Performance Measurements of Automatic Prefetching

James Griffioen
Randy Appleton

Department of Computer Science
University of Kentucky
Lexington, KY 40506
{griff,randy}@dcs.uky.edu
(606) 257-3961

October 18, 1994

Abstract

File systems have become slow relative to processor, memory, and network speeds. The high latencies of distributed file systems, as well as newer, slower devices like CD-ROMs only exacerbates the problem. We believe that *automatic prefetching* [GA94] can effectively minimize the latency problem thereby allowing file systems to keep pace with processor and memory improvements.

This paper examines the performance and stability of automatic prefetching. Our results show the automatic prefetching provides significant performance improvements for any reasonable setting of the algorithm's parameters. The performance measurements show our method can reduce cache miss rates by as much as 70% depending on the cache size. Further, the time needed to complete the average read by a set of benchmark applications can be reduced by up to 42%. Alternatively, a prefetching cache can perform as well as an LRU cache of twice the size.

1 Introduction

File system performance has become slow relative to processor, memory, and network speeds [Ous90]. 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. The popularity of newer, slower file system devices like CD-ROMs only exacerbates the problem. As distributed systems grow in popularity and geographic size, like those envisioned by the Andrew File System [MSC+86], latency and network delays will become the dominant factor in overall file system performance.

We have developed a method called *Automatic Prefetching* that can reduce these high latencies. By observing file access patterns, and predicting future usage, we are able to reduce the cache miss rate by as much as 70% for certain cache sizes. Further studies have shown that the average time needed by read operations across a set of standard applications can be reduced by up to 42%.

This paper examines the performance and stability of automatic prefetching under a variety of file system conditions using trace driven simulations. The paper begins with a brief overview of automatic prefetching and outlines the basic algorithm and prefetching procedure. We refer the interested reader to [GA94] for addition details regarding the algorithm. The remainder of the paper focuses on our simulations and test results.

1.1 The I/O Bottleneck

File system I/O represents one of the biggest obstacles to overall operating system performance [Ous90] and can be divided into two problems: bandwidth and latency. Recent advances in the area of bandwidth have been impressive [PGK88], however latency remains a problem and will only be exacerbated by wide area distributed file systems, wireless file systems, and devices such as CD-ROMS. Moreover, disk and network latencies are not expected to improve much in the future and ultimately are limited by physical constraints.

Caching has been very effective in reducing the number of high latency disk and network accesses by retaining file data in memory in hopes that the data will be used again in the near future. Despite the benefits of caching, latency still contributes heavily to the file system bottleneck. Moreover, it has been noted by many people that the relative benefit (cost effectiveness) of caching decreases as cache size increases [NWO88, App93]. Thus larger caches do not provide much promise for alleviating the latency problem. Moreover, for large cache sizes standard LRU cache replacement is reasonably close to optimal as defined by Belady's Min [BAD+92]. Therefore, we can expect little additional improvement in performance from new cache replacement strategies.

Another shortcoming of caching is the fact that caching does nothing to reduce the latency of file accesses which appear to be the first access to the file. For example, files accessed quite some time ago but no longer in the cache experience large latencies as if it were the first time the file was being accessed. Also, file system caches are often not large enough to hold all the files in a user's (or multiple user's) "working set". For files referenced recently in the "working set" but not present in the cache, each access appears as if it were the first access to the file with latency dominating the overall access time.

Further, most files are quite small with the average file being a few kilobytes in length [OCH+85, BHK+91]. For files of this size, transfer time is minimal when compared to the access latency. As a result, latency becomes the dominate cost in accessing the file.

Finally, 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.

2 Enhanced Caching Via Prefetching

File access latencies are critical to file system performance. Because the average file size is quite small, latency is often even more important than throughput. Moreover, recent trends, like the growth of distributed file systems, will cause the latency problem to continue or become worse.

One way to reduce the perceived latency of a file access is to transfer the file from the file server to the local client cache before an application requests the file. Prefetching files before they are needed also allows future file transfer to overlap the current computation. We are investigating automatic methods for file prefetching that require no user intervention and provide better performance than standard caching.

2.1 Related Work

Prefetching is certainly not a new idea and is already in use in several file systems [KE90, MJLF84]. Many Unix's perform intra-file prefetching, usually 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 represent 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.

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. [PGS93]. 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 open-

ing 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. In combination, we feel these disadvantages will limit the desirability and effectiveness of this solution.

Other researchers have proposed using the history of file accesses to deduce patterns of file activity, and to prefetch files based on these patterns. In particular, Tait and Duchamp employ *trees* of processes and record file activity (working sets) with these trees. The operating system then attempts to find patterns within these trees, and take prefetching actions accordingly. Although their algorithm suffers from a variety of problems [TD91], their initial hit rate analysis using a very simple "fixed number of files cache size" also indicates that prefetching can produce substantial performance gains. However, as we will show later, hit (miss) rates are not necessarily a good or accurate indicator of a system's performance. Consequently, substantially more study is needed to accurately determine the effectiveness of their approach.

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 from a global history of all past file accesses. 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. 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 guide prefetching decisions. Finally, multiple file requests can be batched together to reduce network congestion and file server load. The following section briefly describes the automatic prefetching algorithm.

2.2.1 The Probability Graph

The automatic prefetching algorithm records past file access in a *probability graph* and then uses the graph to predict future file accesses and prefetch the necessary files. Simply stated, the probability graph stores information about past access patterns and maintains the probability of one file being opened "soon" after another. The probability graph is built on-the-fly and continues to evolve and change as access patterns change.

Before describing the details of the probability graph, we must define the *lookahead period* used to construct it. The lookahead period describes what it means for one file to be opened "soon" after another file. We define the lookahead period to be a fixed number of file open operations that occur after the current open. For example, if the lookahead period is one, then the next file opened is the only file considered to have been opened soon after the current file. If the lookahead period is five, then any file opened within five files of the current file is considered to have been opened soon after the current file. We treat Unix exec system calls like opens and thus include them in the probability graph.

We defined the lookahead period using virtual time rather than physical time for a number of reasons. First, precise physical timestamps are not always easily available or may be costly to obtain. Second physical timestamps would cause some additional complexity in the algorithm. Third, physical time may not be a good definition of "soon" and may be a very sensitive parameter given the variation in application execution times and file access patterns. Finally, virtual time is simple and efficient to implement and has worked well in practice. Moreover, we feel it provides a better definition of "soon" for the purposes of prefetching.

The probability graph is an directed graph where each node represents a file in the file system. Arcs between nodes 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. The analyzer weighs each arc by the number of times that the second file is accessed after the first. 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.

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

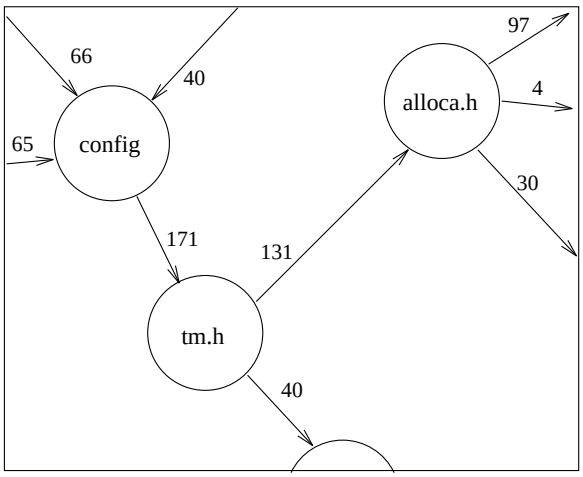

Figure 1: Three nodes of an example probability graph.

estimated chance of the prediction is above a tunable parameter *minimum chance*.

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. Establishing a correct lookahead is also important. A larger lookahead causes more files to be considered related and causes the probability graph to grow in size. Finally, it causes the prefetch algorithm to look further into the future when considering prefetch opportunities.

Each host dynamically builds a single probability graph containing all files accessed on that host. We considered per user graphs but found a single graph acceptable [GA94]. Figure 1 illustrates the structure of an example probability graph.

2.2.2 An Example Probability Graph

To show that history driven analysis and prefetching is indeed possible, we modified the SunOS 4.1.1. kernel to gather file system 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 a timestamped list of every open(), read(), write(), creat(), lseek(), mmap(), exec(), etc. call made by the system, detailing files affected, the process ID of the caller, and various other bits of miscellaneous book-keeping information.

We also designed and implemented a simple analyzer that dynamically builds a probability graph from our traces and attempts to predict future access patterns using various settings of the lookahead and minimum chance.

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 the portion of the trace shown in figure 1, 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`. Therefore it is a good but not perfect guess that the next time `tm.h` is opened `alloca.h` will be needed soon. Inspection of the files in question reveals that they do include each other in the given order. The analyzer was able to deduce this without knowing any other semantic information about the files.

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 'ln' program was followed by the 'rm' program fifty percent of the time. Although the pattern is non-intuitive, the pattern occurred many times over the trace period[1].

3 Test Environment

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 and 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 up until the first one finishes. For our simulations, the local disk latency (seek time) was assumed to be 12ms, after taking into account software latencies. The disk throughput was assumed to be 2 MB/sec, after taking into account both software delays and any need to 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

[1] We eventually tracked this down to a series of reasonably complex shell scripts being run by a user.

requests. Concurrent requests are, however, serialized at the remote disk drive. For simplicity our remote disk model does not include a simulated server cache. The performance improvements resulting from server cache hits, when averaged over a large trace, would effectively result in a slightly lower average remote disk latency for small caches. When client caches are even moderately large (1.6MB), server cache hit rates drops significantly [WEB93]. In addition, our results do not include congestion/contention at the server and network that occurs in the real world, which helps offset the missing server cache.

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 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 [GA].

We used two different traces for these experiments. The first trace consisted of several users performing typical academic activities (e.g., reading mail/news, file cat'ing/scanning, editing, text processing, and some compilation/debugging). It consisted of approximately 400,000 I/O operations involving 400 MB of data and had 1.284 reads for every write. The second trace involved only two users working on a large multimedia programming project (i.e., almost exclusively edit, compile, run/debug cycles). It consisted of approximately 400,000 I/O operations involving 770 MB of data and had 1.52 reads per write. Although all simulations were done starting with a cold cache, the length of these traces was more than sufficient to compensate for any startup effect.

4 Performance Results

We had several goals in doing our performance evaluation. First, we wanted to measure the performance of the prefetch algorithm. In particular, we wanted to measure the performance gains possible relative to standard caching (LRU file caching). Second, we wanted to measure the sen-

sitivity of the prefetch algorithm to changes in parameters and workloads. Because prefetching systems estimate future accesses based on past accesses they are particularly susceptible to instability. Also, the parameters to the prefetch algorithm have the potential to significantly affect the performance either positively or negatively. Third, we felt it important to determine how measured hit rates were related to broader measures of performance, such as the time needed to complete a set of benchmark tasks.

The following section presents the details and rational of the specific performance measures we use. Section 4.2 compares the performance of a prefetching file system with a standard LRU file system. Section 4.3 presents a simple analysis of the stability of our prefetch algorithm and its sensitivity to workload and parameter settings.

4.1 Performance Measurements

Two different measures were employed to evaluate the system's performance: *miss rates* and *read complete times*. *Miss rates* represent the standard measure used to evaluate caching strategies and serves as a good point of comparison with other systems [BHK+91, BAD+92, TD91, PGS93]. Further, miss rates describe how many disk and/or network resources the file system is using. This can be important when these resources are shared with other components of the operating system, for example the virtual memory subsystem.

However, miss rates are not necessarily the best measure for prefetching algorithms. A better measure of performance improvement is the actual speedup realized by the applications; in particular, the speedup resulting from faster reads. To quantify this speedup, we compute the *read complete time* of a trace as the amount of time applications spend waiting for read operations to complete. As we show in section 4.2, miss rates do not always reflect the true speedup achieved, whereas read complete times more accurately portray the true speedup.

We compute the read complete times for a trace by summing all the time all applications spend waiting in the disk queue for read operations to complete. If multiple reads from independent processes occur simultaneously, each read operation contributes its entire wait time to the total to reflect the fact that multiple processes have been delayed. In other words, the read complete time for a trace is the total amount of time that the user who generated the trace would have waited on the file system for all of the applications run during that trace.

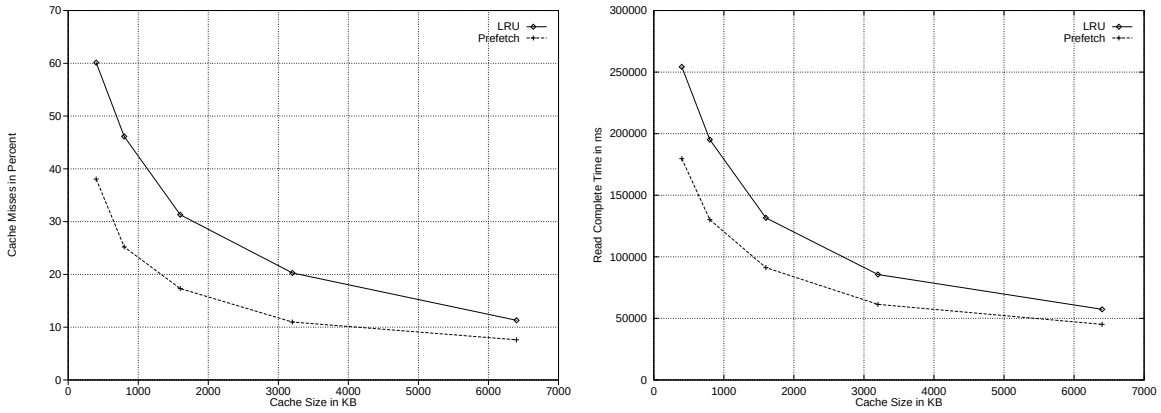

Figure 2: **(a) Cache misses as a function of cache size for local disks. (b) Read Complete Times as a function of cache size for local disks**

4.2 Automatic Prefetching vs. Standard Caching

To illustrate the speedup achievable with automatic prefetching, we ran the simulations using standard LRU caching and then using our automatic prefetching algorithm. To see the effect latency has on performance we also ran the simulations using a local disk model and a remote file system (network disk) model.

Figures 2a and 2b show the hit rates and read complete times under LRU and automatic prefetching when using a local disk model. Prefetching performed better than standard LRU under all cache sizes, at times performing up to 70% better. In fact, a prefetching cache performs about as well as an LRU cache of twice the size. This 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.

Figures 3a and 3b shows the miss rate and read complete times using a networked disk model. Prefetching again performs better than LRU for all cache sizes. Although these graphs are substantially similar to the previous two graphs, there are some differences. The relative performance improvement is greater for a networked disk than for a local disk. We believe this is because there is more latency, and therefore more potential improvement, when using a networked disk.

Also, notice that there are more cache misses when prefetching from a networked disk than from a local disk, even though both simulations ran the same trace, and even though both a local disk and a networked disk have the same number of cache misses under LRU. These extra misses represent prefetches that were initiated such that they had time to complete from a local disk but

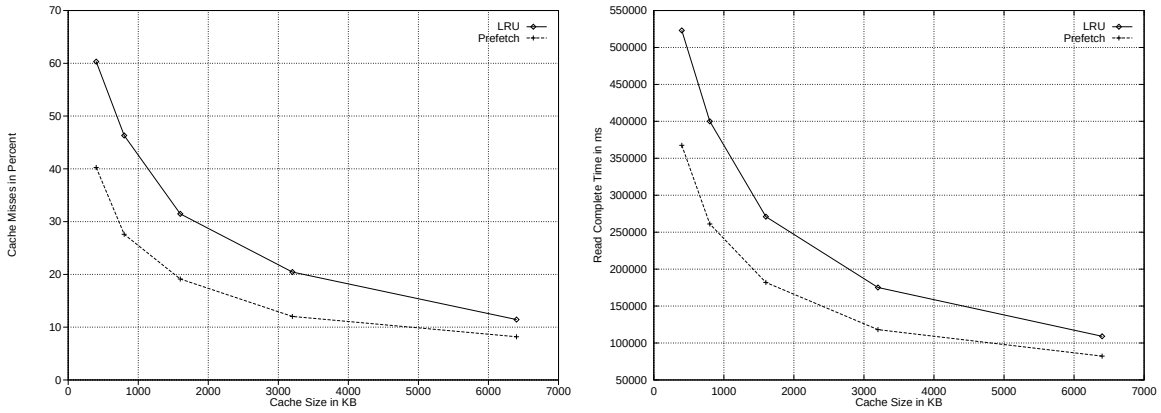

Figure 3: **(a) Cache misses as a function of cache size for an NFS mounted disk. (b) Read Complete Times as a function of cache size for an NFS mounted disk.**

not from the higher latency networked disk.

4.3 Stability of the Algorithm

In order for automatic prefetching techniques to be successful, the algorithms used must be capable of adapting to a wide variety of operating environments. Moreover, the technique must not result in worse performance than a standard (non-prefetching) file system in such environments.

The adaptability of our prefetch algorithm depends partially on the sensitivity of the prefetch parameters lookahead and minimum chance. If performance changes drastically as a result of minor changes to the parameter settings, the algorithm will be unstable. Because the optimal[2] settings are unknown ahead of time the settings must be estimated and are unlikely to be correct. Small deviations from optimal must not produce large performance degradations. Stability also depends on file access patterns and file system characteristics. The algorithm should perform no worse than a non-prefetching algorithm regardless of the file system access patterns, cache size, latency, or bandwidth.

To evaluate the robustness and stability of our prefetching algorithm, we simulated the algorithm across a range of file system environments (cache size, file access patterns, and device latencies and throughputs) and parameter settings (lookahead and minimum chance). The results of these tests also help establish guidelines for selecting optimal parameter settings for each of the simulated environments.

[2]Throughout this section, we use the word *optimal* to denote the parameter settings that produce the best performance.

4.3.1 Stability Results

Figures 4 - 7 show the performance of the prefetching algorithm for various parameter settings (lookahead and minimum chance) and operating environments (cache size and workload).

As described earlier, miss rates and read complete times are only loosely related. Comparing figure 4 and figure 6 clearly show that the miss rate performance measure produces significantly different curves than the read complete time performance measure. Because the two measures are different, we will address each separately. However, note that the simulation results under both measures clearly show that the optimal setting for the lookahead and minimum chance parameters depends heavily on the cache size but only moderately on the workload (i.e., the different traces we collected).

Miss Rates

Figures 4a and 5a shows that the smaller cache (400K) performs best when the lookahead value is low. Smaller caches simply do not have room to hold data needed far into the future. For smaller cache sizes, large lookaheads tend to prefetch data too far in advance of the need. This fills the cache with prematurely loaded data and forces necessary data, needed in the immediate future, from the cache. Since small lookaheads limits prefetches to files need in the near future, smaller caches perform better with smaller lookaheads.

Figures 4b/c and 5b/c indicate that larger cache sizes obtain the best performance when the lookahead value is set reasonably high. Both the 1.6MB cache and the 6.4MB cache achieve the best performance when the algorithm looks 10 files into the future for prefetch opportunities. Although our simulations only show lookaheads out to 10, the performance remains constant (or becomes worse) for lookaheads greater that 10. This occurs because few of the access patterns in our traces benefit from bring in more than 10 files before they are needed. Unlike small caches, larger caches have sufficient space so that the system can load data needed far in the future without removing cache data needed in the immediate future. Further, predicting a data access substantially earlier than the actual access itself allows the system to ensure that the data will be present in the cache before the access occurs.

Notice that optimum performance is obtained by setting the minimum chance value aggressively low (40%) for both large caches and small caches[3]. Setting the minimum chance low increases the number of files considered to have an acceptable probability of being accessed in the near future, resulting in more prefetching. As was the case with the lookahead, large caches can handle

[3]Note that the average probability will be significantly higher than 40%.

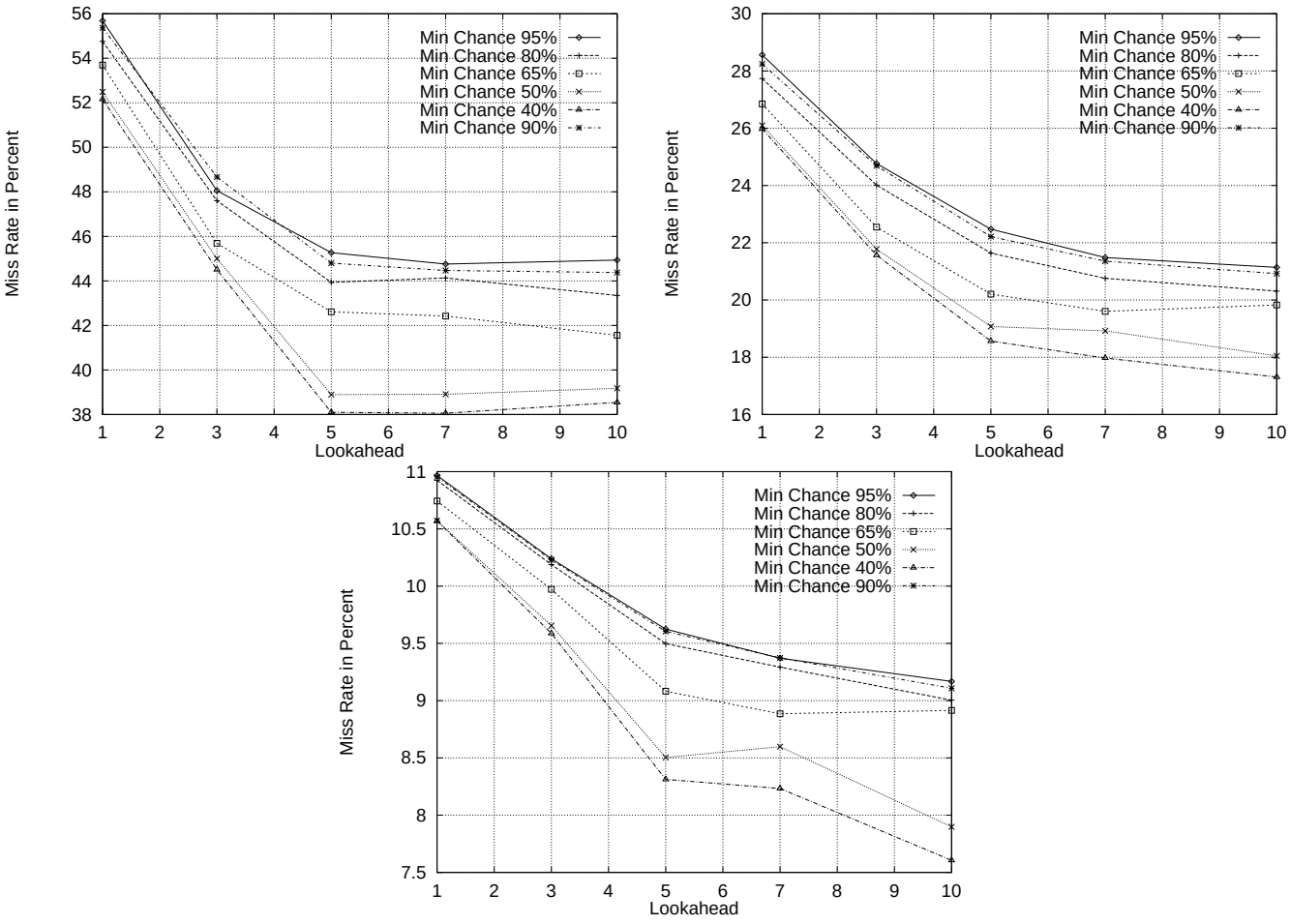

Figure 4: Trace 1: The number of cache misses as a function of the lookahead. Each line represents a different minimum chance value. The caches sizes used are (left to right): (a) 400K, (b) 1600K, (c) 6400K.

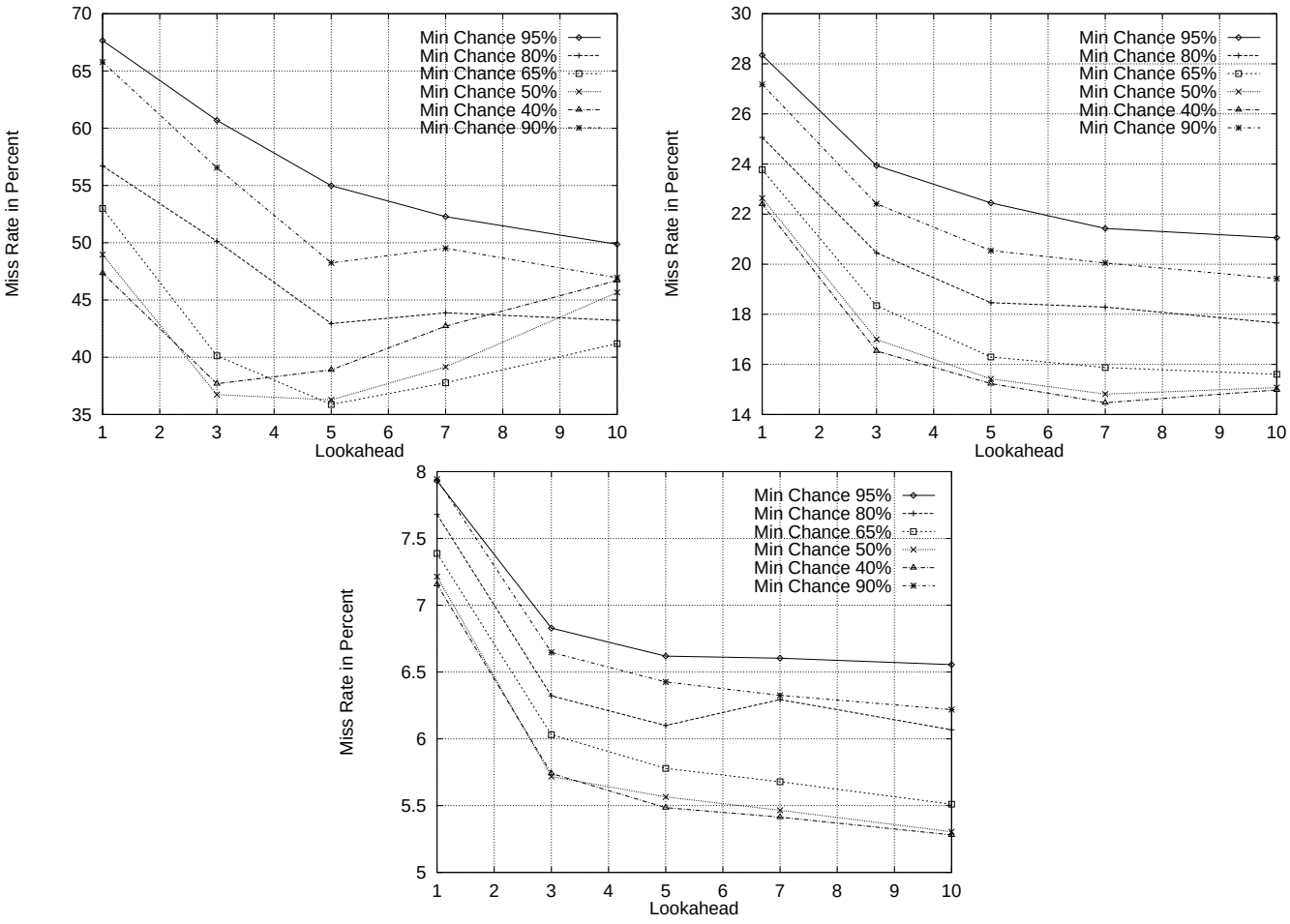

Figure 5: Trace 2: The number of cache misses as a function of the lookahead. Each line represents a different minimum chance value. The caches sizes used are (left to right): (a) 400K, (b) 1600K, (c) 6400K.

substantial amounts of prefetching without forcing necessary data from the cache. However, the same argument does not hold true for smaller cache sizes. Intuitively, smaller caches cannot afford to prefetch large amounts of data because necessary data will be flushed from the cache. However, unlike large lookaheads which bring in data needed far into the future, low minimum chance values bring in data potentially needed in the short term (assuming a small lookahead). Even when the probability is relatively low (40%), these files are more likely to be needed than the oldest files already in the cache. In summary, minimum chance should be low (aggressive) for both large and small caches. Lookahead, on the other hand, should increase as the cache size increases.

The results also indicate that the algorithm provides acceptable stability when "reasonable' lookahead and minimum chance values are chosen. That is, the algorithm is not excessively sensitive to small variations in parameter settings around the optimal value. Small perturbations from the optimal setting only produce small degradations in performance. Also note that the cache size, not the file access pattern, dictates the optimal parameter settings. For any given cache size, both traces, despite their different access patterns, produced approximately the same optimal parameter settings. Low sensitivity to slight changes is particularly important because the system will rarely select the optimal parameter values since file system conditions are not known ahead of time, and often change dynamically.

Read Complete Times

Figure 6 show the read complete times as a function of lookahead and minimum chance for the same cache sizes and disk model as the previous graphs. Figure 7 shows the same data as Figure 6 but displays it as 3-d contour map. Notice that the read complete time performance measure produces substantially different results than the miss rate performance measure. In particular, the read complete times would suggest substantially different optimal settings for the lookahead and minimum change parameters.

Before discussing the results, recall that the disk model queues read and write requests and services them in a sequential fashion. Consequently, if a read request occurs while a prefetch operation is in progress, the read will be delayed until the prefetch completes. This delay may be substantial. However, the miss rate graphs do not reflect such delays because read requests that become blocked by a prefetch requests only count as one cache miss per page, regardless of how long the request is blocked.

Read complete times do not suffer from this inaccuracy. Therefore, when measuring performance in terms of read complete times instead of cache misses, fewer prefetches often produce better

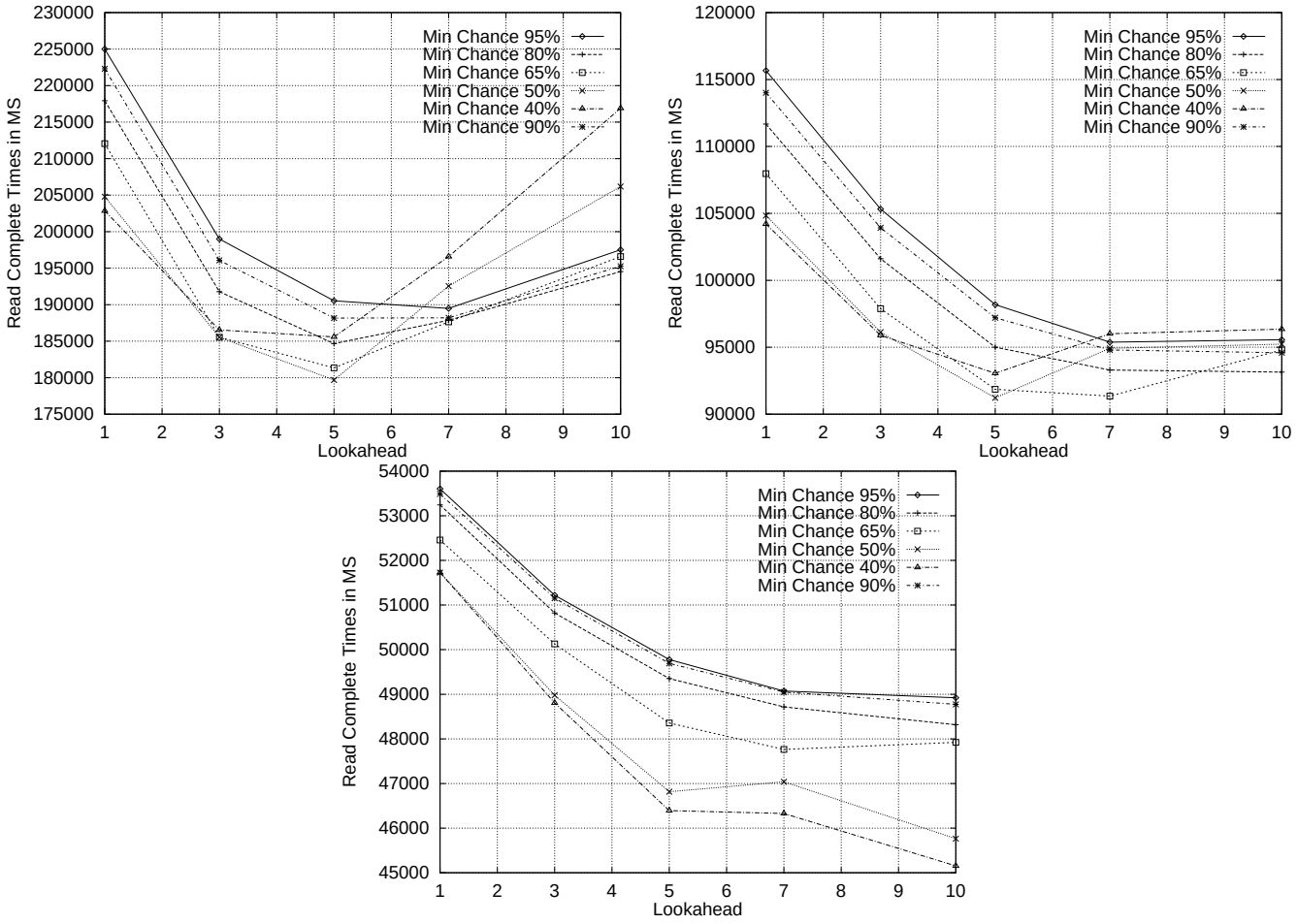

Figure 6: Read Complete Times as a function of the lookahead for various minimum chance values and cache sizes of of (left to right): (a) 400K, (b) 1600K, (c) 6400K.

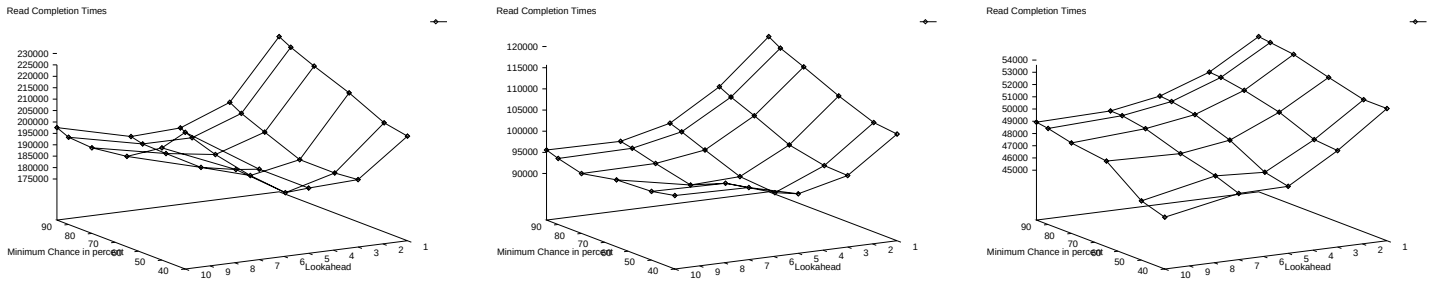

Figure 7: 3D contour of Read Complete Times show above in figure 6. The cache size used (left to right): (a) 400K, (b) 1600K, (c) 6400K.

performance. Few prefetches occur at low lookahead and high minimum chance settings. The read results shown in figures 6 and 7 verify this.

Figures 6a and 7a indicate that small caches should not set the minimum chance value to 40% as the miss rate performance measure suggests. Such a minimum chance does not produce optimum performance and is very sensitive to changes in lookahead.

Instead, the optimum performance occurs at a minimum chance of 50%. Although a lookahead of 5 and a minimum chance of 50% produce the optimal value, small changes to the lookahead value result in substantial performance penalties. Moreover, the read complete times suggest that larger variations in lookahead are acceptable as long as the minimum chance remains relatively high (65-95%). Although such settings do not produce the optimal performance, they offer near optimal performance and are not susceptible to small perturbations. Because small caches already have substantially more disk activity than large caches, small caches cannot afford to further load the disk with prefetches that only have a small chance of being correct.

Figures 6b and 7b indicate that for moderate cache sizes (1,600KB) higher minimum chance values (65%) result in substantially better performance than a low minimum chance value, again contradicting the miss rate recommendation (40%). Moreover, the performance penalty for selecting a relatively low lookahead (say 5) is not as harsh. In fact, a lookahead of 7 produces the optimal performance in this case.

For larger cache sizes (figures 6c and 7c) read complete times and hit rates both indicate that the large lookaheads and low minimum chance values produce the best performance.

In summary, minimum chance values should decrease as cache size increases. Lookahead should increase as cache size increases. And in general, the parameters that optimize performance for read complete times are more moderate that those that optimize the miss rate measure.

5 Conclusion

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, fully utilize the available bandwidth via batched prefetch requests, and improve cache utilization. As wide area distributed file systems, CD-ROMs and other high latency/high bandwidth systems become prevalent, prefetching will become an increasingly important mechanism toward high-performance I/O.

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. Simulations lead us to believe that a prefetching cache can perform as well as an LRU cache of twice the size. We believe that the performance gain of our working system will be substantial.

References

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

[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.

[GA] Jim Griffioen and Randy Appleton. Aging and Space Conservation in Automatic Prefetching System. Technical report. (in preparation).

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

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

[MJLF84] Michael K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *Acm Trans. on Computer Systems*, 2(3):181–197, August 1984.

[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.

[MvRT+90] 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.

[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.

[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] 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.

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

[SGK+85] 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.

[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.

[WEB93] D. Willick, D. Eager, and R. Bunt. Disk Cache Replacement Policies for Network Fileservers. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 2–11, May 1993.