Improving File System Performance via Predictive Caching *

Abstract

Despite impressive advances in file system throughput resulting from technologies such as high-bandwidth networks and disk arrays, file system latency has improved little and in several cases has become worse. Consequently, file system I/O remains one of the major bottlenecks to operating system performance [Ous90].

This paper investigates an automated predictive approach that improves file cache performance, utilizes increased device bandwidth, and reduces file system latency. *Automatic Prefetching* uses past file accesses to predict future file system requests. The objective is to effectively masking access latencies. In short, we want to transform the file cache from a repository of recently accessed data into a staging area for data about to be accessed. We have designed and implemented a system to measure the performance benefits of automatic prefetching. Our current results show that prefetching results in over 60% improvement in file system performance compared to LRU, especially for smaller caches. Alternatively, prefetching can reduce cache size by up to 50%.

1 Introduction

File systems have historically been the primary performance bottleneck of general purpose computing systems[Ous90]. With rapid improvements in processor speeds, memory sizes, and network speeds, solving the file system bottleneck has become even more urgent. The importance of the file system problem has evoked a large amount of research in the last few years. These efforts have primarily focused on improving file system I/O bandwidth and have been quite successful in improving file system performance via improved software and hardware techniques [Kot95, PGK88, HO92, Wil93, PG94].

Despite these advances, high latencies still plague conventional file systems. As a result, file systems continue to be a bottleneck to overall system performance. Moreover, latency is becoming worse rather than better due to several emerging technologies. Campus and corporate scale distributed file systems, wireless computing systems, telecommuting environments that use ISDN, PPP, or SLIP, CD-ROM/WORM storage systems, and even certain RAID systems all exhibit relatively high latencies. However, the new features offered by these systems (e.g., mobility, access from home, massive storage capacity) have made them extremely popular despite their bandwidth and latency limitations. In short, we cannot count on hardware or higher bandwidth techniques and devices to solve the file system bottleneck, especially in light of the increasing popularity of low performance environments. Instead we must develop techniques that fully utilize the available bandwidth resources and hide or avoid latency.

*This work was supported in part by NSF grants number CCR-9309176,CDA-9502645, and CDA-9320179

File caching can be an enormous performance boost, effectively eliminating a large percentage of disk accesses [Smi82, LKKQ89, NWO88, BAD+92]. However, conventional file systems still exhibit enough cache misses so that the file system remains a performance bottleneck [Ous90]. Unfortunately, increasing cache sizes does not yield the type of performance improvements we would like. Experience shows that the relative benefit of caching decreases as cache size and cache cost increases [OCH+85, NWO88]. Often large caches are not even feasible or affordable for emerging computer devices such as PDAs, notebooks, or even low-end workstations and PC's. To complicate things further, the "natural" cache size is rapidly becoming a substantially larger fraction (one forth to one third) of the total memory, due in part to larger files (e.g., big applications, databases, video, audio, etc.) [BHK+91]. This means that larger caches are necessary just to maintain the current hit rates.

We should also note that a cache with a high hit ratio does not necessarily mean fast access. In many distributed file systems, the open() and close() functions represent synchronization points for shared files. Although the file itself may reside in the client cache, each open() and close() call must be executed at the server for consistency reasons. The latency of these calls can be quite large, and tends to dominate other costs, even when the file is in the file cache.

Simply stated, the goal of our research is to transform the conventional file system cache from a "repository of recently accessed data" to a "staging area for data about to be accessed". We would like to pull data through a machine's local memory in an attempt to keep pace with the CPU's demand for data. To optimize file system performance, we must deliver the data "just-in-time". Such delivery insures the data is available when the CPU needs, but yet avoids removing other necessary data or unnecessarily consuming valuable system resources such as network bandwidth, disk bandwidth, or excessive amounts of physical memory for the file cache.

We must develop new methods that anticipate future data requirements if we are to achieve just-in-time delivery and fully utilize the high bandwidth provided by recent hardware. Requesting data in advance of the need for the data can also benefit emerging low-bandwidth systems. Many low bandwidth devices often have significant idle periods of unused bandwidth. By anticipating data needs sufficiently far in advance of the need, the file system can effectively utilize the idle bandwidth of both high and low bandwidth devices.

We have investigated methods for successfully reducing the the perceived latency associated with file system operations in a general purpose computing system. In this paper, we describe a new method for masking file system latency called *automatic prefetching* that attempts to achieve just-in-time delivery. Automatic prefetching takes a heuristic-based approach using knowledge of past accesses to predict future access without user or application intervention. As a result, applications automatically receive reduced perceived latencies, better use of available bandwidth via batched file system requests, and improved cache utilization.

This paper describes our automatic prefetching system. Section 2 provides an overview of our automatic prefetching algorithm [GA94] and the motivation for our design choices. Section 3 uses trace-base simulations to show the potential performance performance benefits of automatic prefetching. To verify these results, we have implemented a prefetching file system in SunOS 4.1.3. Section 4 describes the results of our tests and verifies the simulation results. Section 6 briefly discusses related work. Prefetching is certainly not a new idea and has been used in a wide variety of settings ranging from processor caches to databases to large scale scientific computations. Our work differs from past work in that we focus on a completely automated prefetching system targeted towards general purpose multiuser computer systems.

2 Automatic Prefetching

We are investigating an approach we call *automatic prefetching*, in which the operating system predicts future file requirements and attempts to make the data available in the cache before it is needed. The basic idea behind automatic prefetching is that future file activity can be successfully predicted from past file activity. Our research explores methods that can correctly predict future file access given limited past information. Assuming predictions can be made, the system must employ policies and algorithms to decide when to prefetch a file, how much to prefetch, and the ordering of prefetch requests.

Automatic prefetching has several appealing characteristics. First, existing applications do not need to be modified or rewritten to take advantage of the prefetching system. All applications receive the benefits of automatic prefetching, including existing software. Second, because the operating system automatically performs prefetching on the application's behalf, application writers can concentrate on solving the problem at hand rather than worrying about optimizing file system performance. Third, the operating system monitors file access across application boundaries and can thus detect access patterns that span applications. Consequently, the operating system can prefetch files substantially earlier than the file is actually needed, often before the application even begins to execute. Fourth, predictions can improve the cache management algorithm, effectively reducing cache misses even if no prefetching occurs. Finally, if the operating system can make predictions far enough in advance of the need for the data, a distributed file system client could batch together multiple file requests into a single request which the server can then send during otherwise idle times.

The first goal of our research was to determine whether such an approach is viable. Our second goal was to develop an effective prefetch algorithm and evaluate its performance via simulations. Our third goal was to confirm these results using a working prototype. The following sections consider each of these objectives in turn and describe our results.

2.1 Is Automatic Prefetching Viable?

To determine the viability of automatic prefetching, we analyzed current file system usage patterns. To gather the specific information we needed for our analysis, we modified a SunOS 4.1.3 kernel and monitored the file system activity of several systems over a ten day interval. In addition to recording all file system calls made by the system, the kernel gathers precise information about the issuing process and the time taken for each operation. The timing information not only serves as an indicator of the system's performance, but it also provides information as to whether prefetching can have any substantial effects on performance.

Accurate timing information allowed us to compute the *read complete time* for each trace. We define the read complete time as the total amount of time applications spent waiting for file system reads to complete. The read complete time is easily calculated by summing the read wait times of all the reads issued during the trace. Read complete times are a more accurate measure of file system performance than cache miss rates. Cache miss rates only measure the amount of missing data, not the delay required to bring the data. Read complete times incorporate the delay to access the data by recording the amount of time applications spent waiting for the file system.

We gathered a variety of traces, including the normal daily usage of several researchers, and various synthetic workloads. Traces were collected on a Sun Sparcstation supporting several users executing a variety of tasks. Traces were collected for varying time periods with the longest traces spanning more than ten days and containing over 400,000 operations. Users were not restricted in any way. Typical daily usage included users processing email, editing, compiling, preparing documents and executing other task typical of an academic environment.

The traces were post-processed to eliminate certain file accesses that could interfere with our results. For example, almost all SunOS executables load a standard set of libraries, including `libc.a` on startup. In practice these libraries are present in physical memory. Counting the loading of these libraries as file accesses would overstate the cache hit rate of the file cache. Therefore we eliminated references to these libraries. We also eliminated accesses to devices that appear in the file system, for example `/dev/null`, `/dev/tty` and `/dev/zero`. Finally, we eliminated all files in `/tmp` and `/usr/tmp` as temporary files unsuitable for prefetching.

For brevity, we only present the results of one trace in this paper. Results from other traces with other workloads can be found in [App96, GA95]. Although the workloads differed from trace to trace, a significant percentage of recurring and predictable access patterns were found in all traces. The particular trace presented below contains the normal daily usage of several students, and almost no database activity. Characteristics of the trace used can be found in table 2.1.

Trace Analysis

Our initial analysis of the trace data indicates that typical file system usage can realize substantial

Trace Duration	10 days
Ratio of Write to Read	0.863
Ratio of Exec to Read	0.105
Number of Reads	25,699
Total I/O	396,761KB
Total Unique Files	3,453
Avg. App. Read Complete Time	4.03166ms

Table 1: Characteristics of The Trace

performance improvements from the use of prefetching, and also provides several guidelines for a successful prefetching policy.

First, the data shows that there is relatively little time between the moment when a file is opened and the moment when the first read occurs (see figure 1). In fact, the median time for our traces was less than three milliseconds. Consequently, prefetching must occur significantly earlier than the open operation to achieve any significant performance improvement. Prefetching at open time will only provide minor improvements.

Second, the data shows that the median amount of time between successive opens is substantial (200 ms). If the operating system can accurately predict even the next file that will be accessed, there exists a sufficient amount of time to prefetch the file.

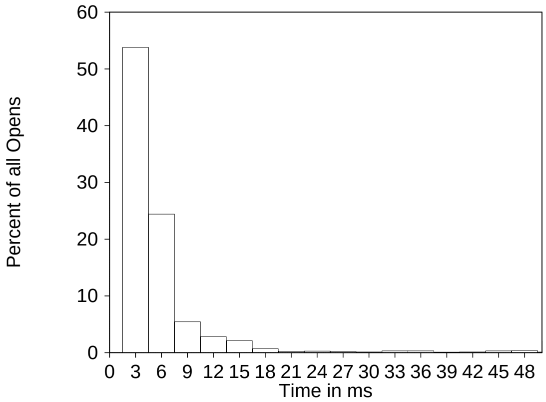

Figure 1: Histogram of times between open and first read of a file.

2.2 The Prediction Algorithm

We have designed and implemented a prediction algorithm that makes informed guesses about future file accesses based on past access patterns. Given past file access information obtained from our trace data, the predictor dynamically creates a logical graph called a *probability graph.* Each node in the graph represents a file in the file system.

Before describing the probability graph, we must define the *Lookahead period* used to construct the graph. The Lookahead period defines what it means for one file to be opened "soon" after another file. The analyzer defines the Lookahead period to be a fixed number of file open operations that occur after the current open. If a file is opened during this period, the open is considered to have occurred "soon" after the current open. A physical time measure rather than a virtual time measure could be used, but the above measure is easily obtained and can be argued to be a better definition of "soon" given the unknown execution times and file access patterns of applications. Our results show that this measure works well in practice.

We say two files are *related* if the files are opened within the same Lookahead period. For example, if the Lookahead period is one, then the next file opened is the only file considered to be related to the current file. If the Lookahead period is five, then any file opened within five files of the current file is considered to be related to the current file.

The predictor allocates a node in the probability graph for each file of interest in the file system. Unix exec system calls are treated like opens and thus are included in the probability graph. Each graph node can be stored on disk along with the other meta-data associated with the file. Alternatively, the graph can be stored in main memory, using the results from [App96] to compact the graph.

Arcs in the probability graph represent related accesses. If the open for one file follows within the Lookahead period of the open for a second file, a directed arc is drawn from the first to the second. Larger Lookaheads produce more arcs. Each arc is weighted by the number of times that the second file is accessed after the first file. Also, each node has associated with it a *total open count*, describing the total number of times the associated file has been opened. Thus, the graph represents an ordered list of files demanded from the file system, and each arc represents the probability of a particular file being opened soon after another file.

Each host dynamically builds a single probability graph containing all files accessed on that host. We considered one graph per user or per file system (e.g. NFS, local disk) but found a single system-wide graph acceptable, and many times preferable [GA94]. Even if files are to be prefetched only from certain file systems, and not all file systems, it is still preferable to have a unified probability graph. Having only one graph per system reduces the total amount of space taken up by the probability graph. Further, building one system wide graph allows access patterns to be shared across users. For example, when Mosaic is started it reads through a very large file for start-up information. When this pattern is discovered by the system for one user, every user gains the advantage of faster Mosaic startup times. Many applications exhibit this behavior. Further, this behavior tends to be quite consistent, with long periods of time between change.

Figure 2 illustrates the structure of an example probability graph.

The probability graph provides the information necessary to make intelligent prefetch decisions. The *likelihood* of file B following file A can be estimated from the probability graph as the ratio of the number of arcs from file A to file B divided by the total open count of file A. We say a prediction such as "B follows A" is *reasonable* if the estimated likelihood of the prediction is above a tunable parameter *MinChance*, the

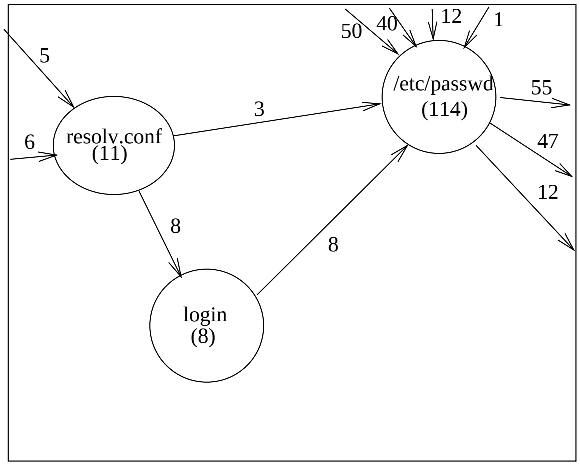

Figure 2: Three nodes of an example probability graph.

minimum chance the algorithm is willing to accept. We say a prediction is *correct* if the file predicted is actually opened within the Lookahead period.

Establishing a minimum chance requirement is crucial to avoid wasting system resources. In the absence of a minimum chance requirement, the predictor would produce several predictions for each file open, consuming network and cache resources with each prediction, many of which would be incorrect.

To measure the success of the predictor we define an *accuracy* value. The accuracy of a set of predictions is the number of correct predictions divided by the total number of predictions made. The accuracy will almost always be at least as large as the minimum chance, and in practice is substantially higher.

Accuracy Tradeoff

The number of predictions made per open call varies with the required accuracy of the predictions. Requiring very accurate predictions (predictions that are almost never wrong) means that only a limited number of predictions can be made. For one typical set of data from the trace, using a relatively low minimum chance value (65%) the probability graph algorithm averages 0.45 files predicted per file open. For higher minimum chance values (95%) the algorithm averages only 0.1 files predicted per open. Even when using a relatively low minimum chance (e.g., 65%), the algorithm can make a prediction about 45% of the time and was correct on approximately 80% of the predictions made. Figure 3 shows this tradeoff.

2.3 The Prefetch Algorithm

The prefetch algorithm works as follows. Each time a file is accessed, all arcs from the corresponding node in the probability graph are examined. All files pointed to by arcs with an estimated likelihood greater than MinChance are added to a *candidate list* of files to prefetch. After the candidate list is built, information about the current open is added to the probability graph as described above (section 2.2).

If the candidate list is empty, the prefetch algorithm has no useful predictions, and therefore no work to

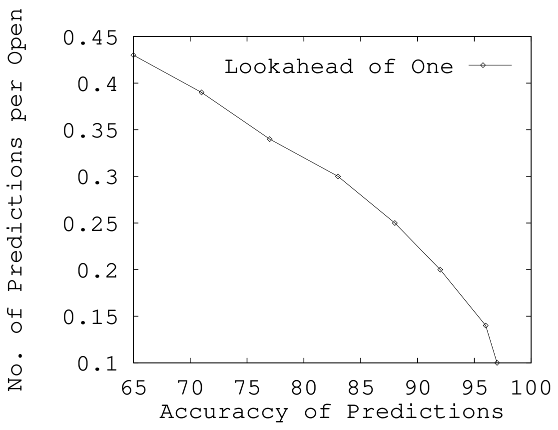

Figure 3: Graph of the Tradeoff Between Accuracy and Quantity.

do. Assuming the candidate list is non-empty, the cache is searched for each file in the list. Each file from the candidate list found in the file cache has its associated LRU access time changed to the current time. Changing the access time to the current time causes the files to remain in the cache longer, preventing them from being flushed before being accessed. Note that just updating the access time of candidate files can improve cache performance, even if no data is ever prefetched.

If a file is not in the cache, the system must decide if the file should be prefetched and, if so, when. The time a file prefetch should occur relative to the time the file will be needed can be controlled by the Lookahead parameter of the probability graph algorithm. That is, to avoid wasting valuable CPU time and memory space predicting files that we have no intention of prefetching, we use the Lookahead value to control the number of predictions and then prefetch all files that are predicted. Consequently the prefetch algorithm will prefetch all files predicted by the predictor. Larger Lookaheads cause more file predictions, effectively increasing the time between file prefetch and the actual need for the data. Setting the Lookahead small causes files to be predicted, and therefore prefetched, closer to the actual need. Section 3.1.3 discusses the effects of Lookahead parameter settings.

The amount of data to prefetch from a predicted file is controlled by the parameter *MaxPrefetch*. The first portion of the file, up to MaxPrefetch bytes, is brought in on each prefetch. MaxPrefetch is set to 300KB for all results shown here. We have experimented with several settings for MaxPrefetch, but even large changes in MaxPrefetch produce minimal changes in performance. Note that most files are prefetched in their entirety because most files are significantly smaller than MaxPrefetch [Irl93]. Any remaining portions can be brought in by read ahead, as described in section 6. Read ahead works well for large files.

3 A Simulation System

To measure the performance of our algorithms, we built a simulator that emulates a file system cache with read-ahead, a local disk, and a remote disk with intervening network. The simulator uses timestamps from the traces to compute the times at which file accesses would occur (adjusted according to the file system model being simulated). As a result, the simulator can calculate both miss rates and total runtimes with or without prefetching for a variety of local and remote file system models.

Our local disk model has two parameters, *latency* and *throughput*. The time needed to access a file is the sum of the access latency and the result of dividing the length of the file by the bandwidth of the device. However, only one file access may use the device at a time. Therefore, if two file accesses are made near-simultaneously, the second access is queued until the first one finishes. For our simulations, the local disk latency (seek time) was assumed to be 12ms, including software latencies. The disk throughput was assumed to be 2 MB/sec, which includes both software delays and any time spent copy buffers.

Our remote disk model has the same parameters as the local disk model with additional *network latency* and *network throughput* for the network portion of the model. For the experiments reported here, the simulated network has a round-trip latency of 2ms and a realizable bandwidth of 8 Mb/sec. Like most real computer networks, the simulated network allows multiple outstanding requests. Concurrent requests are, however, serialized at the remote disk drive.

Under our remote file system model, it is possible for the prefetching component of the file system to request a file and initiate the transfer, but then receive an application's request of the file before the prefetch transfer completes. In this case, the resulting access is considered a cache miss even though the file access completes significantly faster than it would have if prefetching were not use.

The simulated file cache is organized in a basic LRU fashion with a 1KB cache line. The amount of space needed to hold the prefetching graph varies dynamically as the graph is being built. The results from the simulator shown in the following sections do not take into account the space required by the probability graph. However, for all but the smallest cache sizes the space needed to hold the graph is a very small fraction of the total cache size. Our initial results indicate that the space consumed by the algorithm can be sufficiently limited without significantly altering the algorithm's performance [App96].

Notice that the current disk subsystem does no re-ordering of requests. In particular, it does not preempt or defer prefetch requests to satisfy subsequent application requests. Reordering and prioritizing requests represents an area of further potential performance improvements.

3.1 Simulation Results

We performed several simulation tests to measure the performance improvements achieved by automatic prefetching. Results from the prototype system are shown in section 4. For the particular set of tests described below, a trace taken over an ten day period containing the unrestricted activity of multiple users was used. To determine the performance benefits of prefetching, we ran several simulations varying the cache size, Lookahead value, and minimum chance and also measured the LRU performance in each case for comparison purposes.

3.1.1 Timing Results

Recall from section 2.1, that the time between the open of a file and the first read is too small for prefetching to be effective. Figure 4 shows that the simulator is able to predict and begin prefetching files sufficiently far in advance of the first read to the file. Our measurements indicate that 94% of the files that were predicted and then subsequently accessed were prefetched more than 20 ms before the actual need, resulting in cache hits at the time of the first read.

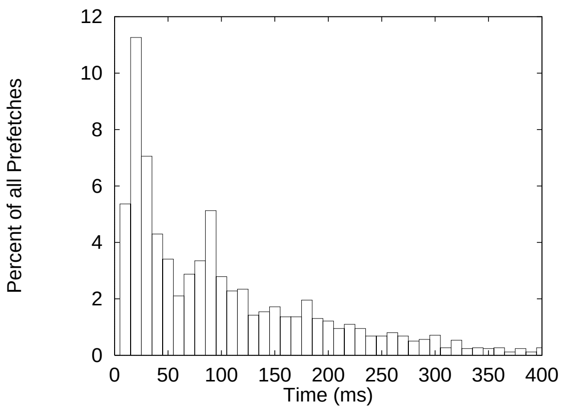

Figure 4: Histogram of times between prefetch and first read access.

3.1.2 Performance Compared to LRU

The primary goal of automatic prefetching is to bring necessary file data into the cache before it is needed. If automatic prefetching is successful we would expect the read complete time using prefetching to be less than that experienced under standard LRU cache management.

Figure 5 shows the read complete time that the file system incurred under LRU and under prefetching for various cache sizes. After tuning MinChance and Lookahead, prefetching performs better than LRU for all cache sizes, in some cases outperforming LRU by as much as 40%. Also note that for the cache sizes shown here, prefetching provided about the the same performance as LRU using a cache half the size. Machines can achive similar file system performance while only using half the cache memory as LRU, freeing memory for applications and enhancing performance. This is particularly important for machines that have only limited amounts of memory available for file caching. However, even large memory machines can benefit from this additional available free memory. This performance boost also indicates that the number of correctly prefetched pages more than offsets any pages incorrectly forced out of the cache by prefetching, even for small cache sizes.

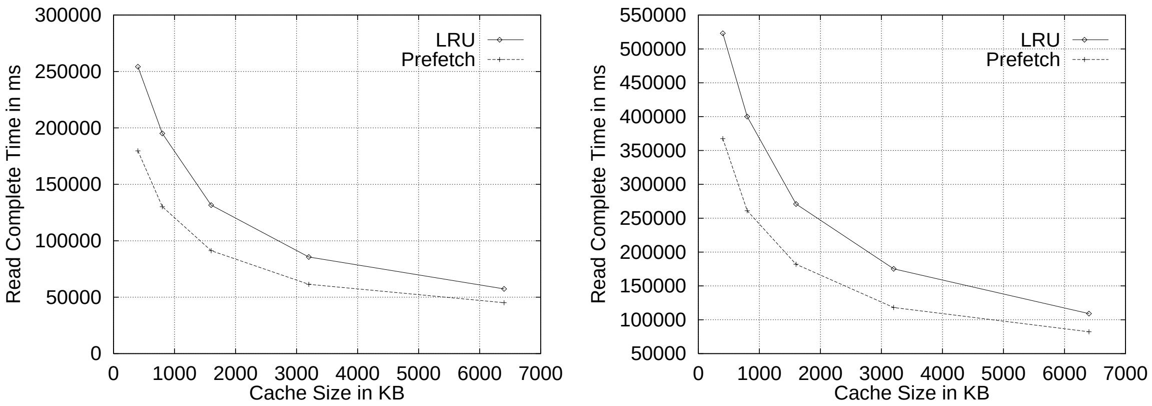

Figure 5: Application read complete time as a function of cache size for an (a) local disk and (b) a networked disk.

3.1.3 Prefetch Parameters Effect on Performance

Two parameters that significantly affect the predictions made by the predictor are the *Lookahead* and *MinChance* values.

Recall that the Lookahead represents how close two file opens need be for the files to be considered related. Setting this value very large increases the number of files that are considered related to each other, and therefore each file open may potentially cause several other files to be prefetched.

Large Lookaheads increase the number of files prefetched since more predictions are made in response to each open request. Moreover, large Lookaheads result in files being prefetched substantially earlier, because predictions can be made much further in advance. As a result, large Lookaheads are inappropriate for smaller cache sizes, but often perform very well with larger caches[1]. In the case of small caches, large Lookaheads tend to prefetch files too far in advance of the need. As a result, data necessary to the current computation may be forced out of the cache and replaced by (useless) data needed far in the future. However, for larger cache sizes, the cache may have sufficient space to load in file data required in the future without disturbing the file data required by the current computation.

MinChance is the minimum estimated probability that a given file will be needed in the near future. For larger cache sizes smaller MinChance values perform better. Setting the MinChance low results in aggressive prefetching. When the cache is large, incorrect prefetches have minimal affect on overall performance. However, for small cache sizes MinChance should be somewhat higher. Small caches suffer from many cache misses. If MinChance is too low, then the file system will prefetch many incorrect files. For small caches, I/O from both application reads and incorrect prefetches can compete for space in the disk queue, slowing application reads. Since large cache suffer from fewer cache misses, this effect is less pronounced for large caches.

[1]Here we use the terms "small" and "large" as relative measures of cache size where the meaning of "small" and "large" depend on the workload. A "small cache" will have many cache misses while a "large cache" will have few misses.

In summary, MinChance should be low (aggressive) for large caches, and somewhat higher (moderate) for small caches. Lookahead should increase with increasing cache size. Figures 6 and 7 illustrate these tradeoffs for a 400 KB cache and a 6,400 KB cache respectively.

Clearly, the Lookahead and MinChance parameters are highly sensitive to the cache size and must be adjusted in accordance with the cache size. Moreover, multiple settings for a particular cache size may result in approximately equal read complete times. In this case, other factors such as network congestion and processing overhead can be used to aid in the selection of appropriate parameter settings.

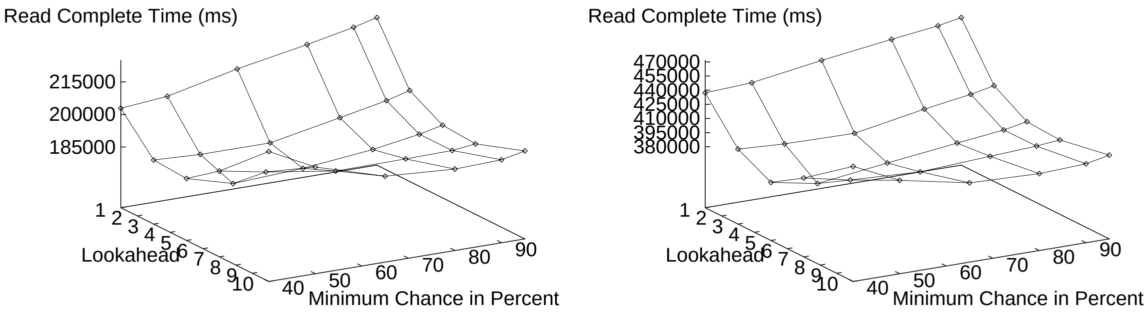

Figure 6: Application read complete times as a function of Lookahead and MinChance for a 400K cache using an (a) local disk or (b) a networked disk.

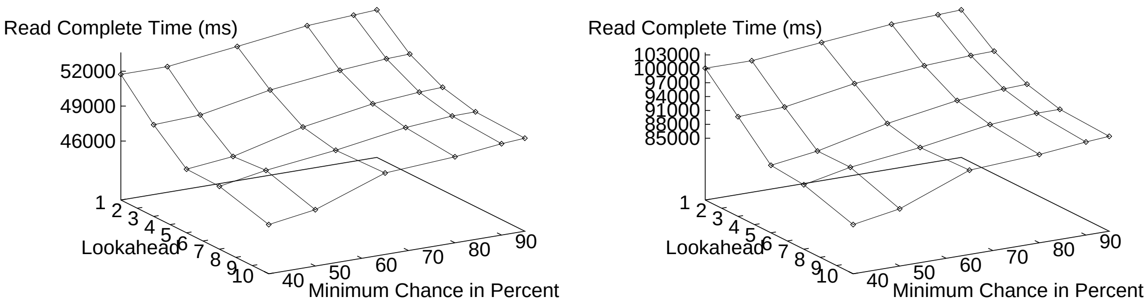

Figure 7: Application read complete times as a function of Lookahead and MinChance for a 6,400K cache using an (a) local disk or (b) a networked disk.

4 Prototype System

To confirm the results of our simulator, we modified the SunOS 4.1.3 kernel to support automatic prefetching. The prototype system supports file prefetching from both the local file system (UFS) and remote file systems (NFS). Approximately 600 lines of kernel code were modified or added to implement prefetching. We made no modifications to the standard SunOS file cache. However a new read call was added to perform prefetch reads. The prefetch read call either pulls the prefetched data into the cache or updates the LRU timestamp if the data is already in the cache.

To obtain information about past accesses, we added tracing code to the *open()* and *exec()* system calls. Every open() and exec() is logged to a pseudo-device in the file system. Accesses to certain files that would interfere with prefetching, such as shared libraries, temporary files, and pseudo-devices are not logged.

A user-level *predictor daemon* tracks current file usage by reading the pseudo-device containing the file access log. The predictor daemon then constructs a probability graph as described in section 2. Periodically, or upon receipt of an appropriate signal from a user, the predictor daemon writes a newly computed *prefetch table* into the kernel. The kernel immediately begins using the new table to make prefetching decisions.

A prefetch table is a list of tuples consisting of file names. The first element in the tuple is the name of a file that when opened should generate prefetch requests. The remainder of the tuple specify the files to be prefetched. Whenever a file is open()'ed or exec()'ed, the prefetch table is consulted. If the file being opened is found in the table (i.e., is the first entry of some tuple) the other files in the tuple are prefetched into the cache. Differing prefetch strategies, such as varying the Lookahead and MinChance parameters, can be tested by computing the appropriate prefetch table and writing that prefetch table into the kernel. Alternatively, prefetching can be disabled by writing the null table into the prefetch table buffer.

A second daemon called the *prefetch daemon* is used to asynchronously prefetch predicted files in parallel with the running program. Each time the running program issues an open() call, the kernel consults the current prefetch table and constructs a candidate list. The kernel then awakens the prefetch daemon to load the cache with the files in the candidate list. The prefetch daemon then invokes the new prefetch read call to load the cache without copying the data to user space.

We tested the prototype by writing two shell scripts that represent activities a typical user may execute repeatedly over time. The first script, *Compilation*, runs a large compile on the prefetching kernel to create a new executable kernel. The Compilation test indicates the type of performance a user compiling source code might except. The second script, *Typesetting*, typesets several large papers using LaTeX, bibTex, dvips, grep, gnuplot, fig2dev, and a few other miscellaneous programs. The Typesetting test shows the type of performance the author of a paper might expect.

The overhead required to compute and store the prefetch table are present in our prototype, just like they would be in a working system. The prefetch table occupies a portion of the physical memory and the kernel consumes CPU cycles examining the prefetch table and calculating the candidate list.

Finally, we modified the kernel to record read complete times of every read operation as defined in section 2.1.

5 Prototype Results

We evaluated the performance of our prototype prefetching system using the Compilation and Typesetting shell scripts described earlier. We executed the test scripts on a SPARCstation 10 running our modified SunOS 4.1.3 kernel. All user files (*.tex files for the Typesetting test and *.[co] files for the Compilation test) resided on a network file server. This configuration reflects the increasingly common situation where user files reside on a file server accessible by all clients. The system files (e.g., executables, font files, header files) were place on a local disk for some of the tests and on a remote file server for the other tests. All tests used the local disk for swap space.

To examine how prefetching performs under various cache sizes, we executed the system with 32 MB of memory and with 12 MB of memory. During the test, no other user applications were running so that test script plus the operating system (and its associated system processes) were the sole memory users.

We ran several sets of tests to measure the performance of the system under a wide variety of conditions. In particular, we varied the Lookahead, MinChance, total amount of free memory, the location of system files. For each test we recorded the total run time (wall time) and read complete time for the test. Each test was repeated several times to obtain an average and standard deviation over several runs. Tables 5 and 5 show the results.

In all cases prefetching significantly reduced the application's read complete time. The improvement ranged between 15% and 58% depending on the test. In general, higher latency (and therefore higher total read time) implies a larger performance gain. Prefetch read complete times in the Compilation test were on average 24% better than non-prefetch times, and prefetch read complete times in the Typesetting test were on average 41% better than non-prefetch times.

For the most part, improvements in read complete times were directly reflected in the total run time (wall time) performance improvements. However, in some cases the improved read complete times did not completely carried through to the total run time. Our profiling showed that prefetching would occasionally increase the number of virtual memory page faults which offset some of the prefetching gains. In addition, the algorithm incurs some, albeit minimal, overhead to store the prefetch table, search the table for prefetch opportunities, and context switch to/from the prefetcher daemon. Despite these added overheads, the performance gains of prefetching outweigh the overheads ultimately reducing total run time by as much as 7%.

Finally, note that a 12 MB system with prefetching exhibits read complete times that are as good as or better than a 32 MB system that does not use prefetching. Similarly, in all cases except the remote Typesetting case, a 12 MB system with prefetching exhibits total run times equal to or better than a 32 MB system without prefetching. This means that similar performance can be achieved with less than half the total memory. Achieving performance similar to or better than that of memory systems twice the size is particularly important to the increasing number of notebook users and a PDA systems where memory is not readily available.

Free Mem Memory	Sys Files Location	Prefetching Used?	Read Comp Time (seconds)	Std Dev (seconds)	Wall Time (seconds)	Std Dev (seconds)
32MB	Local	No	19.3	0.11	132.1	0.66
		Yes	16.5	0.11	128.5	0.28
	Remote	No	37.0	0.21	147.5	0.66
		Yes	24.7	0.25	137.7	0.79
12MB	Local	No	19.8	0.25	137.5	1.11
		Yes	16.8	0.18	132.4	0.82
	Remote	No	37.0	0.12	149.4	0.33
		Yes	25.3	0.11	139.1	0.39

Table 2: Performance results from the Compilation test.

Free Mem Memory	Sys Files Location	Prefetching Used?	Read Comp Time (seconds)	Std Dev (seconds)	Wall Time (seconds)	Std Dev (seconds)
32MB	Local	No	15.2	1.94	227.3	2.18
		Yes	8.0	0.10	214.0	0.42
	Remote	No	4.9	0.05	179.0	1.32
		Yes	2.1	0.02	177.7	1.10
12MB	Local	No	17.5	0.01	242.9	3.74
		Yes	11.1	0.02	228.6	4.02
	Remote	No	6.6	0.16	193.6	3.93
		Yes	4.7	0.08	192.8	4.28

Table 3: Performance results from the Typesetting test.

6 Related work

The concept of prefetching has been used in a variety of environments including microprocessor designs, virtual memory paging, databases, and file read ahead. More recently, long term prefetching has been used in file systems to support disconnected operation [Sat90, SK93, Kue94]. Prefetching has also been used to improve parallel file access on MIMD architectures [KE90].

Many Unix's perform a type of intra-file prefetching known as *read ahead*. When these operating systems notice that a file is being used sequentially, they cause substantial parts of the file to be brought in from disk on each disk access.

Read ahead can substantially boost performance, particularly for large files. However, read ahead does little for small files which are the majority of files [OCH+85, BHK+91]. To achieve performance improvements beyond those already provided by caching-with-read-ahead, we must develop more aggressive and innovative techniques.

A strong performance improvement can come from prefetching files before the application has opened the file. This allows the system time to properly prefetch large files, and allows the system to prefetch the numerous small files that applications use. Unfortunately, no current system can prefetch a file before it is opened; they all prefetch only within a file already opened by an application. Until now there has been great difficulty in determining which files an application will need before that application has opened them.

One relatively straight forward idea is to have each application inform the operating system of its future requirements. This has been proposed and is currently being researched by several researchers including Patterson et. al. [PGG+95]. Using this approach, the application program informs the operating system of its future file requirements, and the operating system then attempts to optimize those accesses. The basic idea behind this approach is that the application knows what files will be needed and when they will be needed.

Although this is a step in the right direction, we see several disadvantages to this solution [App93, GA94]. In particular, this approach assumes that each application will know its file requirements sufficiently far in advance of the real file requirements. However, applications frequently do not know what files they will access until they need to access the file (consider a C compiler opening include files). Further, this approach requires that applications be rewritten to take advantage of these new prefetching system calls. Finally, each application must make prefetching decisions without the knowledge of the activity of other concurrently running processes.

Tait and Duchamp [TD91] have proposed an approach that prefetches all the files in an application's working sets as soon as the specific instance of the application being executed can be uniquely identified. This algorithm employs *trees* of processes, recording file activity (working sets) with these trees. The operating system then attempts to find unique patterns within these trees, and then prefetch the entire working set (subtree) as soon as it can be identified. Automatic prefetching takes a fundamentally different approach. Rather than periodically prefetching an entire working set, automatic prefetching continuously pulls future files into the system at a rate sustainable by the machine's cache. Moreover, automatic prefetching supports prefetching across completely unrelated applications. Tait et. al. make the assumption that prefetching of the working set occurs in the background and completes before the data is needed, but admit that they are uncertain about this assumption. Our experience[GA95], and the results presented here, indicate that this assumption is rarely true, implying that the amount of data prefetched must be carefully regulated. Despite these limitations, their simulated miss rate analysis (which assumes no prefetching overhead and a "fixed number of files cache size") also indicates that prefetching can significantly reduce cache misses.

Kroeger and Long have also recently begun to investigate a tree-based algorithm to predict file access patterns [KL95]. Their approach is motivated by the fact that the study of past events to determine the probability of future events is a problem quite similar to the problem of data compression. Consequently, they employ trees based on a data compression method called *Prediction by Partial Match*. Although their research is still at a very early stage, they report simulated performance gains of 15% over standard LRU miss rates. Their results are significantly worse than the results we have presented here. However, this research may be too preliminary to draw any firm conclusions from.

7 Conclusions

Our results show that reasonable predictions can be made based on past file activity. As a result, automatic prefetching can substantially reduce I/O latency and make better use of the available bandwidth via improved cache utilization. Both our simulator and our prototype show that prefetching can improve file system performance by as much as 60%. As wide area distributed file systems, CDROMs, RAIDs, wireless communication, and other high latency (and possibly low bandwidth) systems become prevalent, predictive caching via prefetching will become increasingly important to reducing latency and achieving high-performance file system I/O.

References

[App93] Randy Appleton. Automatic File System Prefetching. Technical report, University of Kentucky, May 1993.

[App96] Randy Appleton. *An Automatic Prefetching File System*. PhD thesis, University of Kentucky, Lexington, KY, 1996.

[BAD+92] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, October 1992.

[BHK+91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 198–212. Association for Computing Machinery SIGOPS, October 1991.

[GA94] James Griffioen and Randy Appleton. Reducing File System Latency using a Predictive Approach. In *Proceedings of the 1994 Summer USENIX Conference*, June 1994.

[GA95] James Griffioen and Randy Appleton. Performance Measurements of Automatic Prefetching. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 165–170, September 1995.

[HO92] John Hartman and John Ousterhout. Zebra: A Striped Network File System. In *Proceedings of the Usenix File System Workshop*, pages 71–78, May 1992.

[Irl93] Gordon Irlam. Unix File Size Survey. Technical report, Usenet Community Trust, 1993.

[KE90] D. Kotz and C. Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1:218–230, 1990.

[KL95] Thomas Kroeger and Darrell Long. Predicting Future File-System Actions from Prior Events. Technical Report submitted to Usenix Fall 1995, 1995.

[Kot95] D. Kotz. Parallel file system i/o bibliography, August 1995. Seventh Edition,http://www.cs.dartmouth.edu/cs_archive/pario/bib.html.

[Kue94] G. H. Kuenning. The Design of the SEER Predictive Caching System. In *Proceedings of Mobile Computing Systems and Applications*, December 1994.

[LKKQ89] Samuel J. Leffler, Marshal K. Mc Kusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3 BSD Unix Operating System.* Addison Wesley, 1989.

[MSC+86] J. Morris, M. Satyanarayanan, M. Conner, J. Howard, D. Rosenthal, and F. Smith. Andrew: A Distributed Personal Computing Environment. *CACM*, 29:184–201, March 1986.

[NWO88] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[OCH+85] J. Ousterhout, Da Costa, H. Harrison, J Kunze, M. Kupfer, and J. Thompson. A Trace-Driven Analysis of the Unix 4.2 BSD File System. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 15–24, December 1985.

[Ous90] John K. Ousterhout. Why Aren't Operating Systems Getting Faster As Fast as Hardware? In *Proceedings of the Summer 1990 USENIX Conference*, pages 247–256, June 1990.

[PG94] R. Hugo Patterson and Garth A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 7–16, September 1994.

[PGG+95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proc. of the 15th Symposium of Operating Systems Principles*, December 1995.

[PGK88] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD 88*, pages 109–116, June 1988.

[PGS93] R. Hugo Patterson, Garth A. Gibson, and M. Satyanarayanan. Informed prefetching: Converting high throughput to low latency. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 41–55, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies.

[Sat90] M. Satyanarayanan. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. on Computers*, 39:447–459, April 1990.

[SK93] Peter Skopp and Gail Kaiser. Disconnected Operation in a Multi-User Software Development Environment. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 146–151, October 1993.

[Smi82] A. Smith. Cache memories. *Computing Surveys*, 14(3), September 1982.

[TD91] Carl Tait and Dan Duchamp. Detection and Exploitation of File Working Sets. In *Proceedings of the 1991 IEEE 11th International Conference on Distributed Computing Systems*, pages 2–9, May 1991.

[Wil93] John Wilkes. DataMesh, house-building, and distributed systems technology. *ACM Operating Systems Review*, 27(2):104–108, April 1993.