n+2)$ [Shriver97].

The mean disk request size, $\mathbf{E}[\mathsf{disk\_request\_size}]$, can be computed by averaging the request sizes; these can be computed by simulating the algorithm to determine the amount of data prefetched, where the simulation stops when the amount of accessed data is equal to ClusterSize. If the file system is servicing more than one file, the actual amount prefetched can be smaller than expected due to blocks being evicted before use. If the file system is not prefetching data, the $\mathbf{E}[\mathsf{disk\_request\_size}]$ is the file system block size, BlockSize.

Sometimes the requested data are in the disk cache due to readahead; in these cases, the disk response time is

$$\mathsf{DRT}[\text{cached request}] = \mathsf{DiskOverhead} + \mathbf{E}[\mathsf{disk\_request\_size}] / \mathsf{BusTR}. \quad (3)$$

[1]The previous disk model includes additional workload parameters that support specification of spatial locality; these are not needed for our current model since we assume that the files are accessed sequentially. The earlier disk model also supports a read fraction parameter; in this paper, we only model file reads.

Table 2: Workload specification.

parameter	*definition*	*unit*
temporal locality measures		
request_rate	rate at which requests arrive at the storage device	requests/second
cylinder_group_id	cylinder group (location) of the file	integer
arrival_process	inter-request timing (constant [open, closed], Poisson, or bursty)	—
spatial locality measures		
data_span	the span (range) of data accessed	bytes
request_size	length of a host read or write request	bytes
run_length	length of a *run*, a contiguous set of requests	bytes

File system response time. We first compute the amount of time needed for all of the file system accesses TotalFSRT, and then compute the mean response time for each access, FSRT, by averaging:

$$\mathsf{FSRT} = \frac{\mathsf{data\_span}}{\mathsf{request\_size}}\mathsf{TotalFSRT}. \quad (4)$$

The rest of this section discusses approximating TotalFSRT.

Let us first look at the simplest case: reading one file that resides entirely in one cluster, the mean response time to read the cluster contains file system overhead plus the time needed to access the data from disk:

$$\begin{aligned}\mathsf{ClusterRT} = {} & \mathsf{FSOverhead} + \\ & \mathsf{DRT}[\text{first request}] + \\ & \sum_i \mathsf{DRT}[\text{remaining request}_i]\end{aligned}$$

where the first request and remaining requests are the disk requests for the blocks in the cluster and $\mathsf{DRT}[\text{first request}]$ is from equation (2). If n files are being serviced at once, the $\mathsf{DRT}[\text{remaining request}_i]$'s each contain $\mathbf{E}[\mathsf{SeekTime}] + \mathbf{E}[\mathsf{RotLat}]$ if n is more than CacheSegments, the number of disk cache segments. If not, some of the data will be in disk cache and equation (3) is used. The FSOverhead can be measured experimently or computed as SystemCallOverhead + $\mathbf{E}[\mathsf{request\_size}]/\mathsf{MemoryCopyRate}$. The number of requests per cluster can be computed as data_span/disk_request_size.

If the files span multiple clusters, we have

$$\mathsf{TotalFSRT} = \mathsf{NumClusters} \cdot \mathsf{ClusterRT}$$

where we approximate the number of clusters as NumClusters = data_span/ClusterSize. To capture the "extra" cluster due to only the first DirectBlocks blocks being stored on the same cluster, this value is incremented by 1 if (ClusterSize/BlockSize)/DirectBlocks is not 1 and data_span/BlockSize > DirectBlocks.

If the device driver or disk controller scheduling algorithm is CLOOK or CSCAN and the queue is not zero, then there is a large seek time (for CLOOK) or a full stroke seek time (for CSCAN) for each group of n accesses, when n is the number of files being serviced by the file system; we call this time extra_seek_time.

If the n files being read are larger than DirectBlocks, we must include the time required to read the indirect block:

$$\begin{aligned}\mathsf{TotalFSRT} = {} & n \cdot \mathsf{NumClusters} \cdot \mathsf{ClusterRT} + \\ & \mathsf{num\_requests} \cdot \mathsf{extra\_seek\_time} + \\ & \mathsf{DRT}[\text{indirect block}] \quad (5)\end{aligned}$$

where num_requests is the number of disk requests in a file. Since the location of the indirect block is on a random cylinder group, equation (2) is used to compute $\mathsf{DRT}[\text{indirect block}]$. Of course, if the file contains more blocks than can be referenced by both the inode and the indirect block, multiple indirect block terms are needed.

5 Discussion

In the introduction to this paper, we listed the reasons that prefetching improves performance: the disk cache comes into play, the device driver amortizes the fixed cost of an I/O over a larger amount of data, and total disk seek time can be decreased. In this section we discuss the terms introduced by our model and attempt to explain where the time goes, and when and why prefetching works. To do this, we collected detailed traces of a variety of workloads. These traces allowed us to compute the file system and disk response times experienced by the test machines. These response times were also used in our validations as discussed in Appendix A.

Hardware setup and trace gathering. We performed experiments on two hardware configurations: a 200 MHz Pentium Pro processor and a 450 MHz Pentium II processor. Each machine had 128 MB of main memory. We conducted all of our tracing and measurements on a 4 GB Seagate ST34501W (Cheetah) disk, connected via a 10 MB/second PCI SCSI controller (for the 200 MHz processor) or via a 20 MB/second PCI SCSI controller (for the 450 MHz processor). Our test machines were running version 3.1 of the BSD/OS operating system. The file system parameters for our test file systems are found in Table 1 and the disk parameters are in Section 4.2.

We collected our traces by adding trace points to the BSD/OS kernel. At each trace point the kernel wrote a record to an in-memory buffer describing the type of event that had occurred and any related data. The kernel added a time stamp to each trace record, using the CPU's on-chip cycle counter. A user-level process periodically read the contents of the trace buffer.

To document the response time for application-level read requests, we used a pair of trace points at the entry and exit of the FFS read routine. Similarly, we measured disk-level response times using a pair of trace points in the SCSI driver, one when a request is issued to the disk controller, and a second when the disk controller indicates that the I/O has completed. Additional trace points documented the exact sequence (and size) of the prefetch requests issued by the file system, and the amount of time each request spent on the operating system's disk queue.

The numbers discussed in this section are for the machine with the faster bus unless stated otherwise.

Disk cache. When an application makes a read request of the file system, the file system checks to see if the requested data are in its cache, and if not, issues an I/O request to the disk. The data will be found in the file system cache if they were prefetched

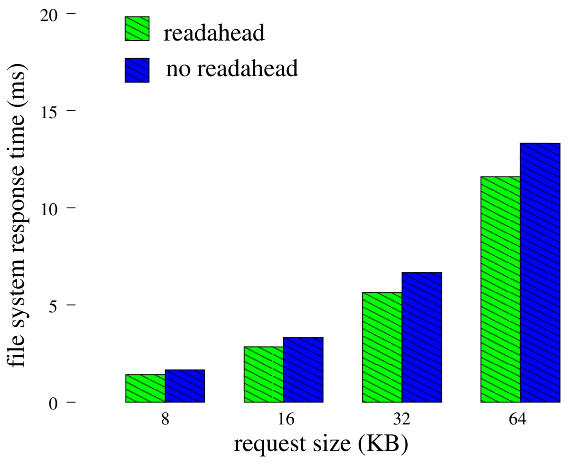

Figure 1: File system response times for a 64 KB file with and without the disk cache performing readahead. The readahead values are measured; the no-readahead values are predicted by the model.

and have not been evicted. If the data are not in the file system's cache, the file system must read it from the disk. There are two possible scenarios:

1. The data are in the disk cache as a result of readahead for a previous command, so the disk does not need to read the data again. The disk sends the data directly from the disk cache.
2. The data are not in the disk cache and must be read from the disk surface.

In an attempt to quantify the effect of the disk cache, Figures 1 and 2 contain the file system response time measurements with the disk cache performing readahead and file system response time predictions without the disk cache performing readahead. The percent improvement in the response time when the disk cache is performing readahead is 17–23%.

Modern disks are capable of caching data for concurrent workloads, where each workload is reading a region of the disk sequentially. If there are enough cache segments for the current number of sequential workloads, the disk will readahead for each workload, and each workload will benefit. However, if there are more sequential workloads than cache segments, depending on the cache replacement algorithm used by the disk, the disk's ability to prefetch may have little or no positive effect on performance. In addition, disk readahead is only valuable when the file system prefetch size is less than the cluster

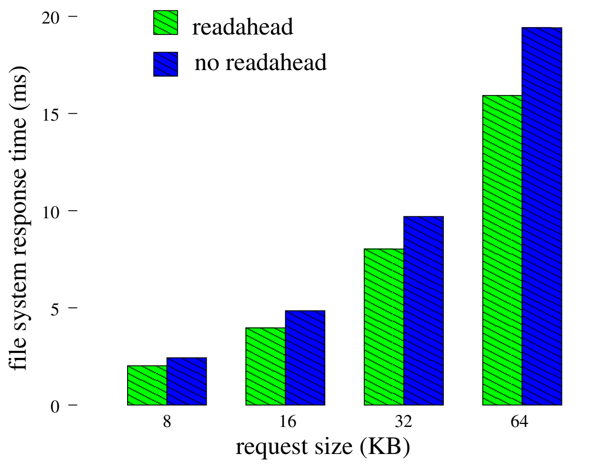

Figure 2: File system response times for a 128 KB file with and without the disk cache performing readahead. The readahead values are measured; the no-readahead values are predicted by the model.

size, since, after that, the entire cluster is fetched with one disk access. In the case of our file system parameters, this occurred when the file was 128 KB or smaller.

I/O cost amortization. On our slow bus configuration, we measured the disk overhead of performing an I/O operation at 1.2 to 1.8 ms, which is on the same order as the time to perform a half-rotation (3 ms). The measured transfer rate of the bus is 9.3 MB/s; by saving an I/O operation, we can transfer an additional 11 to 16 KB. As an example, assume that 64 KB of data will be used by the application. If the requested data are in the disk cache, using a file block of 8 KB will take at least 14.1 ms (1.8 ms overhead four times + 6.9 ms for data transfer);[2] a file block of 64 KB will take 8.7 ms (1.8 ms overhead + 6.9 ms for data transfer), just a little over half the I/O time.

The impact of I/O cost amortization can be seen when comparing the measured file system response time when servicing one file, with and without prefetching. Figure 3 show these times for the slower hardware configuration. With prefetching disabled, the file system requests data in BlockSize units, increasing the number of requests, and the amount of disk and file system overhead. The additional overheads increase the resulting performance by 13–29%.

[2]With the file system performing prefetching, there will be 4 disk requests having a mean disk request size of 16 KB.

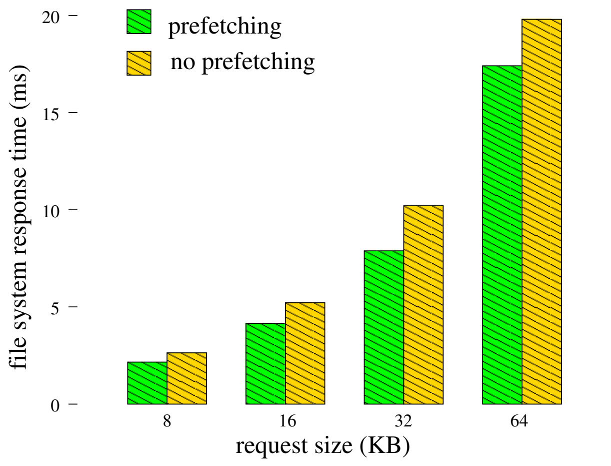

Figure 3: Measured file system response times for a 64 KB file with and without the file system performing prefetching.

When we ran our tests on the machine with the faster bus, we noted anomalous disk behavior that we do not yet understand. According to our measurements, it should (and does, in most cases) take roughly 0.78 ms to read an 8 KB block from the disk cache over the bus on this machine. However, under certain circumstances, 8 KB reads from the disk cache complete more quickly, taking roughly 0.56 ms.[3] This happened only when file system prefetching is disabled, i.e., when the file system requests multiple consecutive 8 KB blocks, and when there is only one application reading from the disk. The net result is that, in these rare situations, performance is slightly *better* with file system prefetching disabled. This behavior also was displayed with the slower bus, but as you can see in Figure 3, the bus is slow enough so that the response time with prefetching is smaller than the response time without prefetching.

Seek time reduction. As the number of active workloads increases, the latency for each workload will increase, but the disk throughput can, paradoxically, increase as well. Due to the type of scheduling algorithms used for the device driver queue, more elements in the read queue can mean smaller seeks between each read, and hence greater disk throughput. On the other hand, a longer queue means that each request will, on average, spend more time in the queue, and thus the read latency will be greater.

[3]We are in communication with Seagate in an attempt to determine why we are seeing this behavior.

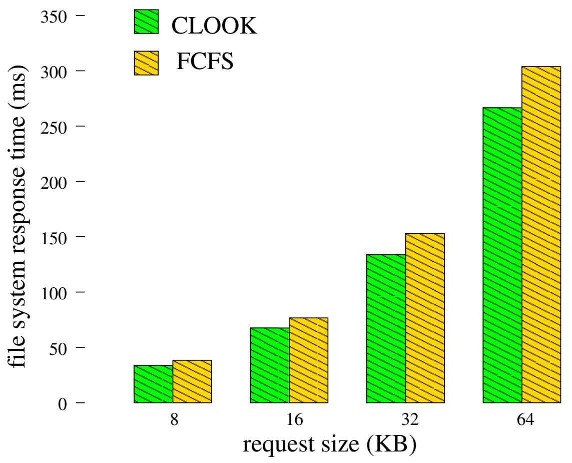

Figure 4: Measured file system response times for 8 64 KB files with the device driver using a CLOOK scheduling algorithm with a FCFS scheduling algorithm.

Figure 4 displays the file system response time with the device driver implementing the CLOOK scheduling algorithm (the standard algorithm), and implementing FCFS, which will not reduce the seek time. The performance gain from using CLOOK over FCFS is 14%.

I/O / computation overlap. As was discussed in Section 1, if an application performs substantial computation as well as I/O, prefetching may allow the application to overlap the two, increasing application throughput and decreasing file system response time. For example, on our test hardware, computing the MD5 checksum of a 10 KB block of data takes approximately one millisecond. A program reading data from the disk and computing the MD5 checksum will exhibit delays between successive read requests, giving the file system time to prefetch data in anticipation of the next read request. Figure 5 shows the file system response times with a request size of 10 KB for files of varying lengths. The figure shows the response time given no delay (representing the application having no I/O / computation overlap), with an application delay of 0.5 ms, and with an application delay of 1.0 ms (as with MD5). As the file size increases, so do the savings due to prefetching. With a 64 KB file there is a 36% improvement, compared to a 114% improvement when reading a 512 KB file.

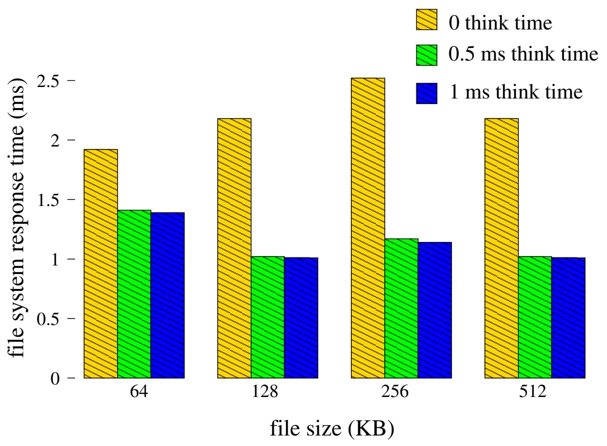

Figure 5: Measured file system response times with the application think time of 0 ms, 0.5 ms, and 1 ms.

Summary. In this section we showed how the components of prefetching combine to improve performance. In our tests, disk caching provided a 17–23% boost, I/O cost amortization yielded, 13–29%, seek time reduction (CLOOK *vs.* FCFS) 14%, and overlapping computation and I/O can save as much as 50%. Depending on the behavior of the specific application, these components can have a greater or lesser benefit; added together, the performance gain is significant.

6 Conclusions

We developed an analytic model that predicts the response time of the file system with a maximum relative error of 9% (see Appendix A). Given the wide range of conceivable file system layouts and prefetching policies, an accurate analytic model simplifies the task of setting system parameters that may improve performance enough to be worth implementing and studying in more detail.

Our model has allowed us to develop two suggestions for decreasing the file system response time. If it is reasonable to assume that the prefetched data will be used, and we have room in the file system cache, once the disk head has been positioned over a cluster, the entire cluster should be read. This will decrease the file system and disk overheads.

The number of disk cache segments restricts the number of sequential workloads for which the disk

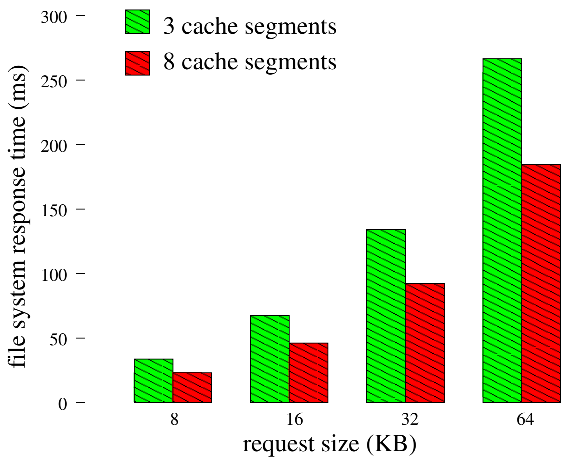

Figure 6: File system response times for 8 64 KB files with 3 and 8 segments. 3-segment values are measured and 8-segment values are predicted by the model.

cache can perform readahead, which means that if the number of disk cache segments is smaller than the number of concurrent workloads, it can be as if the disk had no cache at all. One enhancement that we suggest is for the file system to dynamically modify the number of disk cache segments to be the number of files being concurrently accessed from that disk. This is a simple and inexpensive SCSI operation, and can mean the difference between having the disk cache work effectively and having it not work at all. Figure 6 compares the measured file system response time when servicing 8 workloads with 3 cache segments and the predicted response time with 8 cache segments. This shows us a 44–46% decrease in the response time when the number of cache segments is set to the number of concurrent workloads.

Our current model does not handle file system writes; we would like to extend it to support writes. We would like to use our analytic model to compare different file system prefetching polices and parameter settings to determine the "best" setting for a particular workload. Workloads which seem promising are web server, scientific workloads [Miller91], and database benchmarking workloads.

Acknowledgments. Many thanks to Bruce Hillyer for many interesting conversations.

References

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Asilomar (Pacific Grove), CA), pages 198–212. ACM Press, October 1991.

[Barve99] Rakesh Barve, Elizabeth Shriver, Phillip B. Gibbons, Bruce K. Hillyer, Yossi Matias, and Jeffrey Scott Vitter. Modeling and optimizing I/O throughput of multiple disks on a bus. *Proceedings of the 1999 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems* (Atlanta, GA), May 1999. Available at `http://www.bell-labs.com/~shriver/`.

[Bastian82] A. L. Bastian. Cached DASD performance prediction and validation. *Proceedings of 13th International Conference on Management and Performance Evaluation of Computer Systems* (San Diego, CA), pages 174–177, Mel Boksenbaum, George W. Dodson, Tom Moran, Connie Smith, and H. Pat Artis, editors, December 1982.

[Cao94] Pei Cao, Edward W. Felten, and Kai Li. Implementation and performance of application-controlled file caching. *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA), pages 165–178, November 1994.

[Cao95] Pei Cao, Edward W. Felten, Anna Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems* (Ottawa, Canada), pages 188–197, May 1995.

[Chen93] Shenze Chen and Don Towsley. The design and evaluation of RAID 5 and parity striping disk array architectures. *Journal of Parallel and Distributed Computing*, **17**(1–2):58–74, January–February 1993.

[Feiertag72] R. J. Feiertag and E. I. Organick. The Multics input/output system. *Proceedings of the Third Symposium on Operating Systems Principles* (Palo Alto, CA), pages 35–41, October 1972.

[Griffioen94] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. *Proceedings of the 1994 Summer USENIX Technical Conference* (Boston, MA), pages 197–207, June 1994.

[Hartman93] John Hartman and John Ousterhout. Letter to the editor. *Operating Systems Review*, **27**(1):7–9, January 1993.

[Hofri80] M. Hofri. Disk scheduling: FCFS vs. SSTF revisited. *Communications of the ACM*, **23**(11):645–653, November 1980.

[Hospodor95] Andy Hospodor. Mechanical access time calculation. *Advances in Information Storage Systems*, **6**:313–336, 1995.

[Kimbrel96] Tracy Kimbrel and Anna R. Karlin. Near-optimal parallel prefetching and caching. *Proceedings of the 37th Annual Symposium on Foundations of Computer Science* (Burlington, VT), pages 540–549, October 1996.

[Kotz95] David Kotz. Disk-directed I/O for an out-of-core computation. *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing* (Pentagon City, VA), pages 159–166, August 1995.

[Kroeger96] Thomas Kroeger. Predicting file system actions from reference patterns. Department of Computer Engineering, University of California, Santa Cruz, Santa Cruz, CA, December 1996. Master's thesis.

[Kuratti95] Anand Kuratti and William H. Sanders. Performance analysis of the RAID 5 disk array. *Proceedings of International Computer Performance and Dependability Symposium* (Erlangen, Germany), pages 236–245, April 1995.

[Lei97] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. *Proceedings of the USENIX 1997 Annual Technical Conference* (Anaheim, CA), January 1997.

[Madhyastha97a] Tara M. Madhyastha and Daniel A. Reed. Exploiting global input/output access pattern classification. *Proceedings of Supercomputing '97* (San Jose, CA), November 1997.

[Madhyastha97b] Tara M. Madhyastha and Daniel A. Reed. Input/output access pattern classification using hidden Markov models. *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS)* (San Jose, CA), pages 57–67. ACM Press, November 1997.

[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):181–197, August 1984.

[McKusick96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing, 1996.

[Merchant96] Arif Merchant and Philip S. Yu. Analytic modeling of clustered RAID with mapping based on nearly random permutation. *IEEE Transactions on Computers*, **45**(3):367–373, March 1996.

[Miller91] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputing applications. *Proceedings of Supercomputing '91* (Albuquerque, NM), pages 567–576, November 1991.

[Mowry96] Todd C. Mowry, Angela K. Demke, and Orran Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. *Proceedings of the 1996 Symposium on Operating Systems Design and Implementation* (Seattle, WA), pages 3–17. USENIX Association, October 1996.

[Ng91] Spencer W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, **40**(1):22–30, January 1991.

[Patterson95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, CO), pages 79–95. ACM Press, December 1995.

[Ritchie78] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *The Bell System Technical Journal*, **57**(6):1905–1930, July/August 1978. Part 2.

[Ruemmler94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, **27**(3):17–28, March 1994.

[Seeger96] B. Seeger. An analysis of schedules for performing multi-page requests. *Information Systems*, **21**(5):387–407, July 1996.

[Seltzer90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *Proceedings of Winter 1990 USENIX Conference* (Washington, DC), pages 313–323, January 1990.

[Shriver97] Elizabeth Shriver. *Performance modeling for realistic storage devices.* PhD thesis. Department of Computer Science, New York University, New York, NY, May 1997. Available at `http://www.bell-labs.com/~shriver/`.

[Shriver98] Elizabeth Shriver, Arif Merchant, and John Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. *Joint International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '98/Performance '98)* (Madison, WI), pages 182–191, June 1998. Available at `http://www.bell-labs.com/~shriver/`.

[Small98] Christopher Small. *Building an extensible operating system.* PhD thesis. Division of Engineering and Applied Sciences, Harvard University, Boston, MA, October 1998.

[Wilhelm77] Neil C. Wilhelm. A general model for the performance of disk systems. *Journal of the ACM*, **24**(1):14–31, January 1977.

[Worthington94] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. Scheduling algorithms for modern disk drives. *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Santa Clara, CA), pages 241–251, May 1994.

A Validation of the model

To validate the model, we ran a set of simple microbenchmark workloads, collecting traces of both file system and disk events. From these traces, we determined the mean disk and application-level response times for a variety of synthetic and real workloads. The results in this section are presented for the machine with the slower bus.

The workloads. We created simple workloads which opened files, read them sequentially, and closed them. We recorded only the reads, i.e., not the open and close operations. We varied the request size and size of the file (which, in turn, varied $\mathsf{data\_span}$ and $\mathsf{run\_length}$). Our workloads have a closed arrival process; files greater than 96 KB spanned at least two cylinder groups.

Single files accessed. We ran our workloads using one process to access a single file; we repeated the experiments 100 times and averaged to compute the measured FSRT. Table 3 includes the measured and model-computed FSRT using equations (4) and (5), as well as the relative errors of the model. In all but one case the model's estimate is within 4% of the measured result; when reading 32 KB of a 64 KB file the model error is 9%, which is higher, but still quite good for an analytic model of this type.

Multiple files accessed. We then reran our experiments with multiple concurrent processes, each accessing a different file with the same workload specification (i.e., same request size and same file size). All of the files were opened at the beginning of the experiment; the files which were used during one experiment run were randomly chosen among a set of files that spanned the 4 GB disk. We flushed the disk cache between each of the 50 experiment runs. Table 4 presents our results using equations (4) and (5). We see that the model's error is 6% or less in all cases.

Prefetching disabled. We modified the file system to disable prefetching and reran our single file experiments. Table 5 contains our results using equations (4) and (5) with $\mathbf{E}[\mathsf{disk\_request\_size}] = \mathsf{BlockSize}$. The maximum relative error is 7%.

Table 3: Relative errors with one file accessed.

request size (KB)	*file size* (KB)	*measured* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*computed* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*error*
8	64	2.16	2.16	0%
16		4.15	4.31	4%
32		7.88	8.62	9%
64		17.41	17.24	1%
8	128	2.70	2.69	1%
16		5.41	5.37	1%
32		10.74	10.74	0%
64		21.08	21.48	2%

Table 4: Relative errors with multiple files accessed.

number of files	*request size* (KB)	*file size* (KB)	*measured* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*computed* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*error*
2	8	64	8.80	8.80	0%
	16		17.27	17.61	2%
	32		34.89	35.22	1%
	64		68.96	70.43	2%
3	8	64	12.82	12.90	1%
	16		25.60	25.81	1%
	32		50.85	51.61	1%
	64		99.98	103.23	3%
4	8	64	24.61	23.09	6%
	16		49.16	46.17	6%
	32		98.10	92.35	6%
	64		193.86	184.70	5%
8	8	64	43.83	43.52	1%
	16		87.54	87.04	1%
	32		174.06	174.08	0%
	64		344.04	348.16	1%
16	8	64	77.96	79.50	2%
	16		155.48	159.00	2%
	32		308.20	318.01	3%
	64		604.90	636.01	5%

Table 5: Relative errors with one file accessed and prefetching disabled.

request size (KB)	*file size* (KB)	*measured* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*computed* $\mathbf{E}[\mathsf{FSRT}]$ (ms)	*error*
8	64	2.64	2.46	7%
16	64	5.22	4.93	6%
32	64	10.21	9.86	3%
64	64	19.80	19.72	0%
8	128	3.26	3.07	6%
16	128	6.59	6.14	7%
32	128	13.21	12.28	7%
64	128	26.19	24.56	6%