

Adaptive Filetype Based 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 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 the 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 cached 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 strove to and created a modified prefetching algorithm that takes into account the type of the file being read, and the pressure of file-data already in the file-cache.

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 re-

sult 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. [14] 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.

3.1 Access Patterns

Fig X 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.

Fig X shows the access pattern for a PDF file. The access pattern is overall sequential 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.

Fig X shows the access pattern for a compressed archive(BZ2). The access pattern for uncompressing the whole archive was purely sequential. This represents the case where the existing prefetching algorithm works well.

Fig X 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 parameters, 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.

1. starts at 2, doubles window size every time. stops at 32 blocks. - async.size etc etc 2. problems: random jump stops prefetch behavior. 32 is a static limit. doesn't adapt to cache pressure.

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 date across different sections of the same file. To accommodate these variations we determine long an short term averages of the read rates and compute our predicted rate from these. - faster you read, more we prefetch. Eg: initial/ffd on avis

4.2 Effect of cache-pressure

With many reads happening simultaneously, it may happen that somme 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 a circumstance, 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, and use that to predict the expected lifetime in cache of the pages we prefetch. - more cache pressure, more chance of useless prefetch= \downarrow prefetch less

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 as the product of these two parameters. - ?? implement a tcp back off style algorithms ?? - combine both these factors.

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 last pages of the mp3 file which contain the id3 tags are read in somewhat early in read, which may confuse the existing prefetch algorithm. We decided that such peculiarities of access patterns are best handled on a case by case basis. specifics like last few pages in pdf etc

5 Implementation

5.1 Measuring read-rates

- measure interval between reads of consecutive blocks(use policy to ignore other reads) - maintain long term and short term average - min of averages - aggressive on read rate

5.2 Measuring cache-pressure

- measure average time a block remains in cache - measured from time of last access - maintain long term and short term average - min of averages - conservative

5.3 Updating read-ahead window

- ramp up slowly, ramp down fast - doubles until it reaches ceiling

6 Evaluation

6.1 Workload

simulated environment multiple threads/applications with and without cache pressure

6.2 Results

7 Conclusion

8 Future Work

-persistent -cache policies

References

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

Adaptive Object File System. In *TENCON 2006 - IEEE Region 10 Conference*, 2006.

- [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.
- [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.
- [14] Phunchongharn, Phond Pornnapa, Supart Achalakul, Tiranee. File Type Classification for