Proc. 4th International Information and Telecommunication Technologies Symposium (I2TS '05), Florianopolis, SC, Brazil, pages 122–129, Dec. 2005

Making Early Predictions of File Accesses

Natarajan Ravichandran Jehan-François Pâris
Department of Computer Science
University of Houston
Houston, TX 77204-3010 USA

***Abstract*—Nearly all extant file access predictors attempt to identify the immediate successor to the file being currently accessed. As a result, they leave almost no time to prefetch the predicted file before it is accessed. We present here a Perceptron-based file predictor that identifies the files that will be accessed up to five accesses ahead. Experimental evidence shows that our early predictor can make between 30 and 78 percent of correct predictions, depending on the nature of the file system workload and the position of the access it attempts to predict.**

I. INTRODUCTION

The wide gap between main memory and disk access times is one of the most vexing issues in computer architecture. The problem is not new and is likely to worsen as memory access times continue to decrease at a much faster rate than disk access times.

Two main techniques have been used to alleviate the problem, namely caching and prefetching. Caching keeps in memory the data that are the most likely to be used again while prefetching attempts to bring data in memory before they are needed. Both techniques have been widely used at the block level and start now to be applied at the file level. File-level prefetching is inherently more difficult to implement than file-level caching because prefetching files that are not needed can have a direct negative impact on system performance while keeping in a cache files that will not be reused only reduces the cache effectiveness. A key requirement for a successful implementation of file prefetching is thus a good file access predictor. This predictor should have reasonable space and time requirements, make as many correct predictions as possible and as few false predictions as feasible.

Nearly all file predictors investigated so far have tried to predict the *immediate successor* to the last file being referenced. In that sense, they do very short-term predictions and leave almost no time to prefetch the predicted file before it is accessed. The sole exception is Kroeger and Long's Extended Partitioned Context Model [6], which can predict sequences of future accesses. This approach left more time to prefetch the predicted files and was shown to actually speed up the throughput of the system.

We present here a more direct technique. Rather than predicting sequences of future accesses, our Perceptron-based file predictor attempts to predict which files will be accessed a few accesses in the future without considering the intermediary accesses. As a result, our predictor has lower space requirements. This does not prevent it from being successful in predicting which files will be accessed up to five file accesses ahead.

The remainder of the paper is organized as follows: Section 2 reviews previous work on file access prediction. Section 3 introduces our Perceptron-based file predictor and Section 4 discusses its performance. Finally, Section 5 has our conclusions.

II. PREVIOUS WORK

Palmer *et al.* [9] used an associative memory to recognize access patterns within a context over time. Their predictive cache, named *Fido*, learns file access patterns within isolated access *contexts*. Griffioen and Appleton presented, in 1994, a file prefetching scheme relying on graph-based relationships [5]. Shriver *et al.* [14] proposed an analytical performance model to study the effects of prefetching for file system reads.

Tait and Duchamp [15] investigated a client-side cache management technique used for detecting file access patterns and for exploiting them to prefetch files from servers. Lei and Duchamp [7] later extended this approach and introduced the *Last Successor* predictor. More recent work by Kroeger and Long introduced more effective schemes based on context modeling [6]. Unlike previous predictors, their Extended Partitioned Context Model could successfully predict several references ahead.

Amer *et al.* recently proposed two much simpler predictors [1, 2] that were found to perform very well on a wide variety of workloads. The *Stable Successor* predictor is a refinement of the Last Successor predictor that attempts to filter out noise in the observed file reference stream. Stable Successor keeps track of the last observed successor of every file, but it does not update its past prediction of the successor of file X before having observed m successive instances of file Y immediately following instances of file X. Hence, given the sequence:

S: ABABABACABACABADADADA

Stable Successor with $m = 3$ will first predict that B is the successor of A and will not update its prediction until it encounters three consecutive instances of file D immediately following instances of file A.

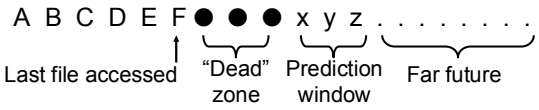

Fig. 1. Our prediction model

The *Recent Popularity* or *k-out-of-m* predictor keeps track of the *m* most recently observed successors of each file. When attempting to make a prediction for a given file, Recent Popularity searches for the most popular successor among these *m* files. If the most popular successor occurs at least *k* times, then it is submitted as a prediction. When more than one file satisfies the criterion, recency is used as the tie-breaker. Whittle *et al.* [16] added a third parameter $l \le k$ to let Recent Popularity predict the last successful *j-out-of-k* if it appears at least *l* times in the list of *m* last successors.

The *Composite File Predictor* [16] applies four independent heuristics to the same context information and select the one that is the most likely to deliver an accurate prediction. These four heuristics are (1) Stable Successor, (2) Predecessor Position, (3) Pre-predecessor Position and (4) Recent Popularity.

Shah *et al.* [13] showed that file predictors that identify stable access patterns, and never alter their predictions, could predict between 50 and 70 percent of next file accesses over a period of one year.

Finally, Brandt *et al.* [4] presented a composite predictor that combines multiple predictors or "experts" to reduce the number of false predictions. Their set of experts includes a *null* prediction expert that suppresses prefetching whenever the likelihood of an accurate prefetch is low.

III. Our Predictor

We want to predict file accesses sufficiently ahead of time to be able to fetch the predicted files before they are actually accessed. Estimating the time delay T_{fetch} required to fetch a specific file is a difficult proposition as we need to know the size of the file, the characteristics of the disk drive on which it is stored and its utilization. We took a much simpler approach in this study. Rather than attempting to predict the file access that will occur T_{fetch} time units after the current access, we try to predict which file will be accessed *n* file accesses after the current one. As seen on Fig. 1, we define a *dead zone* of *n* file accesses that we assume to occur too quickly after the current file access. In addition, we define a *prediction window* of duration *m* and deem our prediction to be a success if the predicted file is accessed within this prediction window. This prediction window represents the fact that correctly predicting the *n*+2th or the *n*+*3*th next file access will have the same beneficial effect as correctly predicting the $n+1^{th}$ next file access.

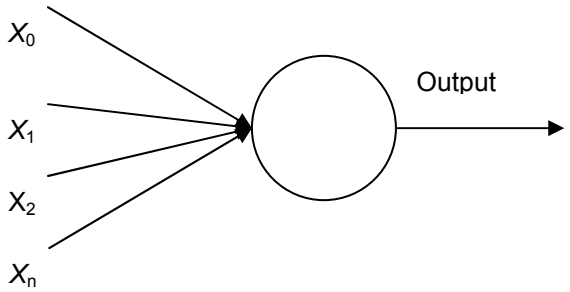

Fig. 2. A simple Perceptron

A. The Perceptron

As seen on Fig. 2, a simple Perceptron [11] has *n* inputs, a weight associated with each input and an output function to determine the output. A pattern is entered through the input, and the output is calculated according to the linear function:

$$Output = \sum_i W_i X_i$$

where the X_i are the inputs to the Perceptron and the W_i are the weights for the corresponding inputs. If the output value is greater than a threshold value, the output of the Perceptron is one else, it is zero. In some cases, fuzzy values are also used to denote the inputs and output of the Perceptron.

The simple Perceptron is capable of solving linearly separable classification problems. If the problem is linearly separable, there exists a learning rule that converges in finite time.

A Perceptron has two modes of operation: training and classification. Training can be done using either unsupervised or supervised learning schemes. In supervised learning, pairs of data consisting of the input pattern and the target output are presented to the Perceptron and the weights are modified according to the training rule. The training rule is as follows:

New Weight = *Old Weight* +
(*Expected Output* – *Actual Output*)×*Learning Rate*

where *Learning Rate* is a constant to be adjusted experimentally. It is important to note that the initial weight values and the learning rate do not affect the classification capability of the Perceptron. Their influence is restricted to the time it will take for the training to be complete.

B. General organization of our predictor

The general organization of our predictor is very simple. We associate one Perceptron to each file whose accesses we want to predict. The inputs of each Perceptron are the files present in the last *k* observed file accesses. We start by running these Perceptrons in training mode and "teach" them which patterns in these last *k* observed file accesses are the best predictors of the $n+1^{th}$ next file access. Ideally, a well-trained Perceptron should return a value close to one when it predicts that its associated file will be accessed and a value close to zero otherwise.

Initialization

For each file in the "hot" list do
Initialize a Perceptron with the file name
End For

Figure 3. The initialization phase

Training

For each file from 4+n^{th} file to last file in the training set do
If the file is not in "hot" list then
Go to the next file
Endif
Identify the four files accessed n file accesses before the current file.
Identify the Perceptron corresponding to the current file.
For each of the four files do
If the file is in the input set of the Perceptron then
New weight = Old weight + 0.3.
Else
Add file to input list.
Weight = random()
Endif
If Perceptron has more than seven inputs then
Keep the inputs with seven highest weights.
Delete all other inputs.
Endif
End for
End for

Fig. 4. The training phase

While keeping this overall structure, we had to introduce some modifications either to reduce the space and time costs of the predictor or to adapt the Perceptron functions to the nature of the problem.

We realized first that it would be impractical to have one Perceptron for each file in the file system and decided instead to limit ourselves to the so-called "hot" files that are referenced much more frequently than the other files and are responsible for 96 percent of the file accesses. This offers the double advantage of reducing the space and time costs of our predictor while focusing our effort on the files for which we would have training samples that would be large enough.

Second, the set of input neurons for each Perceptron was to include all files that were accessed during the training period. To reduce the space and time requirements of our predictor, we decided to use a rather small observation window that would only comprise the last four file accesses (k = 4) and

Continuous Retraining

Select a training window duration D
Do initial training during first window of file accesses
For all successive windows
Predict file accesses using Perceptrons trained during the previous window
Retrain Perceptrons for the next window
End for

Fig. 5. How the Perceptrons are continuously retrained

limit the number of input neurons to seven by only keeping the neuron inputs with the seven highest weights.

Finally, we needed a criterion defining when to make a prediction and when to decline to make one. We decided to let our system predict the file associated with the Perceptron that had the highest output as long as at least two input neurons of that Perceptron were firing. If this condition is not met, no prediction is made.

The initialization phase of our predictor is fairly simple. As Fig. 3 shows, we initialize one Perceptron for each "hot" file.

The training phase consists of a single run through all the training set. As seen on Fig. 4, the update rule is quite simple. At each step of the algorithm, we compare each of the four files that were accessed last with the input neurons of the Perceptron. If they are equal, then the weight of the corresponding input is updated as follows:

New Weight = Old Weight + Learning Rate.

If the file has no corresponding input neuron, then we add a new input neuron to the Perceptron with a random weight. This could lead to very large numbers of input neurons per Perceptron, thus increasing its memory requirements. To avoid this problem, we decided to prune the input space of the Perceptron at that stage by only retaining the seven input neurons that have the seven highest weights. This number of input neurons was assigned arbitrarily.

The outcome of the training phase is one Perceptron for each "hot" file. The input set of this Perceptron will contain the seven most frequently accessed files $n + 1$, $n + 2$, $n + 3$, and $n + 4$ accesses before.

C. *Longevity of file access patterns*

Shah *et al.* [13] found that some workloads had very stable access patterns. Once we have found a good predictor of the stable successor of a file for such a workload, there is hardly any need to update this prediction. Conversely, they also observed that some workloads were much less stable and required the continual update of our predictions.

One of the advantages of the conventional Perceptron learning rule is the option to train the Perceptron online to respond to changing input patterns. Whenever the input patterns observed during the classification phase start to differ from the patterns identified during the training phase, the Perceptron goes through a phase of a reduced degree of

correctness in classification. The Perceptron unlearns the stale input patterns during this phase and the weights of the Perceptron are adjusted to reflect the new patterns according to the Perceptron learning rule. During this online learning phase of mostly negative updates, the utility of the Perceptron is seriously hampered by its reduced correctness.

In the case of file prediction, we are faced by the challenge of changing file patterns. File patterns inevitably change over time and we need to come up with a viable solution so that the performance of the file predictor does not get hampered by these changes.

Two direct solutions may be considered to overcome the problem of changing file patterns. The first solution consists in training the Perceptrons for the first half of the trace and depending on the patterns learnt during that period to predict files in the future. This approach has the double advantage of requiring no online training and eliminating the phase of negative updates when the performance of the Perceptrons are below par. However, this approach assumes that a set of simple Perceptrons can learn patterns encompassing a wide range of changing values. It ignores the dynamics of file pattern changes over time. In addition, the aspect of time in the weights learned is missing. This approach also places unreasonable expectation on a set of simple Perceptrons to learn and classify complex patterns. As expected, the Perceptrons do not perform well for large file traces. This led us to conclude that this kind of training was not specific enough.

A better approach consists of training the Perceptrons with an initial training set and updating the weights online to reflect the changing patterns. Unfortunately, this approach forces the Perceptron to go through a phase where its efficiency will drop. This approach typically works for a simple Perceptron working on a linearly separable classification domain with the conventional learning rule. File access patterns are not simple and do not necessarily confine themselves to a linearly separable domain. Hence, not only is such an approach computationally expensive and affects the accuracy of the predictor it would also not work in most cases due to the nature of the file access patterns.

Keeping the above factors in mind, we came up with a scheme that predicts file accesses while retraining at the same time the Perceptrons for the next prediction.

Our scheme overcomes the disadvantages associated with the two methods explained above and presents a viable way to deal with the problem of changing file patterns. After the conclusion of the initial offline training, we do not carry out any additional I/O for file training. We ensure that the file patterns used for training are not stale by continuously retraining our Perceptrons at regular intervals determined by the prediction window size. This approach also ensures that the prediction for the current window is not affected by the training for the next window as a separate set of inputs and weights are created for predicting in the next window.

In addition, our scheme does not occasion any additional I/O because the accessed files have to be brought in to main memory one way or another (either through file prefetching or caching or via an access to disk in case both fails) and it is this information of which files are accessed that is used to train the set of inputs and weights for prediction in the next window. There is little computational overhead because for every file access after 4+*n*, only one Perceptron has to be updated in the case of the accessed file being a "hot" file. The overhead in terms of storing the weights and inputs for the next window of prediction is reasonable and enables the file predictor to make predictions with improved accuracy. Hence, we believe that combining online prediction with offline training for the next window provides a viable answer to the problem of changing file patterns in file traces. The window can be set large or small depending on the longevity of the file access patterns in the trace.

Prediction Phase

For all file accesses from 4+n^{th} to last in the current window do

For all Perceptrons do

Set to one each input that matches a file identifier in the current input pattern

If at least two of the inputs are one then

Calculate output using current weights

Else

Set output to zero

Endif

If the output of at least one Perceptron is non zero then

Find the Perceptron with the highest output value.

Predict the file associated with that Perceptron as the next file n accessed ahead.

Else

Make no prediction.

Endif

End for

End for

Fig. 6. The prediction phase

Once the initial training is carried out, we are ready to start predicting. Note that the prediction depends on the duration of the dead zone for which the Perceptrons were trained. In other words, we can predict only as far ahead as the Perceptrons were trained.

After 4+*n* file accesses, the four files accessed before *n* file accesses are supplied as inputs to all the Perceptrons. The input(s) for Perceptrons having identical file identifiers in the input set are set to one. The outputs for all Perceptrons are calculated as the sum of the inputs and the corresponding weights. Then the file associated with the Perceptron having the highest output amongst all the Perceptrons in the current calculation is predicted to be the next file. There is no updating of weights in either case of correct or wrong prediction because training is simultaneously carried out for the next window of prediction rendering unnecessary the adjustment of weights for the current set of inputs.

We quickly found out that this simple scheme had a major problem. It always made a prediction unless none of the files in the current input pattern was present in the input set of any of the Perceptrons. As a result, our predictor made too many wrong predictions based on non recurring weak patterns, thus decreasing its overall accuracy. To solve the problem, each Perceptron must now have at least two of its input neurons set to one before making a prediction. This ensures that the Perceptron does not make predictions for stray patterns. Fig. 6 summarizes our final algorithm for the prediction phase.

IV. Experimental Results

We evaluated the performance of our Perceptron predictor by simulating its operation on two sets of file traces. The first set consisted of four file traces collected using Carnegie Mellon University's DFSTrace system [8]. These traces include *mozar*t, a personal workstation, *ives*, a system with the largest number of users, *dvorak*, a system with the largest proportion of write activity, and *barber*, a server with the highest number of system calls per second. They include between four and five million file accesses collected over a time interval of approximately one year. Our second set of traces was collected in 1997 by Roselli [12] at the University of California, Berkeley over a period of approximately three months. To eliminate interleaving issues, these traces were processed to extract the workloads of an instructional machine (*instruct*), a research machine (*research*) and a web server (*web*).

These traces presented the advantage of displaying a wide variety of file access patterns and of having been used in several previous studies [1, 2, 3, 13, 16]. We knew in particular that the *barber* and *mozart* traces from CMU exhibited more stable behaviors than the five other traces, while the *research* and *web* traces from UC Berkeley had the most volatile behaviors.

Fig. 7 summarizes our results concerning the accuracy of our predictor, that is, the number of correct predictions made by the predictor over the total number of predictions it made. As we can see, the determining factor affecting the accuracy of our predictor is the file access patterns exhibited by each trace. Our predictor achieves accuracies between 72 and 85 percent for three of the CMU traces while only obtaining accuracies between 40 and 50 percent for the *web* trace.

We can also observe that the accuracy of the predictions remains good even when we predict one, two, three or four references ahead. The sole counter-example is again obtained with the *web* trace for which we observe a measurable decline between the accuracies of the predictions for the fourth and the fifth next file access.

Another important aspect of the performance of a predictor is its coverage, that is, the number of predictions it makes over the total number of file accesses. As Fig. 8 indicates, our Perceptron predictor makes predictions for between 74 and 94 percent of the file accesses. As expected, the predictor declines more often to make a prediction when the trace has a less stable behavior or when the predictor is asked to predict file accesses that are further ahead. We should also note that the characteristics of each trace affect the coverage of our predictor at least as much as the prediction distance in the future.

Fig. 9 displays the success rates per reference of our Perceptron predictor. This index represents the number of correct predictions our predictor makes over the total number of file accesses and is thus equal to the product of the coverage of the predictor by its accuracy. As we can see, it is significantly more sensitive to the characteristics of each trace and the position of the access it attempts to predict than either of the two other indexes.

Our study would not be complete without discussing how our Perceptron predictor compares with other file access predictors such as Last Successor, Stable Successor and Best *k*-out-of-*m*. Since these predictors only attempt to predict the *next* file access, we will limit our study to that case.

Fig 10 contrasts the overall success rates achieved by our protocol when attempting to predict the next reference with these achieved by Last Successor, Stable Successor and Best *k*-out-of-*m*. As we can see, the Perceptron predictor is in a tie with Last Successor and is always outperformed by both Stable Successor and by Best *k*-out-of-*m* [10].

These results are still good if we consider that they were obtained by a predictor that was not tuned in any fashion, only considered the last four accesses to the file system and had a very simplistic rule for deciding when to decline to make a prediction. More sophisticated predictors would very likely perform better just as Stable Successor and Best *k*-out-of-*m* predictors perform better than the Last Successor.

V. Conclusions

Most conventional file access predictors try to predict the immediate successor of the file being currently accessed. As a result, they leave almost no time to prefetch the predicted file before it is accessed. We have presented here a Perceptron-based file predictor that attempts to predict the files that will be accessed up to five file accesses ahead. Experimental evidence shows that our predictor can make between 30 and 78 of correct predictions, depending on the file system workload and the position of the access it attempts to predict.

We can draw two important conclusions from our data. First, even simple predictors can predict file accesses up to at least six references ahead. Second, making early predictions will not be feasible for all kinds of file access workloads. The technique is likely to work very well with workloads exhibiting stable access patterns and fail with workloads characterized by less predictable access patterns, such as a web access trace.

More work is still needed to reduce the number of incorrect predictions, to investigate other early predictors, and to evaluate the impact of file caching on the performance of our predictor.

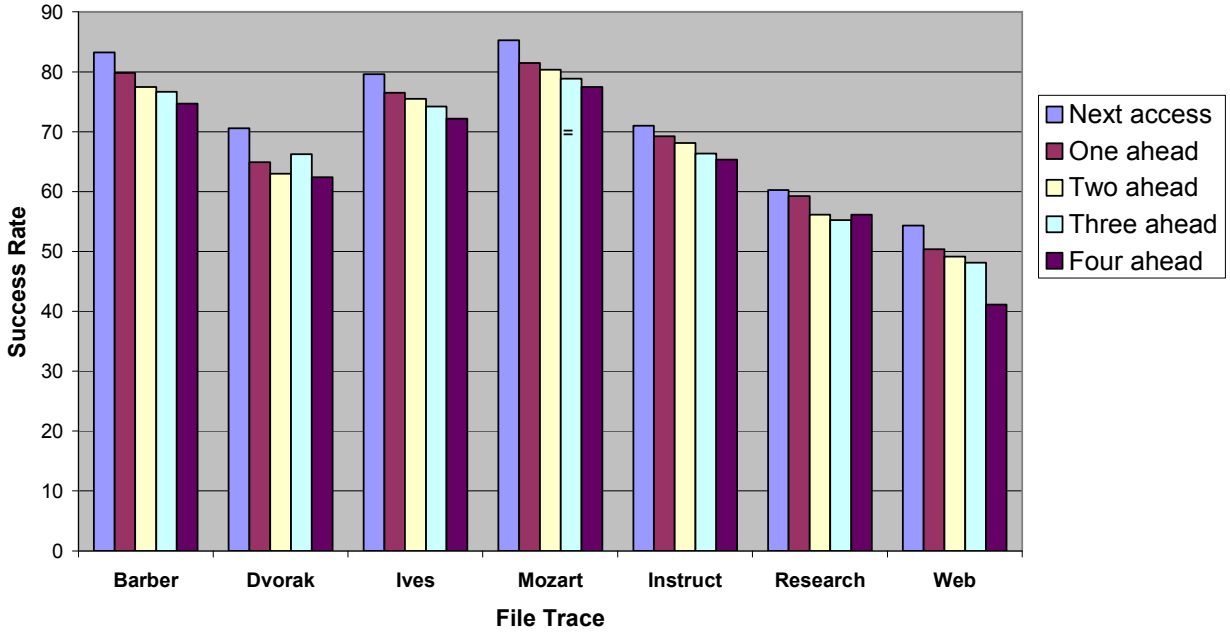

Fig. 7. Accuracy of the predictions made by our Perceptron Predictor

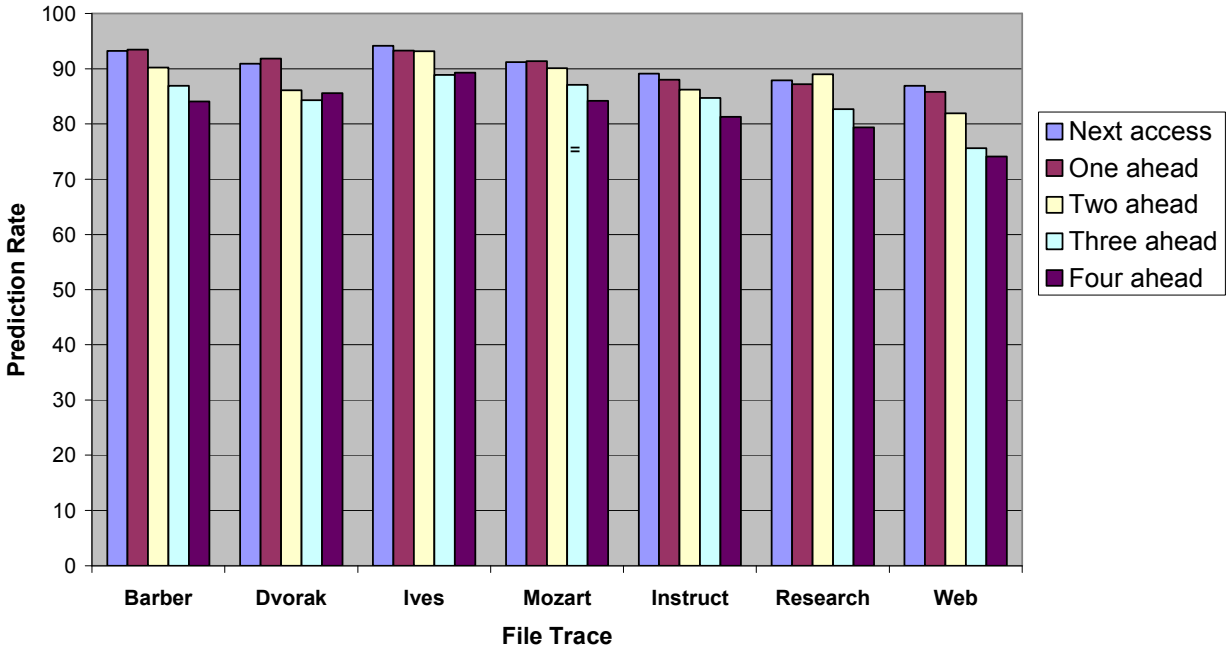

Fig. 8. Coverage of our Perceptron Predictor

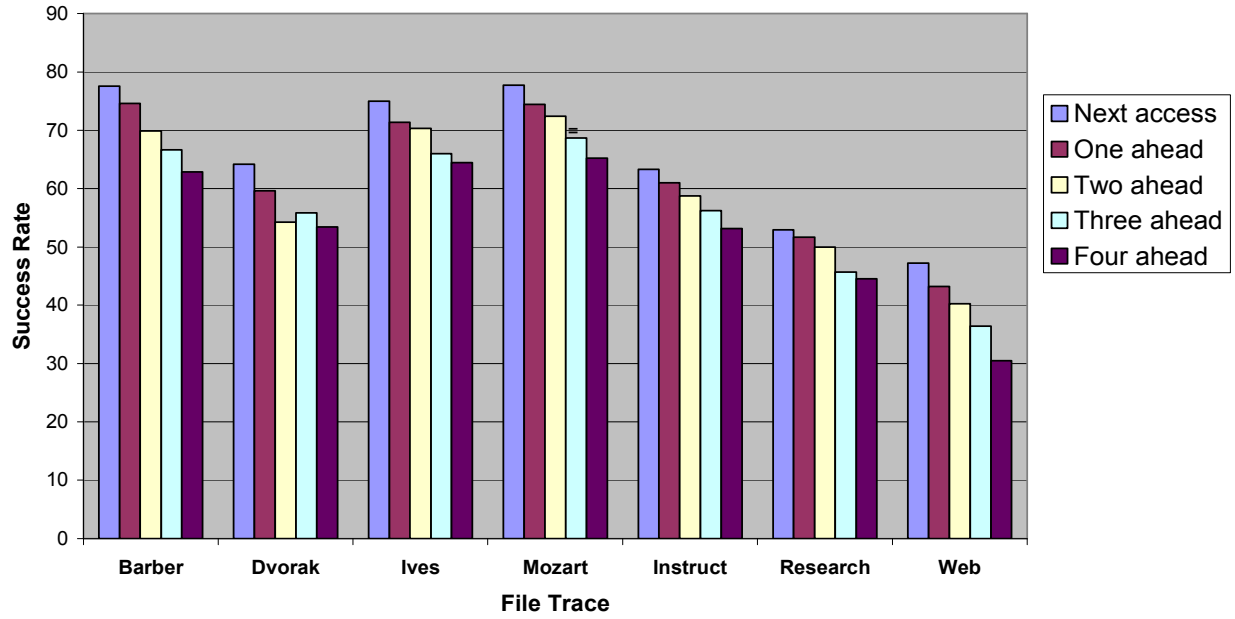

Fig. 9. Overall success rate of our Perceptron Predictor

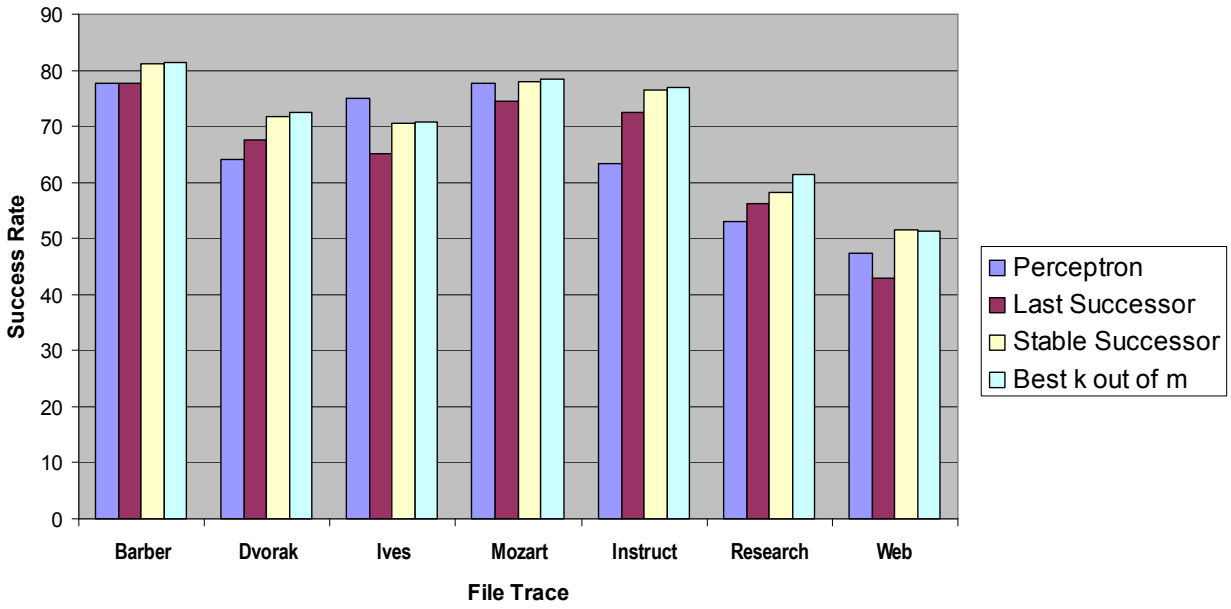

Fig. 10. Compared overall success rate of the Perceptron predictor, Last Successor, Stable Successor and Best *k* out of *m* when predicting the *next* file access.

References

[1] A. Amer and D. D. E. Long, Noah: Low-cost file access prediction through pairs, in *Proc. 20th Int'l Performance, Computing, and Communications Conf.* (IPCCC '01), pp. 27–33, Apr. 2001.

[2] A. Amer, D. D. E. Long, J.-F. Pâris, and R. C. Burns , File access prediction with adjustable accuracy, in *Proc. 21st Int'l Performance of Computers and Communication Conf.*, pp. 131–140, Apr. 2002.

[3] A. Amer, D. Long, and R. Burns. Group-based management of distributed file caches, in *Proc. 17th Int'l Conf. on Distributed Computing Systems* (ICDCS '01), pp. 525–534, July 2002.

[4] K. Brandt, D. D. E. Long and A. Amer, Predicting When Not To Predict, in *Proc. 12th Int'l Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (MASCOTS '04), pp. 419–426, Oct. 2004.

[5] J. Griffioen and R. Appleton, Reducing file system latency using a predictive approach, in *Proc. 1994 Summer USENIX Conf.*, pp. 197–207, June 1994.

[6] T. M. Kroeger and D. D. E. Long, Design and implementation of a predictive file prefetching algorithm, in *Proc. 2001 USENIX Annual Technical Conf.*, pp. 105–118, June 2001.

[7] H. Lei and D. Duchamp, An analytical approach to file prefetching, in *Proc. 1997 USENIX Annual Technical Conf.*, pp. 305–318, Jan. 1997.

[8] L. Mummert and M. Satyanarayanan, Long term distributed file reference tracing: implementation and experience, Technical Report, School of Computer Science, Carnegie Mellon University, 1994.

[9] M. L. Palmer and S. B. Zdonik, FIDO: a cache that learns to fetch, in *Proc. 17th Int'l Conf. on Very Large Data Bases (VLDB)*, Barcelona, pp. 255–264, Sept. 1991.

[10] N. Ravichandran. *Making Early Predictions of File Accesses*. MS Thesis, Department of Computer Science, University of Houston, Aug. 2005.

[11] F. Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain, *Psychological Review*, 65(6):386–408, Nov. 1958

[12] D. Roselli, Characteristics of file system workloads, Technical. Report CSD-98-1029, University of California, Berkeley, 1998.

[13] P. Shah, J.-F. Pâris, A. Amer and D. D. E. Long. Identifying stable file access patterns, in *Proc, 21st IEEE Symp. on Mass Storage Systems* (MSS 2004), pp. 159–163, April 2004

[14] E. Shriver, C. Small, and K. A. Smith, Why does file system prefetching work? in *Proc. 1999 USENIX Technical Conf.*, pp. 71–83**,** June 1999.

[15] C. Tait and D. Duchamp, Detection and exploitation of file working sets, in *Proc. 11th Int'l Conf. on Distributed Computing Systems* (ICDCS '91), pp. 2–9, May 1991.

[16] G. A. S. Whittle, J.-F. Pâris, A. Amer, D. D. E. Long and R. Burns. Using multiple predictors to improve the accuracy of file access predictions, in *Proc. 20th IEEE Symp. on Mass Storage Systems* (MSS 2003), pp. 230–240, April 2003.