Caching Files with a Program-based Last n Successors Model

Tsozen Yeh, Darrell D. E. Long, and Scott A. Brandt
Computer Science Department
Jack Baskin School of Engineering
University of California, Santa Cruz

Abstract

Recent increases in *CPU* performance have outpaced increases in hard drive performance. As a result, disk operations have become more expensive in terms of CPU cycles spent waiting for disk operations to complete. File prediction can mitigate this problem by prefetching files into cache before they are accessed. However, incorrect prediction is to a certain degree both unavoidable and costly. We present the *Program-based Last N Successors* (*PLNS*) file prediction model that identifies relationships between files through the names of the programs accessing them. Our simulation results show that PLNS makes at least 21.11% fewer incorrect predictions and roughly the same number of correct predictions as the last-successor model. We also examine the cache hit ratio of applying PLNS to the *Least Recently Used* (*LRU*) caching algorithm and show that a cache using PLNS and LRU together can perform as well as a cache up to 40 times larger using LRU alone.

1 Introduction

Running programs stall if the data they need is not in memory. As the speed of *CPU* increases, disk I/O becomes more expensive in terms of CPU cycles. File prefetching is a technique that mitigates the speed difference originating from the mechanical operation of disk and the electronic operation of CPU [15] by preloading files into memory before they are needed. The success of file prefetching depends on file prediction accuracy – how accurately an operating system can predict which files to load into memory. Probability and history of file access have been widely used to perform file prediction [3, 4, 7, 8, 10, 12], as have hints or help from programs and compilers [2, 13].

While correct file prediction is useful, incorrect prediction is to a certain degree both unavoidable and costly. An incorrect prediction is worse than no prediction at all. Not only does an incorrectly prefetched file do nothing to reduce the stall time of any program, it also wastes valuable cache space and disk bandwidth. Incorrect prediction can also prolong the time required to bring needed data into the cache if a cache miss occurs while the incorrectly predicted data is being transferred from the disk. Incorrect predictions can lower the overall performance of the system regardless of the accuracy of correct prediction. Consequently reducing incorrect prediction plays a very important role in cache management.

In previous work we examined a file prediction model called *Program-based Last Successor* (*PLS*) [17, 18], inspired by the observation that probability and repeated history of file accesses do not occur for no reason. In particular, programs access more or less the same set of files in roughly the same order every time they execute. PLS uses knowledge about program-specific access patterns to generate more accurate file predictions. In that work, PLS was shown to outperform both the *Last Successor* (*LS*) and *Finite Multi-Order Context* (*FMOC*) models [7].

In this paper we present an extension to PLS called *Program-based Last N Successors* (*PLNS*). PLNS uses additional knowledge about program-based file accesses to determine multiple program-specific last-successors for each file to generate more accurate file predictions. Our results demonstrate that that in general, with only a little more storage and prefetching overhead, PLNS generates more accurate file predictions than either LS or PLS. In particular, compared with LS, PLNS reduces the number of incorrect file predictions and increases the number of correct predictions to provide better overall file prediction and therefore better overall system performance. The "N" in PLNS represents the number of the most recent different program-based last successors that PLNS could predict each time.

We compare PLNS with Last-Successor (LS) for different values of "N" in PLNS. Generally speaking, LS has a high predictive accuracy – our simulation results show that LS can correctly predict the next file to be accessed about 80% of the time in some cases. Our experiments demonstrate that with traces covering as long as 6 months PLNS makes at least 21.11% fewer incorrect predictions than LS, giving PLNS a higher predictive accuracy than

LS. We also examine the cache hit ratios of Least Recently Used (LRU) with no file prediction, and LRU with PLNS. We observe that PLNS always increases the cache hit ratio and in the best case, LRU and PLNS together perform as well as a cache 40 times larger using LRU alone.

2 Related Work

Griffioen and Appleton use probability graphs to predict future file accesses [4]. The graph tracks file accesses observed within a certain window after the current access. For each file access, the probability of its different followers observed within the window is used to make prefetching decision.

Lei and Duchamp use pattern trees to record past execution activities of each program [10]. They maintain different pattern trees for each different accessing pattern observed. A program could require multiple pattern trees to store similar patterns of file accesses in its previous execution.

Vitter, Curewite, and Krishnan adopt the technique of data compression to predict next required page [3, 16]. Their observation is that data compressors assign a smaller code to the next character with a higher predicted probability. Consequently a good data compressing algorithm should also be good at predicting next page more accurately.

Kroeger and Long predict next file based on probability of files in contexts of FMOC [7]. Their research also adopts the idea of data compression like Vitter *et al.* [16], but they apply it to predicting next file instead of next page.

Patterson *et al.* develop *TIP* to do prediction using hints provided from modified compilers [13]. Accordingly, resources can be managed and allocated more efficiently. Chang and Gibson design a tool which can transform UNIX application binaries to perform speculative execution and issues hints [2]. Their algorithm can eliminate the issue of language independence, but it can only be applied to single-thread applications.

Generally speaking probability-based predicting algorithms respond to changes of reference pattern more dynamically than those relying on help from compilers and applications. However over a longer period of time, accumulated probability may not closely reflect the latest accessing pattern and even may mislead predicting algorithms sometimes.

History-based techniques have also been investigated for use in memory systems. Joseph and Grunwald investigated address correlation for cache prefetching [5]. Lai and Falsafi studied per-processor memory sharing correlation in shared-memory multiprocessors [9].

3 LS and PLNS Models

We start with a brief discussion of LS model, followed by details of how to implement PLNS model.

3.1 LS

Given an access to a particular file A, LS predicts that the next file accessed will be the same one that followed the last access to file A. Thus if an access to file B followed the last access to file A, LS predicts that an access to file B will follow this access to file A. This can be implemented by storing the successor information in the metadata of each file. One potential problem with this technique is that file access patterns rely on the temporal order of program execution, and scheduling the same set of programs in different orders may generate totally different file access patterns.

3.2 PLNS

Lacking *a priori* knowledge of file access patterns, many file prediction algorithms use statistical analysis of past file access patterns to generate predictions about future access patterns. One problem with this approach is that executing the same set of programs can produce different file access patterns even if the individual programs always access the same files in the same order. Because it is the individual programs that access files, probabilities obtained from the past file accesses of the system as a whole are ultimately unlikely to yield the highest possible predictive accuracy. In particular, probabilities obtained from a system-wide history of file accesses will not necessarily reflect the access order for any individual program or the future access patterns of the set of running programs. However what could remain unchanged is the order of files accessed by the individual programs. In particular, file reference patterns can describe what has happened more precisely if they are observed for each individual program, and better knowledge about past access patterns leads to better predictions of future access patterns.

PLNS incorporates knowledge about the running programs to generate a better last-successor estimate. More precisely, PLNS records and predicts program-specific last successors for each file that is accessed. The "N" in PLNS represents the number of the most recent different program-specific last successors that PLNS could predict each time. For example, PL1S (N = 1) means only the most recent program-specific last successor is predicted after each file access. In other words, PL1S can be viewed as a program-specific last successor model. PL2S (N = 2) predicts the most recent two different program-specific last successors if a particular program accesses

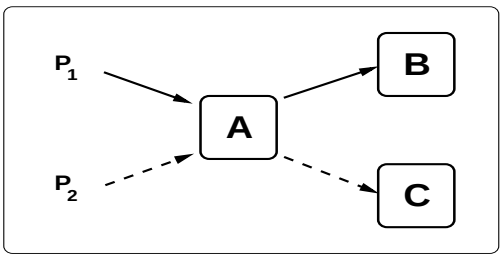

Figure 1: PL1S model

multiple different files after each access of a particular file. However PL2S could still predict only one successor if the program-specific successor for a given file has never changed.

We will use PL1S as an example to explain PLNS. Suppose a file trace at some time shows pattern AB, and pattern AC occurring 60% and 40% of the time respectively. A probability-based prediction will prefer predicting B after A is accessed. If B and C tend to alternate after A, then LS will do especially poorly. But the reason that pattern AB and AC occur may be quite different. For instance, in Figure 1, the file access pattern AB is seen to be caused by program P_1, while the file access pattern AC is caused by program P_2. In other words, what is really behind the numbers 60% and 40% is the execution of two different applications, P_1 and P_2. After we collect this information (a set of pairs consisting of "*program name*" and "*successors*") for file A, next time it is accessed we can predict either B or C depending on P_1 or P_2 is accessing A, or provide no prediction if A is accessed by another program.

Table 1: Metadata of Figure 1 kept under PL1S model

file	⟨*program name*, *successor*⟩
A	⟨P_1, B⟩, ⟨P_2, C⟩
B	⟨P_1, NIL⟩
C	⟨P_2, NIL⟩

One can argue that the same program may access different sets of files each time that it is executed, particularly a system utility program such as a compiler. While it is true that compiling different programs will result in different files being accessed, compiling the same program multiple times will result in many or all of the same files being accessed in the same order. Thus PL1S will make correct predictions for most of these files, even when alternating compilations between two sets of files. Assume, for example, that two programs need to be compiled. The first program needs files X_1, X_2, ..., X_m, in that order, and the second program needs files Y_1, Y_2, ..., Y_n, in that order. If X_1 and Y_1 are different files, then we don't know which file to predict when the compiler starts running, but as soon as either X_1 or Y_1 is accessed we know which file to prefetch next. If X_1 and Y_1 are the same, then we prefetch this file and wait to see whether X_2 or Y_2 is needed, and then we can predict the next file after that.

PLNS can also avoid the slow adaption problem in probability-based prediction models. Probability-based models always predict the same file until the corresponding probability changes. Like LS, PLNS does not rely on probability so it can respond immediately as file access patterns change.

Two issues that need to be addressed are how to collect the metadata in terms of ⟨*program name*, *successors*⟩ for each file, and how big the metadata needs to be in order to make accurate predictions. The first issue is simple. Programs are executed as processes, so we can just store the *program name* in the process control block (*PCB*). For each running program (say P), we also need to keep track of the file (say X), which it has most recently accessed. When P accesses the next file (say Y) after X, we update the metadata of the X with ⟨P, Y⟩, and the next time that P accesses X, PL1S can predict that the next file accessed will be Y.

In the example of Figure 1, when P_1 accesses the next file (say B) after its access to A, we update the metadata of A with ⟨P_1, B⟩, and next time P_1 accesses A, PL1S can predict that the next file accessed will be B. Similarly, A also keeps ⟨P_2, C⟩ as parts of its metadata. If the access to A by P_1 is ever followed by access to different files, for example D, other than B, then the name of file D will be added to the metadata of A. So the corresponding metatdata now becomes ⟨P_1, B, D⟩. The metadata of files in Figure 1 is shown in Table 1.

The second issue is not quite as simple as the first. Ideally, for each file we would like to record the name of every program that has accessed it before, along with the program-specific successors to the file, so that we know which file (or files) to predict when the same program accesses the file again. In reality, this may be too expensive for files used by many different programs. Consequently, we may need to limit the number of ⟨*program name*, *successors*⟩ pairs kept for each file. However, our simulation shows that about 99% of files are accessed by six or fewer programs and thus metadata storage is not a significant problem. In particular, by limiting metadata to at most six programs per file, we can obtain almost all of the benefit provided by PLNS.

A few terms need to be clarified here. The first is that when we use the term "*program*" we mean any running executable file. Thus a driver program that launches different sub-programs at different times is considered by PLNS to be a different program from the sub-programs, each of which is also treated independently. The second

is that both "*program name*" and "*file name*" include the entire pathname of the files. This is important because different programs with the same name can access the same file and different files with the same name can be accessed by different programs, and these accesses must all be handled correctly.

4 Experimental Results

In the section, we will discuss the trace data we used to conduct our experiments, and how we compare performance of LS and PLNS.

4.1 Simulation Trace and Experimental Methodology

In examining PLNS we used the trace data from *DFSTrace* used by the *Coda* project [6, 11]. These traces were collected from 33 machines during the period between February of 1991 and March of 1993. We used data from October 1992 to March 1993, roughly equal to the last quarter of the entire trace, from three machines *Copland*, *Holst*, and *Mozart*. Research has demonstrated that the average life of a file is very short [1]. Therefore, instead of tracking every *READ* or *WRITE* event, we track only the *OPEN* and *EXECVE* events in our simulation.

As mentioned above, PLNS needs to be able to determine the name of a program in order to generate its predictions. Because we cannot obtain the name of any program that started executing before the beginning of the trace, we exclude all *OPEN* events initiated by any *process id* (pid) which started before the beginning of our trace. Intuitively this filtering has no effect on the results of our experiments because the filtering is based only on the time at which the program began. In a real system such filtering is not necessary because all program names are known.

We are interested in how different values of "N" in PLNS could affect the performance and the costs come with it compared with LS. We used the filtered trace data to evaluate LS, PL1S, PL2S, and PL3S respectively.

Both LS and PL1S predict at most one file at a time. We score LS and PL1S by adding one for each correct prediction and zero for each incorrect prediction. We normalize the final scores of PL1S and LS by the number of predictions, not by the number of events. This is because the first time that a file is accessed there is no previous successor to predict and so the failure to make a prediction the first time cannot be considered incorrect. We also score PL2S and PL3S the same way we score LS. Of course scores of PL2S and PL3S do not sit on the same ground as those from PL1S because both PL2S and PL3S could predict more than one file at a time in some cases. A more detailed discussion of PL2S and PL3S will follow later.

4.2 Model Comparison

We begin with PL1S. Figure 2 shows that PL1S has a higher predictive accuracy than LS in all machines. One pitfall in comparing prediction models in terms of predictive accuracy is that higher predictive accuracy does not assure the success of a model because the scores are commonly normalized by the number of predictions made, which does not include those cases where no prediction was made. Consider two prediction models, A and B. If A makes 40 correct predictions, 40 incorrect predictions, and does not make a prediction 20 times out of a total of 100 file accesses, then A's predictive accuracy is 50%. Suppose B makes only 2 correct predictions, 1 incorrect prediction, and does not make a prediction 97 times. B's predictive accuracy is 67%, but model B is almost useless in practice.

Clearly, in order to examine the real performance of a prediction model, we need other information besides predictive accuracy. Thus, we use LS as the baseline to evaluate the performance of PL1S in three categories. The first category is the percentage of total predictions (including correct and incorrect predictions) made by PL1S as compared with LS. This percentage should not be to too small, otherwise PL1S may be an unrealistic model just like the model B above. The second is the percentage of correct predictions made by PL1S as compared with LS. This number should be as high as possible. The last category is the percentage of incorrect predictions made by PL1S as compared with LS.

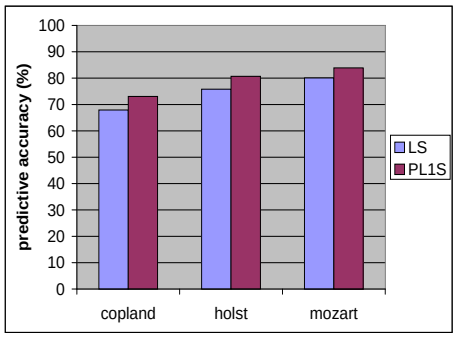

Figure 2: Predictive accuracy of LS and PL1S

4.3 Category Performance

Figure 3 displays the PL1S performance normalized by LS in the three different categories. The columns marked "*total*" show that the total number of predictions made by PL1S is about 92% to 95% of the number made by LS. This is close enough to consider PL1S

to be a practical prediction algorithm in terms of the number of predictions it makes. The middle columns marked "*correct*" are the percentages of correct predictions. PL1S makes more correct predications than LS in all three machines. Percentages from the middle columns demonstrate that PL1S can do roughly as well as LS in correctly predicting files. Finally, the columns marked "*incorrect*" show that PL1S indeed makes about 21% to 25% fewer incorrect predictions as compared with LS, which is a very exciting result. This explains why the PL1S model performs better than LS in Figure 2 .

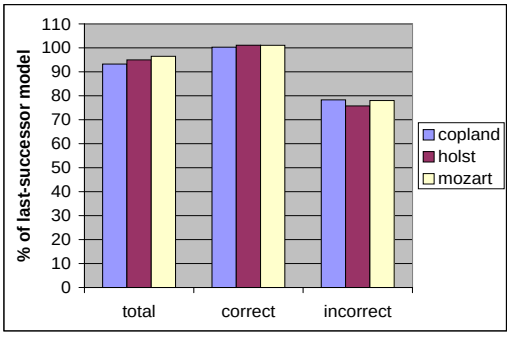

Figure 3: PL1S performance normalized by LS in 3 separate categories

The reduction of incorrect predictions in PL1S is significant enough to be worthy of further exploration. Since the number of predictions made by PL1S is only about five to eight percent less than LS, and the number of correct predictions is roughly same as LS, we conclude that PL1S makes no prediction more often than LS. We collected the percentage of cases where no prediction was made by PL1S compared with LS, and the results are displayed in Figure 4, which confirms this surmise. Figure 4 shows that the percentage of events where no prediction was made by PL1S is roughly two to three times higher than that of LS.

As mentioned earlier we are interested in the cost and benefit of PL2S and PL3S. Table 2 lists the average number of files predicted each time in PL2S and PL3S. It indicates that PL2S actually predicts fewer than two files most of the time. Figure 5 shows how PLNS can increase the percentage of correct prediction (upper part) and reduces incorrect prediction (lower part) respectively. Their predictive accuracy is displayed in Figure 6. The incorrect prediction reduced normalized by LS is shown in Figure 7. Table 2 and Figure 5 to 7 demonstrate that with an average of predicting 1.07 or 1.08 files each time, PL2S provides significant improvements over LS and PL1S. PL2S makes from 38.45% to 46.88% fewer incorrect predictions and from 6.4% to 8.3% more correct predictions as the LS. Nevertheless, PL3S does not offer an obvious improvement over PL2S in our simulation.

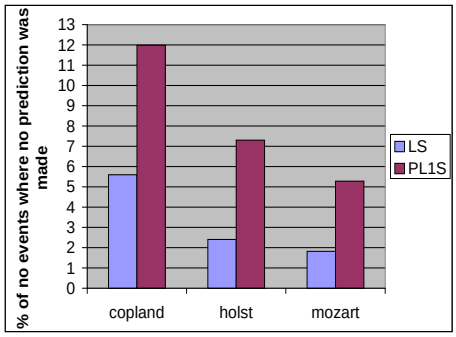

Figure 4: No predictions made by LS and PL1S

Table 2: Average number of files predicted each time in PL2S and PL3S

PLNS	PL2S	PL3S
machine: Copland	1.07	1.19
machine: Holst	1.08	1.18
machine: Mozart	1.08	1.16

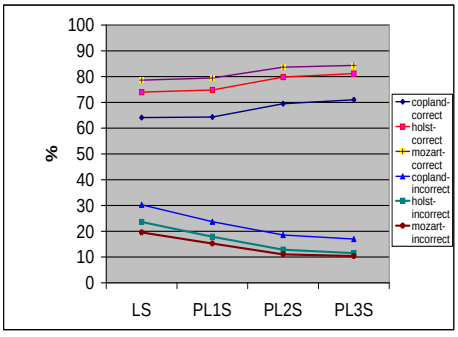

Figure 5: Correct and incorrect predictions made by LS and PLNS

We stated earlier that some events were filtered out of our trace data due to the requirement that PLNS needs to know the program initiating an event, and we claimed that the filtering does not affect the validity of our results. To verify this, we compared the percentage of events filtered out of original trace data with PL1S predictive accuracy for each machine. Our assumption was that if the filtered trace had improved the performance, the effect would be greater for larger amounts of filtered data. However, the results in Figure 8 show that the highest predictive accuracy of PL1S (from Mozart) does not come from the

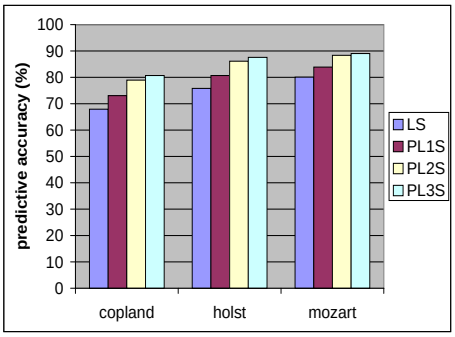

Figure 6: Predictive accuracy of LS and PLNS

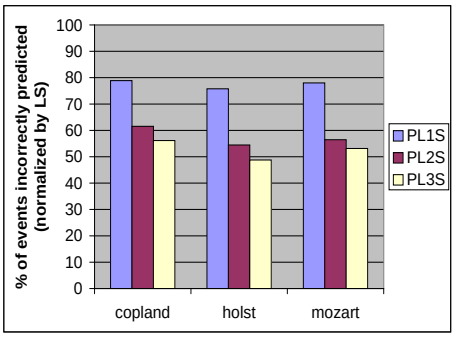

Figure 7: Incorrect prediction reduced by PLNS (normalized by LS)

percentage of events filtered out most (Copland) from the original trace data. In a real system such filtering won't be necessary because all program names are known.

One last note about the number of ⟨*program name*, *successors*⟩ pairs that a file requires to successfully implement PLNS. Our simulation results show that for Copland, more than 99% of files are accessed by four or fewer programs, while more than 99% of files are accessed by six or fewer programs for Holst. For Mozart, more than 99% of files are accessed by five or fewer programs. Thus the amount of data stored for each file in PLNS is not of concern.

In addition to predictive accuracy we also want to know how PLNS performs in terms of cache hit ratio, and additional experiments were conducted to determine this. We set the cache size according to the number of files it can hold for two reasons. The first is that file size is usually small, so the entire file can often be prefetched into cache [14]. The second is that in the case of large files, sequential read is the most common activity. Modern operating systems can already identify sequential read accesses and techniques such as prefetching the next several data blocks for sequential read have been implemented. We simulate cache with different sizes ranging from 25

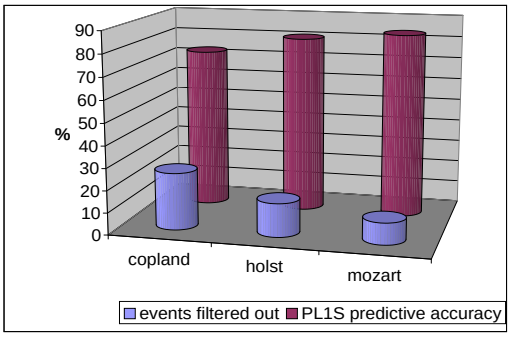

Figure 8: PL1S performance vs. percentage of events filtered out of original trace data

files to 2000 files, and compare the cache hit ratios between the LRU caching algorithm with no prediction and the LRU caching algorithm with PL1S. Figure 9 shows that when using PL1S prediction, the cache always performs better than when using LRU alone, regardless of cache size, and in some cases it performs as well as a cache up to 40 times larger.

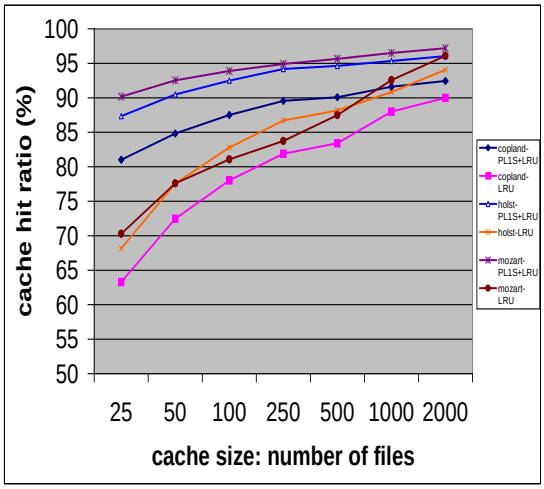

Figure 9: Cache hit ratio of LRU (lablled LRU) and LRU with PL1S (labelled PL1S+LRU)

5 Future Work

Several alternatives may improve the performance of PLNS and are worthy of further exploration. For example, files existing temporarily (such as those in */tmp* directory) usually won't get the same name next time they are created again. If so, then they can never be predicted

correctly by PLNS and there is no need to store their information. PLNS may also use the preceding file together with the ⟨*program name*, *successors*⟩ to improve the performance.

We believe that part of the reason for the dramatic performance improvement shown in Figure 9 is that an incorrect prediction made by PLNS, one that does not correctly predict the next file to be accessed, will still provide benefit if the file is subsequently accessed while it is still in the cache. Because PLNS makes program-based predictions, its incorrect predictions are much more likely to be for a file to be accessed in the near future than are predictions made by non-program-based models, which may predict a file accessed by a program that is no longer even running. In other words, the incorrect predictions by PLS are more likely to be used in the near future and are therefore less wrong than those made by other models. In future work we plan to measure this effect directly.

6 Conclusions

As the speed gap between CPU and the secondary storage device will not be narrowing in the foreseeable future, file prefetching will continue to remain a promising way to keep programs from stalling while waiting for data from disk. Incorrect prediction can be costly in practice. Reducing the number of files incorrectly predicted is very important to cache management and the overall system performance. We propose PLNS, a new Program-based Last N Successors model. Our simulations from PLNS show good results in predicting files, especially in eliminating the cases of incorrect prediction.

By tracking programs initiating file accesses, we successfully avoid many incorrect predictions, which is particularly valuable in a system with limited cache size. Therefore, the overall performance penalty in a system caused by incorrect predictions can be significantly reduced. At least more than 21% of incorrect predictions can be reduced as compared with LS as our results demonstrate. Different values of "N" in PLNS could affect the performance and they come with different costs. As "N" equals to two, PL2S provides an impressive improvement over LS with little overhead. We also compare the cache hit ratios of LRU with and without PL1S. The results show that with PL1S, LRU can deliver a much higher cache hit ratio.

References

[1] Mary Baker, John Hartman, Michael Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *ACM 13th Symposium on Operating Systems Principles*, 1991.

[2] Fay Chang and Garth Gibson. Automatic I/O Hint Generation through Speculative Execution. In *Third Symposium on Operating Systems Design and Implementation*, 1999.

[3] Kenneth Curewite, P. Krishnan, and Jeffrey Scott Vitter. Practical Prefetching via Data Compression. In *ACM SIGMOD*, 1993.

[4] J. Griffioen and R. Appleton. Reducing File System Latency Using a Predictive Approach. In *Proceedings of USENIX summer Technical Conference*, 1994.

[5] Dirk Grunwald and Douglas Joseph. Prefetching Using Markov Predictors. In *ACM/IEEE International Symposium on Computer Architecture*, 1997.

[6] James Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *ACM Transcations on Computer Systems*, 1992.

[7] Tom Kroeger and Darrell Long. The Case for Efficient File Access Pattern Modeling. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, 1999.

[8] Geoffrey H. Kuenning. The Design of the Seer Predictive Caching System. In *Workshop on Mobile Computing Systems and Applications, IEEE Computer Society*, 1994.

[9] An-Chow Lai and Babak Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *ACM/IEEE International Symposium on Computer Architecture*, 1999.

[10] Hui Lei and Dan Duchamp. An Analytical Approach to File Prefetching. In *Proceedings of the USENIX 1997 Annual Techical Conference*, 1997.

[11] L. Mummert and M. Satyanarayanan. Long Term Distributed File Reference Tracing: Implementation and Experience. Technical report, CMU, 1994.

[12] Mark Palmer and Stanley B. Zdonik. Fido: A Cache That Learns to Fetch. In *Proceedings of the 17th International Conference on Very Large Data Base*, 1991.

[13] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the 15th Symposium on Operating Systems Principles*, 1995.

[14] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference*, 2000.

[15] Elizabeth Shriver and Christopher Small. Why does file system prefetching work? In *Proceedings of the 1999 USENIX Annual Technical Conference*, 1999.

[16] Jeffery Scott Vitter and P. Krishnan. Optimal Prefetching via Data Compression. In *Journal of the ACM*, 1996.

[17] Tsozen Yeh, Darrell Long, and Scott Brandt. Conserving Battery Energy Through Making Fewer Incorrect File Predictions. In *Proceedings of the IEEE Workshop on Power Management for Real-Time and Embedded Systems*, 2001.

[18] Tsozen Yeh, Darrell Long, and Scott Brandt. Performing File Prediction with a Program-Based Successor Model. In *Proceedings of the Ninth International Symposium on Modeling, Analysis, and Simulation on Computer and Telecommunication Systems (MASCOTS 2001)*, 2001, *to appear*.