Adaptive Filetype Aware Prefetching

Tushar Khot Varghese Mathew Priyananda Shenoy
Department of Computer Sciences
University of Wisconsin, Madison, WI
{tushar,vmathew,shenoy}@cs.wisc.edu

Abstract

Prefetching and caching of files are the main means by which operating systems strive to strike a match between the high access times of disks and the fast speeds of today's processors. However, current implementations of prefetching algorithms are rather rigid and do not exploit many crucial pieces of information available to them. These include the file-type information, read rates and the load on the file-cache. Through this paper, we describe our efforts and results at developing and deploying a file-type aware enhanced prefetching algorithm for the linux kernel.

1 Introduction

In recent years, processing power of computers have seen exponential increases. On the other hand, when disk capacities have also increased substantially, the data access rates for the disks have improved rather slowly and steadily. This has resulted in a huge chasm between processing speeds and disk speeds, making disk accesses one of the slowest steps in execution of programs.

Now, like with most problems in the Operating system domain, this disparity in speed too has been analyzed and anatomized by many researchers. The primary cause of the slowness of disk accesses is the seek latency involved in moving the disk head over to the data which needs to be read. Thus to amortize this latency, a large number of blocks are prefetched in a batch in advance from the disk. Likewise, in-memory caching of disk pages is used for reducing the actual disk reads in cases where the data had been read already.

However, as we realized from our study of the related work in the field of prefetching and caching, most algorithms do not use the file-type information for fine tuning the prefetching policy. Like wise, the impact of pressure on the file cache is also another hint that can be exploited by the operating system while formulating the prefetch policy.

Thus through this project, we modified the linux prefetching algorithm to take into account the type of the file being read, the rate at which it is being read and the cache-pressure. The rate at which the file is read suggests the extent of prefetching whereas the cache-pressure bounds the same. These two factors can therefore be converged to give an ideal prefetch window at any point in time. Likewise, the file-type information can be used to account for peculiarities in access patterns and provide hints for the prefetching algorithm.

2 Related Work

With rapidly increasing processing speeds that have not quite been matched by rising disk speeds, prefetching and caching gain dominance as mechanisms to mitigate the lag and allow higher utilization of the processor even with I/O bound tasks.

Cao et. al [1] compares two prefetching strategies, aggressive and conservative, and conclude that an aggressive strategy comes closer to the offline-computed optimal strategy than the conservative one. However, we believe that this is overly generalized. In our perspective, the comparison of aggressive vs. conservative must be done at a file-type granularity. For e.g.. a multimedia audio file will not need extremely aggressive access since the read need not take place faster than the rate at which the file is being played. Likewise, an archive file which most likely is accessed in a piecewise sequential manner can again be prefetched somewhat conservatively.

Shih et al. [6] propose to use cache hit histories to determine the sequentiality of access and perform prefetching accordingly. However, the inferences thus drawn are not persistent. And storing them on a per-file basis may not be a scalable approach. However, file-types offer a convenient axis against which inferences about access patterns can be aggregated and stored.

With concurrent I/O by different applications brought into the equation, Li et al. [2] propose a strategy where prefetch depth is determined by the amount of data that can be read in the average time gap between an I/O switch. However, this approach is not without problems. For e.g.. with a multimedia file and a Pdf file contending for prefetch from disk, an equal share is not the best solution.. The multimedia file needs to get priority as it is the one that would suffer more because of delays. This is again a situation where prioritizing based on file-type can possibly yield a solution.

Butt et al. [7] stress on the importance of studying prefetching algorithms and cache management algorithms in conjunction. These two are closely related and therefore an optimization in one without regard to its impact on the other can actually result in deterioration of performance. We therefore look at cache management strategies that can complement the prefetching enhancements at the file-type granularity. Eviction policy is an ideal candidate for improvisation with knowledge of file-type.

A fairly obvious idea is to allow applications to inform the file system about it's prefetching policy. Griffioen [3] uses application specified hints to better tune the prefetching policy. Patterson et al. [4] extends this idea to cover both prefetching and caching policies. To relieve the programmer of the responsibility of giving hints correctly, Chang et al. [5] instrument the application binary to analyze access patterns and generate hints automatically. The problem with hints is that most of the time applications do not themselves know future access patterns, and static analysis of application may not yield the right patterns.

Another approach which has been tried is to build application specific optimizations for complex access patterns. Mitra et al. [8] developed specialized application level prefetch prediction for multimedia programs. They achieve this by generating a prefetch thread which prefetches entirely at the application level. But this would involve major changes to applications. The same effect can be obtained in exploiting similarity in access patterns across applications dealing with the same file-type.

Kim et al. [10] streamline cache management and prefetch policies for multimedia servers. They use system load to determine prefetch depth and cache policies. Their goal is not necessarily to optimize overall performance, but to guarantee quality of service to each client. System load alone is not a good basis for prefetch decisions, especially where files of diverse content types are involved.

File prefetching can be done at both inter-file and intra-file level. Griffioen [12] studied the performance of automatic prefetching which remembers file access patterns and aggressively loads files related to this file. Preload [11] operates as a daemon which looks at file accesses, and uses a predictive probabilistic model to preload files likely to be accessed next. While these techniques work fine for inter-file dependencies, they cannot directly be applied for intra-file block accesses.

A large body of work has been concentrated on analyzing the access patterns of a file and predicting the prefetch depth. Fido [13] uses a dynamic learning algorithm to determine which class the access belongs to, in the specific context of databases. Such approaches suffer from having only local histories, which are forgotten after the session ends. On the other hand, maintaining per-file access histories across sessions involves too much of an overhead. File-type based policies form a nice compromise between local and persistent histories.

Phunchongharn et al. [9] used file type information to optimize various strategies such as disk allocation, redundancy, and caching strategies. But their work doesn't cover prefetching strategies.

3 Motivation

We believe that the access pattern for a certain type of file generally remains the same across different files of that type. This observation can be put to use in determining ideal prefetch depths for different frequently encountered file-types. We expect prefetching driven by such a file-type determined depth to outperform contemporary implementations of prefetching which are rather static. Similarly with caching, the access pattern as suggested by the file-type can hint at the likelihood of the block being accessed again. A cache replacement policy driven by this can make sure that we do not cache blocks that are unlikely to be needed.

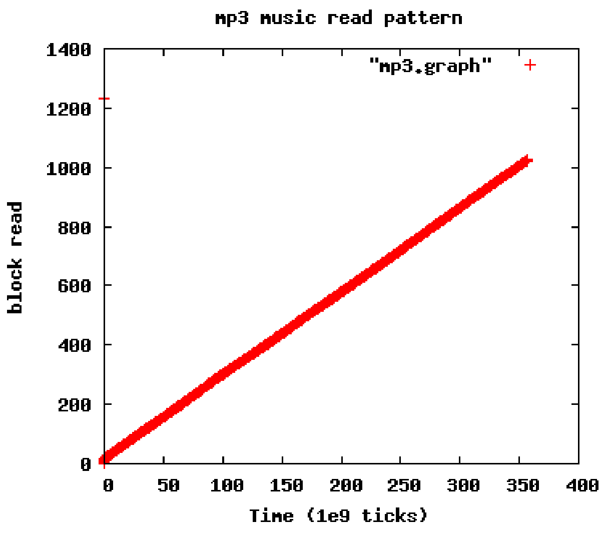

Figure 1: **Access pattern for an MP3 music file.**

3.1 Access Patterns

Figure 1 shows the access pattern for MP3 files. This is a typical example of purely sequential access. The metadata for the MP3 file is stored at the end of file, which is why we see that.

The access pattern for AVI is similar.

Figure 2 shows the access pattern for a PDF file using Adobe reader. The access pattern is overall se-

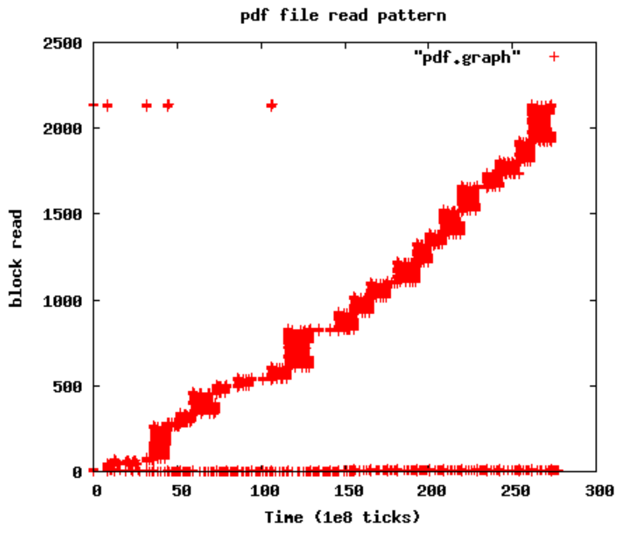

Figure 2: **Accesses for pdf using Acrobat reader.**

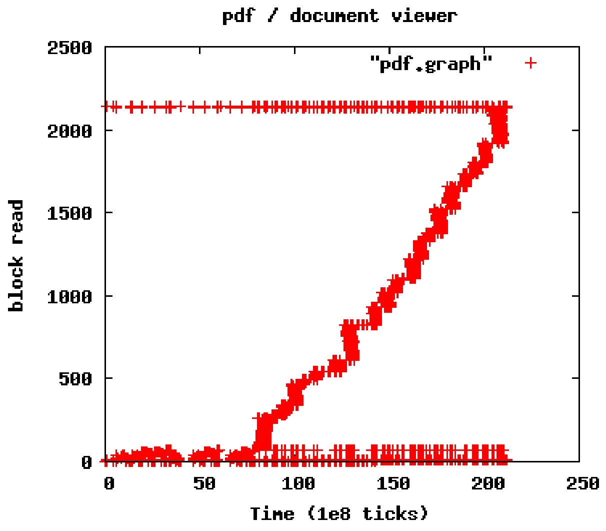

Figure 4: **Access for pdf using Document Viewer.**

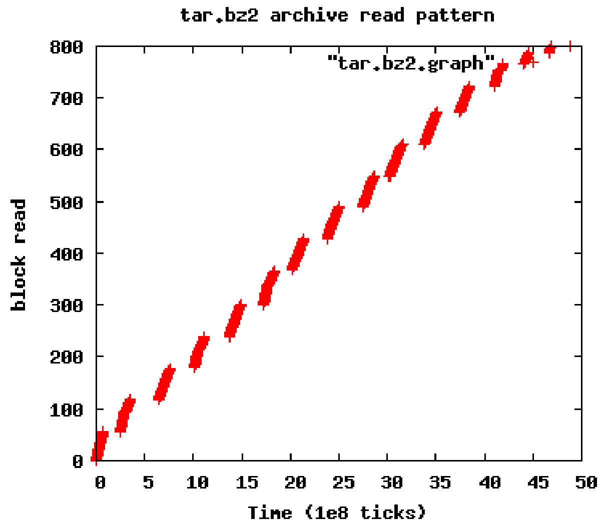

Figure 3: **Access pattern for a tar.bz2 archive.**

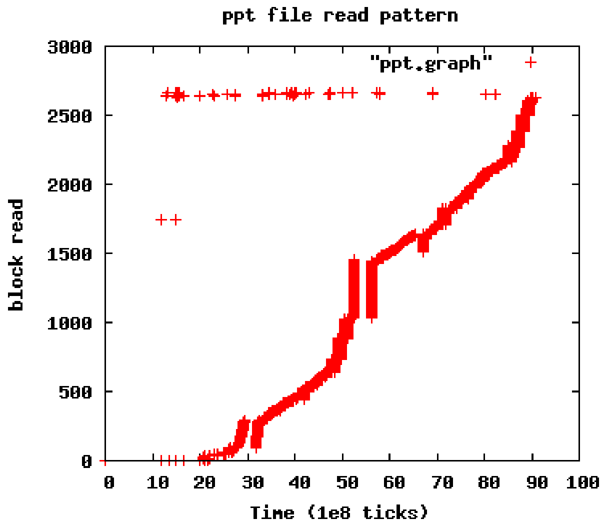

Figure 5: **Access pattern for a powerpoint presentation.**

quential but cut into chunks which are read repeatedly. This evidently gives us an opportunity to adjust prefetch window size close to the chunk size. There are also interleaved accesses to the end of the file, possibly due to accesses to font data, which would confuse the existing prefetching algorithm.

Figure 4 shows the access pattern for the same PDF file using the Gnome Document Viewer. Apart from the prolonged initial processing and an increased number of reads to the head and tail of the file throughout the duration, the access patterns are similar in their chunked nature. This builds upon our case for file-type based patterns in file accesses which lend themselves suitable to expoitation by a prefetching algorithm.

Figure 3 shows the access pattern for a compressed archive(tar.bz2). The access pattern for uncompressing the whole archive was purely sequential. This represents the case where the existing prefetching algorithm works well.

Figure 5 shows the access pattern for a Powerpoint presentation(PPT). The access patterns are similar to PDF, but the chunk sizes show a greater variance. This represents a case where rapid dynamic adaptation of prefetch window is required.

3.2 Existing Implementation

We analyzed the existing implementation of the prefetching algorithm in the linux kernel. The algorithm, as we mentioned, does not account for many of the possible factors, and therefore is quite simplistic.

Any file read starts off with a prefetch window size of 2 pages. If the kernel finds that the file is being read sequentially, this window is doubled every time until either the reads deviate from the sequential pattern or the hard limit of 32 on the prefetch window size is reached. At any point, if the sequential read pattern stops and moves to a random pattern, the algorithm falls back to its initial state.

Quite evidently, one can see the insufficiencies with this approach. First off, particularly with files like pdf, and ppt, where the read pattern is piecewise sequential, every jump causes the prefetching window to go back to the original value. Likewise, the hard static limit of 32 on the prefetch window size can be rather limiting on file types like video files which could benefit in terms of reduced jitter by having a large prefetch window. Also, the prefetch algorithm is currently implemented independent of free space availability in the file-cache, thereby keeping open the possibility that pages that were prefetched get evicted before they are actually used.

4 Methodology

This section describes the methodology we followed in devising our enhancements to the prefetching algorithms. There are three parameters we identify about each file that is being read. The first is the rate at which the file is being read. This gives a measure of the value of prefetching a particular block into memory. The second is the effect of cache-pressure. Through this we estimate the duration for which a block lives in the cache. And finally we use file-type specific aspects of the read pattern to optimize the prefetching algorithm, to the specific file that is being accessed.

4.1 Effect of read-rate

The rate at which a file is being read gives an idea of the rate at which the application processes the file data, and therefore the estimated time at which the next disk request is likely to happen. Of course, there are variations in read rate across different sections of the same file. To accommodate these variations we determine long and short term averages of the read rates and compute our predicted rate from these.

4.2 Effect of cache-pressure

With many reads happening simultaneously, it may happen that some of the data that we prefetched may get evicted from the cache before it is actually used. This kind of a wasteful prefetching is a result of thrashing in the file-cache. Under such circumstances, it would be advisable for the prefetching algorithm to backoff and prefetch less. Towards this direction, we compute long and short term average lifetime of a page in the file-cache. We then use that to predict the expected cache-lifetime of the pages we prefetch.

4.3 Prefetch window calculation

Once we have the estimated read rate of the file and the expected cache lifetime of each prefetched page, we can compute the estimate of the ideal prefetch window based on these factors. We then make use of this estimate to increase / decrease the prefetch window for the next readahead.

4.4 Filetype specific policies

In addition to the above two concerns, we note that different file-types have peculiarities of their own in their access patterns. While simple examples would be files like mp3 and avi which are read sequentially at a relatively constant rate; files like pdf etc tend to have a piecewise sequential read pattern where a chunk of the file is read in and then read over and over a few times before moving to the next chunk of the file. Also, there are peculiarities like for eg: the first and last few pages of a pdf file are repeatedly read throughout the entire application, which may confuse the existing prefetch algorithm. We decided that such peculiarities of access patterns can be used to provide hints to the prefetching algorithm, but are best handled on a case by case basis.

5 Implementation

We implemented the enhanced prefetching algorithm on the linux kernel 2.6.27.5. We employed RDTSC time-stamps within the prefetching and file-caching code to determine the read rate of files and the cache-lifetime of pages. Likewise we modified the open system-call handler to keep track of the file-type of the file for use in the prefetching code. Each of these aspects of implementation are detailed below.

Since our code-changes are above the VFS layer, the behavior is expected to be independent of the filesystem format.

5.1 Measuring read-rates

To determine the read rates, we instrumented the generic file read handler to insert time-stamps and keep track of the previous block read and the time at which it was read. With this information, instantaneous read rates are computed for each block and averaged over a short as well as long interval to compute the short-term and long-term read rates. The larger of these two values is then used as the current read-rate for the file.

Figure 6 depicts the read rate computation. The instantaneous, short and long term time lag between reads as obtained from playing an avi video is graphed in the figure.

5.2 Measuring cache-pressure

Keeping track of cache pressure was the aspect of our implementation which we faced most difficulty with.

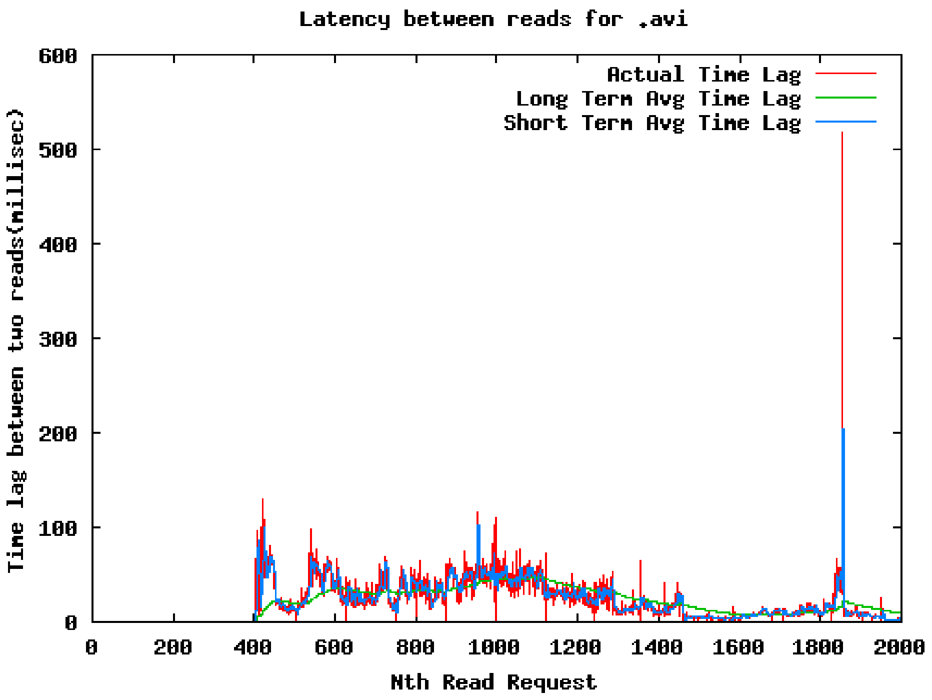

Figure 6: **The instantaneous, short and long term read rates of an avi file.**

This primarily was due to the fact that the linux kernel cache replacement policy is an approximate LRU and not a proper LRU. Thus the page evicted from the cache is only probabilistically guaranteed to be the oldest, making it possible for newer pages to be evicted from the cache with non-zero probability.

However, we still were able to obtain an approximation for the lifetime of a page in the cache by using time-stamps to determine the duration between the last access and eviction times of pages being evicted from the cache. We average this per-page value on a short term and a long term basis to compute two cache-lifetime estimates. We then use the lower of the two to adapt faster to rise in cache pressure while ramp up sluggishly when cache pressure reduces.

For the limiting cases where the system has so far not seen any page eviction or the system was left idle for long or the last page eviction happened considerably long ago, we default to a pre-determined static value for the cache lifetime.

5.3 Updating read-ahead window

Finally we use the read rate and cache lifetime computed above to compute the ideal prefetch window for the next prefetch. We start reading with an initial prefetch window size, and double on each iteration until the prefetch window reaches *EXP_GROWTH_LIMIT*. Thereafter, we increase the window size linearly by *GROWTH_FACTOR*. This increasing phase is done subject to the condition that the current prefetch window lies below the ideal prefetch window size computed for this iteration. We have also placed an absolute upper bound of *WINDOW_LIMIT* on the size of the prefetch window as a worst case throttle.

If the current prefetch window size however is found to lie above the ideal for any iteration, we immediately pull the current window size down to the ideal value to cut back on cache pressure.

In the above process, we make use of the type of the file to adjust the algorithm to discount anomalous behavior expected in the file type (like last few pages read repeatedly in pdf files, as mentioned above).

6 Evaluation

We evaluated the modified prefetching algorithm on an ext3 partition on an Intel Core2Duo 2 GHz. processor. We used a 160GB SATA disk @ 5400 RPM for all the evaluation runs.

6.1 Workload

Measuring and evaluating our file-type based corrections for sequentiality in the prefetching algorithm proved to be difficult. Although we perceived the the delay in loading pdf files to have reduced, we do not have a reliably reproducable benchmark to cite as proof of our improvements.

For testing the impact of our algorithm under various read rates and cache pressure, we adopted the following strategy. We created processes that read a file from the disk at various read rates. With these, we measured the block read latency incurred in the kernel for each read request furnished by the application. In order to simulate cache pressure, we constrained the linux installation to a limited amount of RAM when performing our experiments and made sure that the file sizes we used exceeded the amount of free memory available in the system. The system memory used for runs with cache pressure was 256 MB and the same for runs without cache pressure was 2048 MB.

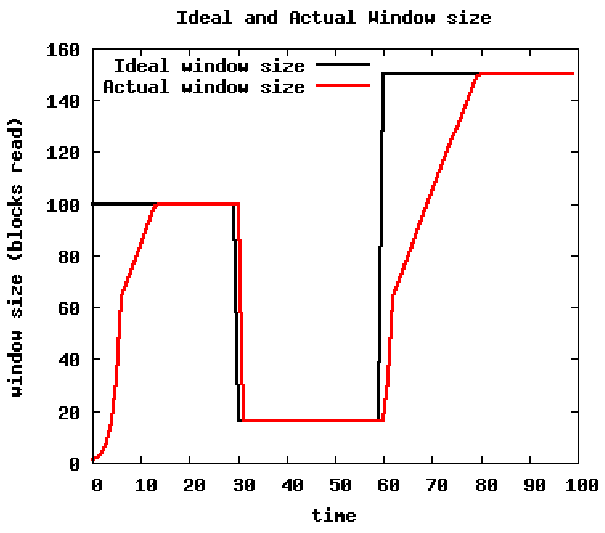

Figure 7: **Plot of how our prefetch window size follows the ideal.**

6.2 Results

Figure 7 plots the ideal prefetch window size computed and how the actual window size used by our algorithm closely follows the same. When the window size is below the EXP_GROWTH_LIMIT (64 here) the increase is exponential. Thereafter the window size increases linearly until it matches the ideal window size. However, in the event that the computed ideal window size falls (due to extraneous factors) we cut back on the actual window size to the same lower ideal window size to quickly respond to cache-pressure.

Figure 8 shows the average time taken to service

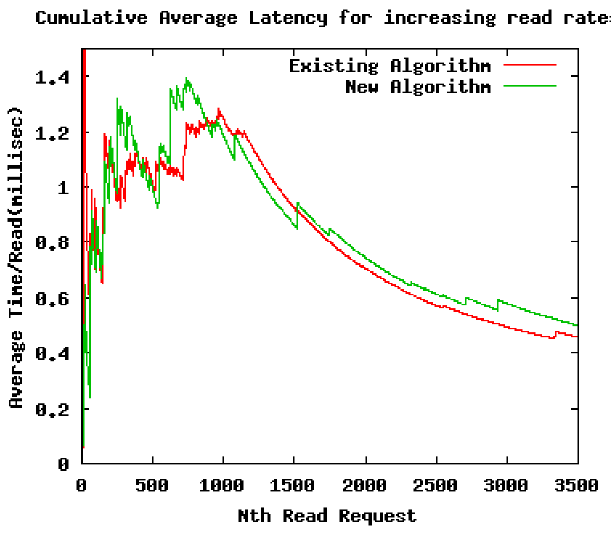

Figure 8: **Cumulative average latency for increasing read rates, without cache pressure.**

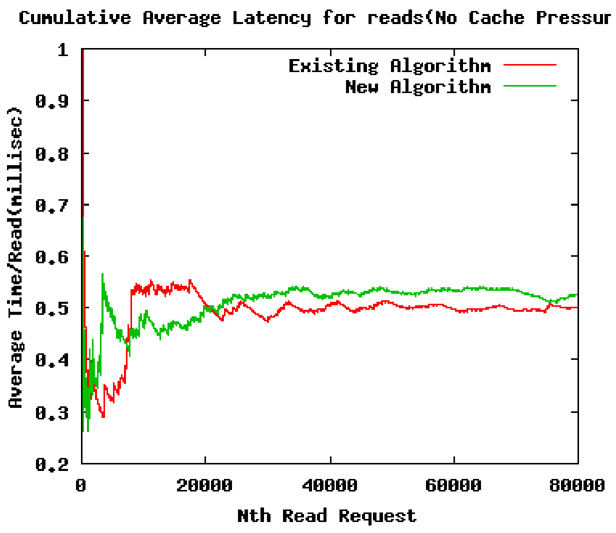

Figure 9: **Cumulative average latency for constant rate reads in the absense of cache pressure.**

requests with increasing read rates on a single file. A program reads 4K bytes of data at a time from a file sequentially with the time lag between two requests dropping form 10000 to 20 microseconds over a period of 3500 reads. Initially both the algorithm have similar approaches and try to ramp up their read ahead window size. The existing algorithm stops after 32 pages, whereas we continue to increase our window size to WINDOW_LIMIT (512 here). But because of the way the LRU cache eviction works in linux, it kicks out some pages from our prefetched window causing intermediate spikes and hence we see an overall performance deterioation. We believe that fixing the LRU eviction algorithm will help us attain a more consistently better performance over the existing prefetching algorithm.

Figure 9 is similar to figure 8 but with the read happening at a constant rate. Since there are not competing reads to the disk, the disk head rarely moves

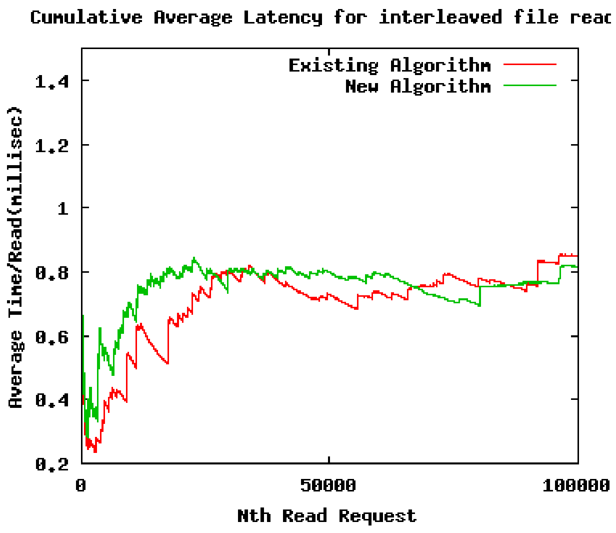

Figure 10: **Comparison of read latencies for interleaved reads in the absensce of cache pressure.**

and hence the existing algorithm performs slightly better than our approach.

Figure 10 shows the interleaved read performance

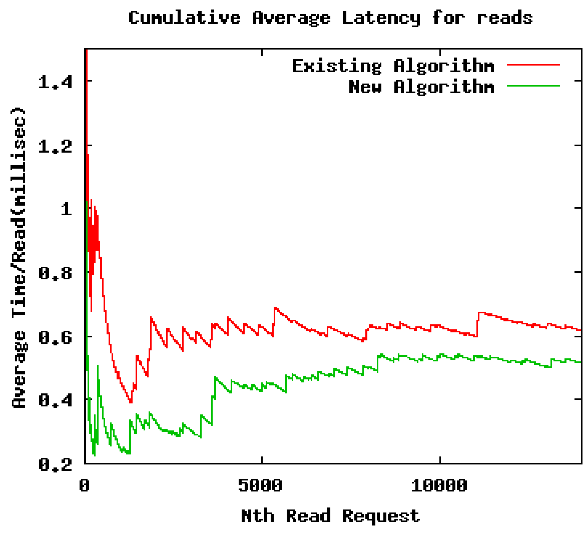

Figure 11: **Comparison of read latencies with existing algorithm and our modified algorithm under cache pressure.**

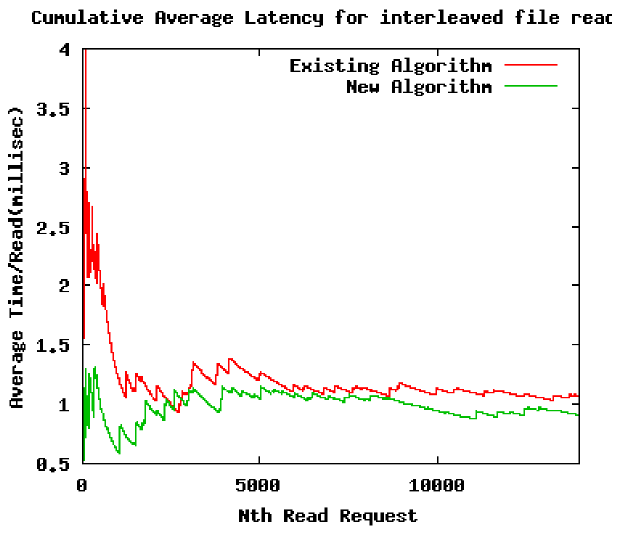

Figure 12: **Comparison of interleaved read latencies with existing algorithm and our modified algorithm under cache pressure.**

in the absence of cache pressure where our approach performs better on average because we prefetch more pages in one go and have overall reduced seek overheads. We got similar patterns on multiple runs and presume the spikes in the graph are caused by cache kicking out prefetched pages (which are always in the inactive list).

Figure 11 depicts the read latencies when a single file is being read at a constant rate with cache presssure arising out of constrained memory limits. In this case we visibly outperform the existing linux prefetching algorithm by a factor of 16%.

Figure 12 depicts the read latencies when multiple files are being read in an interleaved fashion under a constrained cache. In this situation too, our algorithm outperforms the existing linux algorithm as we expected.

7 Conclusion

In conclusion, we find that adapting the prefetching algorithm to accommodate parameters like file-type, read rates and cache pressure improves the performance of file access under normal load patterns.

However, on a lightly loaded system, this strategy may turn out to be less beneficial because of the approximate implementation of the LRU cache eviction policy in linux. Yet, we believe that the tradeoff involved is reasonable since it essentially sums down to a slight increase in read latency under lighter loads as a price for an improvement in read latency under heavier system loads.

8 Future Work

As we mention earlier, one of the roadblocks we faced with our implementation is the approximate nature of the linux LRU cache replacement algorithm. We're working on figuring out ways to circumvent this and to introduce priorities for pages in the cache. With such a facility for eg., pages of an mp3 file which have already been read once can be evicted from the cache in preference to pages from a pdf file.

We also plan to analyze read patterns for more file types and come up with an initial static read ahead window and ramp up size for them. Also currently our long term and short term averages are measured over somewhat arbitrary windows sizes. These may benefit from an empirical determination of the window over which the averages are best computed. Likewise the constants EXP_GROWTH_LIMIT, GROWTH_FACTOR, and WINDOW_LIMIT could also benefit through emperical determination and refinement.

References

[1] Cao P, Edward W. Felten, Anna R and Y Kai Li. A study of integrated prefetching and caching strategies. In *Proceedings of the ACM SIGMETRICS*, 1995.

[2] Chuanpeng Li, Kai Shen, Athanasios E. Papathanasiou Competitive prefetching for concurrent sequential I/O. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[3] James Griffioen. Reducing file system latency using a predictive approach. In *USENIX 94*, 1994.

[4] R. Hugo Patterson and Garth A. Gibson and Eka Ginting and Daniel Stodolsky and Jim Zelenka. Informed Prefetching and Caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[5] Fay W. Chang and Garth A. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[6] Shih, F.W.; Lee, T.-C.; Ong, S. A file-based adaptive prefetch caching design. In *Proceedings 1990 IEEE International Conference on Volume*, 1990.

[7] Ali R. Butt, Chris Gniady, Y. Charlie Hu The performance impact of kernel prefetching on buffer cache replacement algorithms. In *ACM SIGMETRICS Performance Evaluation Review Volume 33 , Issue 1*, 2005.

[8] Tulika Mitra, Chuan-Kai Yang, Tzi-cker Chiueh . APPLICATION-SPECIFIC FILE PREFETCHING FOR MULTIMEDIA PROGRAMS.

[9] Phunchongharn, Phond Pornnapa, Supart Achalakul, Tiranee. File Type Classification for Adaptive Object File System. In *TENCON 2006 - IEEE Region 10 Conference*, 2006.

[10] K.-H. Kim, S.-H. Lim, and K.-H. Park. Adaptive Read-Ahead and Buffer Management for Multimedia Systems. In *Internet and Multimedia Systems and Applications*, 2004.

[11] Behdad Esfahbod. Preload An Adaptive Prefetching Daemon. *Masters Thesis, Graduate Department of Computer Science. University of Toronto*, 2006.

[12] James Griffioen. Performance measurements of automatic prefetching. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, 1995.

[13] Mark Palmer, Stanley B. Zdonik. Fido: A Cache That Learns To Fetch. In *Technical Report: CS-91-15, Brown University*, 1991.