Linear Aggressive Prefetching: A Way to Increase the Performance of Cooperative Caches. [*]

T. Cortes J. Labarta
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya - Barcelona
{toni,jesus}@ac.upc.es

Abstract

Cooperative caches offer huge amounts of caching memory that is not always used as well as it could be. We might find blocks in the cache that have not been requested for many hours. These blocks will hardly improve the performance of the system while the buffers they occupy could be better used to speed-up the I/O operations. In this paper, we present a family of simple prefetching algorithms that increase the file-system performance significantly. Furthermore, we also present a way to make any simple prefetching algorithm into an aggressive one that controls its aggressiveness not to flood the cache unnecessarily. All these algorithms and mechanisms have proven to increase the performance of two state-of-the-art parallel/distributed file systems: PAFS and xFS.

1. Introduction

Cooperative caches are becoming very popular in parallel/distributed environments [3, 5, 7]. A cooperative cache is a big global cache build from the union of the local caches located in each node in the system. As all local caches are globally managed, a more effective cache can be achieved, thus the performance of the file system is greatly improved.

This kind of caches offer huge amounts of caching memory that is not always used as well as it could. We might find blocks in the cache that have not been requested for many hours [4]. These blocks will hardly improve the performance of the system and the buffers they occupy could be better used to improve the performance of the applications. Furthermore, the size of the caches will keep growing as memory sizes increase quite rapidly. For this reason, the above situation will become even more frequent.

In this paper, we present the idea of linear aggressive prefetching. As the size of the caches is quite large, we can implement quite aggressive-prefetching algorithms as the prefetched blocks will replace very old ones. If the blocks are correctly predicted, the system will observe a performance improvement. On the other hand, if the blocks are miss-predicted, they will mostly replace very old ones that nobody expects to find in the cache. These miss-predictions should not decrease the system performance. In conclusion, the larger the cache is, the more aggressive the prefetching can be.

All the performance results presented in this paper have been obtained through simulation so that a wide range of architectures can be tested. The proposed algorithms have been tested on two state-of-the-art parallel/distributed file systems that implement a cooperative cache: PAFS [4] and xFS [1].

1.1. Related Work

First, we would like to mention the prefetching algorithm proposed by Vitter and Krishnan [6]. In their work, they propose a prefetching algorithm based on Markov models. This algorithm has been the start point of the IS_PPM family of prefetching algorithms presented in this paper. In their paper, they also proposed a somewhat aggressive prefetching where the

[*] This work has been supported by the Spanish Ministry of Education (CICYT) under the TIC-95-0429 contract.

most probable blocks where prefetched, although only one of them might be requested. In our case, we trust the predictions and prefetch many blocks assuming that the ones prefetched have been correctly predicted. This means that all prefetched blocks may be requested by the application.

One of the first works on parallel prefetching was developed by Kotz [9]. He presented a study of prefetching mechanisms and algorithms for parallel file systems. In his study, he used synthetic applications and the algorithms were designed for a single application, not for all the applications in the machine as we do in this paper. Furthermore, no aggressive mechanisms were proposed in his work.

In parallel prefetching, there has also been some theoretical work that assumes a knowledge of the access stream. The objective of that work was to make a scheduling of when each block has to be prefetched to achieve the best performance [8]. Our work could be a complement to this one as we propose a way to decide which blocks have to be prefetched and thus no need of knowing the whole access stream is needed.

2. Prefetching Algorithms

In this section, we will describe the two basic algorithms used in this paper. The first one, OBA, is a well-know and widely-used prefetching algorithm while the second one, IS_PPM, is a contribution of this paper. Anyway, we should remember that the main objective of this paper is not to propose the best prefetching algorithm but to build aggressive ones from others much simpler.

2.1. One Block Ahead (OBA)

The first algorithm is the most widely used one in sequential and parallel/distributed file systems. The idea of this simple algorithm is to take advantage of spatial locality. This means that if a given file block is requested, it is very probable for the next sequential block to be requested too.

Using this heuristic, whenever a block *i* is read or written, block *i+1* is also requested for prefetching [13]. This algorithm is a very conservative one as only one block is prefetched after each request.

2.2. Interval and Size Predictor (IS_PPM)

The second algorithm is a more sophisticated one that tries to find the access pattern used by the application to predict the blocks that will be requested in the future. This algorithm is an evolution from the prediction-by-partial-match (PPM), based on Markov models, proposed by Vitter and Krishnan [6].

The first thing we need to do is to describe the way we model the access pattern. In this work, we model a sequence of user requests by a list of pairs formed by a size and an offset interval. The size is the number of file blocks in a request. If a given operation only requests 2 bytes but from two different blocks, we assume that it was a two block request. The second element used to model a sequence of requests is the offset interval. This interval is the difference in blocks between the first block of a given request and the first block of the previous one. With this information, our algorithm will try to detect the access pattern of offset intervals and requested sizes.

As in PPM, we build a graph that will let us predict which blocks will be requested by the application. The main difference between IS_PPM and the original PPM is the information kept in the graph. In the original algorithm, the graph was used to maintain the information of which block had been requested after another given block. This is quite useful for paging (as proposed in that paper) but it is not the most adequate information for file prefetching. In our case, we want to keep the offset intervals between requests. We want to know that after jumping *x* blocks, the following request jumped another *y* blocks. As files are usually accessed using sequential or regular strided patterns [2, 10, 11], keeping the sequence of blocks is not enough as the system would have to wait until a block has been accessed once before being able to prefetch it. In our case, we predict intervals, thus blocks that have never been requested can also be predicted.

A second important difference compared to the original algorithm is the number of blocks to prefetch. In PPM, only one page was prefetched each time. On the other hand, in file-system accesses, we can predict both the position of the next request and the number of blocks that will be requested. This new information can also be predicted using the IS_PPM algorithm presented in this paper.

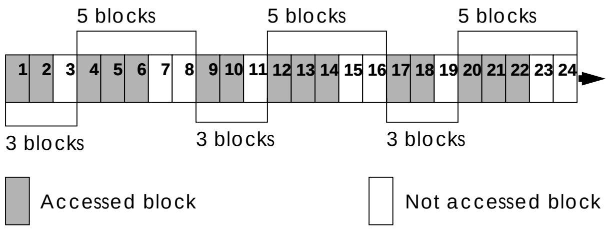

Figure 1. Access pattern used in the example.

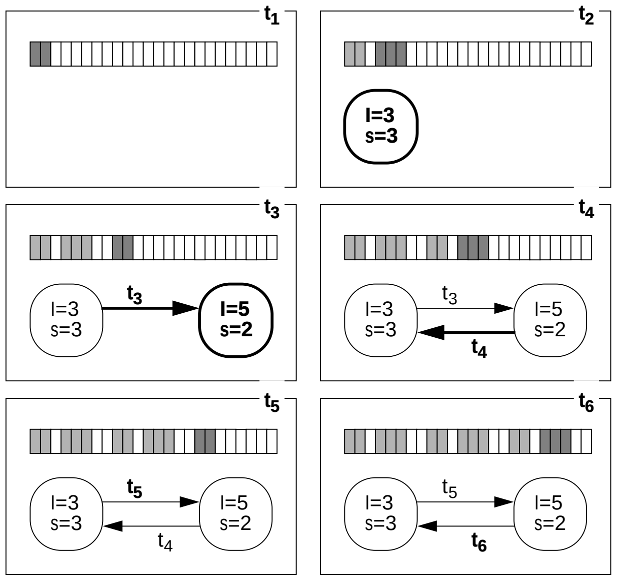

Figure 2. Steps done by the system to construct the graph for the access pattern shown in Figure 1.

Finally, the last difference is that in PPM, a probability was kept in the links between nodes. This was used when more than one block had previously been accessed after a given block. In this case, the one that has been accessed more times was the one prefetched. We have observed that this heuristic is not the most adequate one for file systems [4]. We have found, that following the path that has most recently been followed achieves a more accurate prediction.

The best way to understand how this mechanisms works is to see how the interval-and-size graph is created and the how it can be used to predict the blocks to be accessed. In Figure 1, we show the access pattern we will try to detect and predict as an example. In the example, all contiguous blocks are accessed in a single request. The way the graph is being built is also shown in Figure 2, where the graph after each request is represented.

Let us see how this graph can be used to predict the blocks needed by the application. Whenever the application makes a request, the system computes the offset interval between this request and the previous one. With this interval and the size of the previous request, the system searches for the node that matches this information. If the node is in the graph, the system just has to follow the most-recently updated link and it will have the interval and the size of the next request. Adding this interval to the current file pointer, we get the offset of the predicted request. The number of blocks to prefetch is the size kept in the node.

If we used this algorithm as it is, we would have a startup problem. When there are no nodes that describe the current situation, there is nothing the system can do to predict the next block. As this will happen quite often during the first accesses to a file, we propose using the OBA algorithm whenever not enough information is available in the graph. This should not happen very often as it is only intended for the first accesses to a file. Actually, the number of blocks prefetched using the OBA algorithm was, in average, less than 1% when the files were large (CHARISMA workload) and around 25% when the files were small (Sprite workload). These percentages have been extracted from all the experiments discussed later in this paper.

jth-order Markov Predictor

A *jth-order Markov predictor* uses the last *j* requests from the sequence to make its predictions for the next request. So far, we have only used the last pair (offset interval, size) to predict the next sequence. This means that the predictor we have presented is a 1st-order one. One such predictor may not be able to detect many regular access patterns where a request not only depends on which was the last offset interval, but it depends on what happened in the previous *j* offset intervals. For this reason, we have implemented the whole family of IS_PPM algorithms that have the order as a parameter. For instance, the 3rd order algorithm keeps track of the last-three pairs (offset interval, size). The only

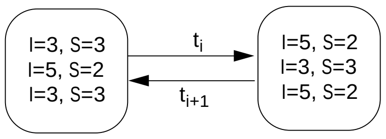

Figure 3. Graph obtained by the 3rd-order predictor for the access pattern in Figure 1.

difference with the one just described is that each node in the graph will have to keep 3 intervals and 3 sizes. Figure 3 shows the graph obtained by this predictor after the access pattern proposed in Figure 3.

To distinguish the order of the algorithms, we will add this order to the name of the predictor. Now the 1st-order predictor will be called **IS_PPM:1** and **IS_PPM:3** will be the name used for the 3rd-order one. These two instances of the family are the ones used the experiments presented later in this paper.

3. Linear Aggressive Prefetching

3.1. Aggressive Prefetching

In this section, we explain how to improve the performance of simple algorithms, such as the ones already presented, by making them aggressive. The idea is very simple, the prefetching mechanism will prefetch blocks continuously as long as it can predict data that is not in the cache yet. On the other hand, if the system detects that a miss-prediction has occurred, it understands that the prediction path is not correct. This triggers a mechanism that updates the prefetching data structures (if needed) and the aggressive prefetching restarts once again from the miss-predicted block. The number of prefetched blocks does not have any limitation except for the disk and network bandwidth.

Agr_OBA is built by constantly prefetching the next block while Agr_IS_PPM prefetched blocks using the prefetching graph without waiting for any user request.

3.2 Limiting the Aggressiveness: Linear Aggressive Prefetching

A too aggressive algorithm may end up flooding the cache. For this reason, propose the **linear** aggressive algorithms to set a limitation to the aggressiveness of this mechanism. The idea behind this limitation is to only allow the prefetching of a single block at a time for each file. We propose that a system with more than one disk should not use them to prefetch more than one block of the same file in parallel. Exploiting the parallelism offered by multiple disks can be achieved by prefetching blocks from different files. We may be prefetching as many blocks as disks but always from different files.

A more intuitive way of aggressive prefetching consists of using the potential parallelism offered by the disks to prefetch several blocks in parallel [8, 12]. This way of prefetching achieves a very good disk utilization when only one file is being accessed. In the same situation, our proposal leaves all disks but one unused. On the other hand, we have the feeling that parallel machines are used to run many applications in parallel minimizing the time that only one file is being used. When as many files as disks are being accessed, our proposal achieves a similar disk utilization. Furthermore, it is a way to limit the aggressiveness and the prefetching is more balanced as blocks from more than one file are being prefetched in parallel. We are not claiming that exploiting the parallelism of several disks ends up in a bad system performance, we are only proposing a different way to do an aggressive prefetching that limits the number of prefetched blocks by only prefetching one block per file at a time.

4. Implementation Details

We will test the behavior of the proposed mechanisms on two different file systems: PAFS [4] and xFS [1]. These file systems have been chosen because both implement a state-of-the-art cooperative cache but it is important to notice that they have important design differences[1].

In PAFS, the management of a given file is handled by a single server. This kind of centralized management allows a simple implementation of the idea of a linear aggressive prefetching. The server in charge of a file keeps all the prefetching information and can limit the number of simultaneously prefetched blocks

[1]In this paper, we are not willing to compare both file systems but to evaluate the viability of the linear aggressive-prefetching algorithms we are presenting. A comparison among both system may be found in earlier papers by our research group [4, 5].

	PM	NOW
Nodes	128	50
Buffer Size	8 KB	8 KB
Memory Bandwidth	500 MB/s	40 MB/s
Network Bandwidth	200 MB/s	19.4 MB/s
Number of Disks	16	8
Disk-Block Size	8 KB	8 KB
Disk Bandwidth	10 MB/s	10 MB/s
Disk Read Seek	10.5 ms	10.5 ms
Disk Write Seek	12.5 ms	12.5 ms

Table 1. Main simulation parameters.

to one. Furthermore, these servers have all the information about the access patterns because all requests have to go through them.

Implementing this kind of linear algorithms is not a simple task in xFS if we want to maintain its original philosophy. In this system, each node manages its own cache and only contacts the manager when the data is not in the cache. This means that only the local node has full information about the access patterns made by an application over a file. The manager only sees part of the operations, the ones that the local system cannot handle locally. For this reason, each node will perform its own prefetching decision. This local prefetching presents a problem in order to implement a linear aggressive prefetching. We can limit the number of blocks that are being prefetched by all the applications located in a single node, but there is no easy way to coordinate all nodes to avoid several prefetches of the same file being done in parallel. This problem could be solved using some external prefetcher, but this solution would modify the general philosophy of xFS. What we have done consists of implementing a linear aggressive prefetching per node. This means that each node will only prefetch one block per file at a time but several blocks of the same file may be prefetched in parallel from different nodes. This imperfect implementation will help us to evaluate whether this linearity is a good idea or not.

For our experiments, we used two sets of traces: CHARISMA [11] and Sprite [2]. The first one characterizes the behavior of a parallel machine (PM) while the second one characterizes the workload that may be found in a network of workstations (NOW). In both cases we have not measured the first part of the trace to warm the cache.

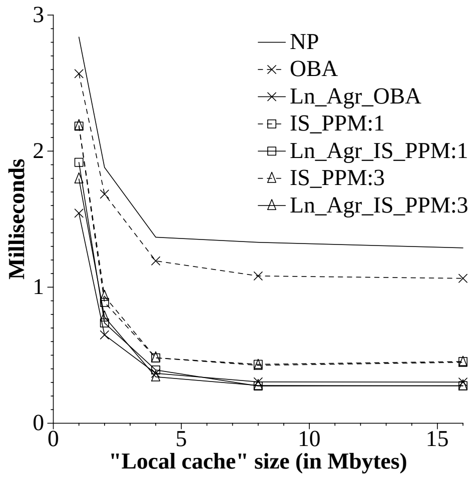

Figure 4. Average read time obtained running the CHARISMA workload under PAFS.

As we have two different workloads we also simulated two basic architectures. While the CHARISMA trace file has been tested on a parallel machine (PM), the Sprite trace file has been simulated on a NOW similar to the one used by Dahlin et al. [7]. The parameters used to simulate both architectures are shown in Table 1.

4.1. Performance Results

This section is divided into four subsections. The first two describe the performance obtained by the CHARISMA workload when run on both PAFS and xFS. The other two sections describe the performance of Sprite also under both file systems.

It is important to notice that only the performance of read operations is presented as write operations are not specially affected by an increase in the hit ratio [5, 4]. Anyway, both of them have been simulated and taken into account.

CHARISMA(PM) on PAFS

Figure 4 shows the average read time obtained by PAFS, and all prefetching algorithms, under the CHARISMA workload. These times have been measured with different cache sizes that vary from 1 to 16 Mbytes per node.

The first thing we observe is that the prefetching algorithms can be divided into three groups. The first one, which only contains OBA, is the one that achieves the worst performance. The reason behind this little improvement in performance is that the algorithm is too conservative. Only one block is prefetched per request thus only a few blocks are prefetched when many more could have been brought from disk.

The second group of algorithms is made by IS_PPM:1 and IS_PPM:3. They achieve very similar performance which is much better than the one obtained by OBA. These algorithms decrease the average read time in a very significant way for two reasons. The first one is that they are intelligent enough to predict most of the access patterns used by the applications. The second reason is that they behave in a quite aggressive manner. These algorithms not only predict the offset of the next request, but the size of that request. This means that in a workload that has large requests, the number of prefetched blocks will also be large. This is the case of the CHARISMA workload that has many large user requests. Another interesting result that can be extracted by examining this group of algorithms is that the order of the Markov predictor does not make a significant difference in the effectiveness of the predictor. This means that most access patterns can be predicted using a first order predictor.

Finally, all aggressive algorithms make the third group. These algorithms are the ones that achieve the best performance results. They nearly double the performance of the previous group and perform up to 4.6 times faster than with no prefetching when large caches are used. The controlled aggressiveness of these algorithms is the key to their excellent performance. Remember that these algorithms prefetch aggressively but only one block per file at a time.

In this third group of algorithms, the behavior depends on whether small or large caches are used. In the first case, Ln_Agr_OBA achieves better performance results than the other two algorithms. On the other hand, when caches are lager than 4 Mbytes, Ln_Agr_IS_PPM:1 and Ln_Agr_IS_PPM:3 obtain better read times. To explain this behavior we need first to describe a very frequently used access pattern. Many of the applications only access the first part of a file and leave the last portion of the file unaccessed. Furthermore, the access of the first part is done using a given access pattern that usually ends up accessing all blocks in this first part (not necessarily in a sequential way). As Ln_Agr_OBA reads all the blocks in a sequential way, it is not frequent for this algorithm to access blocks located in the last part of the file, which will never be requested. It accesses all the blocks in the first part of the file, which might be requested later on. On the other hand, Ln_Agr_IS_PPM can make large jumps and start prefetching from the not-accessed part. This prediction errors are quite important if the caches are small because useful blocks are discarded to keep useless ones. On the other hand, when caches are large enough to support these miss-predictions without discarding useful blocks, the Ln_Agr_IS_PPM algorithms have a better behavior because their knowledge of the access pattern is also better.

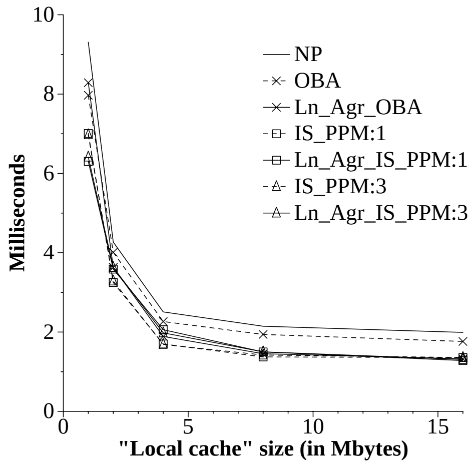

Figure 5. Average read time obtained running the CHARISMA workload under xFS.

CHARISMA(PM) on xFS

Let us now perform the same experiment but using xFS as the base file system (Figure 5).

The first thing that we observe is that there are only two groups of algorithms, one made by OBA and another that contains the rest of the algorithms. As in the previous subsection, OBA increases the performance, but the gain is very small. This behavior can be explained by the same reason we mentioned earlier: it is too conservative.

The second group, made of all the other algorithms, achieves a better performance but the gain is not as good as the ones observed under PAFS. The reason behind this lack of improvement is that these aggressive algorithms are not really controlled by only prefetching one block per file at a time. Remember that this implementation was not easy in a system such as xFS. For this reason, too many blocks are prefetched and the cache is flooded. We have observed that in the xFS executions, the number of prefetched blocks doubles the number observed in the PAFS executions. This means that many useful blocks are discarded by other blocks that might be needed after the ones that were discarded, or even they might never be used. This shows that controlling the aggressiveness of prefetching algorithms by making them linear is a good idea.

If we examine the behavior of this last group in greater detail, we can observe that the behavior with small caches is not the same as when large caches are used. In the first case, the aggressive algorithms end up prefetching too many blocks that flood the cache. This allows the less-aggressive algorithms such as IS_PPM:1 and IS_PPM:3 to achieve better read times. On the other hand, when the size of the cache is large enough, these extra prefetchings become less important and the aggressive algorithm achieve a better performance. We have to keep in mind that many more prefetches are done under xFS as no real linear aggressive algorithms had been implemented.

It is interesting to see that there is an exception to the above explanation. If we examine the graph, we can see that in the 1Mbyte executions both Ln_Agr_IS_PPM:1 and Ln_Agr_IS_PPM:3 behave better than any of the others. This should not happen as the size of the cache is very small. The reason behind this odd behavior is a curious side effect. On one hand, the aggressive algorithms prefetch blocks continuously regardless of the block that is currently being requested. On the other hand, the same file blocks are requested by more than one process at different instants of time. The curious effect is that the blocks prefetched for a given process reach the cache when another process really needs them. This synchronization only happens with very small caches and when this workload is used. Anyway, this is just an anecdotic detail as we are interested in large caches due to the rapid increase in memory sizes.

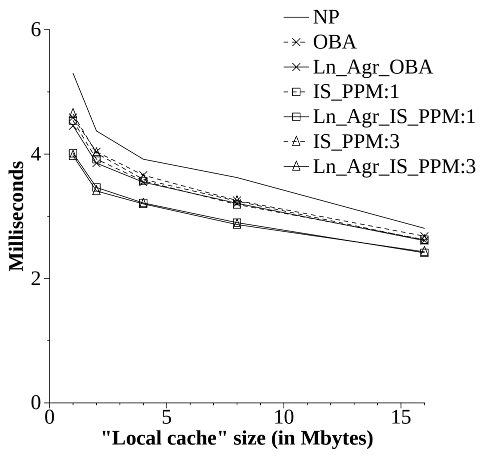

Figure 6. Average read time obtained running the Sprite workload under PAFS.

Sprite(NOW) on PAFS

To extend the results presented so far, we have also tested the prefetching algorithm on another architecture and using a different workload. Figure 6 presents the average read time obtained by the different prefetching algorithms when the Sprite workload was run on PAFS. This experiment has also been done for different cache sizes.

In this graph, we can see that both Ln_Agr_IS_PPM algorithms obtain the best performance as they are intelligent enough to detect the access patterns and aggressive enough to prefetch many needed blocks.

We can also observe that the non-aggressive versions of IS_PPM do not achieve results as impressive as in the CHARISMA experiment. This is due to their lack of aggressiveness. With this workload, requests are much smaller than in the CHARISMA experiment and thus the aggressiveness is also much lower.

Regarding the lack of performance obtained by Ln_Agr_OBA, there is a very simple explanation for it. The algorithm is not smart enough to correctly detect the access patterns used by the applications. As an example, with a 4-Mbyte cache, Ln_Agr_OBA has a miss-prediction ratio of 32% while Ln_Agr_IS_PPM only miss-predicts 15% of the prefetched blocks. This means that Ln_Agr_OBA wastes much time prefetching useless blocks instead of prefetching the ones really needed.

Finally, we can also observe that the improvements

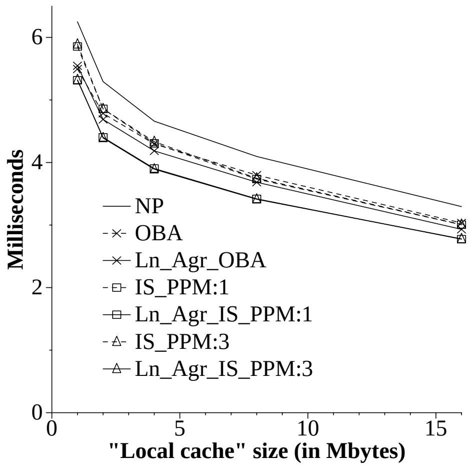

Figure 7. Average read time obtained the running Sprite workload under xFS.

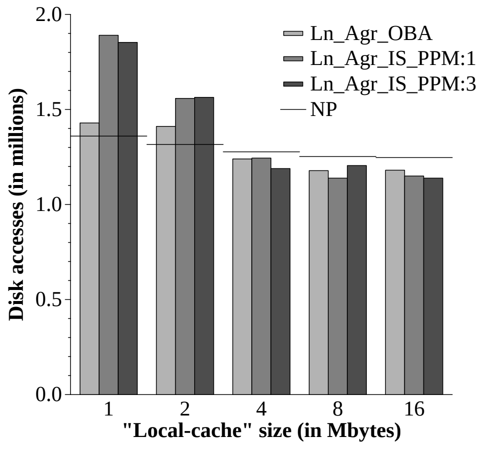

Figure 8. Disk accesses performed when running the CHARISMA workload under PAFS.

Mbytes	1	2	4	8	16
NP	5.9	8.8	11.7	11.7	11.7
Ln_Agr_OBA	5.2	7.9	10.4	10.9	11.0
Ln_Agr_IS_PPM:1	4.2	7.2	10.4	10.5	10.6
Ln_Agr_IS_PPM:3	4.0	7.6	10.1	10.5	10.5

Table 2. Average number of times a block is written to disk when running the CHARISMA workload under PAFS.

obtained by all prefetching algorithms are not as good as the ones observed under the CHARISMA workload. This is basically due the small size of the files. As there are many files that only have a few blocks, and our algorithms cannot predict the first block, the percentage of non-prefetched blocks increases.

Sprite(NOW) on xFS

The last experiment we have performed is to run the Sprite workload on xFS with all the prefetching algorithms (Figure 7). We can see that, in this case, there is no much difference between the results obtained by PAFS (linear) and the ones obtained by xFS (not really linear). This can be easily explained by examining the trace files. In the Sprite workload, there is very little file sharing which means that the non-linear aggressive algorithms will behave much like their linear version even if they are not. Another important characteristic of the workload is the size of the files. As they are quite small, they fit in the cache with no problem. For this reason, even when some file sharing occurs, a block will not be brought several times for different processes even with the non-linear algorithms.

4.2. Disk traffic

So far, we have only been interested in the performance achieved by the linear aggressive algorithms, but this is only one part of the story. It is also important to study how the disk traffic is increased due to these prefetching algorithms. If the disks saturate, this might end up slowing the whole system. For this reason, we have also studied the increase in the number of disk accesses due to the prefetching algorithms described in this paper.

Figure 8 presents the number of accessed blocks during the execution of the CHARISMA workload on top of PAFS. In the graph, we have drawn three bars that correspond to the number of accessed blocks when each of the aggressive algorithms was tested. The horizontal line represents the number of blocks accessed by the system when no prefetching was done.

In the graph, we can see that the extra number of disk blocks accessed is not very high except for very small caches (1 Mbyte). This means that this kind of prefetching can be an adequate solution as good performance is obtained without much disk traffic. Furthermore, we can observe that there are cases where this traffic is even lower than when no prefetching is done. The reason behind this anti-intuitive behavior is quite simple. On one hand, the number of read blocks increases with linear aggressive-prefetching algorithms. On the other hand, the number of blocks written to disk decreases more than the increase observed on disk reads due to the prefetching algorithms. Let us explain the reason behind this lower number of disk writes. There are many blocks that are modified many times during their life in the cache. For fault-tolerance issues, these blocks are periodically sent to the disk. If they remain in the cache for a long time and are modified frequently, they will be written to the disk many times. On the other hand, if they are in the cache only a small amount of time, they will not be written more than once or twice. This is what happens in this experiment. As the applications run much faster with aggressive prefetching algorithms, the time a block is active in the cache is also smaller and thus it is sent to the disk less times. To show this effect, we present the average number of times a block is sent to the disk in table 2.

The disk traffic in the rest of experiments is not shown due to lack of space, but similar results are obtained. The higher disk traffic is not significant compared to the benefits obtained.

5. Conclusions

The first achievement of this work has been the proposal of the IS_PPM family of prefetching algorithms. These algorithms use Markov models based on the offset intervals and size of the requested data.

We have also presented a mechanism that transforms very simple algorithms into aggressive ones that control their aggressiveness in a very simple way. We have also shown that these linear aggressive algorithms perform much better than their non-aggressive versions.

It is also important to remark, that although the proposed algorithms are quite aggressive, they do not load the disk much more than the system with no prefetching. Even more, there are some cases when such algorithms decrease the disk traffic while still achieving excellent performance.

Finally, although we first thought that such aggressive algorithms would only work with large caches, we have also seen that this is not necessarily true. Very good performance results have also been obtained with fairly small ones.

References

[1] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *15th SOSP*, pages 109–126. ACM, 1995.

[2] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *13th SOSP*, pages 198–212. ACM, 1991.

[3] M. Brodowicz and O. Johnson. Paradise: An advanced featured parallel file system. In *ICS*, pages 220–226. ACM, 1998.

[4] T. Cortes. *Cooperative Caching and Prefetching in Parallel/Distributed File Systems*. PhD thesis, Universitat Politècnica de Catalunya, Departament d'Arquitectura de Computadors, 1997. http:// www.ac.upc.es/ homes/ toni/ thesis.html.

[5] T. Cortes, S. Girona, and J. Labarta. Design issues of a cooperative cache with no coherence problems. In *5th IOPADS*, pages 37–46, 1997.

[6] K. M. Curewitz, P. Kristian, and J. S. Vitter. Practical prefetching via data compression. In *SIGMOD Management of Data*, pages 257–266. ACM, 1993.

[7] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *1st OSDI*, pages 267–280. USENIX, 1994.

[8] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bershad, P. Cao, E. W. Felten, G. A. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *2nd OSDI*, pages 19–34. USENIX, 1996.

[9] D. Kotz and C. S. Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, 1990.

[10] T. M. Madhyastha and D. A. Reed. Input/Output pattern classification using hidden markov models. In *5th IOPADS*, pages 57–67. ACM, 1997.

[11] N. Nieuwejaar, D. Kotz, A. Purakayastha, C. S. Elli, and M. L. Best. File-access characteristics of parallel scientific workloads. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1075–1089, 1994.

[12] R. H. Patterson and G. A. Gibson. Exposing I/O concurrency with informed prefetching. In *3rd International Conference on Distributed Information Systems*, pages 7–16, 1994.

[13] A. J. Smith. Disc cache - miss ratio analisys and design considerations. *ACM transactions on Computer Systems*, 3(3):161–203, 1985.