Increasing Predictive Accuracy through Limited Prefetching

Tsozen Yeh, Darrell D. E. Long and Scott A. Brandt
Computer Science Department
University of California, Santa Cruz
{yeh,darrell,sbrandt}@cse.ucsc.edu

Keywords: file prediction, prefetching

Abstract

Prefetching multiple files per prediction can improve the predictive accuracy. However, it comes with the cost of using extra cache space and disk bandwidth. This paper discusses the *most Recent distinct n Successor* (RnS) model and uses it to demonstrate the effectiveness of our earlier work, *Program-based Last n Successor* (PLnS) model, a program-based prediction algorithm [21]. We analyze the simulation results from different trace data and show that PLnS can perform better than RnS while it only predicts at most 59% of the number of files predicted by RnS when the n in PLnS equals to two. PLnS is a good candidate when considering prefetching multiple files per prediction to improve predictive accuracy.

1 INTRODUCTION

As disks operate significantly slower than CPUs, prefetching files to cache memory before they are used remains a promising way to mitigate the problem of speed difference between them. While correct file prediction is useful, incorrect prediction is to a certain degree unavoidable. Incorrect prediction not only wastes cache space and disk bandwidth, it also prolongs 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. Consequently incorrect predictions can lower the overall performance of the system regardless of the accuracy of correct prediction.

Prefetching multiple files for each prediction, on one hand, could take advantage of available cache space and disk bandwidth to potentially increase the predictive accuracy. On the other hand it will consume extra cache space and disk bandwidth, which can bring down the system performance if it is not done wisely. Our goal is to find the cost-effective performance between the number of files predicted per event and the predictive accuracy potentially could be increased.

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 the history of file access have been widely used to perform file prediction [4, 5, 10–12, 16], as have hints or help from programs and compilers [3, 17].

We extend the simple *last-successor* (*LS*) model to the *most Recent distinct n Successor* (*RnS*) model, which predicts multiple files each time to improve predictive accuracy. Our simulation results show that RnS can improve the predictive accuracy. However, the cost comes with it is not negligible. The simulation results also demonstrate that our earlier work, the *Program-based Last n Successor* (PLnS) model [21], can perform as well as RnS, while it predicts less than 59% of the number of files predicted by RnS. Thus, PLnS can generate a better cost-effective performance than RnS. The n in RnS and PLnS indicates the maximum number of files both models can predict per prediction.

Files are accessed by programs. Probability and repeated history of file accesses should not occur for no reason. Our simulations confirm this surmise by showing that the vast majority of files are accessed by one or two programs, which indicates that consecutive accesses of different files can be more predictable on a program-based successor model. We also show that PL2S provides the best cost-effective performance in terms of the number of files predicted each time and the predictive accuracy improved in our simulations.

2 RELATED WORK

Most probability-based prediction algorithms use the history of system-wide file access, which does not consider and take advantage of the corresponding program information like PLnS does.

Griffioen and Appleton use *probability graphs* to predict future file accesses [5]. 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 decisions. Their simulations show that different combinations of window and threshold values could largely affect the performance.

Kroeger and Long predict next file based on probability of files in contexts of FMOC [10]. Their research also adopts the idea of data compression like Vitter *et al.* [20], but they apply it to predicting the next file instead of the next page.

Lei and Duchamp use *pattern trees* to record past execution activities of each program [12]. 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. This imposes keeping duplicated information on the system. Pattern trees of a running program are compared with the current accessing pattern. If a match found, files in that pattern tree are prefetched to memory. One of the main differences between their algorithm and PLnS is that PLnS makes the predicting decision for each individual file, so it can adapts to different patterns of file access more rapidly.

Vitter *et al.* adopt the technique of data compression to predict next required page [4,20]. 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 the next page more accurately.

Patterson *et al.* develop *TIP* to do prediction using hints provided from modified compilers [17]. Accordingly, resources can be managed and allocated more efficiently. Extra coding in programs and language dependence are disadvantages of this type of approach. In the case of no access to source codes there is no way to generate hints. Hints generated statically by compilers sometimes may not be very useful if file accesses cannot be decided until runtime.

Chang and Gibson design a tool which can transform UNIX application binaries to perform speculative execution and issues hints [3]. Their algorithm can eliminate the issue of language independence, but it can only be applied to single-thread applications.

Mowry *et al.* use modified compiler to provide future access patterns for out-of-core applications [14]. Kotz and Ellis define representative parallel file access patterns in parallel disk systems [9]. Cao *et al.* define four properties that optimal predicting and caching model should satisfy [2]. Palmer and Zdonik use *unit pattern* to prefetch data in database applications [16]. Kimbrel *et al.* examine four related algorithms to find out when a prefetching algorithm should act aggressively or conservatively [7].

Prefetching data between different levels of cache, such as moving data from the off-chip cache to the on-chip cache before the processor needs it, can also reduce the latency of memory operations [6].

Probability-based predicting algorithms, in general, 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.

In addition to file prefetching, many researchers have proposed different ways of storing data on disk to improve disk performance from a different angle [13, 18, 19].

3 LS, RnS, AND PLnS MODELS

We start with a brief description of LS, followed by a detailed discussion of RnS and PLnS.

3.1 LS (Last Successor)

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 RnS (Most Recent Distinct n Successor)

RnS is a refinement of LS. Unlike LS tracking the most recent one successor, RnS keeps track of the most recent distinct n successors. In the case of $n = 1$, R1S is the same as LS. RnS differs from LS when n $\geq$ 2. Take $n = 2$ for example: If the current file access pattern is "X, Y, A, B, C, D, A, C, A, C, A", R2S will predict the most recent two distinct successors of file A, which are C and B. LS (or R1S) only predicts C in this case. However, if the current pattern is "A, B, C, B, C, F, B", R2S only predicts file C because an access to B is never followed by an access to files other than C. Building RnS model is easy. It just keeps the most recent distinct n successors for each file.

3.3 PLnS (Program-based Last n Successor)

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 successor estimate [21, 22]. More precisely, PLnS records and predicts program-specific successors for each file that is accessed. The n in PLnS represents the number of the most recent distinct program-specific successors that PLnS could predict each time. For example, PL1S ($n = 1$) means that only the most recent program-specific successor is predicted after each file access. In other words, PL1S can be viewed as a program-based last successor model. PL2S ($n = 2$) predicts the most recent two distinct program-specific successors if a particular program accesses 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 two different pattern AB, and pattern AC after an access to A. If B and C tend to alternate after A, then either the probability-based prediction or the 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 these two patterns of consecutive file accesses 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.

Similarly, for PL2S ($n = 2$), it predicts the most recent two distinct program-specific successors if the program-specific successor ever changed. For example in Figure 2, if program P_1 accesses a different file, say D, other than B after an access to A. PL1S will predict D from now, while PL2S will predict both D and B since they are the most recent two distinct successors that file A keeps for program P_1.

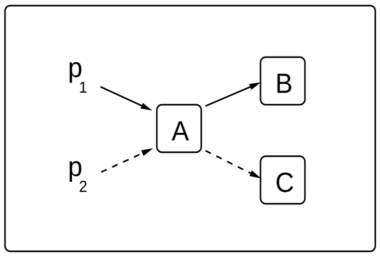

Figure 1. PL1S model

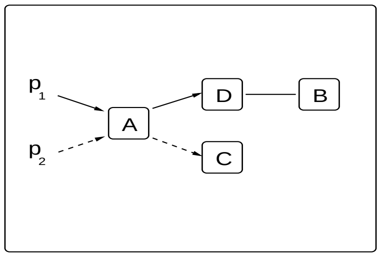

Figure 2. PL2S model

Table 1. Metadata of files in Figure 1 kept under PL1S model

File	⟨*program name, successor*⟩
A	$\langle P_1, B\rangle$, $\langle P_2, C\rangle$
B	$\langle P_1, NIL\rangle$
C	$\langle P_2, NIL\rangle$

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.

There are three issues that must be addressed. The first issue is how to collect the metadata in terms of ⟨*program name*, *successors*⟩ for each file. The solution is simple: Programs are executed as processes, so we

Table 2. Metadata of files in Figure 2 kept under PL2S model

File	⟨*program name*, *successor*⟩
A	$\langle P_1, D, B\rangle, \langle P_2, C\rangle$
B	$\langle P_1, NIL\rangle$
C	$\langle P_2, NIL\rangle$

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 $\langle P, Y\rangle$, and the next time that P accesses X, file Y will be predicted under PL1S model. The metadata of the files in Figure 1 is shown in Table 1. In the case of PL2S ($n = 2$), we keep the most recent two distinct program-specific successors for each file. So the corresponding metadata $\langle P_1, B\rangle$ now becomes $\langle P_1, D, B\rangle$ as seen in Table 2.

The second issue is how large the metadata needs to be in order to make accurate predictions, which 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 the vast of majority of files are accessed by five or fewer programs and thus metadata storage is not a problem. The last issue is that if a program (say X) eventually executes another program (say Y), the information of Y is also added to the metadata of X, and Y will be predicted accordingly in the future.

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 following sections, we first explain how we conduct our experiments. We then discuss the performance of RnS and PLnS, followed by the comparison between these two models.

4.1 Simulation Trace and Experimental Methodology

The key requirement of the file trace we need is the information of corresponding programs for events of file access recorded in the trace. However, the program information cannot be obtained either directly from the traces, or incorrectly by re-processing the data in all the file traces we have access to, except the *DFSTrace* from the *Coda* project [8, 15]. As a result, we selected DFSTrace to evaluate the performance differences between RnS and PLnS.

These traces were collected from 33 machines during the period between February of 1991 and March of 1993. Our earlier study shows that PLnS provides a similar performance improvement over LS among different types of workloads, such as servers and desktop workstations [21,22]. We used data from October 1992 to March 1993, roughly equal to the last quarter of the entire trace, from three desktop workstations, *Copland*, *Holst*, and *Mozart* in our experiments. Research has demonstrated that the average life of a file is very short [1]. So instead of tracking every *READ* or *WRITE* event, we track only the *OPEN* and *EXECVE* events in our simulation.

One may question that the *DFSTrace* might not reflect the file access pattern we see today, particularly in file or program sizes and the rate at which file requests arrive. However, our simulation depends on neither of these, so they won't affect the results. Moreover, 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. Therefore we believe the file traces we used is still adequate to evaluate our algorithm.

As mentioned above, PLnS needs to know 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 *OPEN* events initiated by any *process identifier* (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 used the filtered trace data to evaluate RnS and PLnS. We are interested in how different values of

n in RnS and PLnS could affect the performance and the cost comes with it. The experiments were conducted for the cases when n in RnS and PLnS equals to one, two, and three respectively. As a reminder, LS and R1S are identical. There is no need to have separate results for LS.

Both R1S and PL1S predict one file at a time. We score R1S and PL1S by adding one for each correct prediction and zero for each incorrect prediction. We normalize the final scores of PL1S and R1S by the number of predictions, not by the number of events, to obtain the predictive accuracy. 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 R2S, R3S, PL2S, and PL3S the same way we score R1S and PL1S. Of course their scores do not sit on the same ground as those from R1S and PL1S because they could predict more than one file per prediction in some cases. A more detailed discussion of these four will follow later.

4.2 Performance of RnS and PLnS

We begin with RnS. Table 3 lists the average number of files predicted per event (*i.e.* per file access) for RnS and PLnS. Since we do not count the cases where no prediction was made, the result is that the number of events in the trace is larger than the number of cases where a prediction was made. Because both R1S and PL1S predict one file at a time whenever a prediction was made, so the number of files predicted per event is smaller than one for R1S and PL1S.

Table 3. Average number of files predicted per event in RnS and PLnS

Machines:	Copland	Holst	Mozart
R1S	0.94	0.98	0.98
PL1S	0.88	0.93	0.95
R2S	1.83	1.89	1.90
PL2S	1.07	1.08	1.08
R3S	2.69	2.74	2.79
PL3S	1.19	1.18	1.16

Figure 3 displays the predictive accuracy of RnS for the three machines. It shows that the improvement between R1S and R2S is more noticeable than the improvement between R2S and R3S in all cases. Figure 4 displays the predictive accuracy of PLnS. Figure 5 provides more details for the RnS performance. The upper part of Figure 5 represents the percentage of correct prediction (out of total prediction). The lower half of this figure

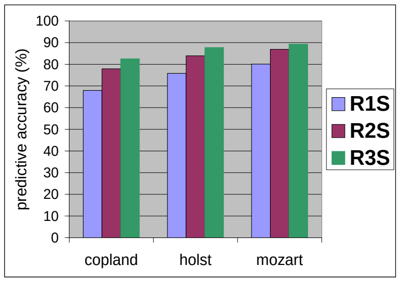

Figure 3. Predictive accuracy of RnS

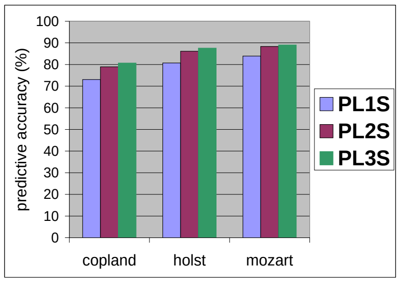

Figure 4. Predictive accuracy of PLnS

shows the percentage of incorrect prediction (out of total prediction). Because we do not count the case where no prediction was made, so it is more informative to put two parts (correct and incorrect) in the same figure to compare the detailed performance among R1S, R2S, and R3S. Figure 6 provides the same detailed performance comparison for the PLnS family. Clearly, R2S and PL2S have a noticeable improvement over R1S and PL1S respectively in making more correct predictions and fewer incorrect predictions. However, the number of files predicted in PL2S (1.07 or 1.08) is significantly smaller than their counterparts (1.83, 1.89, or 1.90) in R2S as seen in Table 3. Obviously PL2S is a better candidate than R2S when considering prefetching multiple files to improve predictive accuracy.

4.3 Comparison by Different Trace Files

There are two criteria used in comparing RnS and PLnS. The first one is the average number of files predicted per event. The second is the predictive accuracy. As mentioned earlier, predicting multiple files each time could increase the probability of correct prediction. However, doing it unwisely could waste cache space and disk bandwidth, which will eventually lower the overall system performance. Another issue is that prefetching files takes time. So prefetching too many files per prediction won't be a feasible way to improve predictive accuracy in practice.

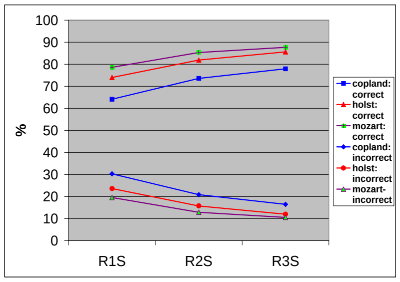

Figure 5. Percentage of correct (upper part) and incorrect (lower part) prediction in RnS

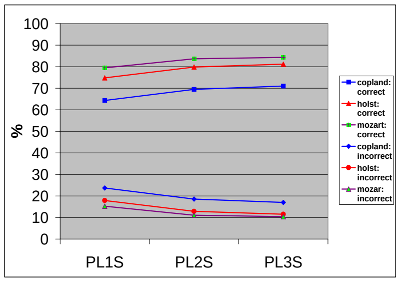

Figure 6. Percentage of correct (upper part) and incorrect (lower part) prediction in PLnS

We will compare RnS and PLnS machine by machine. Figure 7 plots the number of files predicted per event between RnS and PLnS for three traces from Table 3. The x-axis is the different values of n in RnS and PLnS. In average PL2S and PL3S predict a significantly smaller number of files per event than R2S and R3S respectively in all cases. The predictive accuracy of RnS and PLnS for Copland is displayed in Figure 8. PL2S delivers a higher predictive accuracy than R2S. PL1S also outperforms R1S. PL3S is a little bit worse than R3S, but it only predicts 1.19 files per event, instead of 2.69 files as in R3S. Consequently PL2S provides a better cost-efficient performance in terms of the number of files predicted each time and the predictive accuracy increased in our experiments. With predicting 1.07 files per event in average, PL2S can increase the predictive accuracy by 11% over LS (R1S) as seen in Figure 8. The corresponding results for Holst and Mozart are shown in Figure 9 and Figure 10. It is worthy of pointing out that results from all three file traces show the same pattern – PL2S performs better than R2S and it also predicts a smaller number of files per event than R2S. Besides, PL2S and R2S provide a better cost-effective performance increased in the PLnS and RnS family respectively. The good performance of PL2S validates our observation that consecutive accesses of different files can be more accurately predicted given knowledge about which programs are accessing them.

To get a better understanding of the percentage of incorrect prediction can be reduced in PLnS, we normalize the percentage of incorrect prediction in PLnS by that of LS and plot the results in Figure 11. It shows that PL1S reduces the incorrect predictions done by LS approximately between 20% and 25%, PL2S can do about 40% to 45% less, and PL3S can do about 45% to 50% less. Figure 12 shows the percentage of correct predictions PLnS made normalized to those made by LS.

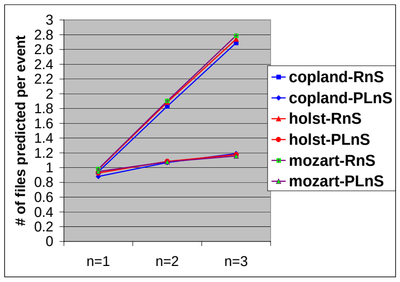

Figure 7. Average number of files predicted per event in RnS and PLnS for three trace files

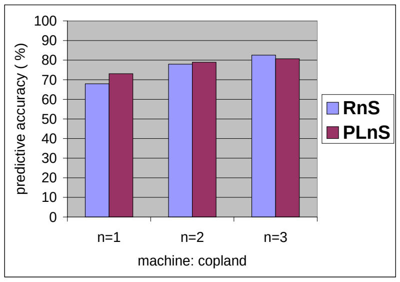

Figure 8. Predictive accuracy of RnS and PLnS for Copland

It is worthy of exploring a little more why, in general, the program-based successor predicting algorithm has a good performance while its cost remains relatively lower than the non program-based successor predicting algorithm. We stated earlier that files are accessed by programs and therefore probability and repeated history of file accesses should not occur for no reason, which implies that a program-based successor model should perform better than non program-based successor models. Table 4 shows the percentage of files which are accessed by up to five different programs in our simulations.

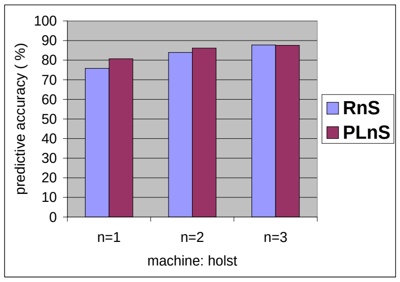

Figure 9. Predictive accuracy of RnS and PLnS for Holst

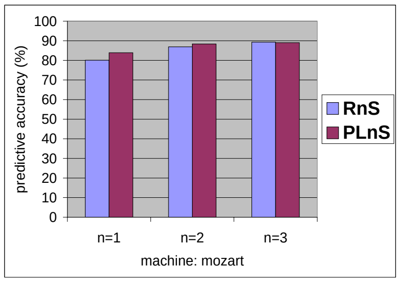

Figure 10. Predictive accuracy of RnS and PLnS for Mozart

The percentage of files accessed by one program is the highest in all three traces, especially for Copland. More than 90% of files are accessed by only one program in Copland, while 2.94% of files are accessed by two different programs. For both Holst and Mozart, the vast majority of files are accessed by either one or two programs. As a result, the file accesses observed for each individual program will be more predictable, and therefore the program-based successor predicting algorithm has a better cost-effective performance than the non program-based successor predicting algorithm.

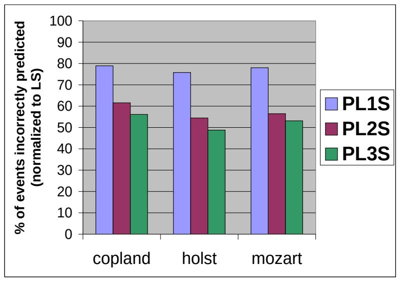

Figure 11. Incorrect predictions made by PLnS (normalized to LS)

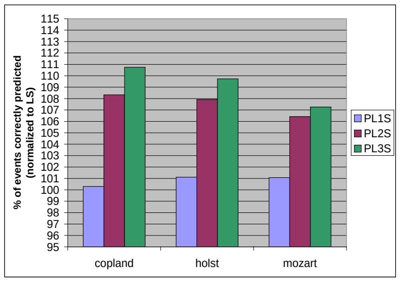

Figure 12. Correct predictions made by PLnS (normalized to LS)

Table 4. Percentage of files accessed by up to five different programs

# of programs	Copland	Holst	Mozart
1	90.5898%	64.5359%	54.9718%
2	2.9498%	19.0526%	26.7582%
3	2.7161%	4.1844%	5.8994%
4	1.3569%	7.8901%	2.0924%
5	0.2552%	1.3574%	2.1018%

5 CONCLUSIONS

As the speed gap between CPU and the secondary storage continues to widen and is unlikely to narrow in the near 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 expensive. Prefetching multiple files per prediction could increase the probability of correction prediction. However, prefetching too many files each time will likely lower the over system performance in practice. Finding the right balance between the number of files predicted per prediction and the predictive accuracy potentially could be increased is very important to the system performance.

Our results demonstrate that R2S and PL2S deliver better cost-effective performance in the RnS and PLnS family respectively. By tracking programs initiating file accesses, we successfully avoid many incorrect predictions. We show that PLnS performs better than RnS when the n is equal to one or two. R3S outperforms PL3S a little, but it comes with the cost of prefetching significantly more files than PL3S per event. On average, PL2S predicts only 1.07 or 1.08 files per event and it delivers noticeably higher predictive accuracy than LS. Compared with LS, about 40% to 45% of incorrect predictions can be reduced in PL2S as seen in Figure 11.

Therefore, PL2S can significantly reduce the overall performance penalty in a system caused by incorrect predictions when considering prefetching multiple files to improve the file predictive accuracy.

6 FUTURE WORK

The DFSTrace is almost 10 years old. We chose it because it contains the program information, which is absolutely necessary to the PLnS model. In the future, we would like to collect our own traces that PLnS can use, and examine how PLnS performs under more recent traces. Ultimately, we will build the PLnS into the filesystem and evaluate its performance in a real system. We would also like to investigate how PLnS performs in a system with multiple levels of cache, and what improvement needs to be made.

7 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments. We are also grateful to the Coda group for providing us access to their traces. This research is supported in part by the National Science Foundation award number PO-10152754 and CCR-0073509, and by Lawrence Livermore National Laboratory under contract B513238.

References

[1] M. Baker, J. Hartman, M. Kupfer, K. Shirriff, and J. Ousterhout. Measurements of a Distributed File System. In *ACM 13th Symposium on Operating Systems Principles*, pages 198–212, 1991.

[2] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of Integrated Prefetching and Caching Strategies. In *ACM SIGMETRICS*, pages 188–197, 1995.

[3] F. Chang and G. Gibson. Automatic I/O Hint Generation through Speculative Execution. In *Third Symposium on Operating Systems Design and Implementation*, pages 1–14, 1999.

[4] K. Curewitz, P. Krishnan, and J. S. Vitter. Practical Prefetching via Data Compression. In *ACM SIGMOD*, pages 257–266, 1993.

[5] J. Griffioen and R. Appleton. Reducing File System Latency Using a Predictive Approach. In *Proceedings of USENIX summer Technical Conference*, pages 197–207, 1994.

[6] D. Joseph and D. Grunwald. Prefetching using Markov Predictors. In *Intl. Symposium on Computer Architecture (ISCA)*, pages 252–263, 1997.

[7] T. Kimbrel, A. Tomkins, H. Patterson, B. Bershad, P. Cao, E. W. Felten, G. A. Gibson, A. R. Karlin, and K. Li. A Trace-Driven Comparison of Algorithms for Parallel Prefetching and Caching. In *Second Symposium on Operating Systems Design and Implementation*, pages 19–34, 1996.

[8] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. In *ACM Transcations on Computer Systems*, pages 3–25, 1992.

[9] D. Kotz and C. S. Ellis. Practical Prefetching Techniques for Parallel File Systems. In *Proceedings of the first Parallel and Distributed Information Systems, IEEE*, pages 182–189, 1991.

[10] T. Kroeger and D. D. E. Long. The Case for Efficient File Access Pattern Modeling. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, pages 14–19, 1999.

[11] G. H. Kuenning. The Design of the Seer Predictive Caching System. In *Workshop on Mobile Computing Systems and Applications, IEEE Computer Society*, pages 37–43, 1994.

[12] H. Lei and D. Duchamp. An Analytical Approach to File Prefetching. In *Proceedings of the USENIX Annual Techical Conference*, pages 275–288, 1997.

[13] L. McVoy and S. Kleiman. Extent-Like Performance from a Unix File System. In *Proceedings of USENIX Annual Technical Conference*, pages 33–43, 1991.

[14] T. Mowry, A. Demke, and O. Krieger. Automatic Compiler-Inserted I/O prefetching for Out-of-Core Applications. In *The Second Symposium on Operating Systems Design and Implementation*, pages 3–17, 1996.

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

[16] M. Palmer and S. B. Zdonik. Fido: A Cache That Learns to Fetch. In *Proceedings of the 17th International Conference on Very Large Data Base*, pages 255–264, 1991.

[17] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 79–95, 1995.

[18] M. Rosenblum and J. Ousterhout. The LFS Storage Manager. In *Proceedings of USENIX Annual Technical Conference*, pages 315–324, 1990.

[19] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of USENIX Annual Technical Conference*, pages 33–44, 1996.

[20] J. S. Vitter and P. Krishnan. Optimal Prefetching via Data Compression. In *Journal of the ACM*, pages 771–793, 1996.

[21] T. Yeh, D. D. E. Long, and S. Brandt. Caching Files with a Program-based Last n Successors Model". In *Proceedings of the Workshop on Caching, Coherency and Consistency (WC3 '01)*, 2001.

[22] T. Yeh, D. D. E. Long, and S. 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)*, pages 193–202, 2001.