Automatic Prefetching in a WAN

James Griffioen

Department of Computer Science
University of Kentucky
Lexington, KY 40506

Randy Appleton

Department of Computer Science
University of Kentucky
Lexington, KY 40506

Abstract

File systems have become slow relative to processor, memory and network speeds. The high latencies of distributed file systems (particularly wide area file systems) have only exacerbated the problem. We believe that file system prefetching provides an effective method for alleviating the latency bottleneck.

File system prefetching requires knowledge of future file accesses. We are currently investigating efficient methods for accurately predicting future file activity automatically without application intervention. We have designed and implement a system which uses past file activity to predict future file system needs. Our initial results indicate that reasonable accuracy can be obtain even when relatively simple algorithms are employed.

1 Introduction

File system performance has become slower relative to processor, memory, and network speeds [10]. One important reason is the prevalence of distributed and networked file systems, which have an added network latency in addition to standard disk access latency. As distributed systems grow in popularity and geographic size, like those envisioned by the Andrew File System [6], latency and network delays will become the dominant factor in overall file system performance for normal applications.

Unfortunately, there seems little hope on the horizon for substantially improved file system performance. Disk access times seem unlikely to improve at any significant rate, and the trend toward wide area distributed file systems and mobile (wireless) distributed systems with high latencies will only increase file access times. Traditionally, file caching has been used to partially solve the latency problem [8, 2]. However, the performance gains of file caching have already been realized. To improve file system performance further requires new techniques.

1.1 The Limitations of Standard Caching

Caching has been very effective in reducing the number of high latency disk accesses by retaining file data in memory in hopes that the data will be used again in the very near future. When the data reference by a file access can be found in the cache, the file access latency is reduced to that of a standard memory access. Reduced file access latency can improve the performance of almost any application, even CPU bound applications, such as a compile or a long-running simulation. Patterson et. al. [11] have noted that compiles of simple programs run 20% faster when file caching is used as opposed to a system without caching, and text searches 30% faster. Many programs show even more benefit. Unfortunately, file system caches suffer from several problems.

One problem with existing file system caches is their inefficient use of large amounts of memory. Experience shows that standard caching is unable to capitalize on the availability of very large memories. It has been noted by many people that the relative benefit of caching decreases as cache size (and cache cost) increases [8, 1]. For example, measured performance of a standard Unix[1] workstation in our lab shows that approximately 72% of all disk access can be met from a two megabyte cache, but doubling the size of the cache only provides an additional 7% improvement. Further doubling provides even smaller improvements. Moreover, for large cache sizes standard LRU cache replacement is reasonably close to optimal [1, 2]. Therefore, we can expect little additional improvement in performance from new cache replacement strategies, since the standard caching techniques already perform at a rate close to optimal (see Figure 1).

A second problem with caching is the fact that file system caches do nothing to reduce the latency of the first access to a file. Obviously, files that have never been referenced before will not be in the cache and

[1] Unix is a registered trademark of AT&T.

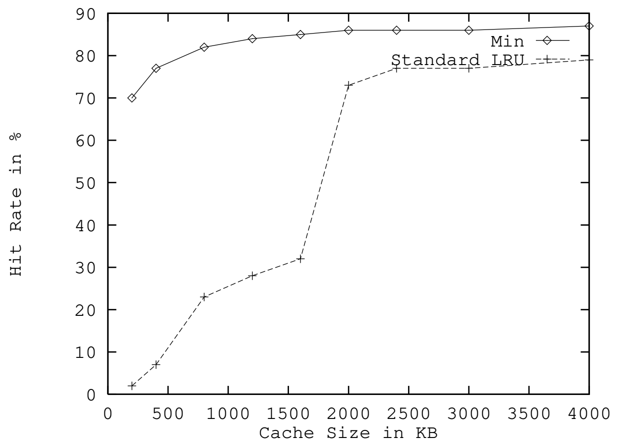

Figure 1: **The measured hit rate for varying size caches of a trace taken over a two day period. For larger cache sizes, LRU approaches the optimal hit rate.**

therefore experience high latencies because they must be retrieved from the file server. However, even files that have been accessed at some time in the past may experience high first access latency. Note that:

- Most file system caches are not large enough to hold all the files in a user's (or multiple user's) "working set". For files in the "working set", but not present in the cache, each access appears as if it were the first access with latency dominating the overall access time.

- 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, 3]. For files of this size, transfer time is minimal when compared to the access latency across a WAN. As a result, access latency becomes the dominate cost in accessing the file.

- 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. This is especially true of small files.

2 Enhanced Caching Via Prefetching

File access latencies are critical to file system performance. For many applications, latency is even more important than throughput. Recent trends, like the growth of distributed file systems, will cause the latency problem to continue or even worsen. We are investigating methods to solve the latency problem.

One way to reduce the perceived latency of a file request is to transfer the file from the file server to the local client cache before the file is really needed. Prefetching files before they are needed also allows the file transfer to overlap the current computation. However, to ask for a file before it is needed requires substantial fore-knowledge.

2.1 Related Research

One relatively straight forward solution is to have each application program inform the operating system of its future requirements. This has been proposed and is currently being researched by several researchers including 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 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 (i.e., it is an attempt to use prefetching to improve cache performance), we see several disadvantages to this solution. The first disadvantage is that it requires that all existing applications be re-written to provide the necessary information. Moreover, the programmer must learn a reasonably complex set of additional system directives that must be strategically place, and, if incorrectly used, can severely degrade performance. Our approach (described below) eliminates this requirement.

A second problem is that the operating system needs a significant lead-time to insure the file is available when needed. 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 it is required. But many applications do not know which files they will need until the actual need arises. For instance, the preprocessor of a compiler cannot know the pattern of nested include files until the files are actually needed, nor will an editor necessarily know which files a user normally edits. Our approach reduces this problem by predicting the need for a file well in advance of when the application could.

A third problem with this approach arises in situations where related file accesses span multiple executables. In these situation, like an edit/compile/run cycle, or certain commonly run shell scripts, no one application knows the cross-application file access patterns, and therefore no application can inform the operating system of them. Our approach uses long term history information across applications to avoid these last two problems.

Finally, the application, because it executes outside the kernel, cannot make informed decisions about when (or even if) a particular file should be prefetched. The information required to make such informed decisions resides in the kernel, inaccessible to the application (e.g., the processor speed, the current load average, the latency to the server, the current server load, etc.). Our approach incorporates system activity information to make well-informed predictions.

2.2 Automatic Prefetching

We are investigating an approach we call *automatic prefetching*, in which the operating system rather than the application predicts future file requirements. Our research indicates that future file access information can often be derived (extrapolated) from a global history of all past file accesses. We are investigating new approaches that make use of past history information to predict future access patterns. For example, a recent history of a programmer's file activity can identify which files include other files, and which files a certain user is likely to edit. The operating system can passively gather this history information as needed, and from it deduce patterns. Once these patterns are known, the operating system can prefetch files to optimize performance without the help of application programs.

Automatic prefetching has several advantages over existing approaches. First, applications need not be rewritten to achieve improved performance. In addition, the programming model is not complicated by new procedure calls which must be strategically placed to inform the file system of files to be prefetched. Second, non-intuitive access patterns, unknown to the average programmer, are automatically identified and used to improve applications performance. Third, performance is optimized across sets of independently written applications executing in sequence or simultaneously. Fourth, because the operating system ultimately performs the prefetching, an application's environmental conditions (e.g., processor speed, current load average, file server latency, current server load, etc.) can be used to further optimize performance. Moreover, multiple file requests can be delayed and batched together to reduce network congestion and file server load.

2.3 The Probability Graph

To show that history driven analysis and prefetching is indeed possible, we modified a SunOS 4.1.1. kernel to gather file access traces. We captured traces from a variety of activities, including the normal daily usage of several researchers, and several synthetic workloads. These traces consist of an ordered listing of every open(), read(), write(), creat(), lseek(), mmap(), exec(), etc. call made by the system, detailing files affected, the process ID of the caller, as various other miscellaneous book-keeping information. Given the traces, we also designed and implemented a simple analyzer that attempts to predict future access patterns from the file system traces. The analyzer works as follows:

Using the trace data, the analyzer creates a logical graph called a *Probability Graph*. Each node in the graph represents a file in the file system. Executables are considered files, and are also represented by nodes. In the Probability Graph directed arcs represent psuedo-consecutive file accesses. If the open for one file follows within a *lookahead* period of the open for a second file, an arc is drawn from the first to the second. These arcs represent an ordered lists of files that were demanded of the file system. If one file exec()s another, the exec() is treated just like an open() system call, and an arc is drawn from previously opened files to the exec()ed file. Treating exec()s in this way includes accesses to executables correctly into the ordered list of all accesses.

The lookahead period represents how close two file opens must be for an arc to be created between the files. That is, it describes the number of intervening opens allowed between two "close" opens. Larger lookaheads will result in more arcs for a given trace. We used lookaheads of one, four, eight and twelve for our experiments.

Clearly, more elaborate graphs and data analysis can be constructed which would use more of the captured data. We are investigating new algorithms that use more complex hueristics; however, if an automatic prefetching system is to be successful, it must be non-intrusive, adding the least amount of additional processing overhead possible. As a result, our initial goal was to evaluate the performance of very basic and simple algorithms and mechanisms to determine the viability and usefulness of automatic prefetching.

3 Results

Given various trace data, each gathered over a period of a few day with various workloads, our analyzer constructed probability graphs for each trace. Given a particular probability graph, we are able to deduce the probability that some file B will be opened, assuming that file A was just opened. For example, in one trace, the file `config.h` was opened 171 times and file `tm.h` followed it immediately all 171 times. Therefore it is a good guess that the next time `config.h` is opened that `tm.h` will be needed soon. To continue the example, `tm.h` was opened 171 times, and 133 times (78%) the next file needed was `/usr/include/alloca.h`. Inspection of the files in question would reveal that they do in fact call upon one another in the given order. The analyzer was able to deduce this without knowing any other semantic information about the files. Figure 2 illustrates the structure of an example probability graph.

Figure 2: **An example probability graph constructed from trace data. The figure shows three of the nodes from the probability graph and their "closeness" in terms of immediately preceeding opens.**

In some cases, the system was able to detect access patterns that a human programmer might be unaware of. For example, in one trace the command 'ln' was followed by the command 'rm' fifty percent of the time. Although the pattern is non-intuitive, it occurred many times over the trace period.

3.1 Making Good Predictions

Given a probability graph, one obvious (and important) question that arises is: "When is a pattern a good predictor of future file accesses?" We say a prediction is *correct* if the file predicted is actually opened within the lookahead period, which starts when the prediction is made. Otherwise the prediction is said to *fail*. We define the *chance* of a prediction 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. We generally defined a 50% chance as reasonable.

To aid the prefetching mechanism, we also define the *accuracy* of a set of predictions. The accuracy of a set of predictions is the number of correct predictions divided by the total number of predictions. Note that the accuracy of a set of predictions is not the same as the *chance* described above. We use the chance value when deciding whether to prefetch a file or not. Consequently, the resulting accuracy will always be at least as large as the minimum chance, and in practice is usually substantially higher. Perfect knowledge would be represented by an accuracy of 100%.

Of course, the ability of the analyzer to make correct predictions or even reasonable predictions depends to some extent on type of workload. The type of file activity is important because it is easier for the analyzer to make predictions when file accesses are regular (like an edit/compile/run cycle) than when they are irregular (completely random accesses). Fortunately our experience and others indicates that users tend to execute the same small set of operations repeatedly [5]. As a result, all of our traces have produced probability graphs with surprisingly high accuracy values.

Establishing some minimum accuracy requirement is crucial to avoid wasting system resources. If there were no accuracy requirements, the analyzer could (and would) produce multiple predictions on every file access, most of which would be incorrect. Assuming each predicted file was automatically prefetched, the system would waste substantial amounts of time prefetching files that would not be needed with the added side-affect of removing necessary files from the cache.

Consequently, selecting an appropriate "reasonable prediction" threshold is critical to overall system performance. Setting the threshold low increases the total number of successful predictions, but reduces the *correct/failure* ratio resulting in many useless prefetches. Setting the threshold high decreases the total number of successful predictions, but increases the *correct/failure* ratio. Figure 3 shows how accuracy falls off as the number of predictions increases.

We used the analyzer to predict file accesses for a variety of traces representing a variety of typical workloads. One trace, representing a large compile, resulted in the analyzer making a prediction on 37%

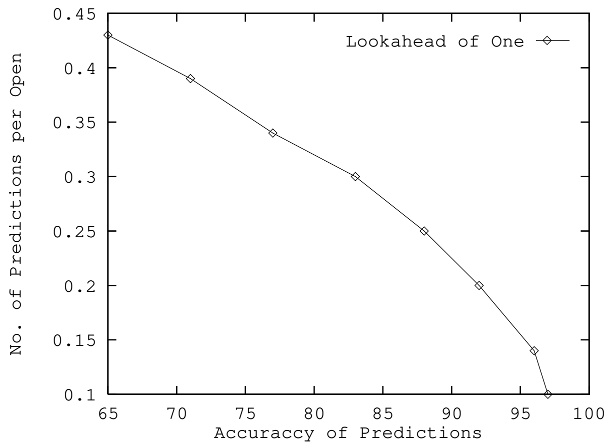

Figure 3: **The number of predictions that can be made drops as the required accuracy increases.**

of the file access with a lookahead of only one, and was right for 80% of all files predicted. In a trace of user activity over two days, the analyzer performed even better, making a reasonable guess 44% of the time, and was correct for 78% of all guesses.

Although the success of the analyzer depends on the workload, we found that in general, the analyzer is able to make a reasonable guess about the next file accessed 40% of the time, and is right about 80% of the time. Even though we are happy with these results, we anticipate future improvements in the ability of our analyzer to predict future file needs as its development continues.

We have also been able to determine that many of the files being predicted do not represent files already in the cache. Many of these files would not be present in the cache under standard LRU, and thus represent strong candidates for file prefetching.

4 Future Work

We are in the process of turning these ideas into a working distributed system that performs file prefetching. We are currently investigating the use of a new user-level daemon similar to the unix 'pagedeamon' daemon. Using this approach, the kernel would be modified to offer needed data on file accesses and cache state to the new daemon. The daemon would analyze the data, and then issue prefetch requests using new system calls to the kernel. The only difference that the average user would see would be a substantially faster file system, especially for files located across the network from the user.

There are several areas left for us to explore. We intend to experiment with differing methods of prefetching files into the cache. We are also interested in improving cache management using the ability of our analyzer to predict future file accesses.

5 Conclusion

File systems have become slow relative to processor, memory and network speeds. Distributed file systems have only exacerbated the problem. We believe that file system prefetching provides an effective method to overcome this bottleneck.

File system prefetching requires knowledge of future file accesses. We have developed a working system that allows us to deduce future file system needs based on past activity. It is our intent to build upon this and create a working distributed system with file prefetching. We believe that the performance gain will be substantial.

References

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

[2] 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.

[3] 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.

[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] Balachander Krishnamurthy. *A Uniform Model of Interaction In Interactive Systems.* PhD thesis, Purdue University, West Lafayette, Indiana, December 1987.

[6] 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.

[7] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba:

A Distributed Operating System for the 1990's. *IEEE Computer*, 23:44–53, May 1990.

[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] 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.

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