Using Program and User Information to Improve File Prediction Performance

Tsozen Yeh, Darrell D. E. Long and Scott A. Brandt[*]
Computer Science Department
Jack Baskin School of Engineering
University of California, Santa Cruz
1156 High Street, Santa Cruz, CA. 95064
{*yeh,darrell,sbrandt*}*@cse.ucsc.edu*

Abstract

Correct prediction of file accesses can improve system performance by mitigating the relative speed difference between CPU and disks. This paper discusses Program-based Last Successor (PLS) and presents Program- and User-based Last Successor (PULS), file prediction algorithms that utilize information about the program and user that access the files. Our simulation results show that PLS makes 21% fewer incorrect predictions and PULS makes 24% fewer incorrect predictions than last-successor with roughly the same number of correct predictions that last-successor makes. The cache space wasted on incorrect predictions can be reduced accordingly. We also show that a cache using the Least Recently Used (LRU) caching algorithm can perform better when the PULS is applied. In some cases, a cache using LRU and either PLS or PULS performs better than a cache up to 40 times larger using LRU alone.

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. Probability and history of file access have been widely used to perform file prediction [10, 11, 13, 4, 5, 12, 16], as have hints or help from programs and compilers [17, 3].

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.

[*]Supported in part by the National Science Foundation award number PO-10152754.

Files are accessed by programs, and programs are executed on behalf of users. This suggests that consecutive accesses of different files should not occur for no reason. We contend the reason is that programs access more or less the same set of files in roughly the same order every time they execute, especially when they are executed by the same user. Therefore consecutive accesses of different files can be more accurately predicted given knowledge about which programs and users are accessing them. We have developed two file prediction algorithms that utilize this information to generate more accurate file predictions; *Program-based Last Successor* (PLS) [20] uses program information while *Program- and User-based Last Successor* (PULS) uses both program and user information. Our results demonstrate that both PLS and PULS generate more accurate file predictions than the other file prediction algorithms examined, with PULS performing somewhat better. In particular, compared with LS, they reduce the number of incorrect file predictions while maintaining roughly the same number of correct predictions to provide better overall file prediction and therefore better overall system performance.

We compare PLS and PULS with Last-Successor (LS) and *Finite Multi-Order Context* (FMOC) [10]. LS can correctly predict the next file to be accessed close to 80% of the time in our experiments. FMOC outperformed LS in a one-month trace in Kroeger's study [10] but it performs slightly worse than LS in our simulations. Our experiments demonstrate that with traces covering as long as 13 months PLS makes about 21% fewer incorrect predictions and PULS makes about 24% fewer incorrect predictions than LS, giving PULS the highest predictive accuracy among all four models in our comparison. We then use a synthesized trace to demonstrate that PULS outperforms LS in a greater scale when there are more users in the system.

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

2 Related Work

Most probability-based predicting algorithms use the history of systemwide file access, which does not consider and take advantage of corresponding program or user information like PULS 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.* [19], 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 [13]. 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 PULS is that PULS makes the predicting decision for each individual file, so it can adapts to different patterns of file access more rapidly.

Vitter, Curewitz, and Krishnan adopt the technique of data compression to predict next required page [4, 19]. 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.

3 Predicting Models Compared

We will briefly discuss different predicting models compared in our study. We start with LS, which is a common benchmark to compare predicting models. We then explain FMOC, which outperformed LS in previous study, followed by the discussion of PLS and PULS.

3.1 Last Successor (LS)

LS is a common benchmark used in comparing different schemes of file prediction. For each file accessed, LS predicts the most recent successor following the last access of the current file as the next successor. Metadata kept in this model is simple. Only one last-successor is needed for each file. However, scheduling the same set of programs in different orders may generate totally different file access patterns. This means the performance of LS could vary dramatically even when the same set of programs executed repeatedly.

3.2 Finite Multi-Order Context (FMOC)

FMOC predicts the next file to be accessed from the files that have been seen so far in "*context*" [10]. Each file seen in a context has a probability indicating the likelihood that it follows that context. FMOC often prefetches multiple files for each prediction. The "*additive accuracy*" was defined to compare the performance between FMOC and LS [10]. If the next file accessed is among those files prefetched, then the predicted probability of that file is added to the score of FMOC. The final score is then normalized by the number of events in the simulation trace to obtain the "additive accuracy" [10]. The additive accuracy therefore indicates the likelihood that the next file actually referenced is among those predicted files.

Kroeger's study showed that using order higher than two resulted in negligible improvements so in this work we only examine the second order FMOC model (denoted as FMOC2).

3.3 Program-based Last Successor (PLS)

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. For example, consider a system with a preemptive scheduler running two programs, P_1 and P_2, where P_1 accesses files A, B, and C, in that order, and P_2 accesses files X, Y, and Z, in that order, and each file is accessed exactly once. While each program has a perfectly predictable access pattern and each file (after the first one in each sequence) follows exactly one other file in the program-based sequence, the system will see one of 20 different file access patterns ($\frac{6!}{3! \times 3!} = 20$) depending on the exact timing of context switches in the system. In particular, with repeated executions of these two programs the history of file accesses observed by the system will vary considerably.

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 "*successor*") 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. Of course, if a particular program accesses multiple different files after each access of a particular file, then the program-specific last successor will change. The metadata of files in Figure 1 is shown in Table 1.

PLS does not have the problem of making incorrect predictions when different file access patterns generated from executing the same set of programs as discussed earlier in LS. However, there are cases where different executions of the same program will likely involve different sequences

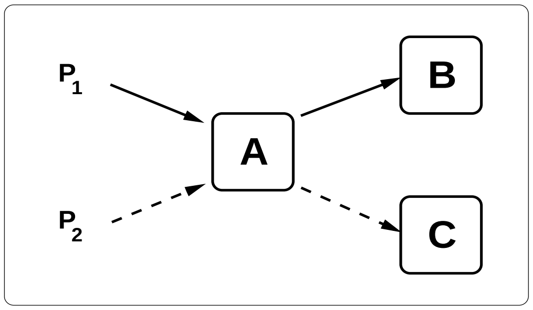

Figure 1. Program-based Last Successor model

of file accesses. Executing system programs such as editors and compilers often falls into this category where PLS could make fewer correct predictions. This is due to different users usually edit and compile their own files. Consequently if we have the knowledge about which user is editing or compiling the files, we then can make predictions more accurately in this case. Database applications could also involve different sequences of file access depending on which user is executing the program.

Table 1. Metadata of Figure 1 kept under PLS model

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

3.4 Program- and User-based Last Successor (PULS)

PULS is a refinement of PLS. PULS takes the user information into account when it makes predictions. As stated earlier knowing which user is executing a particular program can make predictions more accurately than PLS in certain cases. In this section we will discuss how to implement PULS and explain why it can perform better than LS and PLS.

Probability can only tell us what patterns of file access are and how frequently they occur, but not why these patterns exist. Since files are accessed by programs, and programs are executed on behalf of users. Consequently, patterns of file access are largely decided by who is running what program in the system.

Figure 2 shows a case where PULS can outperform PLS. Suppose file B is accessed after an access to file A when the user U_1 runs the program P_1, while an access to A will

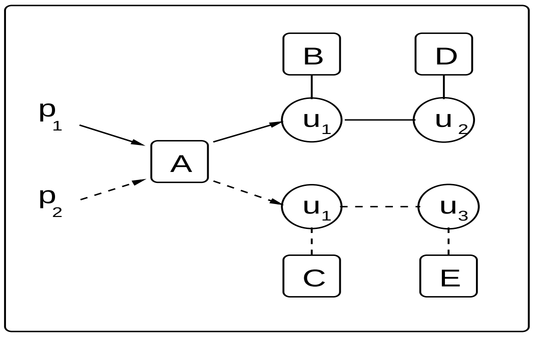

Figure 2. Program- and User-based Last Successor model

be followed by an access to D when P_1 is executed by U_2. In the meanwhile when U_1 executes P_2, an access to A is followed by an access to C, while an access to E will follow an access to A if P_2 is executed by U_3 instead. In this case, PLS will make fewer correct predictions if U_1 and U_2 tend to execute P_1 alternatively. Similarly U_1 and U_3 will face the same problem when they execute P_2. The situation can get worse as there are more users execute the same programs in a system.

However, no matter how many users executing the same programs, PULS can still make correct predictions in the similar cases of Figure 2. This is because PULS collects the information (a set of pairs consisting of "*program name*" and "*user-successor*") for file A, next time it is accessed we can predict either B, C, D, or E depending on which user (U_1, U_2, or U_3) is running P_1 or P_2, or provide no prediction if A is accessed by another program or user. Of course, if a particular program accesses multiple different files after each access of a particular file, then the program- and user-specific last successor will change.

PLS can predict as well as PULS only in a single-user system, or in a system there are no users sharing any programs. The performance PULS over PLS will increase as more uses executing the same programs. In real systems, machines often host multiple users, PULS therefore could predict more accurately than PLS accordingly.

Table 2. Metadata of Figure 2 kept under the PULS model

File	⟨*program name, user-successor*⟩
A	$\langle P_1, U_1B, U_2D\rangle, \langle P_2, U_1C, U_3E\rangle$
B	$\langle P_1, NIL\rangle$
C	$\langle P_2, NIL\rangle$

PLS and PULS can avoid the slow adaptation problem in probability-based prediction models. Probability-based models always predict the same file until the corresponding probability changes. Like LS, both PLS and PULS do not rely on probability so it can respond immediately as patterns of file access change.

There are three issues that need to be addressed. The first issue is how to collect the metadata in terms of ⟨*program name, user-successor*⟩ for each file. Programs are executed as processes, so we can just store the *program name* and *user ID* (uid) in the process control block (*PCB*). For each running program (say P) executed by a user (say U), 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 $\langle P, UY\rangle$, and the next time that P accesses X on behalf of U, PULS can predict that the next file accessed will be Y.

In the example of Figure 2, when P_1 (say executed by U_1) accesses the next file (say B) after its access to A, we update the metadata of A with $\langle P_1, U_1B\rangle$, and next time P_1 accesses A on behalf of U_1, PULS can predict that the next file accessed will be B. Similarly, A also keeps $\langle P_1, U_2D\rangle$ as parts of its metadata. The metadata of files in Figure 2 is shown in Table 2.

The second issue is how big 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- and user-specific successor to the file, so that we know which file to predict when the same program executed by the same user 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, user-successor*⟩ pairs kept for each file. However, our simulation shows that the vast of majority of files are accessed by six or fewer programs and thus metadata storage is not a problem.

The last one 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 it 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 PULS 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 and Performance Evaluation

In this section, we will discuss the trace data we used to conduct our experiments, explain why we choose the particular trace in our simulations, and finally how we compare performance of FMOC2, LS, PLS, and PULS.

4.1 Simulation Trace

The key requirement of the file trace we need is the information of corresponding programs and users for events of file access recorded in the trace. User information sometimes is available in some traces we study. However, the program information cannot be obtained either directly from the traces, or incorrectly by reprocessing the data in all the file traces we have access to, except the *DFSTrace* from the *Coda* project [8, 15]. Therefore we select DFSTrace to evaluate the performance differences among models we compared.

File traces in DFSTrace were collected from 33 machines during the period between February of 1991 and March of 1993. We used data covering between 7 and 13 months from four machines, *Barber*, *Mozart*, *Dvorak*, and *Ives*. Barber was a server, Mozart was a desktop workstation, Dvorak had the highest percentage of write, and Ives hosted the most users. The periods of data selected from Barber, Mozart, Dvorak, and Ives are 11, 13, 7, and 7 months respectively. Research has demonstrated that the average life of a file is very short [1]. Besides, DFSTrace does not contain events of *READ* or *WRITE* most of the time. Therefore, instead of tracking every *READ* or *WRITE* event, we track only the *FORK*, *EXECVE* and *OPEN* events in our simulation.

As mentioned above, PULS needs to be able to determine the name of a program and the user in order to generate its predictions. Because we cannot obtain the name of any program or user that started executing before the beginning of the trace, we exclude *EXECVE* events forked by processes whose user IDs (uid) are unknown. This is because DFSTrace only reports uid of the child process in the *FORK* event. By catching the uid of the new child process, we have the uid we need for all the following *EXECVE* and *OPEN* events from that child process. We also exclude *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 and user names are known.

4.2 Methodology of Performance Evaluation

We score PULS, PLS and LS in the same way by adding one for each correct prediction and zero for each incorrect prediction. We normalize the final scores of these three models by the number of predictions, not by the number of events as in the FMOC2 model. This is because the first time that a file is accessed by a program there is no previous successor to predict and so the failure to make a prediction the first time cannot be considered incorrect. Since our simulation trace is very long (between 7 and 13 months), it turns out that the effect of this compulsory error is negligible and does not affect the comparison of predictive accuracy among these models. The final score (in percentage) is referred as "predictive accuracy" in our experiment. So a predictive accuracy of x% means that in average there are x correct predictions out of 100 predictions. As explained earlier, FMOC2 tends to predict multiple files at a time, the score of FMOC2 is the "additive accuracy", which can be viewed equally to the "predictive accuracy" used in our experiments. In the meanwhile for models predicting only one file at a time such as the other three models compared, predictive accuracy is indeed same as additive accuracy.

4.3 Comparison of Predictive Accuracy

We used the filtered trace data to evaluate FMOC2, LS, PLS, and PULS. Figure 3 shows that PULS has the highest predictive accuracy 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 usually 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 detailed performance of other models in three categories. The first category is the percentage of total predictions (including correct and incorrect predictions) made by PULS as compared with LS. This percentage should not be to too small, otherwise PULS may be an unrealistic model just like the model B above. The second is the percentage of correct predictions made by PULS as compared with LS. This number should be as high as possible. The last category is the percentage of incorrect predictions made by PULS as compared with LS. Ideally this percentage should be less than 100%, indicating that PULS makes fewer incorrect predictions than LS.

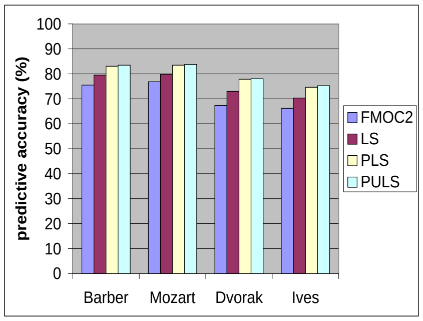

Figure 3. Predictive accuracy of FMOC2, LS, PLS, and PULS

4.4 Performance by Category

We cannot do the same comparison with FMOC2 due to the nature of the FMOC discussed above. Figure 4 displays the performance in the category of total prediction. It shows that the percentage of events where a prediction was made by PULS is only about ten percent less than that of LS. This is close enough to consider PULS to be a practical prediction algorithm in terms of the number of predictions it makes. The percentage of correct predictions is shown in Figure 5. Both percentages for Barber and Ives from PULS are over 98% of the numbers from LS, and it is over 99% for Mozart. For Dvorak, PULS makes more correct predications than LS. Figure 5 demonstrates that both PULS and PLS can do roughly as well as LS in correctly predicting files. Figure 6 shows the percentage of incorrect predictions. To get a closer look at the relative performance in LS, PLS, and PULS in terms of reducing incorrect predictions, data in Figure 6 is normalized to LS and displayed in Figure 7. Figure 7 clearly shows that PULS makes fewer incorrect predictions than both LS and PLS. PULS reduces about 24% of incorrect predictions compared with LS. In the meanwhile PULS cuts incorrect predictions about 3.5% more than PLS can reduce in some cases. This explains why PULS has the highest predictive accuracy among LS, PLS, and PULS seen in Figure 3. As we discussed before, incorrect predictions come with a cost, and avoiding this cost directly translates into better system performance.

The reduction of incorrect predictions in PULS is interesting enough to be worthy of further exploration. Since the number of predictions made by PULS is only about ten percent less than LS, and one percent less than PLS, and the number of correct predictions of these three are roughly the same, we conclude that PULS makes no prediction more

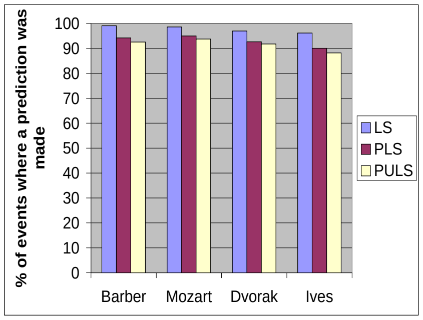

Figure 4. Total predictions made by LS, PLS, and PULS

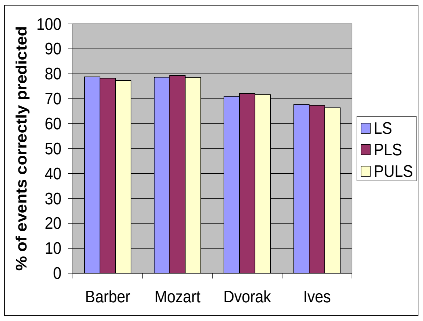

Figure 5. Correct predictions made by LS, PLS, and PULS

often than both LS and PLS . We collected the percentages of cases where no prediction was made by PULS and PLS, and compare them with LS, and the results are displayed in Figure 8, which confirms this surmise. Figure 8 shows that the percentage of events where no prediction was made by PULS is roughly three to seven times higher than that of LS, and is about 1.3 to 1.7 times higher than that of PLS.

4.5 Performance Increase for Multi-Users

Events in each of the four traces are generated by only a small number of users. We believe PULS can surpass LS more as the number of uses goes up in the system. To simulate a system which hosts more users, we synthesized a one-month trace by combining events within the same month from each of the four traces. Because we define a correct prediction as only if the predicted file is the next file needed by the entire system, not by each individual program, therefore both LS and PULS are expected to generate lower predictive accuracy from the synthesized trace. We calculate the “total-weighted” predictive accuracy by sum-

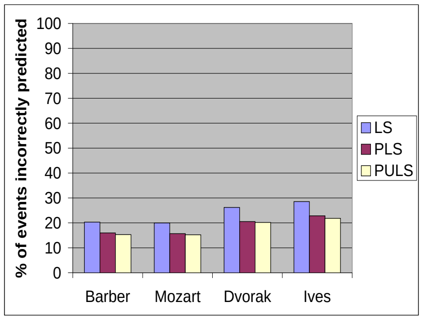

Figure 6. Incorrect predictions made by LS, PLS, and PULS

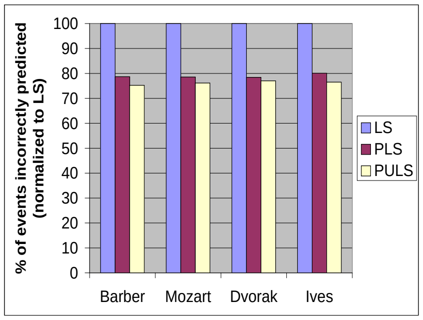

Figure 7. Incorrect predictions made by LS, PLS, and PULS (normalized to LS)

ming up the four "individual-weighted" predictive accuracy, which is the product of multiplying the predictive accuracy of one trace by the percentage of events in the synthesized trace that come from that particular trace. Table 3 and Table 4 show the results for PULS and LS respectively.

Take the Table 3 for example, the original predictive accuracy for Barber is 83.4846%. The number of events from Barber constitutes 25.1210% of the total events in the synthesized trace. So the "individual-weighted" predictive accuracy of Barber is 20.9722% ($83.4846\% \times 25.1210\%$). The "total-weighted" predictive accuracy of PULS, 81.0719% , is the sum of each "individual-weighted" predictive accuracy from the four traces. The "total-weighted" predictive accuracy practically can be viewed as the upper bound of the predictive accuracy that PULS can generate from the synthesized trace. Table 5 compares the practical upper bound of predictive accuracy that PULS and LS can achieve, and the actual predictive accuracy both generate. The results show that PULS can reach 90.88% of its practical limit, while LS can only reach

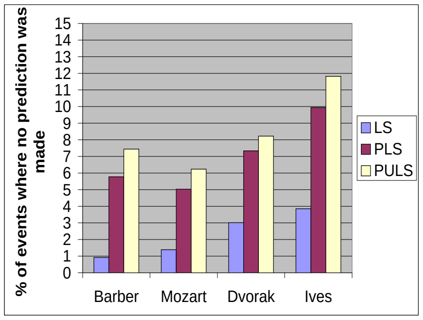

Figure 8. No predictions made by LS, PLS, and PULS

Table 3. Weighted Predictive Accuracy for PULS

Trace	Pred. Accu.	% of events	I. W. Accu.
Barber	83.4846%	25.1210%	20.9722%
Mozart	83.7721%	34.8948%	29.2321%
Dvorak	78.0213%	28.1554%	21.9672%
Ives	75.2438%	11.8288%	8.9004%
Total			81.0719%

88.1546%. This demonstrates that PULS could outperform LS in a greater scale when there are more users in the system. One last note about this evaluation is that the synthesized trace we produced is only one-month long. We expect the performance increase that PULS over LS will be more noticeable when a longer trace applied

4.6 Cache Performance

In addition to predictive accuracy we also want to know how PLS and PULS perform in terms of cache hit ratio. 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 [18]. The second is that in the case of large files, sequential

Table 4. Weighted Predictive Accuracy for LS

Trace	Pred. Accu.	% of events	I. W. Accu.
Barber	79.4912%	25.1210%	19.9690%
Mozart	79.7341%	34.8948%	27.8231%
Dvorak	72.9996%	28.1554%	20.5533%
Ives	70.3224%	11.8288%	8.3183%
Total			76.6637%

Table 5. Performance Comparison of the Synthesized Trace for LS and PULS

	LS	PULS
Total W. Accu.	76.6637%	81.0719%
Syn. Accu	67.5825%	73.6781%
Syn. Accu / Total W. Accu	88.1546%	90.88%

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 files to 2000 files, and compare the cache hit ratios between the LRU caching algorithm with no prediction and the LRU caching algorithm with PLS and with PULS. LRU has been widely employed [21], so it is an appropriate candidate used to evaluate the effectiveness of predicting algorithms in terms of cache hit ratio. Figure 9 shows that when using PULS prediction, the cache always performs better than when using LRU alone, regardless of cache size, and in some cases even better than a cache up to 40 times larger. For this data, because records in each trace are created only by a small number of users, therefore the cache performance of using PLS and PULS are essentially the same.

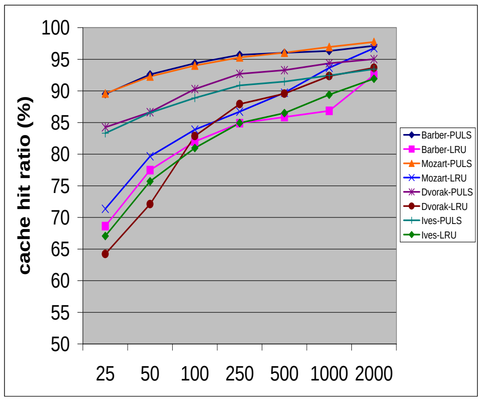

Figure 9. Cache hit ratio of LRU (labelled LRU) and LRU with PULS (labelled PULS)

Part of the reason for the dramatic performance improvement of LRU with PULS is the fact that an incorrect prediction made by PULS, 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 PULS makes program- and user-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 the meanwhile an incorrect prediction for one program executed on behalf of a particular user may more likely be accessed by other programs the same user is running than an incorrect prediction from programs run by others. In other words, the incorrect predictions by PULS are more likely to be used in the near future and are therefore less wrong than those made by other models. The earlier graph displaying predictive accuracy show performance for an effective cache size of one file and therefore do not show the performance benefit of this second-chance effect but Figure 9 clearly shows this effect. In real systems where multiple files can fit in memory at once, the performance will benefit accordingly.

5 Conclusions and Future Work

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 in terms of saving both cache space and disk bandwidth. Our simulations show that using program and user information can generate good results in predicting files, especially in eliminating the cases of incorrect prediction. Therefore, both the file prediction performance and cache hit ratio can be improved.

File accesses are driven by the users and programs using them, not by previous access patterns. By tracking programs and users initiating file accesses, we successfully avoid many incorrect predictions. About 24% of incorrect predictions can be reduced as compared with LS in some cases as our results demonstrate. Therefore, the overall performance penalty in the system caused by incorrect predictions can be significantly reduced. We also compare the cache hit ratios of LRU with and without PULS. The results show that with PULS, LRU can deliver a much higher cache hit ratio.

The DFSTrace is not very new. We chose it because it contains the program and user information, which is absolutely necessary to the PULS model. In the future, we would like to collect our own traces that PULS can use, and examine how PULS performs under more recent traces.

6 Acknowledgments

We would like to thank T. Kroeger and A. Amer for their assistance with FMOC and with their simulators. We are also grateful the comments of the other members of our research group, in particular E. Miller. Finally, we are grateful to the Coda group for providing us access to their traces.

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. 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] T. Kroeger and D. Long. Design and Implementation of a Predictive File Prefetching Algorithm. In *Proceedings of the USENIX Annual Technical Conference*, pages 105–118, 2001.

[12] 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.

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

[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] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–54, 2000.

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

[20] T. Yeh, D. 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.

[21] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of USENIX Annual Technical Conference*, pages 91–104, 2001.