Reducing File System Latency using a Predictive Approach *

James Griffioen, Randy Appleton
Department of Computer Science
University of Kentucky
Lexington, KY 40506

Abstract

Despite impressive advances in file system throughput resulting from technologies such as high-bandwidth networks and disk arrays, file system latency has not improved and in many cases has become worse. Consequently, file system I/O remains one of the major bottlenecks to operating system performance [10].

This paper investigates an automated predictive approach towards reducing file latency. *Automatic Prefetching* uses past file accesses to predict future file system requests. The objective is to provide data in advance of the request for the data, effectively masking access latencies. We have designed and implement a system to measure the performance benefits of automatic prefetching. Our current results, obtained from a trace-driven simulation, show that prefetching results in as much as a 280% improvement over LRU especially for smaller caches. Alternatively, prefetching can reduce cache size by up to 50%.

1 Motivation

Rapid improvements in processor and memory speeds have created a situation in which I/O, in particular file system I/O, has become the major bottleneck to operating system performance [10]. Recent advances in high bandwidth devices (e.g., RAID, ATM networks) have had a large impact on file system throughput. Unfortunately, access latency still remains a problem and is not likely to improve significantly due to the physical limitations of storage devices and network transfer latencies. Moreover, the increasing popularity of certain file system designs such as RAID, CDROM, wide area distributed file systems, wireless networks, and mobile hosts has only exacerbated the latency problem. For example, distributed file systems experience network latency combined with standard disk latency. As distributed file systems scale both numerically and geographically, as envisioned by the Andrew File System designers [7], network delays will become the dominant factor in remote file system access. Similarly, local file systems built on technologies like CD-ROMs also suffer from very high latencies but continue to increase in popularity due to the large amount of storage space they offer.

Although a variety of high bandwidth technologies are now available, it is unlikely that existing (and emerging) low-end technologies such as serial lines running SLIP or PPP, 64/128 Kb ISDN and other slower speed networks will disappear in the near future given their low-cost and wide-spread use. Such communication technologies suffer from both high latencies and low bandwidths. Distributed file systems that build on or incorporate these technologies will experience latencies substantially higher than that of conventional file systems. However, the appeal of low-cost widely available shared access to files will certainly prolong the existence of such file systems, despite their poor performance.

The goal of our research is to investigate methods for successfully reducing the the perceived latency associated with file system operations. In this paper, we describe a new method for masking file system latency called *automatic prefetching*. 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.

2 Related work

Both caching and prefetching have been used in a variety of settings to improve performance. The following briefly describes related work involving caching and prefetching to improve file system performance.

*This work was supported in part by NSF grant number CCR-9309176

2.1 Caching

Caching has been used successfully in many systems to substantially reduce the amount of file system I/O [16, 6, 8, 1]. Despite the success of caching, it is precisely the accesses that cannot be satisfied from the cache that are the current bottleneck to file system performance [10]. Unfortunately, increasing the cache size beyond a certain point only results in minor performance improvements. Experience shows that the relative benefit of caching decreases as cache size (and thus cache cost) increases [9, 8]. There exists a threshold beyond which performance improvements are minor and prohibitively expensive. Moreover, studies show that the "natural" cache size or threshold is 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.) [2]. Consequently, new methods are needed to reduce the perceived latency of file accesses and keep cache sizes in check.

Although machines with large memories are now available, low-end workstations, PCs, mobile laptops/notebooks, and now PDAs (personal data assistants) with limited memory capacities enjoy widespread use. Because of cost or space constraints these machines cannot support large file caches. The desire for smaller portable machines combined with continually increasing files size means that large caches cannot be assumed to be the complete solution to the latency problem.

Finally, as a result of rapid improvements in bandwidth, cache miss service times are dominated by latency. Note that:

- Most files are quite small. In fact, measurements of existing distributed file systems show that the average file is only a few kilobytes long [9, 2]. For files of this size, transmission rate is of little concern when compared to the access latency across a WAN or from a slow device. As a result, access latency, not bandwidth, becomes the dominate cost for references to files not in the cache.

- 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.

In short, the benefits of standard caching have been realized. To improve file system performance further and keep file cache sizes in check, caching will need to be supplemented with new methods and algorithms.

2.2 Prefetching

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 [14, 15, 5]. Prefetching has also been used to improve parallel file access on MIMD architectures [4].

One relatively straight forward method of prefetching is to have each application inform the operating system of its future requirements. This approach has been proposed by Patterson et. al. [11]. 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 is that the application knows what files will be needed and when they will be needed.

Application directed prefetching is certainly a step in the right direction. However, there are several drawbacks to this approach. Using this approach, applications must be rewritten to inform the operating system of future file requirements. Moreover, the programmer must learn a reasonably complex set of additional system directives that must be strategically deployed throughout the program. This implies that the application writer must have a thorough understanding of the application and its file access patterns. Ironically, a key goal of many recent languages, in particular object-oriented languages, is abstraction and encapsulation; hiding the implementation details from the programmer. Even when the details are visible, our experience indicates that the enormity and complexity of many software systems creates a situation in which experts may have difficulty grasping the complete picture of file access patterns. Moreover, incorrectly placed directives or an incomplete set of directives can actually degrade performance rather than improve it.

A second problem is that the operating system needs a significant lead-time to insure the file is available when needed. Therefore, in order to benefit from prefetching, the application must have a significant amount of computation to do between the time the file is predicted and the time the file is accessed. However, many applications do not know which files they will need until the actual need arises. For instance, the preprocessor of a compiler does not know the pattern of nested include files until the files are actually encountered in the input stream, nor will an editor necessarily know which files a user normally edits. Our approach attempts to solve this problem by predicting the need for a file well in advance of when the application could; in some cases long before the application even begins to execute.

A third problem with application driven prefetching arises in situations where related file accesses span multiple executables. Typically applications are written independently and only know file access patterns within the application. In situations where a series of applications execute repeatedly, like an edit/compile/run cycle, or certain commonly run shell scripts, no one application knows the cross-application file access patterns, and therefore cannot inform the operating system of a future application's file requirements. In some cases, batch-type utilities, such as the Unix make facility, can be instrumented to understand cross-application access patterns. However, even in this case, a complete view of the real cross application pattern is often unknown to the user or requires extreme expertise to determine the pattern. Our approach uses long term history information to support prefetching across application boundaries.

3 Automatic Prefetching

We are investigating an approach we call *automatic prefetching*, in which the operating system rather than the application predicts future file requirements. The basic idea and hypothesis underlying automatic prefetching is that future file activity can be successfully predicted from past file activity. This knowledge can then be used to improve overall file system performance.

Automatic prefetching has several advantages over existing approaches. First, existing applications do not need to be rewritten or modified, nor do new applications need to incorporate non-portable prefetching operations. As a result, 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 multiple applications executed repeatedly. Consequently, the operating system can prefetch files substantially earlier than the file is actually needed, often before the application even begins to execute.

Automatic prefetching allows the operating system effectively to overlap processing with file transfers. The operating system can also use past access information to batch together multiple file requests and thus make better use of available bandwidth. Past access information can also be used to improve the cache management algorithm, effectively reducing cache misses even if no prefetching occurs.

The first goal of our research was to determine whether such an approach is viable. Our second goal was to develop effective prefetch policies and quantify the benefits of automatic prefetching. The following sections consider each of these objectives and describe our results.

4 Analysis of Existing Systems

To determine the viability of automatic prefetching, we analyzed current file system usage patterns. Although other researchers have gathered file system traces [9, 2], we decided to modify the SunOS kernel in order to gather our own traces that extract specific information important to our research. In addition to recording all file system calls made by the system, the kernel gathers precise information regarding the issuing process and the timing for every 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.

We gathered a variety of traces, including the normal daily usage of several researchers, and also various synthetic workloads. Traces were collected on a single Sun Sparcstation supporting several users executing a variety of tasks. Traces were collected for varying time periods with the longest traces spanning more than 10 days and containing over 500,000 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. This particular set of traces contains almost no database activity. The data we collected appears to be in line with that of other studies [9, 2] given similar workloads.

Our initial analysis of the trace data indicates that typical file system usage can realize substantial 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 average amount of time between successive opens is substantial (200

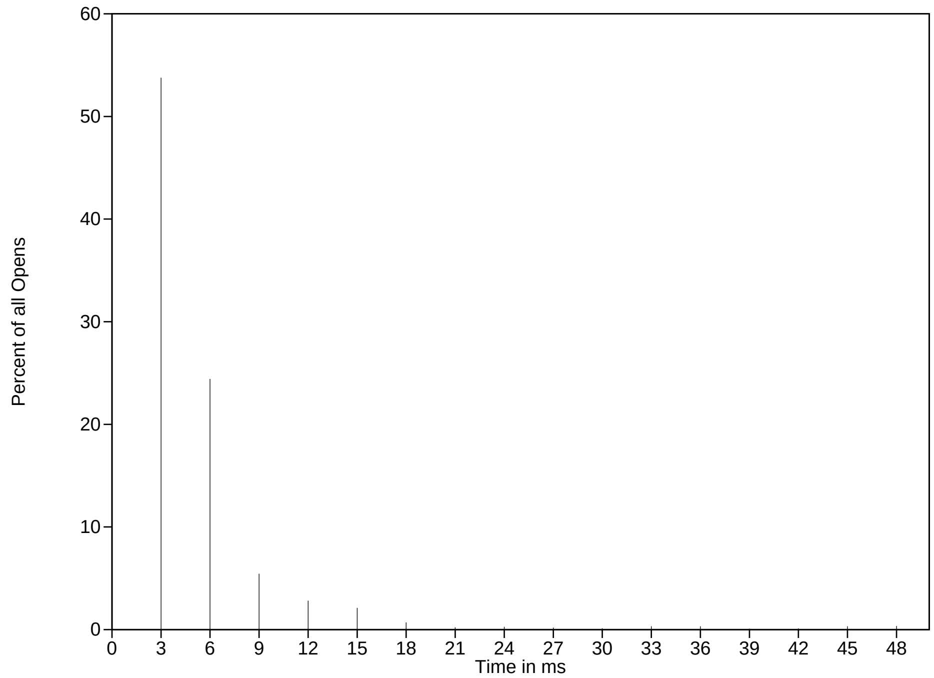

Figure 1: Histogram of times between open and first read of a file.

ms). If the operating system can accurately predict the next file that will be accessed, there exists a sufficient amount of time to prefetch the file.

In a multi-user, multiprogramming environment, concurrently executing tasks may generate an interleaved stream of file requests. In such an environment, reliable access patterns may be difficult to obtain. Even when patterns are discernable, the randomness of the concurrency may render the prefetching effort ineffective. However, analysis of trace data consisting of multiple users (and various daemons) shows that even in a multiprogramming environment accesses tend to be "sequential" where we define sequential as a sensible/predictable uninterrupted progression of file accesses associated with a task. In fact, measurements show that over 94% of the accesses follow logically from the previous access. Thus multiprogramming seems to have little effect on the ability to predict the next file referenced.

5 The Probability Graph

We have designed and implemented a simple analyzer that attempts to predict future accesses based on past access patterns. Driven by trace data, the analyzer 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 a lookahead period of one another. 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 analyzer 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. One graph, derived from the trace described in section 7, generated approximately 6,500 nodes accessed over an eight day period. Each node consumes less than one hundred bytes, and can be efficiently stored on disk in the inode of each associated file, with active portions cached for better performance. Our current graph storage scheme has not been optimized and thus is rather wasteful. We have recently begun investigating methods that will substantially reduce the graph size via graph pruning, aging, and/or compression.

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. The analyzer weighs each arc by the number of times that the second file is accessed after the first file. 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.

Figure 2 illustrates the structure of an example probability graph. The probability graph provides the information necessary to make intelligent prefetch decisions. We define the *chance* of a prediction being correct as the probability of a file (say file B) being opened given the fact that another file (file A) has been opened. The chance of file B following file A can be obtained from the probability graph as the ratio of the number of arcs from file A to file B divided by the total number of arcs leaving file A. We say a prediction is *reasonable* if the estimated chance of the prediction is above a tunable parameter *minimum chance*. We say a prediction is *correct* if the file predicted is actually opened within the lookahead period.

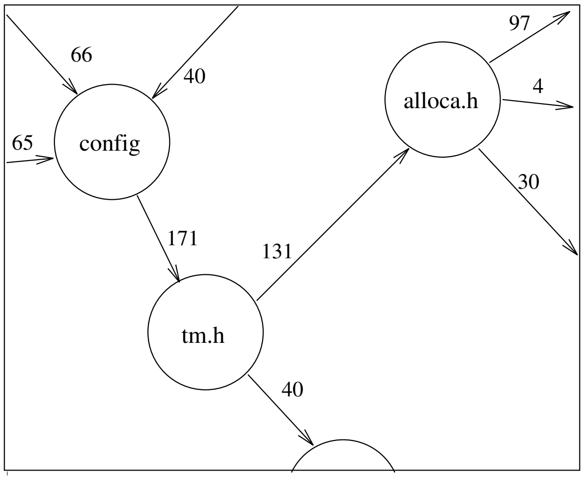

Figure 2: Three nodes of an example probability graph.

Establishing a minimum chance requirement is crucial to avoid wasting system resources. In the absence of a minimum requirement, the analyzer 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 analyzer 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.

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 set of trace data, using a relatively low minimum chance value (65%) the predictor averaged 0.45 files predicted per open. For higher minimum chance values (95%) the predictor averaged only 0.1 files predicted per open. Even when using a relatively low minimum chance (e.g., 65%), the predictor was able to make a prediction about 40% of the time and was correct on approximately 80% of the predictions made.

Figure 3 shows the distribution of estimated chance values with a lookahead of one. The distribution shows that a large number of predictions have an estimated chance of 100%. Setting the minimum chance less than 50% places the system in danger of prefetching many unlikely files. By setting the minimum chance at 50%, very few files that should have been prefetched will be missed. Moreover, the distribution shows how a low minimum chance can still result in a high average accuracy.

6 A Simulation System

To evaluate the performance of systems based on automatic prefetching, we implemented a simulator that models a file system. In order to simulate a variety of file system architectures having a variety of performance characteristics, the simulator is highly parameterized and can be adjusted to model several file system designs. This flexibility allows us to measure and compare the performance of various cache management policies and mechanisms under a wide variety of file system conditions. The simulator consists of four basic components: a *driver*, *cache manager*, *disk subsystem*, and *predictor*.

The *driver* reads a timestamped file system trace and translates each file access into a file system request for the simulator to process. Because the driver generates file requests directly from the trace data, the workload is exactly like that of typical (concurrent) user-level applications. However, the driver must modify the set of requests in a few special cases. Because the simulator is only interested in file system I/O activity, the driver removes accesses made to files representing devices such as terminals or /dev/null. References to

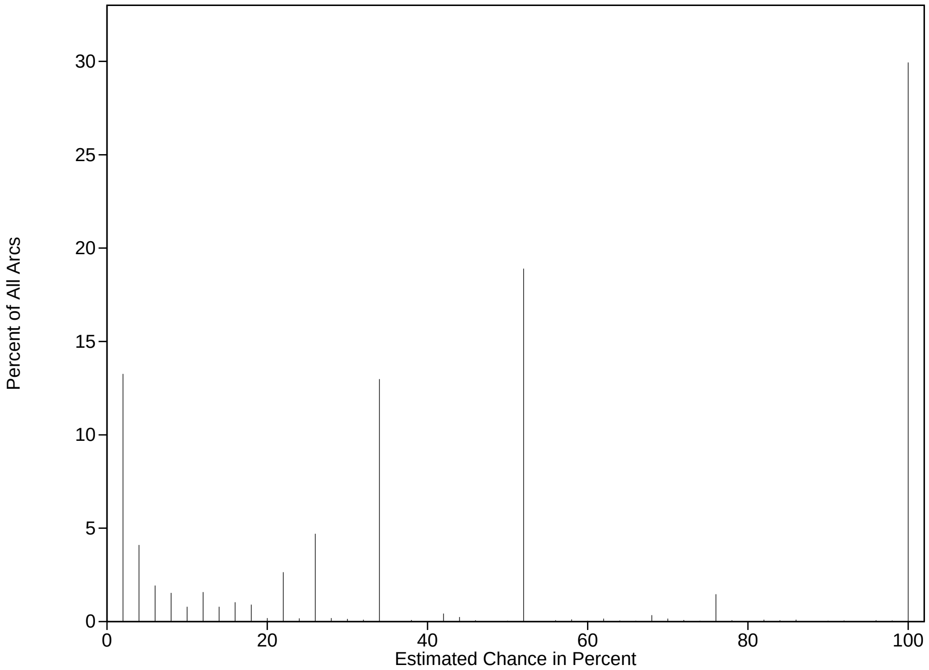

Figure 3: Histogram of estimated chances given a lookahead of one.

certain standard shared libraries such as the C library are also eliminated. Accesses (e.g., mmap() calls) to these libraries rarely require any file system activity, since they are typically already present in the virtual memory cache.

The *cache manager* manages a simulated file cache and services as many requests as possible from the cache without invoking the disk subsystem. We have implemented two cache managers. The first is a standard *LRU cache manager*, where disk pages are replaced in the order of least recent use. The second cache manager is the *prefetch cache manager*. The prefetch cache manager operates much like the LRU manager, updating timestamps on each access and replacing the least recently used page. However, the prefetch manager also updates timestamps based on knowledge of expected accesses from the predictor, thus rescuing some-soon-to-be-accessed pages from replacement. We have found that prefetch cache management can improve performance even if no prefetching occurs (i.e., no pages are actually brought in ahead of time). When run in prefetch mode, the simulator shows that anywhere between 5% and 30% of the performance improvement comes from pages that were rescued rather than actually being prefetched.

The task of the *disk subsystem* is to simulate a file storage device. The current disk subsystem has been configured to emulate local disks. Local disk have relatively low latency when compared to our other target file systems (e.g., wide area distributed file systems, CDROMs, RAIDs, or wireless networks). Consequently, we expect that the performance improvements realized with a local disk model will only be amplified in our other target environments. In the following tests, we assumed a disk model with a first access latency of 15 ms and a transfer rate of 2 MB/sec after factoring in typical file system overhead.

Finally, the simulator contains a *predictor*. The predictor observes open requests that arrive from the driver, and records the data in the probability graph described earlier. The predictor builds the probability graph dynamically just as it would be done in a real system. The longer the simulator executes, the wiser it becomes. On each access the simulator gains a clearer understanding of the true access patterns.

During each open, the probability graph is examined for prefetch opportunities. If an opportunity is discovered, then a read request is sent to the cache manager. If the cache contains the appropriate data, then the data's access time is set to the current time. This ensures that the data will be present for the anticipated need, and possibly rescues the data from an impending flush from the cache. If the prefetch request cannot be satisfied from the cache, then it is prefetched from the disk subject to the characteristics of the disk subsystem.

Notice that the current disk subsystem does no reordering of requests. In particular, it does not preempt or defer prefetch requests to satisfy subsequent appli-

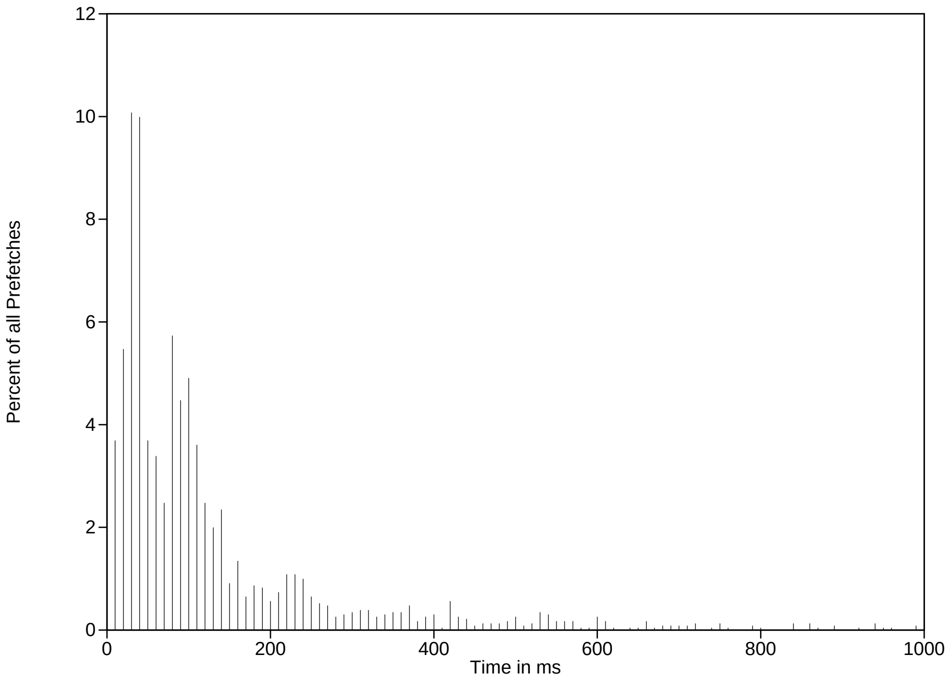

Figure 4: Histogram of times between prefetch and first read access.

cation requests. Reordering and prioritizing requests represents an area of further potential performance improvements.

We are currently in the process of implementing the automatic prefetching system inside a Unix kernel running NFS to measure performance on an actual system.

7 Experimental Results

We performed several tests to measure the performance improvements achieved by automatic prefetching. For the particular set of tests described below, a trace taken over an eight 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.

Recall from section 4, 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 access were prefetched more than 20 ms before the actual need, resulting in cache hits at the time of the first read.

7.1 Prefetch Parameters Effect on Performance

Two parameters that significantly affect the predictions made by the predictor are the *lookahead* and *minimum chance* 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

[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. For the workload in this trace, caches of one megabyte or less would be considered small while caches of three megabytes or more would be considered large. Other traces would produce different values.

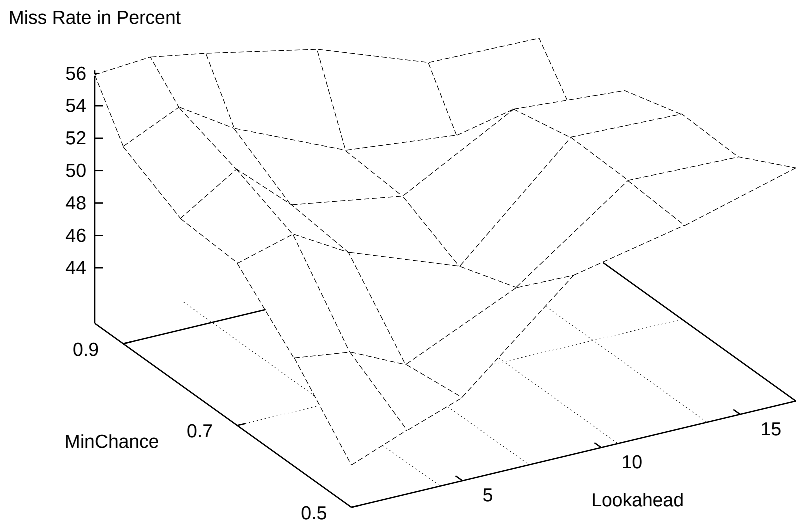

Figure 5: Cache misses as function of lookahead and MinChance for a 400K cache. Performance varies by as much as 13% (between 43% and 56%) depending on the lookahead and minchance settings.

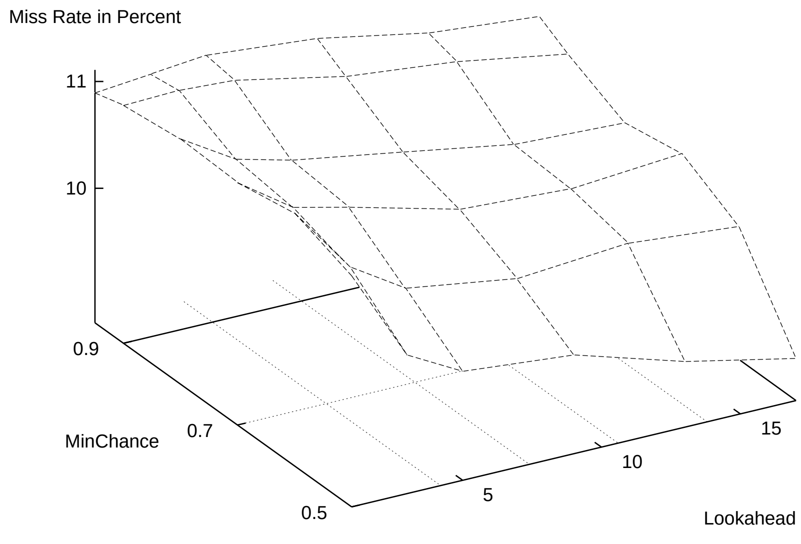

Figure 6: Cache misses as function of lookahead and MinChance for a 4M cache. Performance varies by as much as 2% (between 9% and 11%) depending on the lookahead and minchance settings.

400K Cache Miss Rates (%)							
		Lookahead					
		1	3	5	9	13	17
MinChance	50%	43.2	44.5	45.7	51.6	53.0	54.9
	60%	47.3	46.8	45.2	48.3	53.3	53.1
	70%	50.6	51.6	49.6	47.1	53.5	53.2
	80%	50.8	53.0	50.0	48.9	52.7	52.2
	90%	52.8	54.3	52.2	49.2	48.5	48.9
	95%	55.9	56.2	55.6	54.2	51.8	51.6

Table 1: Data points corresponding to Figure 5.

4000K Cache Miss Rates (%)							
		Lookahead					
		1	3	5	9	13	17
MinChance	50%	10.9	10.0	9.8	9.7	9.4	9.1
	60%	11.1	10.5	10.2	10.0	10.1	9.9
	70%	11.0	10.6	10.5	10.2	10.2	10.2
	80%	11.0	10.7	10.6	10.4	10.2	10.1
	90%	11.0	11.0	10.9	10.7	10.6	10.4
	95%	10.9	10.9	11.0	10.9	10.7	10.6

Table 2: Data points corresponding to Figure 6.

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. Somewhat surprisingly, an aggressively low MinChance value benefits small caches as well. Because the hit rate is low for small caches, correct predictions result in large performance benefits. A low minimum chance increases the total number of correct predictions. For moderate cache sizes, the optimal MinChance is a function of the specific cache size and must limit the number of missed prefetch opportunities without prefetching unnecessary files.

In summary, MinChance should be low (aggressive) for both large and small caches, but higher for intermediate size caches. Lookahead should increase with increasing cache size. Figures 5 and 6 and their associated tables, tables 1 and 2, illustrate these tradeoffs for a 400 KB cache and a 4000 KB cache respectively. Clearly, the Lookahead and MinChance parameters are highly sensative 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 miss ratios. In this case, other factors such as network congestion and processing overhead can be used to aid in the selection of appropriate parameter settings.

7.2 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 number of cache misses to be less than the number of cache misses experienced under standard LRU cache management.

Figure 7 shows the number of page misses that the file system incurred under LRU and under prefetching for various cache sizes. After tuning the above parameters, prefetching performs better than LRU for all cache sizes, in some cases outperforming LRU by as much as 280%. Also note that for the cache sizes shown here, prefetching provided the same or better performance

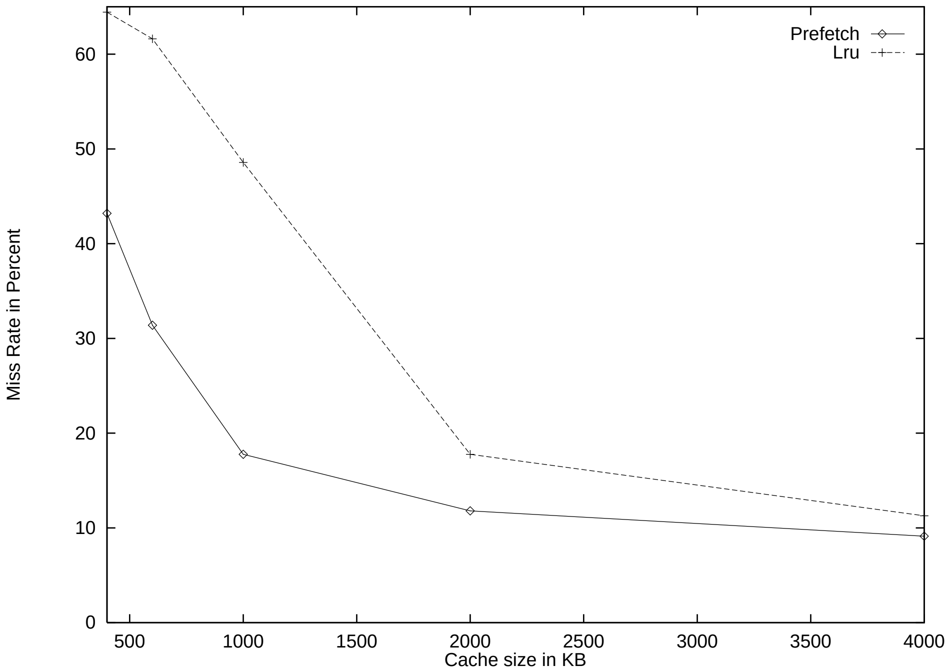

Figure 7: Cache misses as a function of cache size.

than LRU using a cache half the size. This is particularly important for machines that do not have large amounts of memory available for file caching. Even for large memory machines, the ability to achieve similar performance using smaller cache sizes results in more memory for applications. This 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.

For this particular trace, both LRU and prefetching realize relatively little improvement in the miss ratios for caches larger than 4 MB[2]. However, although LRU performance begins to approach prefetch performance as cache size increases, simulations out to cache sizes of 20 MB still show that prefetching results in an 11% reduction in the number of misses as compared to LRU.

8 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, make better use of the available bandwidth via batched prefetch requests, and improve cache utilization. As wide area distributed file systems, CDROM, RAID, and other high latency/high bandwidth systems become prevalent, prefetching will become an increasingly important mechanism toward high-performance I/O.

[2]Like the traces reported in [2], this particular trace consisted of unrestricted real user usage. However, unlike the traces in [2], this trace contained no "heavy users" and thus can achieve reasonable miss rates with a 4 MB cache.

9 Acknowledgements

We would like to thank the reviewers for their helpful comments and suggestions. We would also like to thank Mary Baker for reviewing an early draft of the paper and providing valuable feedback. Finally we would like to thank the DCS users for submitting to being traced.

References

[1] 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.

[2] 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.

[3] James Griffioen and Randy Appleton. Automatic Prefetching in a WAN. In *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 8 – 12, Oct 1993.

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

[5] Geoff Kuenning, Gerald J. Popek, and Peter Reiher. An Analysis of Trace Data for Predictive File Caching in Mobile Computing. In *Proceedings of the 1994 Summer USENIX Conference*, June 1994.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] H. Patterson, G. Gibson, and M. Satyanarayanan. A Status Report on Research in Transparent Informed Prefetching. *SIGOPS, Operating Systems Review*, 27(2):21–34, April 1993.

[12] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9, A Distributed System. In *Proceedings of the Spring 1991 EurOpen Conf.*, pages 43–50, May 1991.

[13] R. Sandberg, D. Goldberg, S. Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network File System. In *Proceedings of the Summer USENIX Conference*, pages 119–130. USENIX Association, June 1985.

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

[15] 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.

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

[17] R. van Renesse, A. S. Tanenbaum, and A. Wilschut. The Design of a High Performance File Server. *Proceedings of the IEEE 9th International Conference on Distributed Computing Systems*, 1989.

10 Author Information

James Griffioen is an Assistant Professor in the Computer Science Department at the University of Kentucky. He received a B.A. in computer science from Calvin College in 1985, and his M.S. and Ph.D in computer science from Purdue University in 1988 and 1991 respectively. He was the recipient of the '89-'90 USENIX scholarship. His research interests include high-performance distributed file systems, scalable distributed shared memory systems, and high-speed network protocols. His email address is `griff@dcs.uky.edu`.

Randy Appleton is a Ph.D student in the Computer Science Department at the University of Kentucky. He received his B.S. degree from the University of Illinois in 1989 and his M.S. from the University of Kentucky in 1992. His research interests are distributed file systems, operating systems, and databases. His email address is `randy@dcs.uky.edu`.