Title

Identifying Stable File Access Patterns

Authors

Shah, Purvi

Paris, Jehan-Francois

Amer, Ahmed

et al.

Publication Date

2004-04-14

Identifying Stable File Access Patterns

Purvi Shah
University of Houston
purvi@cs.uh.edu

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

Ahmed Amer[2]
University of Pittsburgh
amer@cs.pitt.edu

Darrell D. E. Long[3]
U. C. Santa Cruz
darrell@cs.ucsc.edu

1. Introduction

Disk access times have not kept pace with the evolution of disk capacities, CPU speeds and main memory sizes. They have only improved by a factor of 3 to 4 in the last 25 years whereas other system components have almost doubled their performance every other year. As a result, disk latency has an increasingly negative impact on the overall performance of many computer applications.

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. Both techniques are widely implemented at the data block level. More recent work has focused on caching and prefetching entire files.

There are two ways to implement file prefetching. *Predictive prefetching* attempts to predict which files are likely to be accessed next in order to read them before they are needed. While being conceptually simple, the approach has two important shortcomings. First, the prefetching workload will get in the way of the regular disk workload. Second, it is difficult to predict file accesses sufficiently ahead of time to ensure that the predicted files can be brought into main memory before they are needed.

A more promising alternative is to group together on the disk drive files that are often accessed at the same time [3]. This technique is known as *implicit prefetching* and suffers none of the shortcomings of predictive prefetching because each cluster of files can now be brought into main memory in a single I/O operation. The sole drawback of this new approach is the need to identify stable file access patterns in order to build long-lived clusters of related files.

We present here a new file predictor that identifies stable access patterns and can predict between 50 and 70 percent of next file accesses over a period of one year. Our *First Stable Successor* keeps track of the successor of each individual file. Once it has detected *m* successive accesses to file *Y*, each immediately following an access to file *X*, it predicts that file *Y* will always be the successor of file *X* and never alters this prediction.

The remainder of this paper is organized as follows. Section 2 reviews previous work on file access prediction. Section 3 introduces our *First Stable Successor* predictor and Section 4 discusses its performance. Finally, Section 5 states our conclusions

2. Previous Work

Palmer *et al*. [8] 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 [4]. Shriver *et al*. [10] proposed an analytical performance model to study the effects of prefetching for file system reads.

Tait and Duchamp [11] 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 [6] 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 [5].

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. 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*.

The *Recent Popularity* or *k-out-of-n* predictor maintains the *n* most recently observed successors of each file. When attempting to make a prediction for a given file, Recent Popularity searches for the most

[1] Supported in part by the National Science Foundation under grant CCR-9988390.
[2] Supported in part by the National Science Foundation under grant ANI-0325353.
[3] Supported in part by the National Science Foundation under grant CCR-0204358.

popular successor from the list. 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 tiebreaker.

3. The First Stable Successor Predictor

All the predictors are dynamic in the sense that they reflect changes in file access patterns and modify accordingly their predictions. The sole existing static predictor is *First Successor* [1], which always predicts the first encountered successor of file *X* as its successor. It is a rather crude predictor and was found to perform much worse than all Last Successor, Stable Successor or Recent Popularity.

There are two explanations for this poor performance. First, First Successor cannot reflect changes in file access patterns. Second, it bases all its predictions on a single observation.

As shown on Figure 1, the *First Stable Successor* (FSS) predictor remedies this second limitation by requiring m successive instances of file *Y* immediately following instances of file *X* before predicting that file *Y* is the successor of file *X*. Otherwise it makes no prediction. When $m = 1$, the FSS predictor becomes identical to the First Successor protocol and predicts that that file *Y* is the successor of file *X* once it has encountered a single access to file *Y* immediately following an access to file *X*.

A large value of m will result into fewer predictions than a smaller value of m but will also increase the likelihood that these predictions will be correct. This provides us with a relatively easy way to tune the protocol by either increasing m whenever we want to reduce the number of false predictions or decreasing it whenever we want to increase the total number of predictions.

4. Performance Evaluation

When comparing the effectiveness of file predictors, one is often confronted with two primary metrics, *success-per-reference* and *success-per-prediction*. Given the dependent nature of these metrics, it is impossible 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 the extent that it outweighs any improvements due to predictive prefetching.

We will use 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

Assumptions:

G is file being currently accessed
F its direct predecessor
FirstStableSuccessor(*F*) is last prediction made for the successor of *F*
LastSuccessor(*F*) is last observed successor of *F*
Count(*F*) is a counter
m is minimum number of consecutive identical successors to declare a First Stable Sucessor

Algorithm:

```
if FirstStableSuccessor(F) is undefined then
        if LastSuccessor(F) = G then
                Counter(F) ← Counter(F) + 1
        else
                Counter(F) ← 1
        end if
        if Counter(F) = m then
                FirstStableSuccessor(F) ←G
        end if
end if
```

Figure 1 The First Stable Successor Predictor

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 success rate per reference* of a predictor as the ratio:

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

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 success rate per reference* 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.

We evaluated the performance of our FSS 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 [7]. 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,

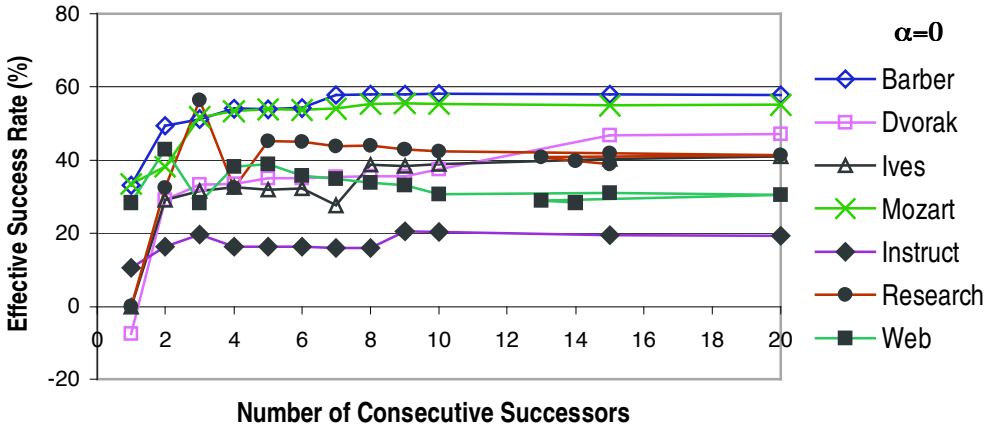

Figure 2. Effective success rate per reference of the FSS predictor for α = 0 and *m* varying between 1 and 20.

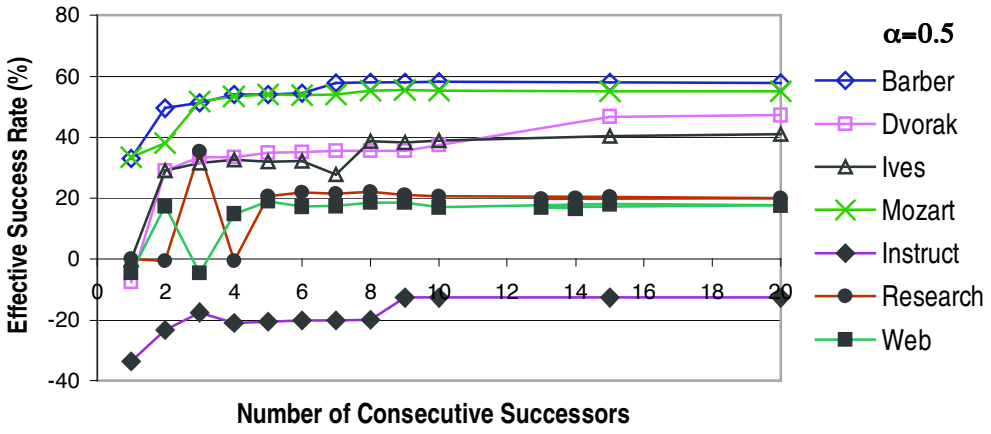

Figure 3. Effective success rate per reference of the FSS predictor for α = 0.5 and *m* varying between 1 and 20.

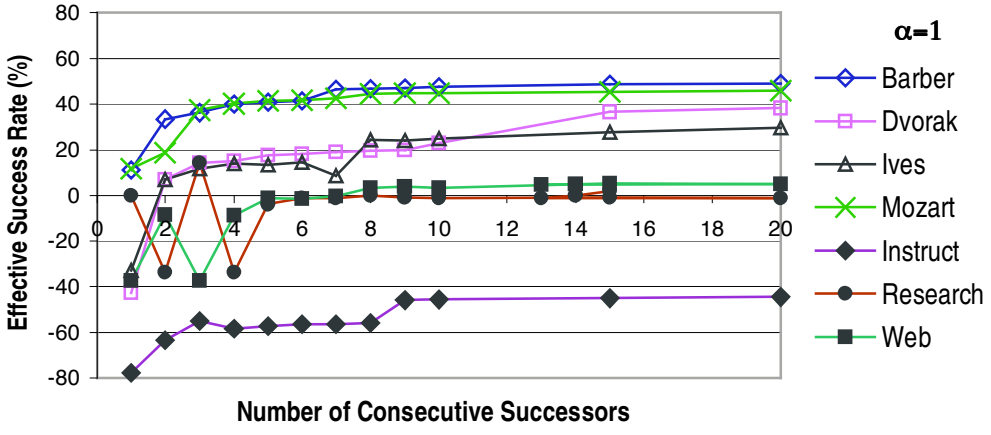

Figure 4. Effective success rate per reference of the FSS predictor for α = 1 and *m* varying between 1 and 20.

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 span of approximately one year. Our second set of traces was collected in 1997 by Roselli [9] 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*).

Figures 2 to 4 represent the effective success rates per reference achieved by our First Stable Successor when the number m of consecutive successors triggering the predictor varies between 1 and 20. Negative success rates correspond to situations where $\alpha > 0$ and the sum of the penalties assessed for incorrect predictions exceeds the number of correct predictions.

As we can see, our First Stable Successor performs much better with the four CMU traces than with the three Berkeley traces even though the Berkeley traces were collected over a much shorter period. In particular, our predictor performs very poorly with the *instruct* trace, which appears to have the least stable reference patterns of all seven traces.

The four CMU traces can be further subdivided into two groups. The first group comprises *barber* and *mozart*, which exhibit rather stable behaviors. As a result, our predictor can successfully predict between 66 and 69 percent of future references. Conversely, *dvorak* and *ives* exhibit less stable behaviors and our predictor can successfully predict between 53 and 57 percent of future references. This should not surprise us because *ives* had the largest number of users and *dvorak* the largest proportion of write activity. Even when we do not penalize incorrect predictions, First Stable Successor requires less consecutive successors to reach their optimum performance on *barber* and *mozart* than on *dvorak* and *ives*.

We can also observe that the number of consecutive successors required to achieve optimum performance increases on all seven traces when α increases from zero to one. It might be therefore indicated to increase the value of the m parameter for workloads that exhibit less stable file access patterns in order to reduce the number of misses.

Figures 5 to 7 compare the effective success rates per reference achieved by our First Stable Successor with $m = 8$ with those achieved by First Successor, Last Successor, Stable Successor with $m = 2$, and k-out-of-m. As we can see, our First Stable Successor predictor performs much better than First Successor but not as well as Last Successor, Stable Successor and k-out-of-m. This gap is especially evident for the *instruct* trace as these last three predictors perform almost as well as with the *mozart* trace while First Successor and First Stable Successor perform very poorly.

We can draw two major conclusions from our measurements. First, there are enough stable access patterns in the six of the seven traces we analyzed to make implicit file prefetching a worthwhile proposition. This is especially true because of the low overhead of the approach, which means that wrong predictions would only incur a minimal penalty ($\alpha \ll 1$). Second, many, if not most, of these stable access patterns are long lived and appear to persist over at least a full year. A file system implementing implicit file prefetching would probably reevaluate its file groups once a week. We can already predict that these weekly group reevaluations will not result in a complete reconfiguration of the whole file system.

5. Conclusions

Identifying and exploiting stable file access patterns is essential to the success of implicit file prefetching as this technique builds long-lived clusters of related files that can be brought into memory in a single I/O operation.

We have presented a new file access predictor that was specifically tailored to identify such stable file access patterns. Trace-driven simulation results indicate that our First Stable Successor can predict up to 70 percent of next file accesses over a period of one year.

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.,* 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,* pp. 525–534, July 2002.

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

[5] 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.

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

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

[8] 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,* pp. 255–264, Sept. 1991.

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

[10] 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.

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

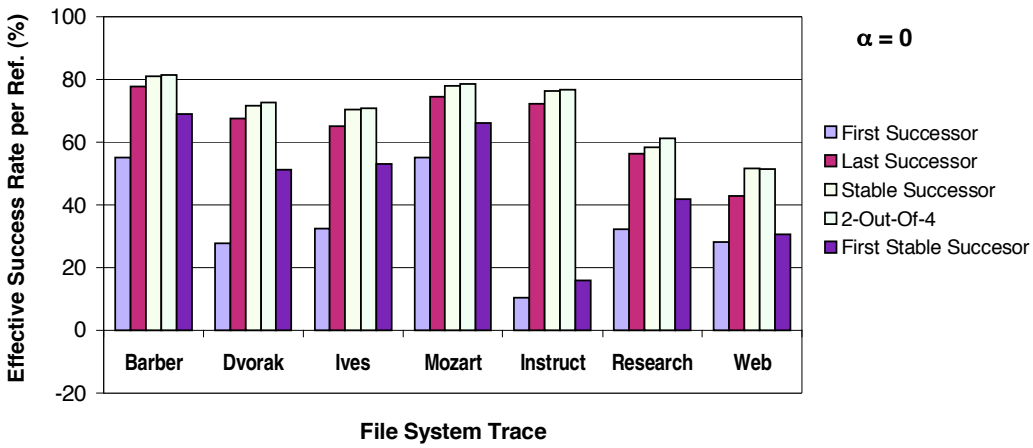

Figure 5. Compared success rates per reference of the five policies for $\alpha = 0$.

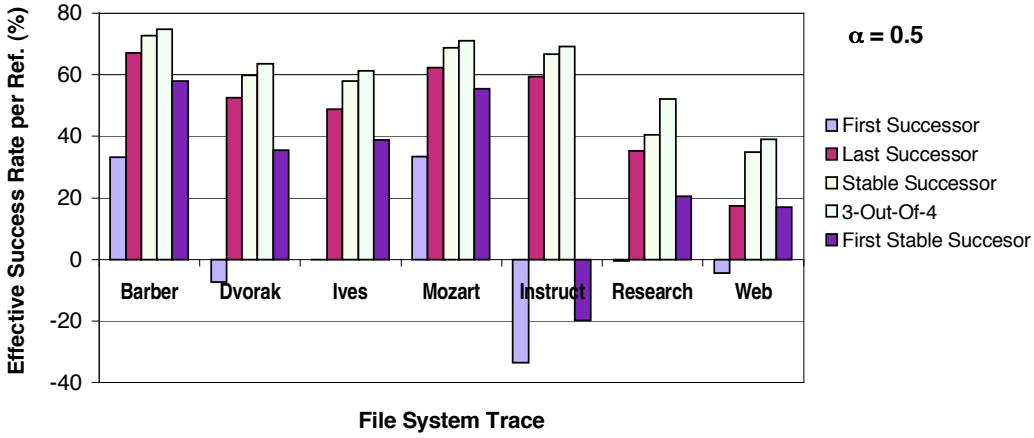

Figure 6. Compared success rates per reference of the five policies for $\alpha = 0.5$.

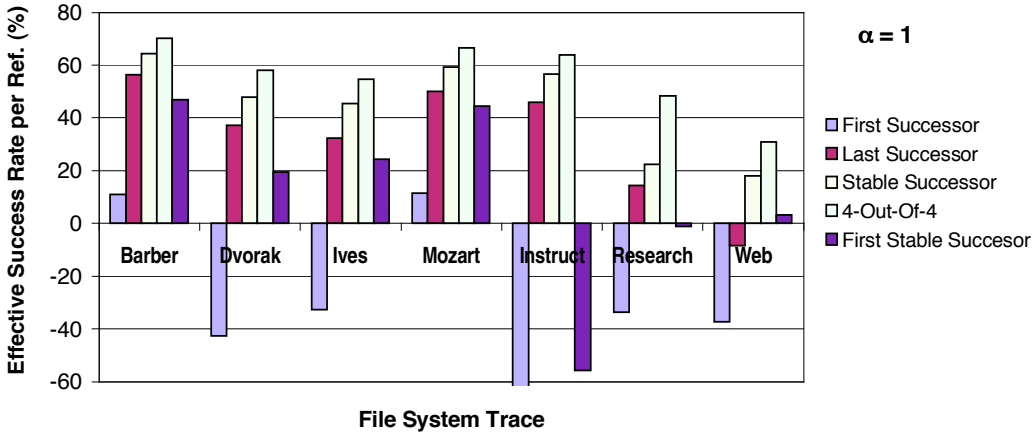

Figure 7. Compared success rates per reference of the five policies for $\alpha = 1$.