Using Multiple Predictors to Improve the Accuracy of File Access Predictions

Gary A. S. Whittle
U. of Houston
gwhittle@hess.com

Jehan-François Pâris[1]
U. of Houston
paris@cs.uh.edu

Ahmed Amer[2]
U. of Pittsburgh
a.amer@acm.org

Darrell D. E. Long[2]
U. C. Santa Cruz
darrell@cs.ucsc.edu

Randal Burns
Johns Hopkins U.
randal@cs.jhu.edu

Abstract

Existing file access predictors keep track of previous file access patterns and rely on a single heuristic to predict which of the previous successors to the file being currently accessed is the most likely to be accessed next. We present here a novel composite predictor that applies multiple heuristics to this selection problem. As a result, it can make use of specialized heuristics that can make very accurate predictions when access patterns are observed to meet their particular criteria. Simulation results involving a total of seven file access traces indicate that our predictor delivers more correct predictions and less inaccurate guesses than predictors relying on a single heuristic for selecting a successor.

1. Introduction

One of the most difficult problems facing operating systems designers is finding the best way to manage memory hierarchies consisting of devices with widely different access times. The problem is not new and is in fact worsening as gains in main memory access times have dramatically outpaced gains in disk access times.

Two main techniques can be used to mitigate this 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. Prefetching is inherently more difficult to implement than caching because prefetched data that are not needed can have a direct negative impact on system performance while keeping in a cache data that will not be reused only reduces the cache effectiveness. As a result, most systems err on the side of caution and do not exploit the full potential of the technique. In particular, no existing system implements *anticipatory file prefetching*, that is, prefetching entire files before they are accessed.

[1] Supported in part by the Texas Advanced Research Program under grant 003652-0124-1999 and the National Science Foundation under grant CCR-9988390.

[2] Supported in part by the National Science Foundation under grant CCR-9972212 and the USENIX Association.

One of the key requirements for a successful implementation of anticipatory file prefetching is a good *file access predictor*. This predictor should have reasonable space and time requirements, make as many successful predictions as possible and as few bad predictions as feasible.

Most early file access predictors [14, 5, 9, 8] relied on sophisticated heuristics that required maintaining a large amount of information about past references to each file. Two more recent contributions [1, 2] have shown that very simple predictors requiring much less context information could provide surprisingly accurate predictions. We propose a novel approach. Rather than relying on a single predictor we apply several independent heuristics to the same context information and select the one that is the most likely to deliver an accurate prediction. This approach has two major advantages. First, we do not have to rely on a single general-purpose method and can make use of several more specialized heuristics that return very accurate predictions when some specific access pattern is present. Second, having all heuristics sharing the same context information means that we do not incur any additional overhead.

Simulation results involving a total of seven file access traces have corroborated the soundness of our approach: combining several heuristics increases the number of good predictions while reducing the number of incorrect guesses.

The remainder is organized as follows. Section 2 reviews previous work on file access prediction. Section 3 introduces the *effective-miss-ratio* criterion that we used in our study. Section 4 presents the four heuristics comprising our predictor and discusses their individual performances. Section 5 introduces our predictor and compares its performance with that of other predictors, namely *Last Successor*, *Stable Successor* and *Recent Popularity*. Finally, Section 6 states our conclusions.

2. Previous Work

Palmer *et al.* [11] 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.* [13] proposed an analytical performance model to study the effect of prefetching for file system reads.

Tait *et al.* [14] 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 [9] 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 and data compression [8].

Two much simpler predictors, *Stable Successor* (or Noah) [1] and *Recent Popularity* [2], have been recently proposed. The *Stable-Successor* predictor is a refinement of the last-successor predictor that attempts to filter out noise in the observed file reference stream. Noah maintains the id of the last file that was a successor to the current file (the "last successor"), as well as a current "stable successor." The stable successor is updated with the id of the last successor only if the last successor is the same for a "stability" count number of references. The *Recent Popularity* or *j-out-of-k* predictor maintains the *k* 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 from the list. If the most popular successor occurs at least *j* times then it is submitted as a prediction. When more than one file qualifies as "most popular," recency is used as the tiebreaker.

Finally, Yeh *et al.* investigated a simple but effective successor model that identifies the relationships between files through identification of the programs accessing them [17].

3. Evaluating the Performance of a File Predictor

When comparing the effectiveness of file predictors, one is often confronted with two primary metrics, *success-per-reference* and *success-per-prediction*. Because of the dependent nature of these metrics, it is not possible to use either of them alone when assessing the performance of any given predictor. For example, a predictor that has a 99% *success-per-prediction* rate would be considered impractical if it could only be used on 5% of the references. Conversely, predictors that have a high *success-per-reference* rate may also give rise to a high number of incorrect predictions that may tax the file system to an extent that outweighs any improvements due to predictive prefetching. Another disadvantage of success-based metrics is that they discount the performance improvement achieved by increasing the success rate of a predictor from, say, 86 to 93%. It makes this improvement appear marginal, even though it represents a 50% reduction in the number of *misses*.

We propose a third metric integrating both aspects of the predictor performance. Consider first the two possible outcomes of an incorrect prediction. If we assume no preemption, the next file access will have to wait while the predicted file is loaded into the cache. The cost of the incorrect prediction is thus one additional cache miss. Allowing preemption would reduce this delay and decrease the penalty. Note that the incorrect prediction will have no other adverse effect on the cache performance as long as the cache replacement policy expels first the files that were never accessed.

We define the *effective-miss-ratio* of a predictor as the ratio:

$$\frac{N_{ref} - N_{corr} + \alpha N_{incorr}}{N_{ref}}$$

where N_{corr} is the number of correct predictions, N_{incorr} the number of incorrect predictions and N_{ref} the number of references and the α factor represents the impact of file fetch preemption on the performance of the predictor. A zero value for α corresponds to the situation where incorrect predictions incur no cost because all predicted file fetches can be preempted when found to be incorrect without any further delay. A unit value assumes that there is no fetch preemption, and all ongoing fetches must be completed, whether correctly predicted or not. An intermediate α value corresponds to situations where preemption is possible, but at some cost less than the cost of a file fetch. Computing the *effective-miss-ratio* for α values of, say, 0.0, 0.5 and 1.0 will permit us to compare predictors for a realistic range of file-system implementations. Predictors that perform well for all α values will be said to be *α-stable* and assumed to be more likely to live up to expectations once implemented.

4. The Four Base Heuristics

As observed by the authors [2, 16], between 80% and 90% of successors for any file reference are present among the last 10 to 20 successors of that file. We can safely reduce the problem of file prediction to that of selecting the next successor from a limited successor history. If the actual successor is not present in that history, it is unlikely that it will occur in an extended successor history, even we increase this history to impractical lengths.

We investigated several possible heuristics and selected four that we evaluated by simulating their operation on two sets of file traces. The first set consisted of four file traces collected using Carnegie Mellon University's DFSTrace system [10]. The traces include *mozart*, 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. These traces provide information at the system-call level, and

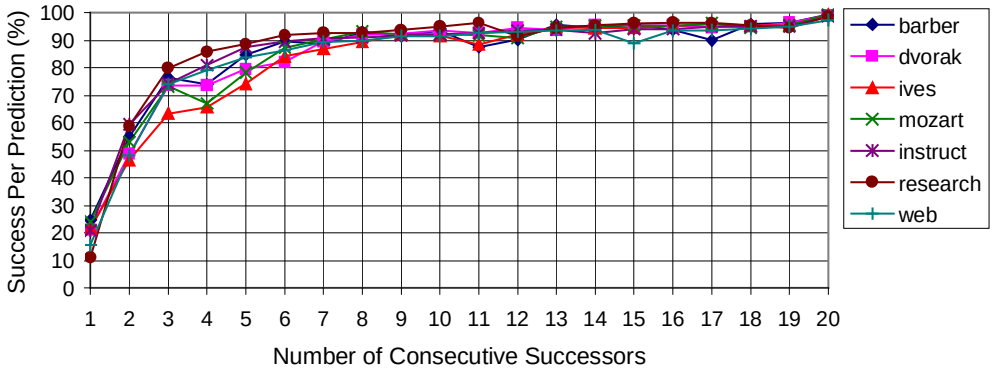

Figure 1: Performance of Most Recent Consecutive Successors Heuristic

represent the original stream of access events not filtered through a cache. They include between four and five million file accesses. 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 any interleaving issues, these traces were processed to extract the workloads of an instructional machine (*instruct*), a research machine (*research*) and a web server (*web*).

It should be pointed out that we based our evaluation of the four heuristics on the *success-per-prediction* metric. For any given reference, we want to determine which heuristic, if it causes a prediction to be made, predicts most successfully. Heuristics that cannot be applied often are still of importance if they predict well, as they can be used in conjunction with other heuristics to achieve the lowest possible *effective-miss-ratio*. This is the main advantage of using several heuristics in our predictor: we do not have to limit ourselves to general-purpose heuristics, but can take advantage of more accurate models when the criteria for their use are met.

4.1. Most recent consecutive successors

The most intuitively powerful heuristic for successor selection is the presence of most recent consecutive successors in the successor history. If we encounter the file reference sequence "ABCBCBCB," it is quite probable that the next reference will also be to file "C." The successor history for file B at this point will end in "CCC." Note that we are interested in the ability of this heuristic to predict the *next* reference, not references beyond that.

We tested this heuristic against all of the file system traces to derive the relationship between *success-per-prediction* and the number of most recent consecutive successors. The results are shown in Figure 1. For the sequence "ABCBCBCB," the *success-per-prediction* percentage if we assume that "C" will be referenced next is between 63% and 80%, depending on the trace. For the sequence "ABCBCBCBCBCBCBCB" this increases to between 87% and 92%. On the whole, the *success-per-prediction* curve increases linearly as the number of consecutive successors increases from one through three, after which it begins to reduce to a fairly shallow slope beyond about six successors. For all traces, more than six most recent consecutive successors are a strong indicator that this successor will be referenced next.

4.2. Predecessor position

Several studies of file prediction have hypothesized that there is considerable correlation between file predecessors and their corresponding successors [11, 15, 4, 6, 7, 3, 8]. If the file reference sequence "ABC" occurred in the recent past, then the reappearance of the sequence "AB" in the present could lead us to expect that a reference to file "C" would follow. We tested this heuristic against all of the file system traces to derive the relationship between *success-per-prediction* and the position of the first predecessor in the history that matches the predecessor of the current reference (if such a match is found). To achieve this, we kept a 20-element successor list and a 20-element predecessor list per file, maintained on a most-recent basis. This is equivalent to a most recent context model of depth equal to one.

The results are shown in Figure 2. Position 1 on the graph represents the most recent element in the list. The graph suggests that use of a *predecessor heuristic* can yield prediction accuracies between 55 percent and 90 percent. There is no apparent relationship between the position of the predecessor in the history and the prediction accuracy, although the accuracy is improved for positions 1 and 2 (most recent) for all traces except *Web*.

4.3. Pre-predecessor position

The predecessor position heuristic was extended to include one additional level of depth. If the file reference sequence "ABCD" occurred in the recent past, then the

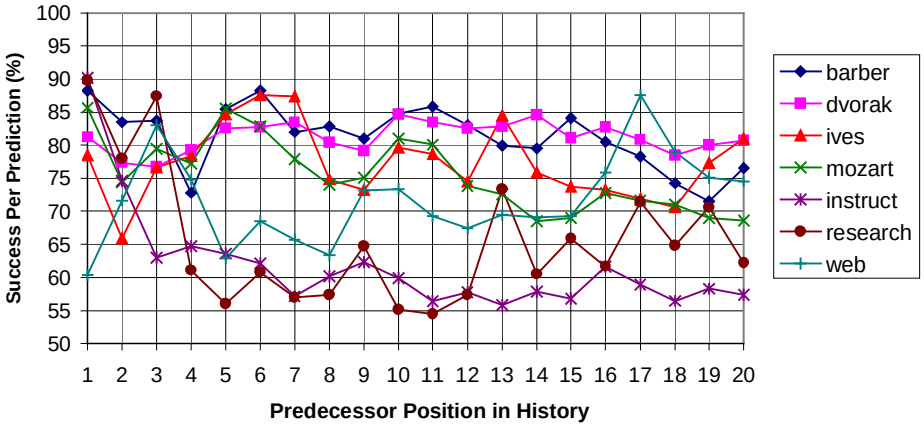

Figure 2: Performance of Predecessor Position Heuristic

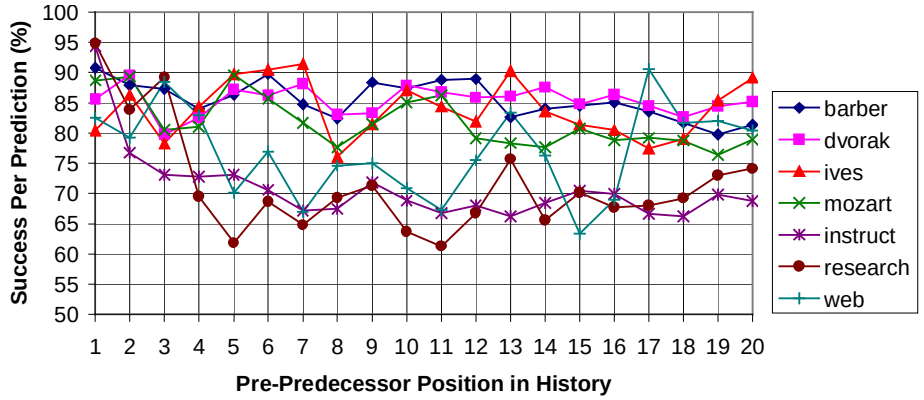

Figure 3: Performance of Pre-Predecessor Position Heuristic

reappearance of the sequence "ABC" in the present could lead us to expect that a reference to file "D" would follow. We tested this heuristic against all of the file system traces to derive the relationship between *success-per-prediction* and the position of the first pre-predecessor and predecessor in the history that matches the pre-predecessor and predecessor of the current reference (if such a match is found). To achieve this, we kept a 20-element successor list, a 20-element predecessor list and a 20 element pre-predecessor list per file, maintained on a most-recent basis. This is equivalent to a most recent context model of depth equal to two.

The results are shown in Figure 3. Position 1 on the graph represents the most recent element in the list. The graph suggests that use of a *pre-predecessor heuristic* can yield prediction accuracies between 65 percent and 95 percent. As we found for the predecessor position heuristic, there is no apparent relationship between the position of the pre-predecessor in the history and the prediction accuracy, although there is a marked improvement in prediction accuracy for positions 1 through 3 (most recent).

4.4. *j*-out-of-*k* ratio for most frequent successor

The last heuristic that was analyzed relates to the dependence of prediction accuracy on the distribution of the most frequent successor in the successor history. This heuristic can be parameterized as a *j-out-of-k Ratio* heuristic, where j is the number of occurrences of the most frequent successor, and k is the successor history length. We tested this heuristic against all of the file system traces to derive the relationship between *success-per-prediction* and values of j/k. k was varied between 1 and 20. For each value of k, j was permitted to vary between 1 and $k - 1$. Cases where j was equal to the history length k were excluded, as these cases are more effectively subsumed by the most recent consecutive successors heuristic analyzed earlier. The results are shown in Figure 4.

Vertical alignment of data points on the graph is seen to occur at the more frequent quotients resulting from

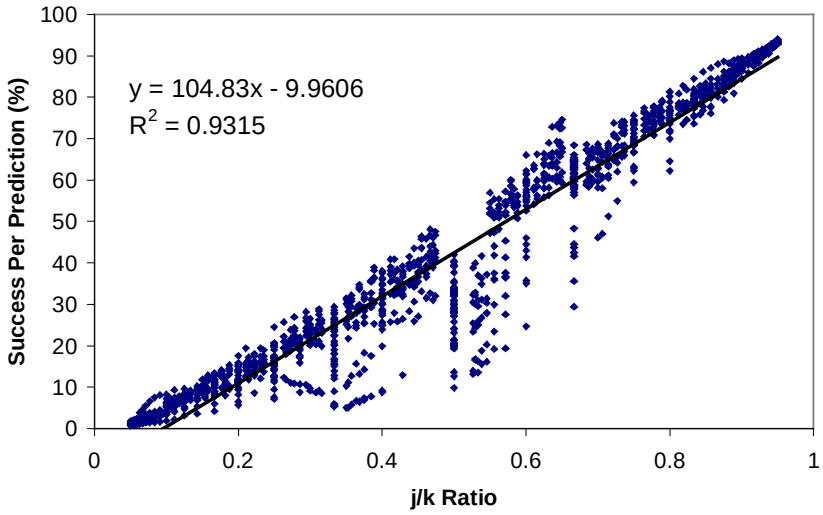

Figure 4: Performance of j/k Ratio Heuristic.

permuted integer ratios (1/6, 1/5, 1/4, 1/3, 1/2, 2/3, 3/4). The graph shows a linear regression line fit to the data points. Although there are a moderate number of outliers, the results suggest that there is a high degree of linear correlation between the j/k ratio and the success-per-prediction (regression coefficient of 0.9315). Variation from the linear trend is seen to increase towards the center of the distribution, around a j/k ratio of 0.5. We hypothesize that this is a natural departure of the data from our underlying linear model, towards the point of maximum uncertainty in the outcome.

5. Our Composite Predictor

Results of the previous sections allowed us to make two hypotheses:

1. Each heuristic, when applied to any given successor history, may give rise to a different success-per-prediction measurement. The heuristics are assumed to be largely independent.
2. Success-per-prediction measurements can be directly related to the *probability* of a successful prediction.

If we assume the validity of these hypotheses, we should be able to devise a composite predictor using the outputs of these heuristics to produce more accurate predictions of the successor of the current file.

Our composite predictor uses a very simple approach:

1. It applies first each of the four analyzed heuristics to obtain independent predictions of the successor of a given file.
2. Any heuristic that can be applied to the current context will return a prediction of the next file id and a *heuristic weight* estimating the probability of success of that prediction.
3. The predictor selects the prediction that came with the highest heuristic weight and compares this weight with a fixed *probability threshold*.
4. If the weight of the prediction is greater than or equal to the threshold, our composite predictor accepts the prediction. Otherwise, it declines to make any prediction.

5.1. Use of Heuristic Weights in Estimating File Occurrence Probabilities

An earlier stated hypothesis was that success-per-prediction measurements could be directly related to the *probability* of a successful prediction. Analysis performed in Section 4 related success-per-prediction, and hence occurrence probability, to variation in the parameters controlling each of four heuristics. From this, we can obtain from the use of each heuristic, not only a predicted file, but also an estimate of its probability of occurrence.

The analysis performed in Section 4 utilized data from all seven traces. When obtaining *heuristic weights* from this data, that relate variation in heuristic parameters to file reference probability, we must eliminate the possibility of *a-priori* information from distorting the objectiveness of our results. To this end, we estimated the heuristic weights using multiple methods prior to running simulations to measure the effectiveness of our composite predictor. These methods include:

1. *Cross-Training*: Only mean heuristic weights from the CMU file system traces were used in simulations run on the Berkeley traces. Simulations for the CMU file system traces only used mean weights derived from the Berkeley traces.

2. *Mean-Weights:* The same set of mean heuristic weights from all seven traces was used in simulations for all traces.
3. *Adjusted Mean-Weights:* The mean heuristic weights from all seven traces were adjusted by plus or minus 10 percent and 20 percent. The resulting heuristic weights were used in simulations for all traces.
4. For the *j-out-of-k ratio* heuristic, a simple ratio of *j/k* was used to estimate the prediction probability for all simulations.

5.2. Selecting a probability threshold

The *probability threshold* utilized in our composite predictor is used to improve its overall performance (as measured by effective-miss-ratio) in situations where there are penalties for missed predictions. When there is no penalty for a missed prediction (the case when $\alpha = 0.0$), the algorithm should always predict, so the threshold is set to zero. When the penalty for a missed prediction is equal to one successful prediction (the case when $\alpha = 1.0$), it is necessary to set the probability threshold to 0.5 in order to exclude predictions that would otherwise reduce the overall effective-success-per-reference. In a similar manner, when α is equal to 0.5, the probability threshold is set to one third.

5.3. Experimental results

We simulated the execution of our composite predictor using the seven file system traces, for α values of 0.0, 0.5 and 1.0. Simulations were run using heuristic weights computed using the *cross-training*, *mean-weights* and *adjusted-mean-weights* methods described previously. All simulations used a successor history length of twenty file identifiers.

As shown in Figure 5, performance of our composite predictor is quite insensitive to the heuristic weight computation method, even when the weights are derived from other file system traces, or when they are adjusted up or down by as much as 20%. The weights derived from a mean of the weights of all of the file system traces gave the best performance in almost all cases, so these will be used in subsequent simulations.

5.4. Use of a Confidence Measure to Improve Performance

Regardless of the file prediction scheme used, it is unlikely that it will be able to predict a successor for all files at all times. For this reason, we added to our predictor a *confidence measure*, whose purpose is to restrict its use to the files that are most benefited by it.

The *confidence measure* is a simple 0.0 to 1.0 saturating counter that is maintained for each file. It is initialized to a value of 0.5. It is incremented by 0.1 on a

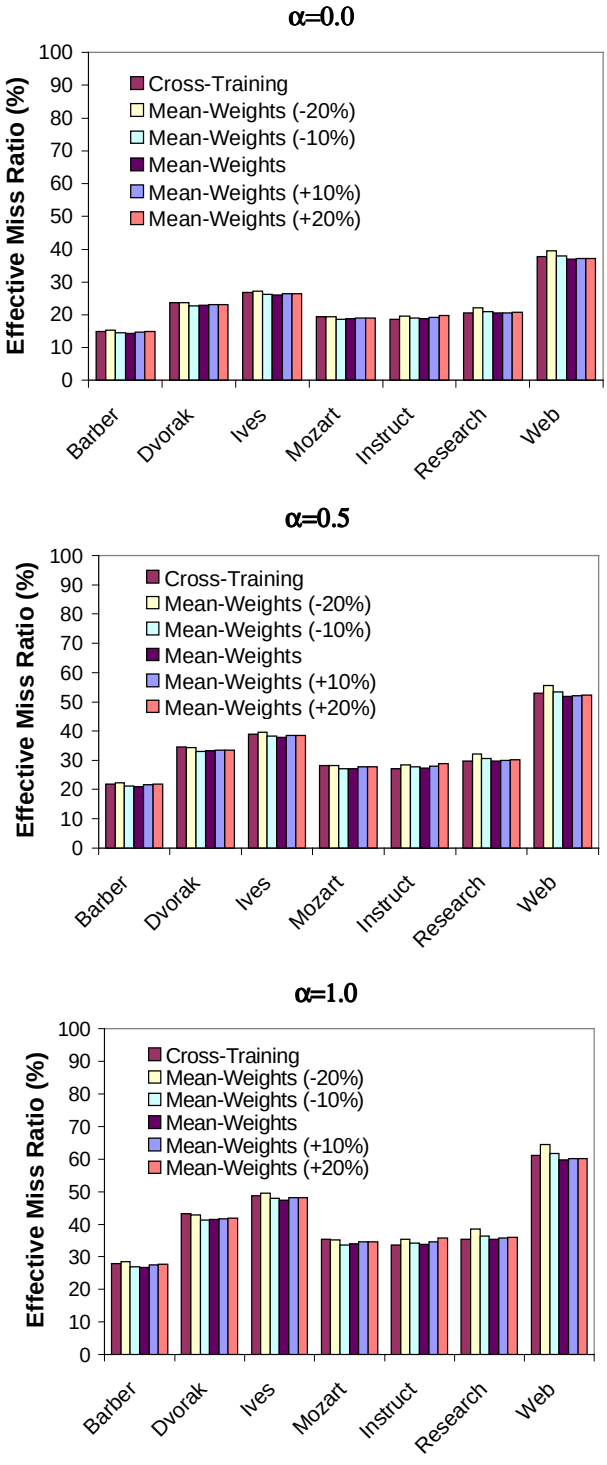

Figure 5: Effect of Heuristic Weight Computation Method on Performance.

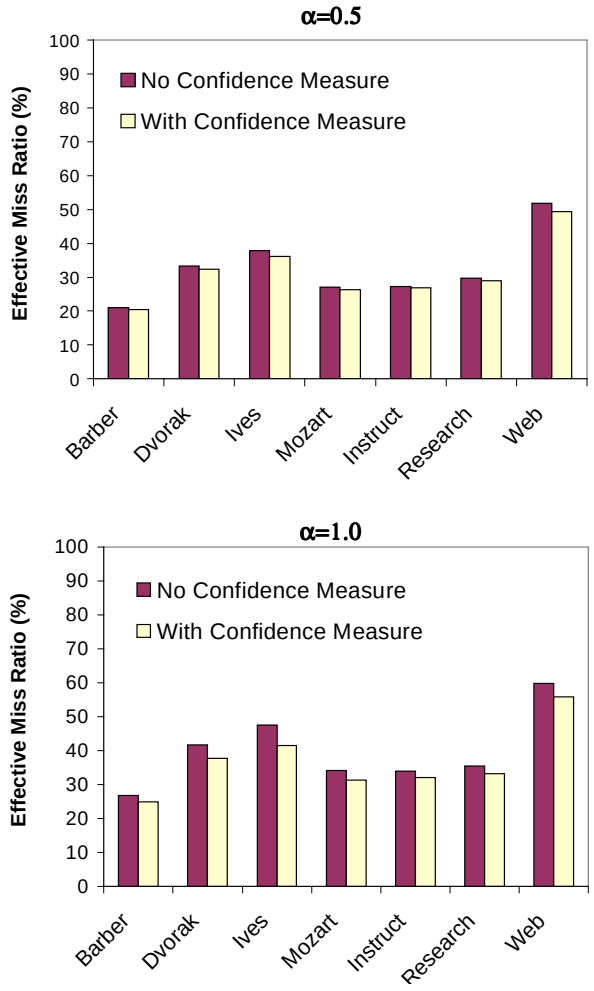

Figure 6: Effect of Use of Confidence Measure on Performance.

successful prediction and is decremented by 0.05 on an incorrect prediction. It is always updated, even if a prediction was not used because the prediction probability failed to exceed the probability threshold. For α values greater than 0.0, the prediction is rejected whenever the current value of the *confidence measure* is less than 0.5.

Figure 6 shows the performance of our composite predictor for α values of 0.5 and 1.0 using the mean heuristic weights and a successor history length of 20 file identifiers, without and with the use of a confidence measure. The confidence measure has no impact when α is zero, as there is no penalty for a missed prediction.

Use of the confidence measure improved performance of the predictor in all cases when α was greater then zero. For the case when α was 0.5, the average improvement in effective-miss ratio was 1.13 percent. For the case when α was 1.0, this increased to 3.23 percent.

5.5. Effect of Successor History Length on Performance of the Composite Predictor

Figure 7 shows the effect of successor history length on the performance of our composite predictor for all seven file system traces and α values of 0.0, 0.5 and 1.0.

The composite predictor used mean heuristic weights and incorporated a confidence measure. As we can see, using a successor history length of nine file identifiers will result in *effective-miss-ratio* measures for all cases that are within one percent of the values obtained with a successor history length of 20 file identifiers. Considering the savings in time and space complexity resulting from using a shorter successor history length, with minimal reduction in prediction performance, we decided to use nine file identifiers in the successor history for subsequent simulation.

5.6. Effect of Heuristic Combinations on Performance of the Composite Predictor

Figure 8 shows the effect of heuristic combinations on the performance of our composite predictor for all seven file system traces and α values of 0.0, 0.5 and 1.0. The letters **CS** denote the *consecutive successors* heuristic, **PR** the *predecessor position* heuristic, **PP** the *pre-predecessor position* heuristic and **JK** the *j-out-of-k ratio* heuristic.

A surprising result of this analysis was that the heuristic combination CS-PP-JK gave results that were marginally better in most of the cases than using all four heuristics, CS-PR-PP-JK. Our current thinking is that the PR (*predecessor position*) heuristic is probably assigned heuristic weights that are slightly higher than they should be. We believe that this is causing it to control predictions that would be better left to one of the other heuristics. More work is still needed in this area.

5.7. Performance of the Composite Predictor When Compared to Others Tested

Figure 9 compares the performance of our composite predictor to that of the *Last Successor*, *Stable Successor* and *Recent Popularity* (*j-out-of-k*) predictors for all seven file traces and α values of 0.0, 0.5 and 1.0. Our composite predictor used all four heuristics, mean heuristic weights, a successor history length of 9 file identifiers, and incorporated a confidence measure.

Note that the results for the *Recent Popularity* predictor represent the best that could be achieved by manual selection of the optimal parameters. This could explain why *Recent Popularity* always outperforms *Last Successor* and outperforms *Stable Successor* in 20 of the 21 experiments.

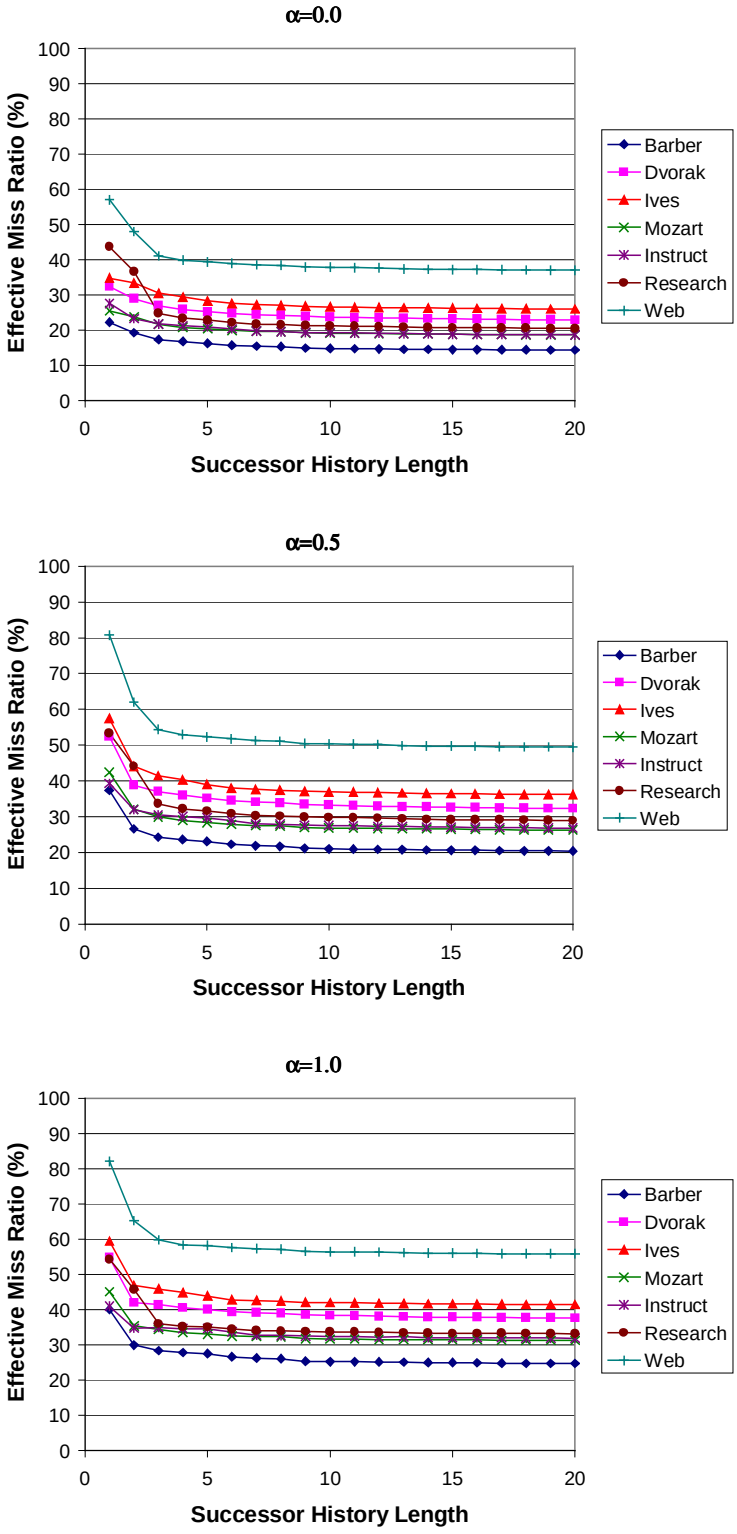

Figure 7: Effect of Successor History Length on Performance of the Composite Predictor.

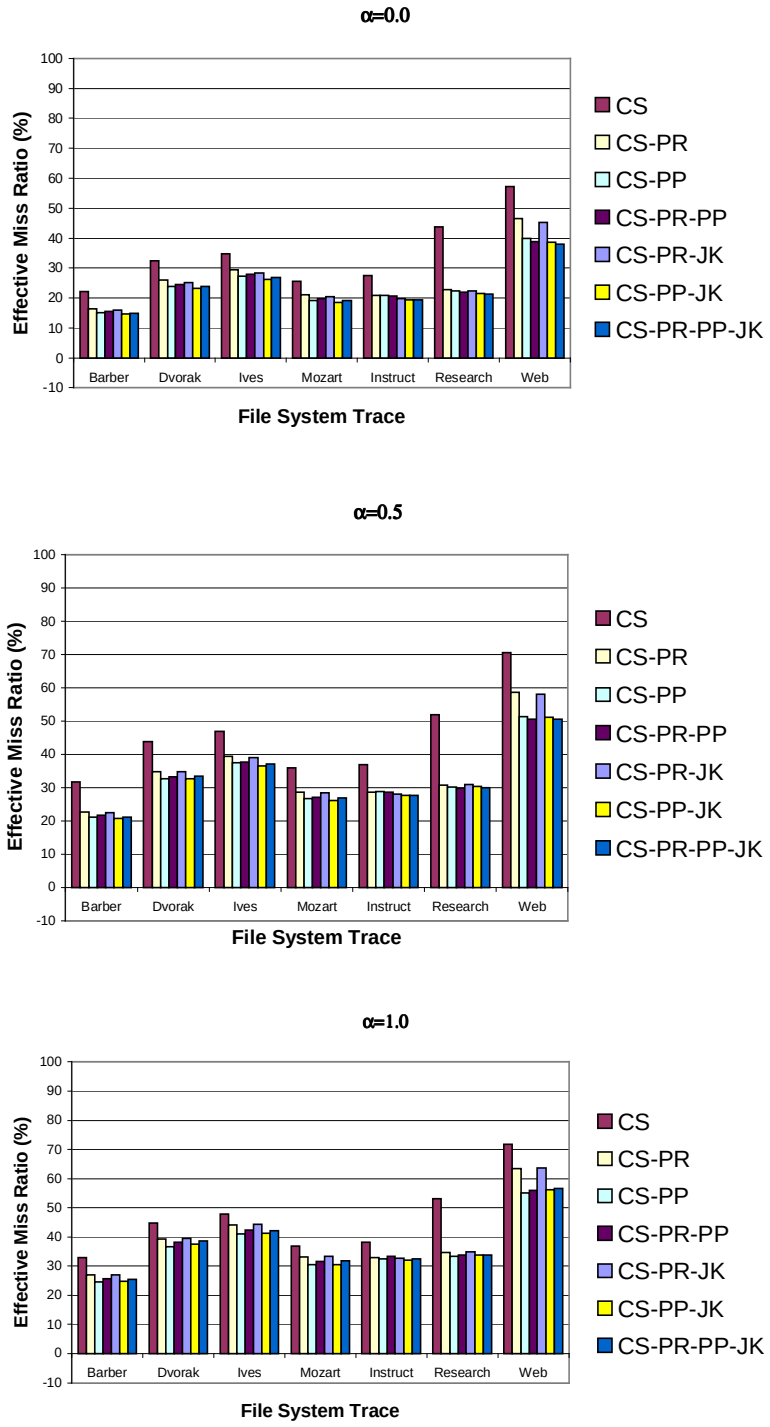

Figure 8: Effect of Heuristic Combinations on Performance of our Composite Predictor.

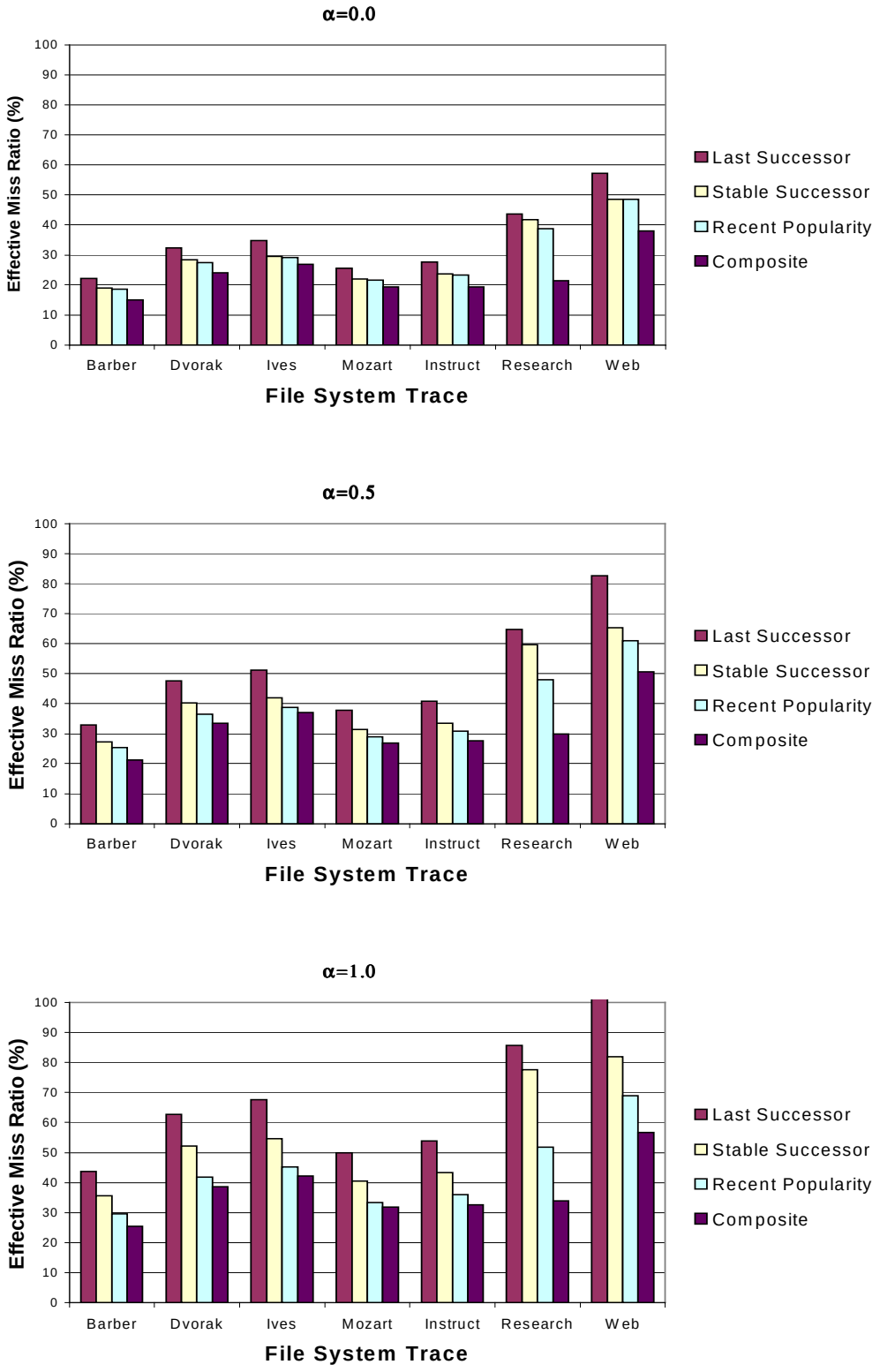

Figure 9: Comparing the Performance of our Composite Predictor to those of the Last Successor, Stable Successor and Recent Popularity Predictors.

As we can see, our hybrid predictor always performs better than the three other predictors: Across all traces and for all values of α, when compared to the best of the three other predictors examined, the minimum reduction in *effective-miss-rate* was 5.09%, the maximum was 44.79% and the average reduction was 13.7%. This is a very satisfactory result as our hybrid protocol operates on the same successor history as the *Stable Successor* and the *Recent Popularity* predictors and, unlike *Recent Popularity*, does not require any manual tuning.

Comparing the performance of our hybrid predictor with that of the *Last Successor* predictor is somewhat more difficult because *Last Successor* requires a much shorter successor history than our hybrid protocol and has thus a lower overhead. We will only note that this lower overhead comes at a significant cost as our hybrid protocol always outperforms *Last Successor* by at least 22.96%.

6. Conclusions

Existing file access predictors keep track of previous file access patterns and rely on a single heuristic to predict which of the previous successors to the file being currently accessed is the most likely to be accessed next. We have presented a composite predictor that applies multiple heuristics to this selection problem. As a result, it can make use of specialized heuristics that make very accurate predictions when some specific access pattern is present.

We simulated the execution of our predictor on seven file access traces and found that our predictor delivered more correct predictions and less inaccurate guesses than *Last Successor*, *Stable Successor* and *Recent Popularity* (*j-out-of-k*). This is a very satisfactory result as our hybrid protocol operates on the same successor history as the *Stable Successor* and the *Recent Popularity* predictors and, unlike *Recent Popularity*, does not require any manual tuning.

We are now investigating the use of more sophisticated methods for selecting the best prediction from the outcomes of our heuristics. We are also planning to use our predictor to define stable clusters of files that could then be co-located on the storage device, allowing for the reduction of data access latencies.

References

[1] A. Amer and D. D. E. Long, Noah: Low-cost file access prediction through pairs, in *Proc. 20th International Performance, Computing, and Communications Conference* pp. 27–33, April 2001.

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

[3] I. C. K. Chen, J. T. Coffey, and T. N. Mudge, Analysis of branch prediction via data compression, in *Proc. International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 128–137, Oct. 1996.

[4] K. M. Curewitz, P. Krishnan, and J. S. Vitter, Practical prefetching via data compression, in *Proc. 1993 ACM SIGMOD Conference on Management of Data*, pp. 257–266, May 1993.

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

[6] P. Krishnan, Online prediction algorithms for databases and operating systems, PhD Thesis, Dept. of Computer Science, Brown University, 1995.

[7] T. M. Kroeger and D. D. E. Long, The case for efficient file access pattern modeling, in *Proc. 1996 USENIX Technical Conference*, pp. 14–19, Jan. 1996.

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

[9] H. Lei and D. Duchamp, An analytical approach to file prefetching, in *Proc. 1997 USENIX Annual Technical Conference*, Jan. 1997.

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

[11] M. L. Palmer and S. B. Zdonik, FIDO: a cache that learns to fetch, in *Proc. 17th International Conference on Very Large Data Bases*, pp. 255–264, Sept. 1991.

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

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

[14] C. Tait and D. Duchamp, Detection and exploitation of file working sets, in *Proc. 11th International Conference on Distributed Computing Systems*, pp. 2–9, May 1991.

[15] J. S. Vitter and P. Krishnan, Optimal prefetching via data compression, in *Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science*, pp. 121–130, Oct. 1991.

[16] G. A. S. Whittle, A hybrid scheme for file system Reference prediction. MS Thesis, Dept. of Computer Science, University of Houston, Texas, May 2002.

[17] T. Yeh, D. Long, and S. Brandt, Performing file prediction with a program-based successor model, in *Proc. 9th International Symposium on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems*, pp. 193–202. Aug. 2001.